SPARC COMPLIANCE DEFINITION 2.4.1

Interface Semantics

SCD 2.4.1 IS

SPARC INTERNATIONAL

June 25, 1999

FINAL DRAFT

The SPARC Compliance Interface Definition 2.3 is published and printed by SPARC International.

Any comments relating to the material contained herein may be submitted to:

SPARC International Inc.

3333 Bowers Ave., Suite 280

Santa Clara, CA 95054-2913

ATTN: Ghassan Abbas (abbas@sparc.org)

SPARC COMPLIANCE DEFINITION 2.4.1 IS

TABLE OF CONTENTS

Introduction

libaio

libc

libc 64 psABI

libdl

libelf

libposix4

libsocket

libthread

libucb

libw

Large Files Interfaces

Execution Environment

Commands

INDEX

SPARC COMPLIANCE DEFINITION 2.4.1 IS

Introduction

Introduction

This is the SPARC Compliance Definitions 2.4 Interface Semantics

This book is a companion volume to the SCD 2.4. It defines the interface semantics for those interfaces that are required by the SCD but are not specified in any other normative reference or whose semantics are different for SCD from that of a normative reference.

It is expected that many of these semantic definitions will eventually be adopted by the committees responsible for the SCD normative references. The definitions here will be deleted when and as they are added to the normative references.

Note that the SCD from 2.4 onward describes two separate ABIs, one for the 32-bit ABI and another for the 64-bit ABI. Not every interface in this document (SCD IS) applies to both ABIs. See the SCD document for specifics.

SPARC COMPLIANCE DEFINITION 2.4.1 IS

libaio

aiocancel

NAME

aiocancel - cancel an asynchronous operation

SYNOPSIS

#include <sys/asynch.h>

int aiocancel (aio_result_t *resultp)*;*

DESCRIPTION

aiocancel() cancels the asynchronous operation associated with the result buffer pointed to by resultp. It may not be possible to immediately cancel an operation which is in progress and in this case, ***aiocancel***() will not wait to cancel it.

Upon successful completion, ***aiocancel***() returns 0 and the requested operation is cancelled. The application will not receive the **SIGIO** completion signal for an asynchronous operation that is successfully cancelled.

RETURN VALUE

aiocancel() returns 0 on success, and -1 on failure and sets errno to indicate the error.

ERRORS

aiocancel() will fail if any of the following are true:

EACCES The parameter *resultp* does not correspond to any outstanding asynchronous operation, although there is at least one currently outstanding.

EINVAL There are not any outstanding requests to cancel.

aioread, aiowrite

NAME

aioread, aiowrite - asynchronous I/O operations.

SYNOPSIS

#include <sys/asynch.h>

int aioread (int fildes, char *bufp, size_t bufs, off_t offset, int whence, aio_result_t *resultp);

int aiowrite (int fildes, const char *bufp, size_t bufs, off_t offset, int whence, aio_result_t *resultp);

DESCRIPTION

aioread() initiates one asynchronous ***read***(**BA_OS**) and returns control to the calling program. The ***read***() continues concurrently with other activity of the process. An attempt is made to read bufs bytes of data from the object referenced by the descriptor fildes into the buffer pointed to by *bufp*.

aiowrite() initiates one asynchronous write(**BA_OS**) and returns control to the calling program. The write() continues concurrently with other activity of the process. An attempt is made to write bufs bytes of data from the buffer pointed to by bufp to the object referenced by the descriptor fildes.

On objects capable of seeking, the I/O operation starts at the position specified by whence and offset. These parameters have the same meaning as the corresponding parameters to the ***lseek*** (**BA_OS**) function. On objects not capable of seeking the I/O operation always start from the current position and the parameters whence and offset are ignored. The seek pointer for objects capable of seeking is not updated by ***aioread***() or ***aiowrite***(). Sequential asynchronous operations on these devices must be managed by the application using the whence and offset parameters.

The result of the asynchronous operation is stored in the structure pointed to by resultp:

```
int    aio_return;    /* return value of read() or write() */
int    aio_errno;     /* value of errno for read() or write() */
```

Upon completion of the operation both ***aio_return*** and ***aio_errno*** are set to reflect the result of the operation. **AIO_INPROGRESS** is not a value used by the system so the client may detect a change in state by initializing ***aio_return*** to this value.

The application supplied buffer bufp should not be referenced by the application until after the operation has completed. While the operation is in progress, this buffer is in use by the operating system.

Notification of the completion of an asynchronous I/O operation may be obtained synchronously through the aiowait function, or asynchronously by installing a signal handler for the SIGIO signal. Asynchronous notification is accomplished by sending the process a **SIGIO** signal. If a signal handler is not installed for the **SIGIO** signal, asynchronous notification is disabled. The delivery of this instance of the **SIGIO** signal is reliable in that a signal delivered while the handler is executing is not lost. If the client ensures that aiowait returns nothing (using a polling timeout) before returning from the signal handler, no asynchronous I/O notifications are lost. The aiowait function is the only way to dequeue an asynchronous notification. Note: **SIGIO** may have several meanings simultaneously: for example, that a descriptor generated **SIGIO** and an asynchronous operation completed. Further, issuing an asynchronous request successfully guarantees that space exists to queue the completion notification.

close(**BA_OS**), ***exit***(**BA_OS**) and ***execve***() (see ***exec***(**BA_OS**)) will block until all pending asynchronous I/O operations can be canceled by the system.

It is an error to use the same result buffer in more than one outstanding request. These structures may only be reused after the system has completed the operation.

RETURN VALUE

aioread() and ***aiowrite***() return 0 on success, and -1 on failure and set errno to indicate the error.

ERRORS

EAGAIN The number of asynchronous requests that the system can handle at any one time has been exceeded

EBADF fildes is not a valid file descriptor open for reading.

EINVAL The parameter *resultp* is currently being used by an outstanding asynchronous request.

ENOMEM Memory resources are unavailable to initiate request.

aiowait

NAME

aiowait - wait for completion of asynchronous I/O operation

SYNOPSIS

#include <sys/asynch.h>

#include <sys/time.h>

aio_result_t *aiowait (const struct timeval *timeout);

DESCRIPTION

aiowait() suspends the calling process until one of its outstanding asynchronous I/O operations completes. This provides a synchronous method of notification.

If timeout is a non-NULL pointer, it specifies a maximum interval to wait for the completion of an asynchronous I/O operation. If timeout is a NULL pointer, then aiowait() blocks indefinitely. To effect a poll, the timeout parameter should be non-zero, pointing to a zero-valued timeval structure.

The timeval structure is defined in <*sys/time.h*> and contains the following members:

long	***tv_sec; /* seconds */***
long	***tv_usec; /* and microseconds */***

The value of tv_usec is restricted to the range [0:1000000].

RETURN VALUE

On success, ***aiowait***() returns a pointer to the result structure used when the completed asynchronous I/O operation was requested, or a **NULL** pointer if the time limit expires. On failure, it returns (*aio_result_t* *)-1 and sets errno to indicate the error.

ERRORS

EINTR A signal was delivered before an asynchronous I/O operation completed.

EINVAL There are no outstanding asynchronous I/O requests (or, all outstanding asynchronous **EINVAL**. There are no outstanding asynchronous I/O requests (or, all outstanding /O requests were cancelled via aiocancel.); or tv_usec is outside of the range [0:1000000].

NOTES

aiowait() is the only way to dequeue an asynchronous notification. It may be used either inside a **SIGIO** signal handler or in the main program. One **SIGIO** signal may represent several queued events.

SPARC COMPLIANCE DEFINITION 2.4.1 IS

libc

_cleanup

NAME

_cleanup - flush all open files for writing

SYNOPSIS

void _cleanup();

DESCRIPTION

_cleanup is used to flush all open files for writing, functionally it is equivalent to ***fflush***(**NULL**).

SEE ALSO

fflush(**BA_OS**)

addseverity

NAME

addseverity - build a list of severity levels for an application for use with fmtmsg

SYNOPSIS

int addseverity(int value, const char *string)

DESCRIPTION

The function ***addseverity*** adds a new severity level of *value*. *value* must be greater than 4.

The function associates string with the level *value* so that *string* is produced with messages of that *value* yielded by ***fmtmsg()***. If a severity of *value* already exists it is replaced by the new description.

If *string* is (char *)0 then the severity level is deleted.

If *string* is a NULL string "" then the severity level is deleted.

DIAGNOSTICS

Under the following conditions, ***addseverity*** fail by returning -1, and setting errno to:

EINVAL	Using a value smaller or equal to 4.
EIVAL	If an attempt is made to delete a currently undefined severity level.

crypt, encrypt, setkey

NAME

crypt, setkey, encrypt - generate string encoding

SYNOPSIS

char *crypt (char *key, char *salt);

void setkey (char *key);

void encrypt (char *block, int edflag);

DESCRIPTION

The function ***crypt*** is a string-encoding function. The argument *key* is a string to be encoded. The argument *salt* is a two-character string chosen from the set [a-zA-Z0-9./]; this string is used to perturb the encoding algorithm, after which the string that *key* points to is used as the key to repeatedly encode a constant string. The returned value points to the encoded string. The first two characters are the *salt* itself, the remaining characters shall not be identical to the original value of *key*. The functions ***setkey*** and ***encrypt*** provide (rather primitive) access to the encoding algorithm. The argument to ***setkey*** is a 64-bit string represented by a character array of length 64 containing only the characters with numerical value 0 and 1. The string is divided into groups of 8 and the low-order bit in each group is ignored; this gives a 56-bit key. This is the key that may be used with the above mentioned algorithm to encode the string *block* with the function ***encrypt***; the encryption algorithm provided by the system may not actually use *key*. The argument *block* to encrypt is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the encoding algorithm using the key set by setkey. If the argument *edflag* is zero, the string *block* is encoded. If the *edflag* is non-zero and the implementation supports decryption then the string *block* is decoded. If the *edflag* is non-zero and the implementation does not support decryption then errno is set to **ENOSYS**.

DIAGNOSTICS

Under the following conditions, these functions fail, and set errno to:

ENOSYS ***encrypt*** was called with a non-zero value for *edflag* on a system that does not support decryption.

USAGE

The return value of the function crypt points to static data that are overwritten by each call. A portable application shall not depend on portability of encrypted data, nor assume that decryption is supported on all **SCD** conforming platforms. Also, portable applications must set errno to zero before calling any of the functions since there are no function return values for setkey or encrypt.

RATIONALE

Encryption capability is often needed by an application that wants to provide some of its own license protection. The application needs to be able to depend on the system to provide an encryption service to do this even if the system does not provide a mechanism for decryption. This standard does not require any particular underlying encryption algorithm, but only requires that the ***crypt*** function return a value that is not identical to the original. This leaves it to the system vendors to chose whatever algorithm they find to be appropriate, and alleviates any requirement for a system vendor to choose one that has export restrictions.

setlabel

NAME

setlabel - define the label for standard format messages.

SYNOPSIS

#include <pfmt.h>

int setlabel (const char *label);

DESCRIPTION

The routine ***setlabel()*** defines the label for messages produced in standard format.

label is a character string no more than 25 characters in length.

No label is defined before ***setlabel()*** is called. A NULL pointer or an empty string passed as argument will reset the definition of the *label*.

RETURN VALUE

setlabel() returns 0 in case of success, non-zero otherwise.

USAGE

The *label* should be set once at the beginning of a utility and remain constant.

SEE ALSO

getopt(BA_LIB)

sysinfo

NAME

sysinfo - get system information strings

SYNOPSIS

#include <sys/systeminfo.h>

long sysinfo (int command, char *buf, long count);

DESCRIPTION

sysinfo copies information relating to the **UNIX** system on which the process is executing into the buffer pointed to by *buf*. *count* is the size of the buffer.

The *command*s available are:

SI_SYSNAME Copy into the array pointed to by *buf* the string that would be returned by uname [see ***uname***(**BA_OS**)] in the *sysname* field. This is the name of the implementation of the operating system, for example, **UNIX_SV**.

SI_HOSTNAME Copy into the array pointed to by *buf* a string that names the present host machine. This is the string that would be returned by uname in the *nodename* field. This hostname or nodename is often the name the machine is known by locally.

The *hostname* is the name of this machine as a node in some network; different networks may have different names for the node, but presenting the nodename to the appropriate network Directory or name-to-address mapping service should produce a transport end point address. The name may not be fully qualified.

Internet host names may be up to 256 bytes in length (plus the terminating null).

SI_RELEASE Copy into the array pointed to by *buf* the string that would be returned by uname in the *release* field. Typical values might be 4.2, 4.0, 3.2.

SI_VERSION Copy into the array pointed to by *buf* the string that would be returned by uname in the *version* field. The syntax and semantics of this string are defined by the system provider.

SI_MACHINE Copy into the array pointed to by *buf* the string that would be returned by uname in the *machine* field.

SI_ARCHITECTURE Copy into the array pointed to by *buf* a string describing the instruction set architecture of the current system, for example, sparc. These names may not match predefined names in the C language compilation system.

SI_HW_PROVIDER Copies the name of the hardware manufacturer into the array pointed to by *buf*.

SI_HW_SERIAL Copy into the array pointed to by buf a string which is the **ASCII** representation of the hardware-specific serial number of the physical machine on which the system call is executed. Note that this may be implemented in Read-Only Memory, via software constants set when building the operating system, or by other means, and may contain non-numeric characters. It is anticipated that manufacturers will not issue the same “serial number” to more than one physical machine. The pair of strings returned by **SI_HW_PROVIDER** and **SI_HW_SERIAL** is likely to be unique across all vendors' System V implementations.

SI_SRPC_DOMAIN Copies the Secure Remote Procedure Call domain name into the array pointed to by *buf*.

DIAGNOSTICS

Upon successful completion, the value returned indicates the buffer size in bytes required to hold the complete value and the terminating null character. If this value is no greater than the value passed in *count*, the entire string was copied; if this value is greater than *count*, the string copied into *buf* has been truncated to *count*-1 bytes plus a terminating null character. Otherwise, a value of -1 is returned and errno is set to indicate the error.

___errno, asctime_r , ctime_r , flockfile funlockfile, getc_unlocked, getchar_unlocked gmtime_r , localtime_r, putc_unlocked putchar_unlocked, rand_r, strtok_r

NAME

___errno, asctime_r, ctime_r, gmtime_r, localtime_r, flockfile, funlockfile, getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked, rand_r, strtok_r - Support routines for multithreading added to libsys and libc.

SYNOPSIS

#include <errno.h>

int   *___errno(void);

#include <time.h>

char   *asctime_r (const struct tm *tm, char *buf, int buflen);

char   *ctime_r (const time_t *clock, char *buf, int buflen);

struct tm   *gmtime_r (const time_t *clock, struct tm *res);

struct tm   *localtime_r (const time_t *clock, struct tm *res);

#include <stdio.h>

void   flockfile (FILE *stream);

void   funlockfile (FILE *stream);

int   getc_unlocked (FILE *stream);

int   getchar_unlocked (void);

int   putc_unlocked (int c, FILE *stream);

int   putchar_unlocked (int c);

#include <stdlib.h>

int   rand_r(unsigned int *seed);

#include <string.h>

char   *strtok_r(char *s1, const char *s2, char **lasts);

DESCRIPTION and RETURN VALUES

These functions are “reentrant” versions of existing functions. They exist as the definition of the existing functions prevents the transparent implementation of multithreading, usually because of the use of a static storage area. In general, these functions are exactly equivalent to the non-reentrant versions in terms of function and results, but differ in providing for the implementation the necessary storage for completion of the function.

___errno returns the address of errno for the “calling thread”. The location labelled errno provides the storage for the “main thread” in the process. In all references to “errno” which follow, it is implied that the storage used will be that for the thread invoking the operation.

asctime_r is equivalent to ***asctime***, however the caller must supply a buffer buf in which to store the resulting string. buflen indicates the length which must be at least 26 bytes. The return value of ***asctime_r*** is a pointer to buf on success. On failure, NULL is returned and errno is set. If the operation fails because buflen is not large enough, errno will be set to **ERANGE**.

ctime_r is equivalent to ctime, however the caller must supply a buffer buf in which to store the resulting string. buflen indicates the length of buf which must be at least 26 bytes. If the operation fails because buflen is not long enough, ***ctime_r*** will return NULL and errno will be set to **ERANGE**. ***flockfile*** and ***funlockfile*** are new functions which allow the caller to gain or release exclusive access, respectively, to stream. They can be used in conjunction with a sequence of calls to ***getc*** et al. so as to avoid the overhead of locking the stream on each access to the buffers managed by stream. ***getc_unlocked***, ***getchar_unlocked***, ***putc_unlocked***, and ***putchar_unlocked*** implement an unlocked access to stream (or, for getchar the standard input and for putchar the standard output). ***gmtime_r*** is equivalent to *gmtime* but the caller must supply a result buffer res, which is the return value of the function. ***localtime_r*** is equivalent to *localtime* but the caller must supply a result buffer res, which is the return value of the function. ***rand_r*** is equivalent to ***rand*** except that a pointer to a seed seed must be supplied by the caller ***strtok_r*** is equivalent to strtok except that a pointer to a string place holder lasts must be supplied by the caller. The lasts pointer is to keep track of the next substring in which to search for the next token.

NOTES

asctime_r and ***ctime_r*** are designated as **EXPERMIMENTAL** since they have interfaces which are different from the ones in **POSIX 1003.1c**. The interfaces of these functions are in **POSIX** as following:

char	**asctime_r**	*(*	*const struct tm*	**tm,*
			char	**buf);*
char	**ctime_r**	*(*	*const time_t*	**clock,*
			char	**buf);*

priocntl

NAME

priocntl - process scheduler control

SYNOPSIS

#include <sys/types.h>

#include <sys/priocntl.h>

#include <sys/rtpriocntl.h>

#include <sys/tspriocntl.h>

long priocntl(idtype_t idtype, id_t id, int cmd, /* arg */ ...);

DESCRIPTION

priocntl() provides for control over the scheduling of an active light weight process (**LWP**). **LWPs** fall into distinct classes with a separate scheduling policy applied to each class. The two classes currently supported are the real-time class and the time-sharing class.The characteristics of these classes are described under the corresponding headings below. The class attribute of an **LWP** is inherited across the ***fork()***, ***exec()*** and ***_lwp_create()*** system calls.***priocntl()*** can be used to dynamically change the class and other scheduling parameters associated with a running **LWP** or set of **LWPs** given the appropriate permissions as explained below. In the default configuration, a runnable real-time **LWP** runs before any other **LWP**.Therefore, inappropriate use of real-time **LWP** can have a dramatic negative impact on system performance. ***priocntl()*** provides an interface for specifying a process, set of processes or an **LWP** to which the function is to apply. The ***priocntlset()*** system call provides the same functions as ***priocntl()***, but allows a more general interface for specifying the set of **LWPs** to which the function is to apply. For ***priocntl()***, the *idtype* and *id* arguments are used together to specify the set of **LWPs**. The interpretation of id depends on the value of *idtype*. The possible values for idtype and corresponding interpretations of id are as follows:

P_LWPID id is an **LWP** ID. The ***priocntl()*** system call applies to the **LWP** with the specified ID within the calling process.

P_PID id is a process ID specifying a single process.The ***priocntl()*** system call applies to all **LWPs** currently associated with the specified process.

P_PPID id is a parent process ID. The ***priocntl()*** system call applies to all LWPs currently associated with processes with the specified parent process ID.

P_PGID id is a process group ID. The ***priocntl()*** system call applies to all **LWPs** currently associated with processes in the specified process group.

P_SID id is a session ID.The ***priocntl()*** system call applies to all LWPs currently associated with processes in the specified session.

P_CID id is a class ID (returned by ***priocntl()*** **PC_GETCID** as explained below). The ***priocntl()*** system call applies to all **LWPs** in the specified class.

P_UID id is a user ID. The ***priocntl()*** system call applies to all **LWPs** with this effective user ID.

P_GID id is a group ID. The ***priocntl()*** system call applies to all **LWPs** with this effective group ID.

P_ALL The ***priocntl()*** system call applies to all existing **LWPs**.The value of id is ignored. The permission restrictions described below still apply.

An id value of **P_MYID** can be used in conjunction with the *idtype* value to specify the calling **LWP**'s **LWP** ID, parent process ID, process group ID, session ID, class ID, user ID, or group ID. In order to change the scheduling parameters of an **LWP** (using the **PC_SETPARMS** command as explained below) the real or effective user ID of the **LWP** calling ***priocntl()*** must match the real or effective user ID of the receiving **LWP** or the effective user ID of the calling **LWP** must be super-user. These are the minimum permission requirements enforced for all classes. An individual class may impose additional permissions requirements when setting **LWPs** to that class and/or when setting class-specific scheduling parameters. A special sys scheduling class exists for the purpose of scheduling the execution of certain special system processes (such as the swapper process). It is not possible to change the class of any **LWP** to sys. In addition, any processes in the sys class that are included in a specified set of processes are disregarded by ***priocntl()***. For example, an idtype of **P_UID** and an id value of zero would specify all processes with a user ID of zero except processes in the sys class and (if changing the parameters using **PC_SETPARMS**) the ***init()*** process. The *init* process is a special case.In order for a ***priocntl()*** call to change the class or other scheduling parameters of the init process (process ID 1), it must be the only process specified by idtype and id. The *init* process may be assigned to any class configured on the system, but the time-sharing class is almost always the appropriate choice. The data type and value of arg are specific to the type of command specified by cmd. A structure with the following members is used by the **PC_GETCID** and **PC_GETCLINFO** commands.

id_t	*pc_cid;*	*/* Class id */*
char	*pc_clname[PC_CLNMSZ];*	*/* Class name */*
long	*pc_clinfo[PC_CLINFOSZ];*	*/* Class information */*

pc_cid is a class ID returned by ***priocntl()*** **PC_GETCID**. *pc_clnameis* a buffer of size **PC_CLNMSZ** (defined in <*sys/priocntl.h*>) used to hold the class name (**RT** for realtime or **TS** for time-sharing). *pc_clinfo* is a buffer of size **PC_CLINFOSZ** (defined in <*sys/priocntl.h*>) used to return data describing the attributes of a specific class.The format of this data is class-specific and is described under the appropriate heading (**REAL-TIME CLASS** or **TIME-SHARING CLASS**) below. A structure with the following elements is used by the **PC_SETPARMS** and **PC_GETPARMS** commands.

id_t	*pc_cid;*	*/* **LWP** class */*
long	*pc_clparms[PC_CLPARMSZ];*	*/* Class-specific params */*

pc_cid is a class ID (returned by ***priocntl()*** **PC_GETCID**). The special class ID **PC_CLNULL** can also be assigned to *pc_cid* when using the **PC_GETPARMS** command as explained below. The *pc_clparms* buffer holds class-specific scheduling parameters.The format of this parameter data for a particular class is described under the appropriate heading below. **PC_CLPARMSZ** is the length of the *pc_clparms* buffer and is defined in <*sys/priocntl.h*>.

Commands

Available ***priocntl()*** commands are:

PC_GETCID

Get class ID and class attributes for a specific class given class name.The *idtype* and id arguments are ignored. If *arg* is non-null, it points to a structure of type *pcinfo_t*. The *pc_clname* buffer contains the name of the class whose attributes you are getting. On success, the class ID is returned in *pc_cid*, the class attributes are returned in the *pc_clinfo* buffer, and the ***priocntl()*** call returns the total number of classes configured in the system (including the sys class). If the class specified by *pc_clname* is invalid or is not currently configured the ***priocntl()*** call returns -1 with errno set to **EINVAL**. The format of the attribute data returned for a given class is defined in the <*sys/rtpriocntl.h*> or <*sys/tspriocntl.h*> header file and described under the appropriate heading below. If arg is a NULL pointer, no attribute data is returned but the ***priocntl()*** call still returns the number of configured classes.

PC_GETCLINFO

Get class name and class attributes for a specific class given class ID. The idtype and id arguments are ignored. If arg is non-null, it points to a structure of type *pcinfo_t.pc_cid* is the class ID of the class whose attributes you are getting. On success, the class name is returned in the *pc_clname* buffer, the class attributes are returned in the *pc_clinfo* buffer, and the ***priocntl()*** call returns the total number of classes configured in the system (including the sys class). The format of the attribute data returned for a given class is defined in the <*sys/rtpriocntl.h*> or <*sys/tspriocntl.h*> header file and described under the appropriate heading below. If arg is a NULL pointer, no attribute data is returned but the ***priocntl()*** call still returns the number of con figured classes.

PC_SETPARMS

Set the class and class-specific scheduling parameters of the specified **LWP**(s) associated with the specified process(es). When this command is used with the *idtype* of **P_LWPID**, it will set the class and class-specific scheduling parameters of the **LWP**. arg points to a structure of type *pcparms_t*. *pc_cid* specifies the class you are setting and the *pc_clparms* buffer contains the class-specific parameters you are setting. The format of the class-specific parameter data is defined in the <*sys/rtpriocntl.h*>or<*sys/tspriocntl.*h> header and described under the appropriate class heading below. When setting parameters for a set of **LWPs**, ***priocntl()*** acts on the **LWPs** in the set in an implementation-specific order. If ***priocntl()*** encounters an error for one or more of the target processes, it may or may not continue through the set of **LWPs**, depending on the nature of the error.If the error is related to permissions (**EPERM**), ***priocntl()*** continues through the **LWP** set, resetting the parameters for all target **LWPs** for which the calling **LWP** has appropriate permissions. ***priocntl()*** then returns -1 with errno set to **EPERM** to indicate that the operation failed for one or more of the target **LWPs**. If ***priocntl()*** encounters an error other than permissions, it does not continue through the set of target **LWPs** but returns the error immediately.

PC_GETPARMS

Get the class and/or class-specific scheduling parameters of an **LWP**. arg points to a structure of type *pcparms_t*. If *pc_cid* specifies a configured class and a single **LWP** belonging to that class is specified by the idtype and id values or the *procset* structure, then the scheduling parameters of that **LWP** are returned in the pc_clparms buffer. If the **LWP** specified does not exist or does not belong to the specified class, the ***priocntl()*** call returns -1 with errno set to **ESRCH**. If *pc_cid* specifies a configured class and a set of **LWPs** is specified, the scheduling parameters of one of the specified **LWP** belonging to the specified class are returned in the *pc_clparms* buffer and the ***priocntl()*** call returns the process ID of the selected **LWP**. The criteria for selecting an **LWP** to return in this case is class dependent. If none of the specified **LWPs** exist or none of them belong to the specified class the ***priocntl()*** call returns -1 with errno set to **ESRCH**. If pc_cid is **PC_CLNULL** and a single **LWP** is specified the class of the specified **LWP** is returned in pc_cid and its scheduling parameters are returned in the *pc_clparms* buffer.

PC_ADMIN

This command provides functionality needed for the implementation of the ***dispadmin()*** command.It is not intended for general use by other applications.

REAL-TIME CLASS

The real-time class provides a fixed priority preemptive scheduling policy for those LWPS requiring fast and deterministic response and absolute user/application control of scheduling priorities. If the real-time class is configured in the system it should have exclusive control of the highest range of scheduling priorities on the system. This ensures that a runnable real-time **LWP** is given CPU service before any **LWP** belonging to any other class. The real-time class has a range of real-time priority (*rt_pri*) values that may be assigned to an **LWP** within the class. Real-time priorities range from 0 to x, where the value of x is configurable and can be determined for a specific installation by using the ***priocntl()*** **PC_GETCID** or **PC_GETCLINFO** command. The real-time scheduling policy is a fixed priority policy. The scheduling priority of a real-time **LWP** is never changed except as the result of an explicit request by the user/application to change the *rt_pri* value of the **LWP**. For an **LWP** in the real-time class, the rt_pri value is, for all practical purposes, equivalent to the scheduling priority of the **LWP**. The rt_pri value completely determines the scheduling priority of a real-time **LWP** relative to other **LWPs** within its class.Numerically higher *rt_pri* values represent higher priorities. Since the real-time class controls the highest range of scheduling priorities in the system it is guaranteed that the runnable real-time **LWP** with the highest rt_pri value is always selected to run before any other **LWPs** in the system.

In addition to providing control over priority, ***priocntl()*** provides for control over the length of the time quantum allotted to the **LWP** in the real-time class. The time quantum value specifies the maximum amount of time an **LWP** may run assuming that it does not complete or enter a resource or event wait state (sleep).Note that if another **LWP** becomes runnable at a higher priority, the currently running **LWP** may be preempted before receiving its full time quantum. The system's process scheduler keeps the runnable real-time **LWPs** on a set of scheduling queues. There is a separate queue for each configured real-time priority and all realtime **LWPs** with a given rt_pri value are kept together on the appropriate queue. The **LWPs** on a given queue are ordered in **FIFO** order (that is, the **LWP** at the front of the queue has been waiting longest for service and receives the CPU first).Real-time **LWPs** that wake up after sleeping, **LWPs** which change to the real-time class from some other class, **LWPs** which have used their full time quantum, and runnable **LWPs** whose priority is reset by ***priocntl()*** are all placed at the back of the appropriate queue for their priority. An **LWP** that is preempted by a higher priority **LWP** remains at the front of the queue (with whatever time is remaining in its time quantum) and runs before any other **LWP** at this priority. Following a ***fork()*** or ***_lwp_create()*** system call by a real-time **LWP**, the parent **LWP** continues to run while the child **LWP** (which inherits its parent's *rt_pri* value) is placed at the back of the queue. A structure with the following members (defined in <*sys/rtpriocntl.h*>) defines the format used for the attribute data for the real-time class.

short rt_maxpri; / Maximum real-time priority */*

The ***priocntl()*** **PC_GETCID** and **PC_GETCLINFO** commands return real-time class attributes in the pc_clinfo buffer in this format. rt_maxpri specifies the configured maximum rt_pri value for the real-time class (if *rt_maxpri* is x, the valid real-time priorities range from 0 to x). A structure with the following members (defined in <*sys/rtpriocntl.h*>) defines the format used to specify the

real-time class-specific scheduling parameters of an **LWP**.

short	*rt_pri;*	*/* Real-Time priority */*
ulong	*rt_tqsecs;*	*/* Seconds in time quantum */*
long	*rt_tqnsecs;*	*/* Additional nanoseconds in quantum */*

When using the ***priocntl()*** **PC_SETPARMS** or **PC_GETPARMS** commands, if pc_cid specifies the realtime class, the data in the pc_clparms buffer is in this format. The above commands can be used to set the real-time priority to the specified value or get the current rt_pri value. Setting the rt_pri value of an **LWP** that is currently running or runnable (not sleeping) causes the **LWP** to be placed at the back of the scheduling queue for the specified priority. The **LWP** is placed at the back of the appropriate queue regardless of whether the priority being set is different from the previous rt_pri value of the **LWP**. Note that a running **LWP** can voluntarily release the CPU and go to the back of the scheduling queue at the same priority by resetting its *rt_pri* value to its current real-time priority value. In order to change the time quantum of an **LWP** without setting the priority or affecting the **LWP**'s position on the queue, the *rt_pri* field should be set to the special value **RT_NOCHANGE** (defined in <*sys/rtpriocntl.h*>). Specifying **RT_NOCHANGE** when changing the class of an **LWP** to real-time from some other class results in the real-time priority being set to zero.

For the ***priocntl()*** **PC_GETPARMS** command, if pc_cid specifies the real-time class and more than one real-time **LWP** is specified, the scheduling parameters of the real-time **LWP** with the highest rt_pri value among the specified **LWPs** are returned and the **LWP** ID of this **LWP** is returned by the ***priocntl()*** call. If there is more than one **LWP** sharing the highest priority, the one returned is implementation-dependent.

The *rt_tqsecs* and *rt_tqnsecs* fields are used for getting or setting the time quantum associated with an **LWP** or group of **LWPs**. *rt_tqsecs* is the number of seconds in the time quantum and *rt_tqnsecs* is the number of additional nanoseconds in the quantum. For example setting *rt_tqsecs* to 2 and *rt_tqnsecs* to 500,000,000 (decimal) would result in a time quantum of two and one-half seconds. Specifying a value of 1,000,000,000 or greater in the *rt_tqnsecs* field results in an error return with errno set to **EINVAL**.Although the resolution of the *tq_nsecs* field is very fine, the specified time quantum length is rounded up by the system to the next integral multiple of the system clock's resolution. The maximum time quantum that can be specified is implementation-specific and equal to **LONG_MAX** ticks (defined in <*limits.h*>). Requesting a quantum greater than this maximum results in an error return with errno set to **ERANGE** (although infinite quantum may be requested using a special value as explained below).Requesting a time quantum of zero (setting both *rt_tqsecs* and *rt_tqnsecs* to 0) results in an error return with errno set to **EINVAL**. The rt_tqnsecs field can also be set to one of the following special values (defined in <*sys/rtpriocntl.h*>), in which case the value of *rt_tqsecs* is ignored.

RT_TQINF Set an infinite time quantum.

RT_TQDEF Set the time quantum to the default for this priority (see ***rt_dptbl()***).

RT_NOCHANGE Do not set the time quantum.This value is useful when you wish to change the real-time priority of an **LWP** without affecting the time quantum.Specifying this value when changing the class of an **LWP** to real-time from some other class is equivalent to specifying **RT_TQDEF**.

In order to change the class of an **LWP** to real-time (from any other class) the **LWP** invoking ***priocntl()*** must have super-user privileges. In order to change the priority or time quantum setting of a real-time **LWP**, the **LWP** invoking ***priocntl()*** must have super-user privileges or must itself be a real-time **LWP** whose real or effective user ID matches the real of effective user ID of the target **LWP**. The real-time priority and time quantum are inherited across the ***fork()*** and ***exec()*** system calls.

TIME-SHARING CLASS

The time-sharing scheduling policy provides for a fair and effective allocation of the CPU resource among **LWPs** with varying CPU consumption characteristics. The objectives of the time-sharing policy are to provide good response time to interactive **LWPs** and good throughput to CPU-bound jobs while providing a degree of user/application control over scheduling. The time-sharing class has a range of time-sharing user priority (see *ts_upri* below) values that may be assigned to **LWPs** within the class. A *ts_upri* value of zero is defined as the default base priority for the time-sharing class. User priorities range from -x to +x where the value of x is configurable and can be determined for a specific installation by using the ***priocntl()*** **PC_GETCID** or **PC_GETCLINFO** command.

The purpose of the user priority is to provide some degree of user/application control over the scheduling of **LWPs** in the time-sharing class. Raising or lowering the *ts_upri* value of an **LWP** in the time-sharing class raises or lowers the scheduling priority of the **LWP**. It is not guaranteed however, that an **LWP** with a higher *ts_upri* value will run before one with a lower *ts_upri* value. This is because the *ts_upri* value is just one factor used to determine the scheduling priority of a time-sharing **LWP**. The system may dynamically adjust the internal scheduling priority of a time-sharing **LWP** based on other factors such as recent CPU usage. In addition to the system-wide limits on user priority (returned by the **PC_GETCID** and **PC_GETCLINFO** commands) there is a per **LWP** user priority limit (see *ts_uprilim* below), which specifies the maximum *ts_upri* value that may be set for a given **LWP**; by default, *ts_uprilim* is zero. A structure with the following members (defined in <*sys/tspriocntl.h*>) defines the format used for the attribute data for the time-sharing class.

short ts_maxupri; / Limits of user priority range */*

The ***priocntl()*** **PC_GETCID** and **PC_GETCLINFO** commands return time-sharing class attributes in the *pc_clinfo* buffer in this format. ts_maxupri specifies the configured maximum user priority value for the time-sharing class. If *ts_maxupri* is x, the valid range for both user priorities and user priority limits is from -x to +x. A structure with the following members (defined in <*sys/tspriocntl.h*>) defines the format used to specify the time-sharing class-specific scheduling parameters of an **LWP**.

short	*ts_uprilim;*	*/* Time-Sharing user priority limit */*
short	*ts_upri;*	*/* Time-Sharing user priority */*

When using the ***priocntl()*** **PC_SETPARMS** or **PC_GETPARMS** commands, if pc_cid specifies the time-sharing class, the data in the *pc_clparms* buffer is in this format. For the ***priocntl()*** **PC_GETPARMS** command, if *pc_cid* specifies the time-sharing class and more than one time-sharing **LWP** is specified, the scheduling parameters of the time-sharing **LWP** with the highest ts_upri value among the specified **LWPs** is returned and the **LWP** ID of this **LWP** is returned by the ***priocntl()*** call. If there is more than one **LWP** sharing the highest user priority, the one returned is implementation-dependent.

Any time-sharing **LWP** may lower its own *ts_uprilim* (or that of another **LWP** with the same user ID). Only a time-sharing **LWP** with super-user privileges may raise a *ts_uprilim*. When changing the class of an **LWP** to time-sharing from some other class, super-user privileges are required in order to set the initial *ts_uprilim* to a value greater than zero. Attempts by a non-super-user **LWP** to raise a *ts_uprilim* or set an initial ts_uprilim greater than zero fail with a return value of -1 and errno set to **EPERM**. Any time-sharing **LWP** may set its own ts_upri (or that of another **LWP** with the same user ID) to any value less than or equal to the **LWP**'s *ts_uprilim*. Attempts to set the *ts_upri* above the *ts_uprilim* (and/or set the *ts_uprilim* below the *ts_upri*) result in the *ts_upri* being set equal to the *ts_uprilim*.

Either of the ts_uprilim or *ts_upri* fields may be set to the special value **TS_NOCHANGE** (defined in <*sys/tspriocntl.h*>) in order to set one of the values without affecting the other. Specifying **TS_NOCHANGE** for the *ts_upri* when the *ts_uprilim* is being set to a value below the current *ts_upri* causes the *ts_upri* to be set equal to the ts_uprilim being set. Specifying **TS_NOCHANGE** for a parameter when changing the class of an **LWP** to time-sharing (from some other class) causes the parameter to be set to a default value. The default value for the *ts_uprilim* is 0 and the default for the *ts_upri* is to set it equal to the *ts_uprilim* which is being set. The time-sharing user priority and user priority limit are inherited across the fork and exec functions.

RETURN VALUES

Unless otherwise noted above, ***priocntl()*** returns a value of 0 on success.***priocntl()*** returns -1 on failure and sets errno to indicate the error.

ERRORS

priocntl() fails if one or more of the following are true :

EAGAIN An attempt to change the class of an **LWP** failed because of insufficient resources other than memory (for example, class-specific kernel data structures).

EFAULT One of the arguments points to an illegal address.

EINVAL The argument cmd was invalid, an invalid or unconfigured class was specified, or one of the parameters specified was invalid.

EFAULT

ENOMEM An attempt to change the class of an **LWP** failed because of insufficient memory. The effective user of the calling **LWP** is not super-user.

ERANGE The requested time quantum is out of range.

ESRCHN one of the specified **LWPs** exist.

SEE ALSO

priocntl(), ***dispadmin()***, ***init()***, ***_lwp_create()***, ***exec()***, ***fork()***, ***nice()***, ***priocntlset()***, ***rt_dptbl()***

strftime, cftime, ascftime

NAME

strftime, cftime, ascftime - convert date and time to string

SYNOPSIS

#include <time.h>

size_t strftime(const char *s, size_t maxsize, const char *format, const struct tm *timeptr);

int cftime(char *s, char *format, const time_t *clock);

int ascftime(char *s, const char *format, const struct tm *timeptr);

DESCRIPTION

strftime(), ***ascftime()***, and ***cftime()*** place bytes into the array pointed to by s as controlled by the string pointed to by format. The format string consists of zero or more conversion specifications and ordinary characters.A conversion specification consists of a '%' (percent) character and one or two terminating conversion characters that determine the conversion specification's behavior.All ordinary characters (including the terminating null byte) are copied unchanged into the array pointed to by s.If copying takes place between objects that overlap, the behavior is undefined. For ***strftime()***, no more than maxsize bytes are placed into the array. If format is (char *)0, then the locale's default format is used.For ***strftime()*** the default format is the same as %c; for ***cftime()*** and ***ascftime()*** the default format is the same as %C.***cftime()*** and ***ascftime()*** first try to use the value of the environment variable **CFTIME**, and if that is undefined or empty, the default format is used. Each conversion specification is replaced by appropriate characters as described in the following list.The appropriate characters are determined by the **LC_TIME** category of the program's locale and by the values contained in the structure pointed to by timeptr for ***strftime()*** and ***ascftime()***, and by the time represented by clock for ***cftime()***.

%%	same as %
%a	locale's abbreviated weekday name
%A	locale's full weekday name
%b	locale's abbreviated month name
%B	locale's full month name
%c	locale's appropriate date and time representation Default
%C	locale's date and time representation as produced by ***date()***

<u>Standard-conforming</u>

%C	century number (the year divided by 100 and truncated to an integer as a decimal number [1,99]); single digits are preceded by 0; see ***standards()***
%d	day of month [1,31]; single digits are preceded by 0
%D	date as %m/%d/%y
%e	day of month [1,31]; single digits are preceded by a space
%h	locale's abbreviated month name
%H	hour (24-hour clock) [0,23]; single digits are preceded by 0
%I	hour (12-hour clock) [1,12]; single digits are preceded by 0

%j day number of year [1,366]; single digits are preceded by 0

%k hour (24-hour clock) [0,23]; single digits are preceded by a blank

%l hour (12-hour clock) [1,12]; single digits are preceded by a blank

%m month number [1,12]; single digits are preceded by 0

%M minute [00,59]; leading zero is permitted but not required

%n insert a newline

%p locale's equivalent of either a.m. or p.m.

%r appropriate time representation in 12-hour clock format with %p

%R time as %H:%M

%S seconds [00,61]

%t insert a tab

%T time as %H:%M:%S

%u weekday as a decimal number [1,7], with 1 representing Sunday

%U week number of year as a decimal number [00,53], with Sunday as the first day of week 1

%V week number of the year as a decimal number [01,53], with Monday as the first day of the week. If the week containing 1 January has four or more days in the new year, then it is considered week 1; otherwise, it is week 53 of the previous year, and the next week is week 1.

%w weekday as a decimal number [0,6], with 0 representing Sunday

%W week number of year as a decimal number [00,53], with Monday as the first day of week 1

%x locale's appropriate date representation

%X locale's appropriate time representation

%y year within century [00,99]

%Y year, including the century (for example 1993)

%Z time zone name or abbreviation, or no bytes if no time zone information exists

If a conversion specification does not correspond to any of the above or to any of the modified conversion specifications listed below, the behavior is undefined and 0 is returned. The difference between %U and %W (and also between modified conversion specifications %OU and %OW) lies in which day is counted as the first of the week.Week number 1 is the first week in January starting with a Sunday for %U or a Monday for %W. Week number 0 contains those days before the first Sunday or Monday in January for %U and %W, respectively.

Modified Conversion Specifications

Some conversion specifications can be modified by the E and O modifiers to indicate that an alternate format or specification should be used rather than the one normally used by the unmodified conversion specification. If the alternate format or specification does not exist in the current locale, the behavior will be as if the unmodified specification were used.

%Ec locale's alternate appropriate date and time representation

%EC name of the base year (period) in the locale's alternate representation

%Ex locale's alternate date representation

%EX	locale's alternate time representation
%Ey	offset from %EC (year only) in the locale's alternate representation
%EY	full alternate year representation
%Od	day of the month using the locale's alternate numeric symbols
%Oe	same as %Od
%OH	hour (24-hour clock) using the locale's alternate numeric symbols
%OI	hour (12-hour clock) using the locale's alternate numeric symbols
%Om	month using the locale's alternate numeric symbols
%OM	minutes using the locale's alternate numeric symbols
%OS	seconds using the locale's alternate numeric symbols
%Ou	weekday as a number in the locale's alternate numeric symbols
%OU	week number of the year (Sunday as the first day of the week) using the locale's alternate numeric symbols
%Ow	number of the weekday (Sunday=0) using the locale's alternate numeric symbols
%OW	week number of the year (Monday as the first day of the week) using the locale's alternate numeric symbols
%Oy	year (offset from %C) in the locale's alternate representation and using the locale's alternate numeric symbols

Selecting the Output Language

By default, the output of ***strftime()***, ***cftime()***, and ***ascftime()*** appear in U.S. English. The user can request that the output of ***strftime()***, ***cftime()***, or ***ascftime()*** be in a specific language by setting the **LC_TIME** category using ***setlocale()***.

Time Zone

Local time zone information is used as though ***tzset()*** were called.

RETURN VALUES

strftime(), ***cftime()***, and ***ascftime()*** return the number of characters placed into the array pointed to by s, not including the terminating null character.If the total number of resulting characters including the terminating null character is more than maxsize, ***strftime()*** returns 0 and the contents of the array are indeterminate.

SEE ALSO

date(), ***ctime()***, ***mktime()***, ***setlocale()***, ***strptime()***, ***tzset()***, ***TIMEZONE()***, ***attributes()***, ***environ()***

NOTES

The range of values for %S is [00,61] rather than [00,59] to allow for the occasional leap second and even more occasional double leap second.

syslog, openlog, closelog, setlogmask

NAME

syslog, openlog, closelog, setlogmask - control system log

SYNOPSIS

#include <syslog.h>

void openlog(const char *ident, int logopt, int facility);

void syslog(int priority, const char *message,... /* arguments */);

void closelog(void);

int setlogmask(int maskpri);

DESCRIPTION

The ***syslog()*** function sends a message to ***syslogd()***, which, depending on the configuration of /etc/syslog.conf, logs it in an appropriate system log, writes it to the system console, forwards it to a list of users, or forwards it to syslogd on another host over the network. The logged message includes a message header and a message body. The message header consists of a facility indicator, a severity level indicator, a timestamp, a tag string, and optionally the process ID. The message body is generated from the message and following arguments in the same manner as if these were arguments to ***printf()***, except that occurrences of %m in the format string pointed to by the message argument are replaced by the error message string associated with the current value of errno. A trailing **NEWLINE** character is added if needed. Values of the priority argument are formed by ORing together a severity level value and an optional facility value. If no facility value is specified, the current default facility value is used. Possible values of severity level include:

LOG_EMERG	A panic condition.This is normally broadcast to all users.
LOG_ALERT	A condition that should be corrected immediately, such as a corrupted system database.
LOG_CRIT	Critical conditions, such as hard device errors.
LOG_ERR	Errors.
LOG_WARNING	Warning messages.
LOG_NOTICE	Conditions that are not error conditions, but that may require special handling.
LOG_INFO	Informational messages.
LOG_DEBUG	Messages that contain information normally of use only when debugging a program.

The facility indicates the application or system component generating the message. Possible facility values include:

LOG_KERN	Messages generated by the kernel. These cannot be generated by any user processes.
LOG_USER	Messages generated by random user processes. This is the default facility identifier if none is specified.
LOG_MAIL	The mail system.
LOG_DAEMON	System daemons, such as in.***ftpd()***.

LOG_AUTH The authorization system: ***login()***, ***su()***, ***getty()***.

LOG_LPR The line printer spooling system: ***lpr***(1B), ***lpc***(1B).

LOG_NEWS Reserved for the **USENET** network news system.

LOG_UUCP Reserved for the **UUCP** system; it does not currently use syslog.

LOG_CRON The cron/at facility; ***crontab()***, ***at()***,***cron()***.

LOG_LOCAL0-7 Reserved for local use.

The ***openlog()*** function sets process attributes that affect subsequent calls to ***syslog()***.The ident argument is a string that is pre-appended to every message. The logopt argument indicates logging options. Values for logopt are constructed by a bitwise-inclusive OR of zero or more of the following:

LOG_PID Log the process ID with each message. This is useful for identifying specific daemon processes (for daemons that fork).

LOG_CONS Write messages to the system console if they cannot be sent to ***syslogd()***. This option is safe to use in daemon processes that have no controlling terminal, since ***syslog()*** forks before opening the console.

LOG_NDELAY Open the ***connectiontosyslogd()*** immediately. Normally the open is delayed until the first message is logged.This is useful for programs that need to manage the order in which file descriptors are allocated.

LOG_ODELAY Delay open until ***syslog()*** is called.

LOG_NOWAIT Do not wait for child processes that have been forked to log messages onto the console. This option should be used by processes that enable notification of child termination using **SIGCHLD**, since ***syslog()*** may otherwise block waiting for a child whose exit status has already been collected.

The facility argument encodes a default facility to be assigned to all messages that do not have an explicit facility already encoded.The initial default facility is **LOG_USER**. The ***openlog()*** and ***syslog()*** functions may allocate a file descriptor.It is not necessary to call ***openlog()*** prior to calling ***syslog()***. The ***closelog()*** function closes any open file descriptors allocated by previous calls to ***openlog()*** or ***syslog()***. The ***setlogmask()*** function sets the log priority mask for the current process to *maskpri* and returns the previous mask. If the maskpri argument is 0, the current log mask is not modified.Calls by the current process to ***syslog()*** with a priority not set in maskpri are rejected. The mask for an individual priority *pri* calculated by the macro **LOG_MASK**(*pri*); the mask for all priorities up to and including *toppri* is given by the macro **LOG_UPT**(*toppri*). The default log mask allows all priorities to be logged. Symbolic constants for use as values of the logopt, facility, priority, and maskpri arguments are defined in the <*syslog.h*> header.

RETURN VALUES

The ***setlogmask()*** function returns the previous log priority mask.The ***closelog()***, ***openlog()*** and ***syslog()*** functions return no value.

SEE ALSO

at(), ***crontab()***, ***logger()***, ***login()***, ***lpc***(), ***lpr***(), ***cron()***,***getty()***,in.***ftpd()***, ***su()***, ***syslogd()***, ***printf()***, ***syslog.conf()***

dbm_clearerr, dbm_close, dbm_delete, dbm_error, dbm_fetch dbm_firstkey, dbm_nextkey, dbm_open, dbm_store

NAME

dbm_clearerr, dbm_close, dbm_delete, dbm_error, dbm_fetch, dbm_firstkey, dbm_nextkey, dbm_open, dbm_store - database functions

SYNOPSIS

#include <ndbm.h>

int dbm_clearerr(DBM *db);

void dbm_close(DBM *db);

int dbm_delete(DBM *db, datum key);

int dbm_error(DBM *db);

datum dbm_fetch(DBM *db, datum key);

datum dbm_firstkey(DBM *db);

datum dbm_nextkey(DBM *db);

DBM *dbm_open(const char *file, int open_flags, mode_t file_mode);

int dbm_store(DBM *db, datum key, datum content, int store_mode);

DESCRIPTION

These functions create, access and modify a database.They maintain key/content pairs in a database. The functions will handle large databases (up to a billion blocks) and will access a keyed item in one or two file system accesses. This package replaces the earlier ***dbm()*** library, which managed only a single database. keys and contents are described by the datum typedef. A datum consists of at least two members, dptr and dsize. The dptr member points to an object that is dsize bytes in length.Arbitrary binary data, as well as ASCII character strings, may be stored in the object pointed to by dptr. The database is stored in two files. One file is a directory containing a bit map of keys and has .dir as its suffix. The second file contains all data and has *.pag* as its suffix.

The ***dbm_open()*** function opens a database. The file argument to the function is the pathname of the database. The function opens two files named *file.dir* and *file.pag*.The open_flags argument has the same meaning as the flags argument of ***open()*** except that a database opened for write-only access opens the files for read and write access. The file_mode argument has the same meaning as the third argument of ***open()***. The ***dbm_close()*** function closes a database. The argument db must be a pointer to a dbm structure that has been returned from a call to ***dbm_open()***. The ***dbm_fetch()*** function reads a record from a database. The argument db is a pointer to a database structure that has been returned from a call to ***dbm_open()***.The argument key is a datum that has been initialized by the application program to the value of the key that matches the key of the record the program is fetching. The ***dbm_store()*** function writes a record to a database. The argument db is a pointer to a database structure that has been returned from a call to ***dbm_open()***. The argument key is a datum that has been initialized by the application program to the value of the key that identifies (for subsequent reading, writing or deleting) the record the program is writing. The argument content is a datum that has been initialized by the application program to the value of the record the program is writing. The argument store_mode controls whether ***dbm_store()*** replaces any pre-existing record that has the same key that is specified by the key argument. The application program must set store_mode to either **DBM_INSERT** or **DBM_REPLACE**.If the database contains a record that matches the key argument and store_mode is **DBM_REPLACE**, the existing record is replaced with the new record.If the database contains a record that matches the key argument and store_mode is **DBM_INSERT**, the existing record is not replaced with the new record. If the database does not contain a record that matches the key argument and store_mode is either **DBM_INSERT** or **DBM_REPLACE**, the new record is inserted in the database.

The ***dbm_delete()*** function deletes a record and its key from the database.The argument db is a pointer to a database structure that has been returned from a call to ***dbm_open()***. The argument key is a datum that has been initialized by the application program to the value of the key that identifies the record the program is deleting. The ***dbm_firstkey()*** function returns the first key in the database. The argument db is a pointer to a database structure that has been returned from a call to ***dbm_open()***. The ***dbm_nextkey()*** function returns the next key in the database.The argument db is a pointer to a database structure that has been returned from a call to ***dbm_open()***.The ***dbm_firstkey()*** function must be called before calling. Subsequent calls to ***dbm_nextkey()*** return the next key until all of the keys in the database have been returned. The ***dbm_error()*** function returns the error condition of the database. The argument db is a pointer to a database structure that has been returned from a call to ***dbm_open()***. The ***dbm_clearerr()*** function clears the error condition of the database.The argument db is a pointer to a database structure that has been returned from a call to ***dbm_open()***. These database functions support key/content pairs of at least 1024 bytes.

RETURN VALUES

The ***dbm_store()*** and ***dbm_delete()*** functions return 0 when they succeed and a negative value when they fail. The ***dbm_store()*** function returns 1 if it is called with a flags value of **DBM_INSERT** and the function finds an existing record with the same key. The ***dbm_error()*** function returns 0 if the error condition is not set and returns a non-zero value if the error condition is set. The return value of ***dbm_clearerr()*** is unspecified. The ***dbm_firstkey()*** and ***dbm_nextkey()*** functions return a key datum.When the end of the database is reached, the dptr member of the key is a null pointer.If an error is detected, the dptr member of the key is a null pointer and the error condition of the database is set. The ***dbm_fetch()*** function returns a content datum.If no record in the database matches the key or if an error condition has been detected in the database, the dptr member of the content is a null pointer. The ***dbm_open()*** function returns a pointer to a database structure.If an error is detected during the operation, ***dbm_open()*** returns a (**DBM** *)0.

SEE ALSO

ar(), cat(), cp(), tar(), open(), dbm(), netconfig()

NOTES

The *.pag* file will contain holes so that its apparent size may be larger than its actual content. Older versions of the UNIX operating system may create real file blocks for these holes when touched. These files cannot be copied by normal means (***cp()***, ***cat()***, ***tar()***, ***ar()***) without filling in the holes. The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover all key/content pairs that hash together must fit on a single block.***dbm_store()*** will return an error in the event that a disk block fills with inseparable data. The order keys represented by ***dbm_firstkey()*** and ***dbm_nextkey()*** depends on a hashing function. There are no interlocks and no reliable cache flushing; thus concurrent updating and reading is risky. The database files (file.dir and file.pag) are binary and are architecture-specific (for example, they depend on the architecture's byte order.) These files are not guaranteed to be portable across architectures.

decimal_to_floating, decimal_to_single, decimal_to_double decimal_to_extended, decimal_to_quadruple

NAME

decimal_to_floating, decimal_to_single, decimal_to_double, decimal_to_extended, decimal_to_quadruple - convert decimal record to floating-point value

SYNOPSIS

#include <floatingpoint.h>

void decimal_to_single(single *px, decimal_mode *pm, decimal_record *pd, fp_exception_field_type *ps);

void decimal_to_double(double *px, decimal_mode *pm, decimal_record *pd, fp_exception_field_type *ps);

void decimal_to_extended(extended *px, decimal_mode *pm, decimal_record *pd, fp_exception_field_type *ps);

void decimal_to_quadruple(quadruple *px, decimal_mode *pm, decimal_record *pd, fp_exception_field_type *ps);

DESCRIPTION

The ***decimal_to_floating()*** functions convert the decimal record at *pd into a floating-point value at *px, observing the modes specified in *pm and setting exceptions in **ps*. If there are no IEEE exceptions, *ps will be zero. pd->sign and *pd->fpclass* are always taken into account. *pd->exponent, pd->ds* and pd->ndigits are used when pd->fpclass is *fp_normal* or *fp_subnormal*. In these cases pd->ds must contain one or more ascii digits followed by a NULL and pd->ndigits is assumed to be the length of the string pd->ds.Notice that for efficiency reasons, the assumption that pd->ndigits == ***strlen***(pd->ds) is NEVER verified.

On output, *px is set to a correctly rounded approximation to (*pd->sign*)*(pd->ds)*10**(pd->exponent) Thus if *pd->exponent* == -2 and pd->ds == "1234", *px will get 12.34 rounded to storage precision. pd->ds cannot have more than **DECIMAL_STRING_LENGTH**-1 significant digits because one character is used to terminate the string with a NULL. If pd->more != 0 on input then additional nonzero digits follow those in pd->ds; *fp_inexact is* set accordingly on output in *ps.

*px is correctly rounded according to the IEEE rounding modes in pm->rd.*ps is set to contain *fp_inexact*, *fp_underflow*, or *fp_overflow* if any of these arise. ***decimal_to_floating()*** C Library Functions ***decimal_to_floating()***pm->*df* and *pm->ndigits* are not used. ***strtod()***, ***scanf()***, ***fscanf()***, and ***sscanf()*** all use ***decimal_to_double()***.

SEE ALSO

fscanf(), scanf(), sscanf(), strtod()

floating_to_decimal, single_to_decimal, double_to_decimal, extended_to_decimal, quadruple_to_decimal

NAME

floating_to_decimal, single_to_decimal, double_to_decimal, extended_to_decimal, quadruple_to_decimal - convert floating-point value to decimal record

SYNOPSIS

#include <floatingpoint.h>

void single_to_decimal(single *px, decimal_mode *pm, decimal_record *pd, fp_exception_field_type *ps);

void double_to_decimal(double *px, decimal_mode *pm, decimal_record *pd, fp_exception_field_type *ps);

void extended_to_decimal(extended *px, decimal_mode *pm, decimal_record *pd, fp_exception_field_type *ps);

void quadruple_to_decimal(quadruple *px, decimal_mode *pm, decimal_record *pd, fp_exception_field_type *ps);

DESCRIPTION

The ***floating_to_decimal()*** functions convert the floating-point value at *px into a decimal record at *pd, observing the modes specified in *pm and setting exceptions in *ps. If there are no IEEE exceptions, *ps will be zero. If *px is zero, infinity, or NaN, then only *pd->sign* and *pd->fpclas*s are set. Otherwise *pd->exponent* and *pd->ds* are also set so that *(pd->sign)*(pd->ds)*10**(pd->exponent)* is a correctly rounded approximation to **px. pd->ds* has at least one and no more than **DECIMAL_STRING_LENGTH**-1 significant digits because one character is used to terminate the string with a NULL. pd->ds is correctly rounded according to the IEEE rounding modes in pm->rd. *ps has *fp_inexact* set if the result was inexact, and has *fp_overflow* set if the string result does not fit in *pd->ds* because of the limitation **DECIMAL_STRING_LENGTH**. If *pm->df == floating_form*, then *pd->ds* always contains pm->ndigits significant digits. Thus if **px == 12.34* and *pm->ndigits == 8*, then *pd->ds* will contain 12340000 and pd->exponent will contain -6. If *pm->df == fixed_form* and *pm->ndigits >= 0*, then *pd->ds* always contains pm->ndigits after the point and as many digits as necessary before the point. Since the latter is not known in advance, the total number of digits required is ***floating_to_decimal()*** C Library Functions ***floating_to_decimal()*** returned in *pd->ndigits*; if that number >= **DECIMAL_STRING_LENGTH**, then ds is undefined. *pd->exponent* always gets *-pm->ndigits*. Thus if **px == 12.34* and pm->ndigits == 1, then *pd->ds* gets 123, *pd->exponent* gets -1, and *pd->ndigits* gets 3. If *pm->df == fixed_form* and *pm->ndigits < 0*, then *pd->ds* always contains *-pm->ndigits* trailing zeros; in other words, rounding occurs -*pm->ndigits* to the left of the decimal point, but the digits rounded away are retained as zeros. The total number of digits required is in *pd->ndigits.pd->exponent* always gets 0. Thus if **px == 12.34* and *pm->ndigits == -1*, then pd->ds gets 10, pd->exponent gets 0, and pd->ndigits gets 2. pd->more is not used. ***econvert()***, ***fconvert()***, ***gconvert()*** ,***printf()***, and ***sprintf()*** all use ***double_to_decimal()***.

SEE ALSO

econvert(), fconvert(), gconvert(), printf(), sprintf(), attributes()

string_to_decimal, file_to_decimal, func_to_decimal

NAME

string_to_decimal, file_to_decimal, func_to_decimal - parse characters into decimal record

SYNOPSIS

#include <floatingpoint.h>

void string_to_decimal(char **pc, int nmax, int fortran_conventions, decimal_record *pd,
enum decimal_string_form *pform, char **pechar);

void func_to_decimal(char **pc, int nmax, int fortran_conventions, decimal_record *pd,
enum decimal_string_form *pform, char **pechar,
int (*pget)(void), int *pnread, int (*punget)(int c));

#include <stdio.h>

void file_to_decimal(char **pc, int nmax, int fortran_conventions, decimal_record *pd,
enum decimal_string_form *pform,
char **pechar, FILE *pf, int *pnread);

DESCRIPTION

The ***char_to_decimal*** functions parse a numeric token from at most *nmax* characters in a string **pc or file *pf or function (**pget*)***()*** into a decimal record **pd*, classifying the form of the string in **pform* and **pechar*. The accepted syntax is intended to be sufficiently flexible to accommodate many languages: whitespace value or whitespace sign value where whitespace is any number of characters defined by *isspace* in <*ctype.h*>, sign is either of [+-], and value can be number, nan, or *inf*. *inf* can be INF (*inf_form*) or INFINITY (*infinity_form*) without regard to case. nan can be **NAN** (*nan_form*) or NAN(*nstring*) (*nanstring_form*) without regard to case; *nstring* is any string of characters not containing ')' or NULL; *nstring* is copied to pd->ds and, currently, not used subsequently. number consists of significand or significand *efield* where significand must contain one or more digits and may contain one point; possible forms are

digits (int_form)

digits.(intdot_form)

.digits(dotfrac_form)

digits.digits (intdotfrac_form)

efield consists of *echar* digits or *echar* sign digits where *echar* is one of [Ee], and digits contains one or more digits.

When *fortran_converstion* is nonzero, additional input forms are accepted according to various Fortran conventions:

0 no Fortran conventions

1 Fortran list-directed input conventions

2 Fortran formatted input conventions, ignore blanks (BN)

3 Fortran formatted input conventions, blanks are zeros (BZ)

When *fortran_conventions* is nonzero, *echar* may also be one of [**DdQq**], and *efield* may also have the form sign digits. When *fortran_conventions*>= 2, blanks may appear in the digits strings for the integer, fraction, and exponent fields and may appear between *echar* and the exponent sign and after the infinity and NaN forms. If *fortran_conventions*== 2, the blanks are ignored.When *fortran_conventions*== 3, the blanks that appear in digits strings are interpreted as zeros, and other blanks are ignored.

When *fortran_conventions* is zero, the current locale's decimal point character is used as the decimal point; when *fortran_conventions* is nonzero, the period is used as the decimal point. The form of the accepted decimal string is placed in **pform*. If an *efield* is recognized, **pechar* is set to point to the *echar*.

On input, *pc points to the beginning of a character string buffer of *length* >= *nmax*. On output, *pc points to a character in that buffer, one past the last accepted character. ***string_to_decimal()*** gets its characters from the buffer; ***file_to_decimal()*** gets its characters from **pf* and records them in the buffer, and places a null after the last character read. ***func_to_decimal()*** gets its characters from an int function (**pge*)***()***

The scan continues until no more characters could possibly fit the acceptable syntax or until nmax characters have been scanned. If the nmax limit is not reached then at least one extra character will usually be scanned that is not part of the accepted syntax. ***file_to_decimal()*** and ***func_to_decimal()*** set **pnread* to the number of characters read from the file; if greater than nmax, some characters were lost.If no characters were lost, ***file_to_decimal()*** and ***func_to_decimal()*** attempt to push back, with ***ungetc()*** or (****punget***)***()*** as many as possible of the excess characters read, adjusting *pnread accordingly. If all ***unget*** calls are successful, then ***pc* will be NULL. No push back will be attempted if (****punget***)***()*** is NULL.

Typical declarations for ****pget()*** and ****punget()*** are:

```
int xget(void)
{ ...  }
int (*pget)(void) = xget;
int xunget(int c)
{ ... }
int (*punget)(int) = xunget;
```

If no valid number was detected, pd->fpclass is set to *fp_signaling*, *pc is unchanged, and *pform is set to *invalid_form*. ***atof()*** and ***strtod()*** use ***string_to_decimal()***.***scanf()*** uses ***file_to_decimal()***.

SEE ALSO

ctype(), localeconv(), scanf(), setlocale(), strtod(), ungetc()

econvert, fconvert, gconvert, seconvert, sfconvert sgconvert, qeconvert, qfconvert, qgconvert ecvt, fcvt, gcvt

NAME

econvert, fconvert, gconvert, seconvert, sfconvert, sgconvert, qeconvert, qfconvert, qgconvert, ecvt, fcvt, gcvt - output conversion

SYNOPSIS

#include <floatingpoint.h>

char *econvert(double value, int ndigit, int *decpt, int *sign, char *buf);

char *fconvert(double value, int ndigit, int *decpt, int *sign, char *buf);

char *gconvert(double value, int ndigit, int trailing, char *buf);

char *seconvert(single *value, int ndigit, int *decpt, int *sign, char *buf);

char *sfconvert(single *value, int ndigit, int *decpt, int *sign, char *buf);

char *sgconvert(single *value, int ndigit, int trailing, char *buf);

char *qeconvert(quadruple *value, int ndigit, int *decpt, int *sign, char *buf);

char *qfconvert(quadruple *value, int ndigit, int *decpt, int *sign, char *buf);

char *qgconvert(quadruple *value, int ndigit, int trailing, char *buf);

char *ecvt(double value, int ndigit, int *decpt, int *sign);

char *fcvt(double value, int ndigit, int *decpt, int *sign);

char *gcvt(double value, int ndigit, char *buf);

DESCRIPTION

The ***econvert()*** function converts the value to a null-terminated string of ndigit ASCII digits in buf and returns a pointer to buf. buf should contain at least *ndigit*+1 characters.The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt*. Thus buf == "314" and **decpt* == 1 corresponds to the numerical value 3.14, while buf == "314" and **decpt* == -1 corresponds to the numerical value .0314. If the sign of the result is negative, the word pointed to by sign is nonzero; otherwise it is zero. The least significant digit is rounded.

The ***fconvert()*** function works much like ***econvert()***, except that the correct digit has been rounded as if for ***sprintf***(%w.nf) output with n=*ndigit* digits to the right of the decimal point.ndigit can be negative to indicate rounding to the left of the decimal point. The return value is a pointer to buf.buf should contain at least 310+max(0,*ndigit*) characters to accommodate any double-precision value.

The ***gconvert()*** function converts the value to a null-terminated ASCII string in buf and returns a pointer to buf. It produces ndigit significant digits in fixed-decimal format, like sprintf(%w.nf), if possible, and otherwise in floating-decimal format, like sprintf(%w.ne); in either case buf is ready for printing, with sign and exponent. The result corresponds to that obtained by (void) sprintf(buf,``%w.ng",value) ; If *trailing*= 0, trailing zeros and a trailing point are suppressed, as in sprintf(%g). If trailing!= 0, trailing zeros and a trailing point are retained, as in sprintf(%#g).

The ***seconvert()***, ***sfconvert()***, and ***sgconvert()*** functions are single-precision versions of these functions, and are more efficient than the corresponding double-precision versions. pointer rather than the value itself is passed to avoid C's usual conversion of single-precision arguments to double.

The ***qeconvert()***, ***qfconvert()***, and ***qgconvert()*** functions are quadruple-precision versions of these functions. The ***qfconvert()*** function can overflow the decimal_record field ds if value is too large. In that case, buf[0] is set to zero.

The ***ecvt()*** and ***fcvt()*** functions are versions of ***econvert()*** and ***fconvert()*** that create a string in a static data area, overwritten by each call, and return values that point to that static data. These functions are therefore not reentrant.

The ***gcvt()*** function is an version of ***gconvert()*** that always suppresses trailing zeros and point.

IEEE Infinities and **NaNs** are treated similarly by these functions.``NaN'' is returned for NaN, and ``Inf'' or ``Infinity'' for Infinity. The longer form is produced when ndigit >= 8.

SEE ALSO

sprintf()

getspnam, getspnam_r, getspent, getspent_r, setspent endspent, fgetspent, fgetspent_r

NAME

getspnam, getspnam_r,getspent,getspent_r,setspent, endspent, fgetspent, fgetspent_r - get password entry

SYNOPSIS

#include <shadow.h>

struct spwd	**getspnam(const char *name);*
struct spwd	**getspnam_r(const char *name, struct spwd *result, char *buffer, int buflen);*
struct spwd	**getspent(void);*
struct spwd	**getspent_r(struct spwd *result, char *buffer, int buflen);*
void	*setspent(void);*
void	*endspent(void);*
struct spwd	**fgetspent(FILE *fp);*
struct spwd	**fgetspent_r(FILE *fp, struct spwd *result, char *buffer, int buflen);*

DESCRIPTION

These functions are used to obtain shadow password entries. An entry may come from any of the sources for shadow specified in the /etc/nsswitch.conf file (see nsswitch.***conf()***). ***getspnam()*** searches for a shadow password entry with the login name specified by the character string parameter name. The functions ***setspent()***, ***getspent()***, and ***endspent()*** are used to enumerate shadow password entries from the database. ***setspent()*** sets (or resets) the enumeration to the beginning of the set of shadow password entries. This function should be called before the first call to ***getspent()***.Calls to ***getspnam()*** leave the enumeration position in an indeterminate state. Successive calls to ***getspent()*** return either successive entries or NULL, indicating the end of the enumeration. ***endspent()*** may be called to indicate that the caller expects to do no further shadow password retrieval operations; the system may then close the shadow password file, deallocate resources it was using, and so forth. It is still allowed, but possibly less efficient, for the process to call more shadow password functions after calling ***endspent()***. ***fgetspent()***, unlike the other functions above, does not use nsswitch.conf; it reads and parses the next line from the stream f, which is assumed to have the format of the shadow file (see ***shadow()***).

Reentrant Interfaces

The functions ***getspnam()***, ***getspent()***, and ***fgetspent()*** use static storage that is re-used in each call, making these routines unsafe for use in multithreaded applications. The functions: ***getspnam_r()***, ***getspent_r()***, and ***fgetspent_r()*** provide reentrant interfaces for these operations. Each reentrant interface performs the same operation as its non-reentrant counterpart, named by removing the ``_r'' suffix. The reentrant interfaces, however, use buffers supplied by the caller to store returned results, and are safe for use in both single-threaded and multithreaded applications. Each reentrant interface takes the same parameters as its non-reentrant counterpart, as well as the following additional parameters. The parameter result must be a pointer to a struct *spwd* structure allocated by the caller. On successful completion, the function returns the shadow password entry in this structure.The parameter buffer must be a pointer to a buffer supplied by the caller. This buffer is used as storage space for the shadow password data. All of the pointers within the returned struct *spwd* result point to data stored within this buffer. The buffer must be large enough to hold all of the data associated with the shadow password entry. The parameter buflen should give the size in bytes of the buffer indicated by buffer. For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. ***setspent()*** may be used in a multithreaded application but resets the enumeration position for all threads.If multiple threads interleave calls to ***getspent_r()***, the threads will enumerate disjoint subsets of the shadow password database. Like its non-reentrant counterpart, ***getspnam_r()*** leaves the enumeration position in an indeterminate state.

RETURN VALUES

Password entries are represented by the struct *spwd* structure defined in <*shadow.h*>:

```
struct spwd{
        char            *sp_namp;       /* login name */
        char            *sp_pwdp;       /* encrypted passwd */
        long            sp_lstchg;      /* date of last change */
        long            sp_min;         /* min days to passwd change */
        long            sp_max;         /* max days to passwd change*/
        long            sp_warn;        /* warning period */
        long            sp_inact;       /* max days inactive */
        long            sp_expire;      /* account expiry date */
        unsigned long   sp_flag;        /* not used */
};
```

See ***shadow()*** for more information on the interpretation of this information. The functions ***getspnam()*** and ***getspnam_r()*** each return a pointer to a struct *spwd* if they successfully locate the requested entry; otherwise they return NULL. The functions ***getspent()***, ***getspent_r()***, ***fgetspent()***, and ***fgetspent()*** each return a pointer to a struct spwd if they successfully enumerate an entry; otherwise they return NULL, indicating the end of the enumeration. The functions ***getspnam()***, ***getspent()***, and ***fgetspent()*** use static storage, so returned data must be copied before a subsequent call to any of these functions if the data is to be saved. When the pointer returned by the re-entrant functions ***getspnam_r()***, ***getspent_r()***, and ***fgetspent_r()*** is non-NULL, it is always equal to the result pointer that was supplied by the caller.

ERRORS

The reentrant functions ***getspnam_r()***, ***getspent_r()***, and ***fgetspent_r()*** will return NULL and set errno to **ERANGE** if the length of the buffer supplied by caller is not large enough to store the result. See ***intro()*** for the proper usage and interpretation of errno in multithreaded applications.

FILES

/etc/shadow, /etc/nsswitch.conf, /etc/passwd

SEE ALSO

nispasswd(), ***passwd()***, ***yppasswd()***, ***intro() getlogin()***, ***getpwnam()***, nsswitch.***conf()***, ***passwd()***, ***shadow()***,

WARNINGS

The reentrant interfaces ***getspnam_r()***, ***getspent_r()***, and ***fgetspent_r()*** are included in this release on an uncommitted basis only, and are subject to change or removal in future minor releases.

NOTES

Programs that use the interfaces described in this manual page cannot be linked statically since the implementations of these functions employ dynamic loading and linking of shared objects at run time. When compiling multithreaded applications, see ***intro()***, Notes On Multithread Applications, for information about the use of the **_REENTRANT** flag. Use of the enumeration interfaces ***getspent()*** and ***getspent_r()*** is not recommended; enumeration is supported for the shadow file, NIS, and NIS+, but in general is not efficient and may not be supported for all database sources. The semantics of enumeration are discussed further in nsswitch.***conf()***. Access to shadow password information may be restricted in a manner depending on the database source being used. Access to the /etc/shadow file is generally restricted to processes running as the super-user (root). Other database sources may impose stronger or less stringent restrictions. When NIS is used as the database source, the information for theshadowpasswordentriesisobtained from the ``passwd.byname'' map. This map stores only the information for the sp_namp and sp_pwdp fields of the struct spwd structure. Shadow password entries obtained from NIS will contain the value -1 in the remainder of the fields. When NIS+ is used as the database source, and the caller lacks the permission needed to retrieve the encrypted password from the NIS+ ``*passwd.org_dir*'' table, the NIS+ service returns the string ``*NP*'' instead of the actual encrypted password string. The functions described on this page will then return the string ``*NP*'' to the caller as the value of the member *sp_pwdp* in the returned shadow password structure.

gettimeofday, settimeofday

NAME

gettimeofday, settimeofday - get or set the date and time

SYNOPSIS

#include <sys/time.h>

int gettimeofday(struct timeval *tp, void *);

int settimeofday(struct timeval *tp, void *);

DESCRIPTION

The ***gettimeofday***() function gets and the ***settimeofday***() function sets the system's notion of the current time. The current time is expressed in elapsed seconds and microseconds since 00:00 Universal Coordinated Time, January 1, 1970. The resolution of the system clock is hardware dependent; the time may be updated continuously or in clock ticks.

The *tp* argument points to a *timeval* structure, which includes the following members:

long tv_sec; / seconds since Jan. 1, 1970 */*

long tv_usec; / and microseconds */*

If *tp* is a null pointer, the current time information is not returned or set.

The **TZ** environment variable holds time zone information.

The second argument to ***gettimeofday***() and ***settimeofday***() should be a pointer to NULL.

Only the super-user may set the time of day.

RETURN VALUES

Upon successful completion, 0 is returned. Otherwise, -1 is returned and errno is set to indicate the error.

ERRORS

The ***gettimeofday***() function will fail if:

EINVAL The structure pointed to by *tp* specifies an invalid time.

EPERM A user other than the privileged user attempted to set the time or time zone.

Additionally, the ***gettimeofday***() function will fail for 32-bit interfaces if:

EOVERFLOW The system time has progressed beyond 2038, thus the size of the *tv_sec* member of the *timeval* structure pointed to by *tp* is insufficient to hold the current time in seconds.

USAGE

If the *tv_usec* member of *tp* is > 500000, ***settimeofday***() rounds the seconds upward. If the time needs to be set with better than one second accuracy, call ***settimeofday***() for the seconds and then ***adjtime***() for finer accuracy.

SEE ALSO

adjtime, ***asctime***, **TIMEZONE**

getutxent, getutxid, getutxline, pututxline, setutxent endutxent, utmpxname, getutmp, getutmpx updwtmp, updwtmpx

NAME

getutxent, getutxid, getutxline, pututxline, setutxent, endutxent, utmpxname, getutmp, getutmpx, updwtmp, updwtmpx - access utmpx file entry

SYNOPSIS

#include <utmpx.h>

struct utmpx	****getutxent(void);***
struct utmpx	****getutxid(const struct utmpx *id);***
struct utmpx	****getutxline(const struct utmpx *line);***
struct utmpx	****pututxline(const struct utmpx *utmpx);***
void	***setutxent(void);***
void	***endutxent(void);***
int	***utmpxname(const char *file);***
void	***getutmp(struct utmpx *utmpx, struct utmp *utmp);***
void	***getutmpx(struct utmp *utmp, struct utmpx *utmpx);***
void	***updwtmp(char *wfile, struct utmp *utmp);***
void	***updwtmpx(char *wfilex, struct utmpx *utmpx);***

DESCRIPTION

getutxent(), ***getutxid()***, and ***getutxline()*** each return a pointer to a *utmpx* structure with the following members:

char	*ut_user[32]; /* user login name */*
char	*ut_id[4]; /* /etc/inittab id */*
/ (usually line #) */*	
char	*ut_line[32]; /* device name (console, lnxx) */*
pid_t	*ut_pid; /* process id */*
short	*ut_type;/* type of entry */*
struct	*exit_status ut_exit;/* exit status of a process */*
/ marked as DEAD_PROCESS */*	
struct timeval	*ut_tv; /* time entry was made */*
long	*ut_session; /* session ID, used for windowing */*
long	*pad[5]; /* reserved for future use */*
short	*ut_syslen;/* significant length of ut_host */*
/ including terminating null */*	
char	*ut_host[257];/* host name, if remote */*

The structure exit status includes the following members:

short	*e_termination;*	*/* termination status */*
short	*e_exit;*	*/* exit status */*

getutxent()

Reads in the next entry from a *utmpx*-like file. If the file is not already open, it opens it. If it reaches the end of the file, it fails.

getutxid()

Searches forward from the current point in the utmpx file until it finds an entry with a *ut_type* matching *id->ut_type* if the type specified is **RUN_LVL**, **BOOT_TIME**, **OLD_TIME**, or **NEW_TIME**.If the type specified in id is **INIT_PROCESS**, **LOGIN_PROCESS**,**USER_PROCESS**, or **DEAD_PROCESS**, then ***getutxid()*** will return a pointer to the first entry whose type is one of these four and whose *ut_id* field matches id->ut_id.If the end of file is reached without a match, it fails.

getutxline()

Searches forward from the current point in the utmpx file until it finds an entry of the type **LOGIN_PROCESS** or **USER_PROCESS** which also has a *ut_line* string matching the *line->ut_line* string. If the end of file is reached without a match, it fails.

pututxline()

Writes out the supplied utmpx structure into the utmpx file. It uses ***getutxid()*** to search forward for the proper place if it finds that it is not already at the proper place. It is expected that normally the user of ***pututxline()*** will have searched for the proper entry using one of the ***getutx()*** routines.If so, ***pututxline()*** will not search. If ***pututxline()*** does not find a matching slot for the new entry, it will add a new entry to the end of the file. It returns a pointer to the utmpx structure. When called by a non-root user, ***pututxline()*** invokes a ***setuid()*** root program to verify and write the entry, since */etc/utmpx* is normally writable only by root.In this event, the *ut_name* field must correspond to the actual user name associated with the process; the *ut_type* field must be either **USER_PROCESS** or **DEAD_PROCESS**; and the *ut_line* field must be a device special file and be writable by the user.

setutxent()

Resets the input stream to the beginning of the file.This should be done before each search for a new entry if it is desired that the entire file be examined.

endutxent()

Closes the currently open file.

utmpxname()

Allows the user to change the name of the file examined, from */var/adm/utmpx* to any other file. It is most often expected that this other file will be */var/adm/wtmpx*.If the file does not exist, this will not be apparent until the first attempt to reference the file is made.***utmpxname()*** does not open the file. It just closes the old file if it is currently open and saves the new file name. The new file name must end with the ``x'' character to allow the name of the corresponding utmp file to be easily obtainable; otherwise, an error code of 1 is returned.

getutmp()

Copies the information stored in the fields of the utmpx structure to the corresponding fields of the utmp structure. If the information in any field of utmpx does not fit in the corresponding utmp field, the data is truncated. (See ***getutent()*** for utmp structure)

getutmpx()

Copies the information stored in the fields of the utmp structure to the corresponding fields of the utmpx structure. (See ***getutent()*** for utmp structure)

updwtmp()

Checks the existence of wfile and its parallel file, whose name is obtained by appending an ``x" to wfile. If only one of them exists, the second one is created and initialized to reflect the state of the existing file. utmp is written to wfile and the corresponding utmpx structure is written to the parallel file.

updwtmpx()

Checks the existence of wfilex and its parallel file, whose name is obtained by truncating the final ``x" from wfilex. If only one of them exists, the second one is created and initialized to reflect the state of the existing file. utmpx is written to wfilex, and the corresponding utmp structure is written to the parallel file.

RETURN VALUES

A null pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write.

FILES

/var/adm/utmp	contains current user access and administrative information (old format)
/var/adm/utmpx	contains current user access and administration information (new format)
/var/adm/wtmp	contains a history of user access and administrative information.
/var/adm/wtmpx	contains a history of user access and administrative information.

SEE ALSO

getutent(), ttyslot(), utmp(), utmpx()

NOTES

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. On each call to either ***getutxid()*** or ***getutxline()***, the routine examines the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason, to use ***getutxline()*** to search for multiple occurrences it would be necessary to zero out the static after each success, or ***getutxline()*** would just return the same structure over and over again. There is one exception to the rule about emptying the structure before further reads are done. The implicit read done by ***pututxline()*** (if it finds that it is not already at the correct place in the file) will not hurt the contents of the static structure returned by the ***getutxent()***, ***getutxid()***, or ***getutxline()*** routines, if the user has just modified those contents and passed the pointer back to ***pututxline()***. These routines use buffered standard I/O for input, but ***pututxline()*** uses an unbuffered write to avoid race conditions between processes trying to modify the utmpx and *wtmpx* files.

ffs

NAME

ffs - find first set bit

SYNOPSIS

#include <strings.h>

int ffs(const int i);

DESCRIPTION

The ***ffs()*** function finds the first bit set (beginning with the least significant bit) and returns the index of that bit. Bits are numbered starting at one (the least significant bit).

RETURN VALUES

The ***ffs()*** function returns the index of the first bit set. If i is 0, then ***ffs()*** returns 0.

isnan, isnand, isnanf, finite, fpclass, unordered

NAME

isnan, isnand, isnanf, finite, fpclass, unordered - determine type of floating-point number

SYNOPSIS

#include <ieeefp.h>

int	***isnand(double dsrc);***
int	***isnanf(float fsrc);***
int	***finite(double dsrc);***
fpclass_t	***fpclass(double dsrc);***
int	***unordered(double dsrc1, double dsrc2);***

#include <math.h>

int	***isnan(double dsrc);***

DESCRIPTION

The functionality of ***isnan()*** is identical to that of ***isnand()***. ***isnanf()*** is implemented as a macro included in the <*ieeefp.h*> header. ***fpclass()*** returns the class the *dsrc* belongs to.The 10 possible classes are as follows:

FP_SNAN	signaling NaN
FP_QNAN	quiet NaN
FP_NINF	negative infinity
FP_PINF	positive infinity
FP_NDENORM	negative de-normalized non-zero
FP_PDENORM	positive de-normalized non-zero
FP_NZERO	negative zero
FP_PZERO	positive zero
FP_NNORM	negative normalized non-zero
FP_PNORM	positive normalized non-zero

None of these routines generate any exception, even for signaling NaNs.

RETURN VALUES

isnan(), ***isnand()***, and ***isnanf()*** return true ***()*** if the argument dsrc or *fsrc* is a NaN; otherwise they return false (0). ***finite()*** returns true ***()*** if the argument dsrc is neither infinity nor NaN; otherwise it returns false (0). isnan ***isnan()unordered()*** returns true ***()*** if one of its two arguments is unordered with respect to the other argument. This is equivalent to reporting whether either argument is NaN. If neither of the arguments is NaN, false (0) is returned.

SEE ALSO

fpgetround()

fpgetround, fpsetround, fpgetmask fpsetmask, fpgetsticky, fpsetsticky

NAME

fpgetround, fpsetround, fpgetmask, fpsetmask, fpgetsticky, fpsetsticky - IEEE floating-point environment control

SYNOPSIS

#include <ieeefp.h>

fp_rnd	***fpgetround(void);***
fp_rnd	***fpsetround(fp_rnd rnd_dir);***
fp_except	***fpgetmask(void);***
fp_except	***fpsetmask(fp_except mask);***
fp_except	***fpgetsticky(void);***
fp_except	***fpsetsticky(fp_except sticky);***

DESCRIPTION

There are five floating-point exceptions: divide-by-zero, overflow, underflow, imprecise (inexact) result, and invalid operation. When a floating-point exception occurs, the corresponding sticky bit is set, and if the mask bit is enabled, the trap takes place. These routines let the user change the behavior on occurrence of any of these exceptions, as well as change the rounding mode for floating-point operations.

The following floating-point exception masks are OR-ed together to form mask.

FP_X_INV	/* invalid operation exception */
FP_X_OFL	/* overflow exception */
FP_X_UFL	/* underflow exception */
FP_X_DZ	/* divide-by-zero exception */
FP_X_IMP	/* imprecise (loss of precision) */

The following floating-point rounding modes are passed to ***fpsetround()*** and returned by ***fpgetround()***.

FP_RN	/* round to nearest representative number */
FP_RP	/* round to plus infinity */
FP_RM	/* round to minus infinity */
FP_RZ	/* round to zero (truncate) */

The default environment is rounding mode set to nearest (**FP_RN**) and all traps disabled. Individual bits may be examined using the constants defined in <*ieeefp.h*>.

RETURN VALUES

fpgetround() returns the current rounding mode.

fpsetround() sets the rounding mode and returns the previous rounding mode.

fpgetmask() returns the current exception masks.

fpsetmask() sets the exception masks and returns the previous setting.

fpgetsticky() returns the current exception sticky flags.

fpsetsticky() sets (clears) the exception sticky flags and returns the previous setting.

SEE ALSO

isnan()

NOTES

fpsetsticky() modifies all sticky flags. ***fpsetmask()*** changes all mask bits. ***fpsetmask()*** clears the sticky bit corresponding to any exception being enabled. C requires truncation (round to zero) for floating point to integral conversions.The current rounding mode has no effect on these conversions. One must clear the sticky bit to recover from the trap and to proceed.If the sticky bit is not cleared before the next trap occurs, a wrong exception type may be signaled.

truncate, ftruncate

NAME

truncate, ftruncate - set a file to a specified length

SYNOPSIS

#include <unistd.h>

int truncate(const char *path, off_t length);

int ftruncate(int fildes, off_t length);

DESCRIPTION

The ***truncate()*** function causes the regular file named by path to have a size of length bytes. The ***ftruncate()*** function causes the regular file referenced by fildes to have a size of length bytes. The effect of ***ftruncate()*** and ***truncate()*** on other types of files is unspecified.If the file previously was larger than length, the extra data is lost. If it was previously shorter than length, bytes between the old and new lengths are read as zeroes. With ***ftruncate()***, the file must be open for writing; for ***truncate()***, the process must have write permission for the file. If the request would cause the file size to exceed the soft file size limit for the process, the request will fail and the implementation will generate the **SIGXFSZ** signal for the process. These functions do not modify the file offset for any open file descriptions associated with the file. On successful completion, if the file size is changed, these functions will mark for update the *st_ctime* and *st_mtime* fields of the file, and if the file is a regular file, the **S_ISUID** and **S_ISGID** bits of the file mode may be cleared.

RETURN VALUES

Upon successful completion, ***ftruncate()*** and ***truncate()*** return 0.Otherwise, -1 is returned and errno is set to indicate the error.

ERRORS

The ***ftruncate()*** and ***truncate()*** functions will fail if:

EINTRA	signal was caught during execution.
EINVAL	The length argument was less than 0.
EFBIG or **EINVAL**	The length argument was greater than the maximum file size.truncate ***truncate()***
EIO	An I/O error occurred while reading from or writing to a file system.

The ***truncate()*** function will fail if:

EACCES	A component of the path prefix denies search permission, or write permission is denied on the file.
EFAULT	The path argument points outside the process' allocated address space.
EINVAL	The path argument is not an ordinary file.
EISDIR	The named file is a directory.
ELOOP	Too many symbolic links were encountered in resolving path.
EMFILE	The maximum number of file descriptors available to the process has been reached.

EMULTIHOP Components of path require hopping to multiple remote machines and file system type does not allow it.

ENAMETOOLONG The length of the specified pathname exceeds **PATH_MAX** bytes, or the length of a component of the pathname exceeds **NAME_MAX** bytes.

ENOENT A component of path does not name an existing file or path is an empty string.

ENFILE Additional space could not be allocated for the system file table.

ENOTDIRA component of the path prefix of path is not a directory.

ENOLINK The path argument points to a remote machine and the link to that machine is no longer active.

EROFS The named file resides on a read-only file system.

The ***ftruncate()*** function will fail if:

EAGAIN The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file (see ***chmod()***).

EBADF or **EINVAL** The fildes argument is not a file descriptor open for writing.

EFBIG The file is a regular file and length is greater than the offset maximum established in the open file description associated with fildes.

EINVAL The fildes argument references a file that was opened without write permission.

EINVAL The fildes argument does not correspond to an ordinary file.

ENOLINK The fildes argument points to a remote machine and the link to that machine is no longer active.

The ***truncate()*** function may fail if:

ENAMETOOLONG Pathname resolution of a symbolic link produced an intermediate result whose

SEE ALSO

chmod(), fcntl(), open()

getdents

NAME

getdents - read directory entries and put in a file system independent format

SYNOPSIS

#include <sys/dirent.h>

int getdents(int fildes, struct dirent *buf, size_t nbyte);

DESCRIPTION

The ***getdents()*** function attempts to read *nbyte* bytes from the directory associated with the file descriptor fildes and to format them as file system independent directory entries in the buffer pointed to by buf. Since the file system independent directory entries are of variable length, in most cases the actual number of bytes returned will be strictly less than *nbyte*. See ***dirent()*** to calculate the number of bytes. The file system independent directory entry is specified by the dirent structure.For a description of this see ***dirent()***. On devices capable of seeking, ***getdents()*** starts at a position in the file given by the file pointer associated with fildes. Upon return from ***getdents()***, the file pointer is incremented to point to the next directory entry. This function was developed in order to implement the readdir routine (for a description, see ***opendir()***), and should not be used for other purposes.

RETURN VALUES

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. A value of 0 indicates the end of the directory has been reached.If the function failed, -1 is returned and errno is set to indicate the error.

ERRORS

The ***getdents()*** function will fail if one or more of the following are true:

EBADF	fildes is not a valid file descriptor open for reading.
EFAULT	buf points to an illegal address.
EINVAL	*nbyte* is not large enough for one directory entry.
EIO	An I/O error occurred while accessing the ***getdents()*** System Calls ***getdents()*** file system.
ENOENT	The current file pointer for the directory is not located at a valid entry.
ENOLINK	fildes points to a remote machine and the link to that machine is no longer active.
ENOTDIR	fildes is not a directory.
EOVERFLOW	The value of the dirent structure member *d_ino* or *d_off* cannot be represented in an *ino_t* or *off_t*.

SEE ALSO

opendir(), dirent()

getmntent, getmntany, hasmntopt, putmntent

NAME

getmntent, getmntany, hasmntopt, putmntent - get mnttab file information

SYNOPSIS

#include <stdio.h>

#include <sys/mnttab.h>

int getmntent(FILE *fp, struct mnttab *mp);

int getmntany(FILE *fp, struct mnttab *mp, struct mnttab *mpref);

char *hasmntopt(struct mnttab *mnt, char *opt);

int putmntent(FILE *iop, struct mnttab *mp);

DESCRIPTION

getmntent() and ***getmntany()*** each fill in the structure pointed to by *mp* with the broken-out fields of a line in the */etc/mnttab* file. Each line in the file contains a mnttab structure, which is declared in the <*sys/mnttab.h*> header. The structure contains the following members:

*char *mnt_special;*

*char *mnt_mountp;*

*char *mnt_fstype;*

*char *mnt_mntopts;*

*char *mnt_time;*

The fields have meanings described in ***mnttab()***.

getmntent() returns a pointer to the next *mnttab* structure in the file; so successive calls can be used to search the entire file. ***getmntany()*** searches the file referenced by *fp* until a match is found between a line in the file and *mpref*. *mpref* matches the line if all non-null entries in *mpref* match the corresponding fields in the file. Note that these routines do not open, close, or rewind the file.

hasmntopt() scans the *mnt_mntopts* field of the *mnttab* structure *mnt* for a sub-string that matches *opt*. It returns the address of the sub-string if a match is found, otherwise it returns 0.

The ***putmntent()*** macro formats the contents of the mnttab structure according to the layout required for the /etc/mnttab file and writes the entry to the file.Note: the file should be opened in append mode (***fopen()*** with an "a" mode) so that the entry is appended to the file.

RETURN VALUES

If the next entry is successfully read by ***getmntent()*** or a match is found with ***getmntany()***, 0 is returned. If an EOF is encountered on reading, these functions return -1. If an error is encountered, a value greater than 0 is returned. The possible error values are:

MNT_TOOLONG A line in the file exceeded the internal buffer size of **MNT_LINE_MAX**.

MNT_TOOMANY A line in the file contains too many fields.

MNT_TOOFEW A line in the file contains too few fields.

On success, ***putmntent()*** returns the number of bytes printed to the specified file and on failure returns **EOF**.

FILES

/etc/mnttab

SEE ALSO

mnttab()

NOTES

The members of the mnttab structure point to information contained in a static area, so it must be copied if it is to be saved.

getpw

NAME

getpw - get passwd entry from UID

SYNOPSIS

#include <stdlib.h>

int getpw(uid_t uid, char *buf);

DESCRIPTION

getpw() searches the user data base for a user id number that equals uid, copies the line of the password file in which uid was found into the array pointed to by buf, and returns 0. ***getpw()*** returns non-zero if uid cannot be found. This routine is included only for compatibility with prior systems and should not be used; see ***getpwnam()*** for routines to use instead.

RETURN VALUES

getpw() returns non-zero on error.

FILES

/etc/passwd

SEE ALSO

getpwnam(), passwd()

NOTES

If the */etc/passwd* and the */etc/group* files have the ``+'' for the NIS entry, then ***getpwent()*** and ***getgwent()*** will not return NULL when the end of file is reached.

getvfsent, getvfsfile, getvfsspec, getvfsany

NAME

getvfsent, getvfsfile, getvfsspec, getvfsany - get vfstab file entry

SYNOPSIS

#include <stdio.h>

#include <sys/vfstab.h>

int getvfsent(FILE *fp, struct vfstab *vp);

int getvfsfile(FILE *fp, struct vfstab *vp, char *file);

int getvfsspec(FILE *, struct vfstab *vp, char *spec);

int getvfsany(FILE *, struct vfstab *vp, vfstab *vref);

DESCRIPTION

getvfsent(), ***getvfsfile()***, ***getvfsspec()***, and ***getvfsany()*** each fill in the structure pointed to by vp with the broken-out fields of a line in the /etc/vfstab file.Each line in the file contains a vfstab structure, declared in the <*sys/vfstab.h*> header:

*char *vfs_special;*

*char *vfs_fsckdev;*

*char *vfs_mountp;*

*char *vfs_fstype;*

*char *vfs_fsckpass;*

*char *vfs_automnt;*

*char *vfs_mntopts;*

The fields have meanings described in ***vfstab()***. ***getvfsent()*** returns a pointer to the next vfstab structure in the file; so successive calls can be used to search the entire file. ***getvfsfile()*** searches the file referenced by fp until a mount point matching file is found and fills vp with the fields from the line in the file.***getvfsspec()*** searches the file referenced by fp until a special device matching spec is found and fills vp with the fields from the line in the file.spec will try to match on device type (block or character special) and major and minor device numbers.If it cannot match in this manner, then it compares the strings. ***getvfsany()*** searches the file referenced by fp until a match is found between a line in the file and vref. vref matches the line if all non-null entries in vref match the corresponding fields in the file. Note that these routines do not open, close, or rewind the file.

RETURN VALUES

If the next entry is successfully read by ***getvfsent()*** or a match is found with ***getvfsfile()***, ***getvfsspec()***, or ***getvfsany()***, 0 is returned. If an end-of-file is encountered on reading, these functions return -1. If an error is encountered, a value greater than 0 is returned. The possible error values are:

VFS_TOOLONG A line in the file exceeded the internal buffer size of **VFS_LINE_MAX**.

VFS_TOOMANY A line in the file contains too many fields.

VFS_TOOFEW A line in the file contains too few fields.

NOTES

The members of the vfstab structure point to information contained in a static area, so it must be copied if it is to be saved.

FILES

/etc/vfstab

SEE ALSO

vfstab()

iconv

NAME

iconv - code conversion function

SYNOPSIS

#include <iconv.h>

size_t iconv(iconv_t cd, const char **inbuf, size_t *inbytesleft, char **outbuf, size_t *outbytesleft);

DESCRIPTION

The ***iconv()*** function converts the sequence of characters from one *codeset*, in the array specified by *inbuf*, into a sequence of corresponding characters in another *codeset*, in the array specified by outbuf.The *codesets* are those specified in the ***iconv_open()*** call that returned the conversion descriptor, *cd*. The *inbuf* argument points to a variable that points to the first character in the input buffer and *inbytesleft* indicates the number of bytes to the end of the buffer to be converted. The *outbuf* argument points to a variable that points to the first available byte in the output buffer and *outbytesleft* indicates the number of the available bytes to the end of the buffer.

For state-dependent encoding, the conversion descriptor *cd* is placed into its initial shift state by a call for which *inbuf* is a null pointer, or for which *inbuf* points to a null pointer.When ***iconv()*** is called in this way, and if *outbuf* is not a null pointer or a pointer to a null pointer, and *outbytesleft* points to a positive value, ***iconv()*** will place, into the output buffer, the byte sequence to change the output buffer to its initial shift state. If the output buffer is not large enough to hold the entire reset sequence, ***iconv()*** will fail and set errno to **E2BIG**. Subsequent calls with *inbuf* as other than a null pointer or a pointer to a null pointer cause the conversion to take place from the current state of the conversion descriptor. If a sequence of input bytes does not form a valid character in the specified codeset, conversion stops after the previous successfully converted character. If the input buffer ends with an incomplete character or shift sequence, conversion stops after the previous successfully converted bytes. If the output buffer is not large enough to hold the entire converted input, conversion stops just prior to the input bytes that would cause the output buffer to overflow. The variable pointed to by *inbuf* is updated to point to the byte following the last byte successfully used in the conversion. The value pointed to by *inbytesleft* is decremented to reflect the number of bytes still not converted in the input buffer. The variable pointed to by *outbuf* is updated to point to the byte following the last byte of converted output data. The value pointed to by *outbytesleft* is decremented to reflect the number of bytes still available in the output buffer. For state-dependent encodings, the conversion descriptor is updated to reflect the shift state in effect at the end of the last successfully converted byte sequence. If ***iconv()*** encounters a character in the input buffer that is legal, but for which an identical character does not exist in the target code set, ***iconv()*** performs an implementation-defined conversion on this character.

RETURN VALUES

The ***iconv()*** function updates the variables pointed to by the arguments to reflect the extent of the conversion and returns the number of non-identical conversions performed. If the entire string in the input buffer is converted, the value pointed to by *inbytesleft* will be 0.If the input conversion is stopped due to any conditions mentioned above, the value pointed to by *inbytesleft* will be non-zero and errno is set to indicate the condition. If an error occurs ***iconv()*** returns (*size_t*) -1 and sets errno to indicate the error.

ERRORS

The ***iconv()*** function will fail if:

EILSEQ Input conversion stopped due to an input byte that does not belong to the input codeset.

E2BIG Input conversion stopped due to lack of space in the output buffer.

EINVAL Input conversion stopped due to an incomplete character or shift sequence at the end of the input buffer.

The ***iconv()*** function may fail if:

EBADF The *cd* argument is not a valid open conversion descriptor.

FILES

/usr/lib/iconv/.so* conversion modules

SEE ALSO

iconv(), iconv_close(), iconv_open()

iconv_close

NAME

iconv_close - code conversion de-allocation function

SYNOPSIS

#include <iconv.h>

int iconv_close(iconv_t cd);

DESCRIPTION

The ***iconv_close()*** function de-allocates the conversion descriptor *cd* and all other associated resources allocated by the ***iconv_open()*** function.

If a file descriptor is used to implement the type *iconv_t*, that file descriptor will be closed.

RETURN VALUES

Upon successful completion, ***iconv_close()*** returns 0; otherwise, it returns -1 and sets errno to indicate the error.

ERRORS

The ***iconv_close()*** function may fail if:

EBADF The conversion descriptor is invalid.

SEE ALSO

iconv(), iconv_open()

iconv_open

NAME

iconv_open - code conversion allocation function

SYNOPSIS

#include <iconv.h>

iconv_t iconv_open(const char *tocode, const char *fromcode);

DESCRIPTION

The ***iconv_open()*** function returns a conversion descriptor that describes a conversion from the codeset specified by the string pointed to by the *fromcode* argument to the codeset specified by the string pointed to by the *tocode* argument. For state-dependent encodings, the conversion descriptor will be in a codeset-dependent initial shift state, ready for immediate use with the ***iconv()*** function. Settings of *fromcode* and *tocode* and their permitted combinations are implementation-dependent.

A conversion descriptor remains valid in a process until that process closes it.

RETURN VALUES

Upon successful completion ***iconv_open()*** returns a conversion descriptor for use on subsequent calls to ***iconv()***. Otherwise, ***iconv_open()*** returns (*iconv_t*) -1 and sets errno to indicate the error.

ERRORS

The ***iconv_open*** function may fail if:

EMFILE {**OPEN_MAX**} files descriptors are currently open in the calling process.

ENFILE Too many files are currently open in the system.

ENOMEM Insufficient storage space is available.

EINVAL The conversion specified by *fromcode* and *tocode* is not supported.

SEE ALSO

iconv(), ***iconv_close()***, ***malloc()***

NOTES

iconv_open() uses ***malloc()*** to allocate space for internal buffer areas.***iconv_open()*** may fail if there is insufficient storage space to accommodate these buffers. Portable applications must assume that conversion descriptors are not valid after a call to one of the exec functions.

insque, remque

NAME

insque, remque - insert/remove element from a queue

SYNOPSIS

include <search.h>

void insque(struct qelem *elem, struct qelem *pred);

void remque(struct qelem *elem);

DESCRIPTION

insque() and ***remque()*** manipulate queues built from doubly linked lists. Each element in the queue must be in the following form:

```
struct qelem {
        struct qelem    *q_forw;
        struct qelem    *q_back;
        char            q_data[];
};
```

insque() inserts *elem* in a queue immediately after *pred*. ***remque()*** removes an entry *elem* from a queue.

madvise

NAME

madvise - provide advice to VM system

SYNOPSIS

#include <sys/types.h>

#include <sys/mman.h>

int madvise(caddr_t addr, size_t len, int advice);

DESCRIPTION

madvise() advises the kernel that a region of user mapped memory in the range [addr, addr + len) will be accessed following a type of pattern. The kernel uses this information to optimize the procedure for manipulating and maintaining the resources associated with the specified mapping range.

Values for advice are defined in <*sys/mman.h*> as:

#define ***MADV_NORMAL***	*0x0*	*/* No further special treatment */*
#define ***MADV_RANDOM***	*0x1*	*/* Expect random page references */*
#define ***MADV_SEQUENTIAL***	*0x2*	*/* Expect sequential page references */*
#define ***MADV_WILLNEED***	*0x3*	*/* Will need these pages */*
#define ***MADV_DONTNEED***	*0x4*	*/* Don't need these pages */*

MADV_NORMAL

The default system characteristic where accessing memory within the address range causes the system to read data from the mapped file. The kernel reads all data from files into pages which are retained for a period of time as a "cache." System pages can be a scarce resource, so the kernel steals pages from other mappings when needed. This is a likely occurrence, but adversely affects system performance only if a large amount of memory is accessed.

MADV_RANDOM

Tells the kernel to read in a minimum amount of data from a mapped file on any single particular access. If **MADV_NORMAL** is in effect when an address of a mapped file is accessed, the system tries to read in as much data from the file as reasonable, in anticipation of other accesses within a certain locality.

MADV_SEQUENTIAL

Tells the system that addresses in this range are likely to be accessed only once, so the system will free the resources mapping the address range as quickly as possible.This is used in the ***cat()*** and ***cp()*** utilities.

MADV_WILLNEED

Tells the system that a certain address range definitely needed so the kernel will start reading the specified range into memory. This can benefit programs wanting to minimize the time needed to access memory the first time, as the kernel would need to read in from the file.

MADV_DONTNEED

Tells the kernel that the specified address range is no longer needed, so the system starts to free the resources associated with the address range.

madvise() should be used by programs with specific knowledge of their access patterns over a memory object, such as a mapped file, to increase system performance.

RETURN VALUES

madvise() returns:

0 on success.

-1 on failure and sets errno to indicate the error.

ERRORS

EINVAL *addr* is not a multiple of the page size as returned by ***sysconf()***.

The length of the specified address range is less than or equal to 0, or the advice was invalid.

EIO An I/O error occurred while reading from or writing to the file system.

ENOMEM Addresses in the range [*addr*, *addr* + *len*) are outside the valid range for the address space of a process, or specify one or more pages that are not mapped.

ESTALE Stale NFS file handle.

SEE ALSO

cat(), cp(), mmap(), sysconf()

malloc, calloc, realloc, valloc, alloca free, memalign

NAME

malloc, calloc, free, memalign, realloc, valloc, alloca - memory allocator

SYNOPSIS

#include <stdlib.h>

void *malloc(size_t size);

void *calloc(size_t nelem, size_t elsize);

void free(void *ptr);

void *memalign(size_t alignment, size_t size);

void *realloc(void *ptr, size_t size);

void *valloc(size_t size);

#include <alloca.h>

void *alloca(size_t size);

DESCRIPTION

malloc() and ***free()*** provide a simple general-purpose memory allocation package.***malloc()*** returns a pointer to a block of at least size bytes suitably aligned for any use.

The argument to ***free()*** is a pointer to a block previously allocated by ***malloc()*** , ***calloc()*** or ***realloc()***. After ***free()*** is performed this space is made available for further allocation. If *ptr* is a NULL pointer, no action occurs. Undefined results will occur if the space assigned by ***malloc()*** is overrun or if some random number is handed to ***free()***. ***calloc()*** allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

memalign() allocates size bytes on a specified alignment boundary, and returns a pointer to the allocated block. The value of the returned address is guaranteed to be an even multiple of alignment. Note: the value of alignment must be a power of two, and must be greater than or equal to the size of a word.

realloc() changes the size of the block pointed to by *ptr* to size bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If *ptr* is NULL, ***realloc()*** behaves like ***malloc()*** for the specified size. If size is zero and ptr is not a null pointer, the object pointed to is freed. ***valloc()*** is equivalent to memalign (***sysconf***(**_SC_PAGESIZE**), size).

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

malloc(), ***realloc()***, ***memalign()***, and ***valloc()*** will fail if there is not enough available memory.

alloca() allocates size bytes of space in the stack frame of the caller, and returns a pointer to the allocated block. This temporary space is automatically freed when the caller returns. If the allocated block is beyond the current stack limit, the resulting behavior is undefined.

RETURN VALUES

If there is no available memory, ***malloc()***, ***realloc()***, ***memalign()***, ***valloc()***, and ***calloc()*** return a null pointer. When ***realloc()*** returns NULL, the block pointed to by *ptr* is left intact.If size, *nelem*, or elsize is 0, a unique pointer to the arena is returned.

ERRORS

If ***malloc()***, ***calloc()***, or ***realloc()*** returns unsuccessfully, errno will be set to indicate the following:

ENOMEM *size* bytes of memory exceeds the physical limits of your system, and cannot be allocated.

EAGAIN There is not enough memory available at this point in time to allocate size bytes of memory; but the application could try again later.

SEE ALSO

brk(),getrlimit(),bsdmalloc(),malloc(), mapmalloc(), watchmalloc(), attributes()

WARNINGS

Undefined results will occur if the size requested for a block of memory exceeds the maximum size of a process's heap, which may be obtained with ***getrlimit()***.

alloca() is machine-, compiler-, and most of all, system-dependent. Its use is strongly discouraged.

NOTES

Comparative Features of ***malloc()***, ***bsdmalloc()***, and ***malloc()***:

* The ***bsdmalloc()*** routines afford better performance, but are space-inefficient.

* The ***malloc()*** routines are space-efficient, but have slower performance.

* The standard, fully SCD-compliant malloc routines are a trade-off between performance and space-efficiency.

free() does not set errno.

mincore

NAME

mincore - determine residency of memory pages

SYNOPSIS

#include <sys/types.h>

int mincore(caddr_t addr, size_t len, char *vec);

DESCRIPTION

mincore() determines the residency of the memory pages in the address space covered by mappings in the range [*addr*, *addr* + *len*]. The status is returned as a character-per-page in the character array referenced by *vec (which the system assumes to be large enough to encompass all the pages in the address range). The least significant bit of each character is set to 1 to indicate that the referenced page is in primary memory, 0 if it is not. The settings of other bits in each character are undefined and may contain other information in future implementations.

Because the status of a page can change after ***mincore()*** checks it, but before ***mincore()*** returns the information, returned information might be outdated. Only locked pages are guaranteed to remain in memory; see ***mlock()***.

RETURN VALUES

mincore() returns 0 on success, -1 on failure and sets errno to indicate the error.

ERRORS

mincore() fails if:

EFAULT *vec* points to an illegal address.

EINVAL addr is not a multiple of the page size as returned by ***sysconf()***.

EINVAL The argument *len* has a value less than or equal to 0.

ENOMEM Addresses in the range [*addr*, *addr* + *len*] are invalid for the address space of a process, or specify one or more pages which are not mapped.

SEE ALSO

mmap(), mlock(), sysconf()

modf, modff

NAME

modf, modff - decompose floating-point number

SYNOPSIS

#include <math.h>

double modf(double x, double *iptr);

float modff(float x, float *iptr);

DESCRIPTION

The ***modf()*** and ***modff()*** functions break the argument x into integral and fractional parts, each of which has the same sign as the argument. ***modf()*** stores the integral part as a double in the object pointed to by *iptr*. ***modff()*** stores the integral part as a float in the object pointed to by *iptr*.

RETURN VALUES

Upon successful completion, ***modf()*** and ***modff()*** return the signed fractional part of x.

If x is NaN, NaN is returned and *iptr is set to **NaN**.

If the correct value would cause underflow to 0.0, ***modf()*** returns 0 and errno may be set to **ERANGE**.

ERRORS

The ***modf()*** function may fail if:

ERANGE The result underflows.

USAGE

An application wishing to check for error situations should set errno to 0 before calling ***modf()***. If errno is non-zero on return, or the return value is **NaN**, an error has occurred.

SEE ALSO

frexp(), isnan(), ldexp()

p_online

NAME

p_online - change processor operational status

SYNOPSIS

#include <sys/types.h>

#include <sys/processor.h>

int p_online(processorid_t processorid, int flag);

DESCRIPTION

The processor specified by the first argument is set on-line or off-line or is unchanged, depending on whether the flag argument is **P_ONLINE**, **P_OFFLINE**, or **P_STATUS**. When **P_ONLINE** is specified and the processor is off-line, the processor is brought on-line and allowed to process **LWPs** (lightweight processes) and perform system activities.

When **P_ONLINE** or **P_OFFLINE** is specified and the processor is powered off, it is powered on. In the **P_ONLINE** case, the processor is also brought on-line and allowed to process **LWPs** (lightweight processes) and perform system activities. When **P_OFFLINE** is specified and the processor is on-line, it is taken off-line and not allowed to process **LWPs**. The processor will become as inactive as possible.

When **P_STATUS** is specified, no change occurs, but the current status is returned.

RETURN VALUES

On successful completion, the value returned is the previous state of the processor, **P_ONLINE**, **P_OFFLINE**, or **P_POWEROFF**. Otherwise, a value of -1 is returned and errno is set to indicate the error.

ERRORS

EPERM	The effective user of the calling process is not super user.
EINVAL	A non-existent processor ID was specified or flag was invalid.
EBUSY	The flag was **P_OFFLINE** and the specified processor is the only on-line processor, there are currently **LWPs** bound to the processor, or the processor performs some essential function that cannot be performed by another processor.
EBUSY	The specified processor is powered off and cannot be powered on because some platform specific resource is not available.
ENOTSUP	The specified processor is powered off, and the platform does not support power on of individual processors.

SEE ALSO

psradm(), ***psrinfo()***, ***processor_bind()***, ***processor_info()***, ***pset_create()***, ***sysconf()***

read, readv, pread

NAME

read, ***readv***, ***pread*** - read from file

SYNOPSIS

#include <unistd.h>

ssize_t read(int fildes, void *buf, size_t nbyte);

ssize_t pread(int fildes, void *buf, size_t nbyte,off_t offset);

#include <sys/uio.h>

ssize_t readv(int fildes, const struct iovec *iov, int iovcnt);

DESCRIPTION

The ***read()*** function attempts to read nbyte bytes from the file associated with the open file descriptor, fildes, into the buffer pointed to by buf. If *nbyte* is 0, ***read()*** will return 0 and have no other results. On files that support seeking (for example, a regular file), the ***read()*** starts at a position in the file given by the file offset associated with fildes.The file offset is incremented by the number of bytes actually read. Files that do not support seeking, for example, terminals, always read from the current position. The value of a file offset associated with such a file is undefined. No data transfer will occur past the current end-of-file. If the starting position is at or after the end-of-file, 0 will be returned. If the file refers to a device special file, the result of subsequent ***read()*** requests is implementation-dependent. If the value of *nbyte* is greater than **SSIZE_MAX**, the result is implementation-dependent. When attempting to read from a regular file with mandatory file/record locking set (see ***chmod()***), and there is a write lock owned by another process on the segment of the file to be read:

* If **O_NDELAY** or **O_NONBLOCK** is set, ***read()*** returns -1 and sets errno to **EAGAIN**.

* If **O_NDELAY** and **O_NONBLOCK** are clear, ***read()*** sleeps until the blocking record lock is removed.

When attempting to read from an empty pipe (or **FIFO**):

* If no process has the pipe open for writing, ***read()*** returns 0 to indicate end-of-file.

* If some process has the pipe open for writing and **O_NDELAY** is set, ***read()*** returns 0.

* If some process has the pipe open for writing and **O_NONBLOCK** is set, ***read()*** returns -1 and sets errno to **EAGAIN**.

* If **O_NDELAY** and **O_NONBLOCK** are clear, ***read()*** blocks until data is written to the pipe or the pipe is closed by all processes that had opened the pipe for writing.

When attempting to read a file associated with a terminal that has no data currently available:

* If **O_NDELAY** is set, ***read()*** returns 0.

* If **O_NONBLOCK** is set, ***read()*** returns -1 and sets errno to **EAGAIN**.

* If **O_NDELAY** and **O_NONBLOCK** are clear, ***read()*** blocks until data become available.

When attempting to read a file associated with a socket or a stream that is not a pipe, a **FIFO**, or a terminal, and the file has no data currently available:

* If **O_NDELAY** or **O_NONBLOCK** is set, ***read()*** returns -1 and sets errno to **EAGAIN**.

* If **O_NDELAY** and **O_NONBLOCK** are clear, ***read()*** blocks until data becomes available.

The ***read()*** function reads data previously written to a file. If any portion of a regular file prior to the end-of-file has not been written, ***read()*** returns bytes with value 0. For example, ***lseek()*** allows the file offset to be set beyond the end of existing data in the file.If data is later written at this point, subsequent reads in the gap between the previous end of data and the newly written data will return bytes with value 0 until data is written into the gap. For regular files, no data transfer will occur past the offset maximum established in the open file description associated with fildes. Upon successful completion, where nbyte is greater than 0, ***read()*** will mark for update the *st_atime* field of the file, and return the number of bytes read. This number will never be greater than nbyte. The value returned may be less than nbyte if the number of bytes left in the file is less than nbyte, if the ***read()*** request was interrupted by a signal, or if the file is a pipe or **FIFO** or special file and has fewer than nbyte bytes immediately available for reading. For example, a ***read()*** from a file associated with a terminal may return one typed line of data.

If a ***read()*** is interrupted by a signal before it reads any data, it will return -1 with errno set to **EINTR**. If a ***read()*** is interrupted by a signal after it has successfully read some data, it will return the number of bytes read. A ***read()*** from a STREAMS file can read data in three different modes: byte-stream mode, message-nondiscard mode, and message-discard mode. The default is byte-stream mode. This can be changed using the **I_SRDOPT** ***ioctl()*** request, and can be tested with the **I_GRDOPT** ***ioctl()***.In byte-stream mode, ***read()*** retrieves data from the **STREAM** until as many bytes as were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries. In STREAMS message-nondiscard mode, ***read()*** retrieves data until as many bytes as were requested are transferred, or until a message boundary is reached.If ***read()*** does not retrieve all the data in a message, the remaining data is left on the **STREAM**, and can be retrieved by the next ***read()*** call.Message-discard mode also retrieves data until as many bytes as were requested are transferred, or a message boundary is reached.However, unread data remaining in a message after the ***read()*** returns is discarded, and is not available for a subsequent ***read()***, ***readv()*** or ***getmsg()*** call. How ***read()*** handles zero-byte **STREAMS** messages is determined by the current read mode setting. In byte-stream mode, ***read()*** accepts data until it has read nbyte bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The ***read()*** function then returns the number of bytes read, and places the zero-byte message back on the **STREAM** to be retrieved by the next ***read()***, ***readv()*** or ***getmsg()***.In message-nondiscard mode or message-discard mode, a zero-byte message returns 0 and the message is removed from the **STREAM**. When a zero-byte message is read as the first message on a **STREAM**, the message is removed from the **STREAM** and 0 is returned, regardless of the read mode.

A ***read()*** from a STREAMS file returns the data in the message at the front of the **STREAM** head read queue, regardless of the priority band of the message. By default, STREAMs are in control-normal mode, in which a ***read()*** from a STREAMS file can only process messages that contain a data part but do not contain a control part.The ***read()*** fails if a message containing a control part is encountered at the **STREAM** head. This default action can be changed by placing the **STREAM** in either control-data mode or control-discard mode with the **I_SRDOPT** ***ioctl()*** command.In control-data mode, ***read()*** converts any control part to data and passes it to the application before passing any data part originally present in the same message. In control-discard mode, ***read()*** discards message control parts but returns to the process any data part in the message. In addition, ***read()*** and ***readv()*** will fail if the **STREAM** head had processed an asynchronous error before the call. In this case, the value of errno does not reflect the result of ***read()*** or ***readv()*** but reflects the prior error. If a hang-up occurs on the **STREAM** being read, ***read()*** continues to operate normally until the **STREAM** head read queue is empty. Thereafter, it returns 0.

readv()

The ***readv()*** function is equivalent to ***read()***, but places the input data into the iovcnt buffers specified by the members of the iov array: *iov0*, *iov1*, ..., *iov*[*iovcnt*-1].The *iovcnt* argument is valid if greater than 0 and less than or equal to **IOV_MAX**.

The *iovec* structure contains the following members:

caddr_t	*iov_base;*
int	*iov_len;*

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. The ***readv()*** function always fills an area completely before proceeding to the next. Upon successful completion, ***readv()*** marks for update the *st_atime* field of the file.

pread()

The ***pread()*** function performs the same action as ***read()***, except that it reads from a given position in the file without changing the file pointer. The first three arguments to ***pread()*** are the same as ***read()*** with the addition of a fourth argument offset for the desired position inside the file. ***pread()*** will read up to the maximum offset value that can be represented in an *off_t* for regular files.An attempt to perform a ***pread()*** on a file that is incapable of seeking results in an error.

RETURN VALUES

Upon successful completion, ***read()*** and ***readv()*** return a non-negative integer indicating the number of bytes actually read. Otherwise, the functions return -1 and set errno to indicate the error.

ERRORS

These functions will fail if:

EAGAIN Mandatory file/record locking was set, **O_NDELAY** or **O_NONBLOCK** was set, and there was a blocking record lock.

EAGAIN Total amount of system memory available when reading using raw I/O is temporarily insufficient.

EAGAIN No data is waiting to be read on a file associated with a *tty* device and **O_NONBLOCK** was set.

EAGAIN No message is waiting to be read on a stream and **O_NDELAY** or **O_NONBLOCK** was set.

EBADF fildes is not a valid file descriptor open for reading.

EBADMSG Message waiting to be read on a stream is not a data message.

EDEADLK The read was going to go to sleep and cause a deadlock to occur.

EFAULT buf points to an illegal address.

EINTRA signal was caught during the read operation and no data was transferred.

EINVAL Attempted to read from a stream linked to a multiplexor.

EIO A physical I/O error has occurred, or the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group of the process is orphaned.

EISDIR fildes refers to a directory on a file system type that does not support read operations on directories.

ENOLCK The system record lock table was full, so the ***read()*** or ***readv()*** could not go to sleep until the blocking record lock was removed.

ENOLINK fildes is on a remote machine and the link to that machine is no longer active.

ENXIO The device associated with *fildes* is a block special or character special file and the value of the file pointer is out of range.

The ***read()*** and ***readv()*** functions will fail if:

EOVERFLOW The file is a regular file, nbyte is greater than 0, the starting position is before the end-of-file, and the starting position is greater than or equal to the offset maximum established in the open file description associated with fildes.

In addition, ***readv()*** may return one of the following errors:

EFAULT iov points outside the allocated address space.

EINVAL iovcnt was less than or equal to 0, or greater than or equal to {**IOV_MAX**}. (See ***intro()*** for a definition of {**IOV_MAX**}).

~~**EINVAL** The sum of the *iov_len* values in the iov array overflowed an int.~~

EINVAL The sum of the *iov_len* values in the iov array overflowed an ssize_t.

In addition, ***pread()*** fails and the file pointer remains unchanged if the following is true:

ESPIPE fildes is associated with a pipe or **FIFO**.

SEE ALSO

Intro(), chmod(), creat(), dup(), fcntl(), getmsg(), ioctl(), lseek(), open(), pipe(), streamio, termio

processor_bind

NAME

processor_bind - bind **LWPs** to a processor

SYNOPSIS

#include <sys/types.h>

#include <sys/processor.h>

#include <sys/procset.h>

int processor_bind(idtype_t idtype, id_t id, processorid_t processorid, processorid_t *obind);

DESCRIPTION

The **LWP** (lightweight process) or set of **LWPs** specified by *idtype* and *id* are bound to the processor specified by processorid.Additionally, if *obind* is not NULL, the *processorid_t* variable pointed to by *obind* will be set to the previous binding of one of the specified **LWPs**, or to **PBIND_NONE** if the selected **LWP** was not bound.

If *idtype* is **P_PID**, the binding effects all **LWPs** of the process with process ID (PID) id.

If *idtype* is **P_LWPID**, the binding effects the **LWP** of the current process with **LWP** ID id.

If *id* is **P_MYID**, the specified **LWP** or process is the current one.

If *processorid* is **PBIND_NONE**, the processor bindings of the specified **LWPs** are cleared.

If *processorid* is **PBIND_QUERY**, the processor bindings are not changed.

The effective user of the calling process must be super user, or its real or effective user ID must match the real or effective user ID of the **LWPs** being bound. If the calling process does not have permission to change all of the specified **LWPs**, the bindings of the **LWPs** for which it does have permission will be changed even though an error is returned.

RETURN VALUES

processor_bind returns 0 if successful; otherwise, -1 is returned and errno is set to reflect the error.

ERRORS

ESRCH	No processes or **LWPs** were found to match the criteria specified by *idtype* and id.
EINVAL	The specified processor is not on-line.
EINVAL	idtype was not **P_PID** or **P_LWPID**.
EFAULT	The location pointed to by *obind* was not NULL and not writable by the user.
EPERM	The effective user of the calling process is not super user, and its real or effective user ID does not match the real or effective user ID of one of the **LWPs** being bound.

SEE ALSO

psradm(), psrinfo(), p_online(), pset_bind(), sysconf()

processor_info

NAME

processor_info - determine type and status of a processor

SYNOPSIS

#include <sys/types.h>

#include <sys/processor.h>

int processor_info(processorid_t processorid, processor_info_t *infop);

DESCRIPTION

The status of the processor specified by *processorid* is returned in the *processor_info_t* structure pointed to by *infop*. The structure contains the following members:

int	*pi_state;*	*/* **P_ONLINE**, **P_OFFLINE** or **P_POWEROFF** */*
char	*pi_processor_type[PI_TYPELEN];*	
char	*pi_fputypes[PI_FPUTYPE];*	
int	*pi_clock;*	*/* CPU clock freq in MHz */*

The fields have the following meanings:

pi_state	is the current state of the processor, either **P_ONLINE**, **P_OFFLINE** or **P_POWEROFF**.
pi_processor_type	is a NULL-terminated ASCII string specifying the type of the processor.
pi_fputypes	is a NULL-terminated ASCII string containing the comma-separated types of floating-point units (**FPUs**) attached to the processor. This string will be empty if no FPU is attached.
pi_clock	is the processor clock frequency rounded to the nearest megahertz. It may be 0 if not known.

RETURN VALUES

processor_info() returns 0 if successful.Otherwise -1 is returned and errno is set to reflect the error.

ERRORS

EINVAL	An non-existent processor ID was specified.
EFAULT	The *processor_info_t* structure pointed to by *infop* was not writable by the user.

SEE ALSO

psradm(), psrinfo(), p_online(), sysconf(

psignal, psiginfo

NAME

psignal, psiginfo - system signal messages

SYNOPSIS

#include <siginfo.h>

void psignal(int sig, const char *s);

void psiginfo(siginfo_t *pinfo, char *s);

DESCRIPTION

psignal() and ***psiginfo()*** produce messages on the standard error output describing a signal. *sig* is a signal that may have been passed as the first argument to a signal handler. *pinfo* is a pointer to a *siginfo* structure that may have been passed as the second argument to an enhanced signal handler (see ***sigaction()***). The argument string s is printed first, then a colon and a blank, then the message and a newline.

SEE ALSO

sigaction(), gettext(), perror(), setlocale(), siginfo(), signal()

NOTES

If the application is linked with *-lintl*, then messages printed from these functions are in the native language specified by the **LC_MESSAGES** locale category; see ***setlocale()***.

write, pwrite, writev

NAME

write, pwrite, writev - write on a file

SYNOPSIS

#include <unistd.h>

ssize_t write(int fildes, const void *buf, size_t nbyte);

ssize_t pwrite(int fildes, const void *buf, size_t nbyte,off_t offset);

#include <sys/uio.h>

int writev(int fildes, const struct iovec *iov, int iovcnt);

DESCRIPTION

The ***write()*** function attempts to write nbyte bytes from the buffer pointed to by buf to the file associated with the open file descriptor, fildes. If nbyte is 0, ***write()*** will return 0 and have no other results if the file is a regular file; otherwise, the results are unspecified. On a regular file or other file capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file offset associated with fildes. Before successful return from ***write()***, the file offset is incremented by the number of bytes actually written.On a regular file, if this incremented file offset is greater than the length of the file, the length of the file will be set to this file offset. If the **O_SYNC** flag of the file status flags is set and fildes refers to a regular file, a successful ***write()*** does not return until the data is delivered to the underlying hardware. On a file not capable of seeking, writing always takes place starting at the current position.The value of a file offset associated with such a device is undefined. If the **O_APPEND** flag of the file status flags is set, the file offset will be set to the end of the file prior to each write and no intervening file modification operation will occur between changing the file offset and the write operation. For regular files, no data transfer will occur past the offset maximum established in the open file description with fildes.

A ***write()*** to a regular file is blocked if mandatory file/record locking is set (see ***chmod()***), and there is a record lock owned by another process on the segment of the file to be written:

* If **O_NDELAY** or **O_NONBLOCK** is set, ***write()*** returns -1 and sets errno to **EAGAIN**.

* If **O_NDELAY** and **O_NONBLOCK** are clear, ***write()*** sleeps until all blocking locks are removed or the ***write()*** is terminated by a signal.

If a ***write()*** requests that more bytes be written than there is room for-for example, if the write would exceed the process file size limit (see ***getrlimit()*** and ***ulimit()***), the system file size limit, or the free space on the device-only as many bytes as there is room for will be written.For example, suppose there is space for 20 bytes more in a file before reaching a limit. A ***write()*** of 512-bytes returns 20. The next ***write()*** of a non-zero number of bytes gives a failure return (except as noted for pipes and **FIFO** below). If ***write()*** is interrupted by a signal before it writes any data, it will return -1 with errno set to **EINTR**.

If ***write()*** is interrupted by a signal after it successfully writes some data, it will return the number of bytes written. If the value of nbyte is greater than **SSIZE_MAX**, the result is implementation-dependent. After a ***write()*** to a regular file has successfully returned:

* Any successful ***read()*** from each byte position in the file that was modified by that write will return the data specified by the ***write()*** for that position until such byte positions are again modified.

* Any subsequent successful ***write()*** to the same byte position in the file will overwrite that file data. Write requests to a pipe or **FIFO** are handled the same as a regular file with the following exceptions:

* There is no file offset associated with a pipe, hence each write request appends to the end of the pipe.

* Write requests of {**PIPE_BUF**} bytes or less are guaranteed not to be interleaved with data from other processes doing writes on the same pipe.Writes of greater than {**PIPE_BUF**} bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the **O_NONBLOCK** or **O_NDELAY** flags are set.

* If **O_NONBLOCK** and **O_NDELAY** are clear, a write request may cause the process to block, but on normal completion it returns nbyte.

* If **O_NONBLOCK** and **O_NDELAY** are set, ***write()*** does not blocktheprocess.If a ***write()*** request for {**PIPE_BUF**} or fewer bytes succeeds completely ***write()*** returns nbyte.Otherwise, if **O_NONBLOCK** is set, it returns -1 and sets errno to **EAGAIN** or if **O_NDELAY** is set, it returns 0. A ***write()*** request for greater than {**PIPE_BUF**} bytes transfers what it can and returns the number of bytes written or it transfers no data and, if **O_NONBLOCK** is set, returns -1 with errno set to **EAGAIN** or if **O_NDELAY** is set, it returns 0. Finally, if a request is greater than {**PIPE_BUF**} bytes and all data previously written to the pipe has been read, ***write()*** transfers at least {**PIPE_BUF**} bytes. When attempting to write to a file descriptor (other than a pipe, a **FIFO**, a socket, or a **STREAM**) that supports non-blocking writes and cannot accept the data immediately:

* If **O_NONBLOCK** and **O_NDELAY** are clear, ***write()*** blocks until the data can be accepted.

* If **O_NONBLOCK** or **O_NDELAY** is set, ***write()*** does not block the process.If some data can be written without blocking the process, ***write()*** writes what it can and returns the number of bytes written. Otherwise, if **O_NONBLOCK** is set, it returns -1 and sets errno to **EAGAIN** or if **O_NDELAY** is set, it returns 0. Upon successful completion, where nbyte is greater than 0, ***write()*** will mark for update the *st_ctime* and *st_mtime* fields of the file, and if the file is a regular file, the **S_ISUID** and **S_ISGID** bits of the file mode may be cleared. For STREAMS files (see ***intro()*** and ***streamio***()), the operation of ***write()*** is determined by the values of the minimum and maximum nbyte range ("packet size") accepted by the **STREAM**.These values are contained in the topmost **STREAM** module, and can not be set or tested from user level. If *nbyte* falls within the packet size range, nbyte bytes are written. If nbyte does not fall within the range and the minimum packet size value is zero, ***write()*** breaks the buffer into maximum packet size segments prior to sending the data downstream (the last segment may be smaller than the maximum If nbyte does not fall within the range and the minimum value is non-zero, ***write()*** fails and sets errno to **ERANGE**. Writing a zero-length buffer (nbyte is zero) to a STREAMS device sends a zero length message with zero returned. However, writing a zero-length buffer to a pipe or **FIFO** sends no message and zero is returned. The user program may issue the **I_SWROPT** ***ioctl()*** to enable zero-length messages to be sent across the pipe or **FIFO**.

When writing to a **STREAM**, data messages are created with a priority band of zero.When writing to a socket or to a **STREAM** that is not a pipe or a **FIFO**:

* If **O_NDELAY** and **O_NONBLOCK** are not set, and the **STREAM** cannot accept data (the **STREAM** write queue is full due to internal flow control conditions), ***write()*** blocks until data can be accepted.

* If **O_NDELAY** or **O_NONBLOCK** is set and the **STREAM** can not accept data, ***write()*** returns -1 and sets errno to **EAGAIN**.

* If **O_NDELAY** or **O_NONBLOCK** is set and part of the buffer has already been written when a condition occurs in which the **STREAM** cannot accept additional data, ***write()*** terminates and returns the number of bytes written.

In addition, ***write()*** and ***writev()*** will fail if the **STREAM** head had processed an asynchronous error before the call. In this case, the value of errno does not reflect the result of ***write()*** or ***writev()*** but reflects the prior error.

pwrite()

The ***pwrite()*** function performs the same action as ***write()***, except that it writes into a given position without changing the file pointer. The first three arguments to ***pwrite()*** are the same as ***write()*** with the addition of a fourth argument offset for the desired position inside the file.

writev()

The ***writev()*** function performs the same action as ***write()***, but gathers the output data from the iovcnt buffers specified by the members of the iov array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt*-1]. The iovcnt buffer is valid if greater than 0 and less than or equal to {**IOV_MAX**}.See ***intro()*** for a definition of {**IOV_MAX**}.

The iovec structure contains the following members:

caddr_t	*iov_base;*
int	*iov_len;*

Each *iovec* entry specifies the base address and length of an area in memory from which data should be written. ***writev()*** always writes all data from an area before proceeding to the next. If fildes refers to a regular file and all of the *iov_len* members in the array pointed to by iov are 0, ***writev()*** will return 0 and have no other effect. For other file types, the behavior is unspecified. If the sum of the *iov_len* values is greater than **SSIZE_MAX**, the operation fails and no data is transferred.

RETURN VALUES

Upon successful completion, ***write()*** returns the number of bytes actually written to the file associated with fildes. This number is never greater than nbyte. Otherwise, -1 is returned and errno is set to indicate the error.

Upon successful completion, ***writev()*** returns the number of bytes actually written.Otherwise, it returns -1, the file-pointer remains unchanged, and errno is set to indicate an error.

ERRORS

The ***write()***, ***pwrite()***, and ***writev()*** function fail and the file pointer remains unchanged if one or more of the following are true:

EAGAIN Mandatory file/record locking is set, **O_NDELAY** or **O_NONBLOCK** is set, and there is a blocking record lock; total amount of system memory available when reading using raw I/O is temporarily insufficient; an attempt is made to write to a **STREAM** that can not accept data with the **O_NDELAY** or **O_NONBLOCK** flag set; or a write to a pipe or **FIFO** of {**PIPE_BUF**} bytes or less is requested and less than *nbytes* of free space is available.

EBADF *fildes* is not a valid file descriptor open for writing.

EDEADLK The write was going to go to sleep and cause a deadlock situation to occur.

EDQUOT The user's quota of disk blocks on the file systemcontainingthefilehasbeen exhausted.

EFAULT buf points to an illegal address.

EFBIG An attempt is made to write a file that exceeds the process' file size limit or the maximum file size (see ***getrlimit()*** and ***ulimit()***).

EFBIG The file is a regular file, *nbyte* is greater than 0, and the starting position is greater than or equal to the offset maximum established in the file description associated with fildes.

EINTR A signal was caught during the write operation and no data was transferred.

EIO The process is in the background and is attempting to write to its controlling terminal whose **TOSTOP** flag is set, or the process is neither ignoring nor blocking **SIGTTOU** signals and the process group of the process is orphaned.

ENOLCK Enforced record locking was enabled and {**LOCK_MAX**} regions are already locked in the system, or the system record lock table was full and the write could not go to sleep until the blocking record lock was removed.

ENOLINK fildes is on a remote machine and the link to that machine is no longer active.

ENOSPC During a write to an ordinary file, there is no free space left on the device.

ENOSR An attempt is made to write to a STREAMS with insufficient STREAMS memory resources available in the system.

ENXIO A hang-up occurred on the **STREAM** being written to.

EPIPE An attempt is made to write to a pipe or a **FIFO** that is not open for reading by any process, or that has only one end open (or to a file descriptor created by ***socket()***, using type **SOCK_STREAM** that is no longer connected to a peer endpoint). A **SIGPIPE** signal will also be sent to the process.The process dies unless special provisions were taken to catch or ignore the signal.

ERANGE The transfer request size was outside the range supported by the STREAMS file associated with fildes.

The ***pwrite()*** function fails and the file pointer remains unchanged if:

ESPIPE fildes is associated with a pipe or **FIFO**.

The ***writev()*** function will fail if:

EINVAL The sum of the *iov_len* values in the iov array would overflow an ssize_t.

The ***write()*** and ***writev()*** functions may fail if:

EINVAL The **STREAM** or multiplexer referenced by fildes is linked (directly or indirectly) downstream from a multiplexer.

ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the device.

ENXIO A hang-up occurred on the **STREAM** being written to.

A write to a **STREAMS** file may fail if an error message has been received at the **STREAM** head. In this case, errno is set to the value included in the error message.

The ***writev()*** function may fail and set errno to:

EINVAL iovcnt was less than or equal to 0 or greater than {**IOV_MAX**}; one of the *iov_len* values in the iov array was negative; or the sum of the *iov_len* values in the iov array overflowed an int.

SEE ALSO

chmod(), creat(), dup(), fcntl(), getrlimit(), ioctl(), lseek(), open(), pipe(), ulimit(), socket(), streamio()

realpath

NAME

realpath - resolve pathname

SYNOPSIS

#include <stdlib.h>

char *realpath(const char *file_name, char *resolved_name);

DESCRIPTION

The ***realpath()*** function derives, from the pathname pointed to by *file_name*, an absolute pathname that names the same file, whose resolution does not involve ".", "..", or symbolic links. The generated pathname is stored, up to a maximum of {**PATH_MAX**} bytes, in the buffer pointed to by resolved_name.

The ***realpath()*** function can handle both relative and absolute path names. For absolute path names and the relative names whose resolved name cannot be expressed relatively (for example, *../../reldir*), it returns the resolved absolute name. For the other relative path names, it returns the resolved relative name.

RETURN VALUES

On successful completion, ***realpath()*** returns a pointer to the resolved name.Otherwise, ***realpath()*** returns a null pointer and sets errno to indicate the error, and the contents of the buffer pointed to by *resolved_name* are undefined.

ERRORS

The ***realpath()*** function will fail if:

EACCES	Read or search permission was denied for a component of file_name.
EINVAL	Either the *file_name* or *resolved_name* argument is a null pointer.
EIO	An error occurred while reading from the file system.
ELOOP	Too many symbolic links were encountered in resolving path.
ENAMETOOLONG	The *file_name* argument is longer than {**PATH_MAX**} or a pathname component is longer than **NAME_MAX**.
ENOENT	A component of *file_name* does not name an existing file or *file_name* points to an empty string. ***realpath()*** **ENOTDIRA** component of the path prefix is not a directory.

The ***realpath()*** function may fail if:

ENAMETOOLONG	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {**PATH_MAX**}.
ENOMEM	Insufficient storage space is available.

SEE ALSO

getcwd(), sysconf()

NOTES

realpath() operates on null-terminated strings. One should have execute permission on all the directories in the given and the resolved path. ***realpath()*** may fail to return to the current directory if an error occurs.

select, FD_SET, FD_CLR, FD_ISSET, FD_ZERO

NAME

select, ***FD_SET***, ***FD_CLR***, ***FD_ISSET***, ***FD_ZERO*** - synchronous I/O multiplexing

SYNOPSIS

#include <sys/time.h>

int ***select(int nfds,*** **fd_set** ****readfds,*** **fd_set** ****writefds,*** **fd_set** ****errorfds, struct timeval *timeout);***

void ***FD_SET(int fd,*** **fd_set** ****fdset);***

void ***FD_CLR(int fd,*** **fd_set** ****fdset);***

int ***FD_ISSET(int fd,*** **fd_set** ****fdset);***

void ***FD_ZERO(*****fd_set** ****fdset);***

DESCRIPTION

The ***select()*** function indicates which of the specified file descriptors is ready for reading, ready for writing, or has an error condition pending. If the specified condition is false for all of the specified file descriptors, ***select()*** blocks, up to the specified timeout interval, until the specified condition is true for at least one of the specified file descriptors. The ***select()*** function supports regular files, terminal and pseudo-terminal devices, STREAMS-based files, **FIFO**s and pipes. The behavior of ***select()*** on file descriptors that refer to other types of file is unspecified. The *nfds* argument specifies the range of file descriptors to be tested. The ***select()*** function tests file descriptors in the range of 0 to *nfds*-1. If the *readfs* argument is not a null pointer, it points to an object of type *fd_set* that on input specifies the file descriptors to be checked for being ready to read, and on output indicates which file descriptors are ready to read.

If the *writefs* argument is not a null pointer, it points to an object of type **fd_set** that on input specifies the file descriptors to be checked for being ready to write, and on output indicates which file descriptors are ready to write. If the *errorfds* argument is not a null pointer, it points to an object of type **fd_set** that on input specifies the file descriptors to be checked for error conditions pending, and on output indicates which file descriptors have error conditions pending. On successful completion, the objects pointed to by the *readfs*, *writefs*, and *errorfds* arguments are modified to indicate which file descriptors are ready for reading, ready for writing, or have an error condition pending, respectively. For each file descriptor less than *nfds*, the corresponding bit will be set on successful completion if it was set on input and the associated condition is true for that file descriptor. If the timeout argument is not a null pointer, it points to an object of type struct timeval that specifies a maximum interval to wait for the selection to complete.If the timeout argument points to an object of type struct timeval whose members are 0, ***select()*** does not block.If the timeout argument is a null pointer, ***select()*** blocks until an event causes one of the masks to be returned with a valid (non-zero) value.If the time limit expires before any event occurs that would cause one of the masks to be set to anon-zero value, ***select()*** completes successfully and returns 0. If the *readfs*, *writefs*, and *errorfds* arguments are all null pointers and the timeout argument is not a null pointer, ***select()*** blocks for the time specified, or until interrupted by a signal. If the *readfs*, *writefs*, and *errorfds* arguments are all null pointers and the timeout argument is a null pointer, ***select()*** blocks until interrupted by a signal. File descriptors associated with regular files always select true for ready to read, ready to write, and error conditions. On failure, the objects pointed to by the *readfs*, *writefs*, and *errorfds* arguments are not modified. If the timeout interval expires without the specified condition being true for any of the specified file descriptors, the objects pointed to by the *readfs*, *writefs*, and *errorfds* arguments have all bits set to 0. Selecting true for reading on a socket descriptor upon which a ***listen()*** call has been performed indicates that a subsequent ***accept()*** call on that descriptor will not block.

File descriptor masks of type **fd_set** can be initialized and tested with the macros ***FD_CLR()***, ***FD_ISSET()***, ***FD_SET()***, and ***FD_ZERO()***. ***FD_CLR*** (*fd*, &*fdset*) Clears the bit for the file descriptor *fd* in the file descriptor set *fdset*. ***FD_ISSET*** (*fd*, &*fdset*) Returns a non-zero value if the bit for the file descriptor fd is set in the file descriptor set pointed to by fdset, and 0 otherwise. ***FD_SET***(fd, &*fdset*) Sets the bit for the file descriptor fd in the file descriptor set fdset. ***FD_ZERO*** (&*fdset*) initializes the file descriptor set *fdset* to have zero bits for all file descriptors. The behavior of these macros is undefined if the *fd* argument is less than 0 or greater than or equal to **FD_SETSIZE**.

RETURN VALUES

FD_CLR(), ***FD_SET()***, and ***FD_ZERO()*** return no value. ***FD_ISSET()*** returns a non-zero value if the bit for the file descriptor fd is set in the file descriptor set pointed to by *fdset*, and 0 otherwise. On successful completion, ***select()*** returns the total number of bits set in the bit masks. Otherwise, -1 is returned, and errno is set to indicate the error.

ERRORS

Under the following conditions, ***select()*** fails and sets errno to:

EBADF One or more of the file descriptor sets specified a file descriptor that is not a valid open file descriptor.

EINTR The ***select()*** function was interrupted before any of the selected events occurred and before the timeout interval expired.

If **SA_RESTART** has been set for the interrupting signal, it is implementation-dependent whether ***select()*** restarts or returns with **EINTR**.

EINVAL An invalid timeout interval was specified.

EINVAL The *nfds* argument is less than 0, or greater than or equal to **FD_SETSIZE**.

EINVAL One of the specified file descriptors refers to a **STREAM** or multiplexer that is linked (directly or indirectly) downstream from a multiplexer.

EINVAL A component of the pointed-to time limit is outside the acceptable range: *t_sec* must be between 0 and 10^8, inclusive. *t_usec* must be greater than or equal to 0, and less than 10^6.

USAGE

The ***poll()*** function is preferred over this function, particularly when the number of file descriptors exceeds **FD_SETSIZE**. The use of a timeout does not affect any pending timers set up by ***alarm()***, ***ualarm()*** or ***setitimer()***. On successful completion, the object pointed to by the timeout argument may be modified.

SEE ALSO

alarm(), fcntl(), poll(), read(), setitimer(), write(), accept(), listen(), ualarm()

NOTES

The default value for **FD_SETSIZE** (currently 1024) is larger than the default limit on the number of open files. In order to accommodate programs that may use a larger number of open files with ***select()***, it is possible to increase this size within a program by providing a larger definition of **FD_SETSIZE** before the inclusion of <*sys/types.h*>.

setuid, setegid, seteuid, setgid

NAME

setuid, setegid, seteuid, setgid - set user and group IDs

SYNOPSIS

#include <sys/types.h>

#include <unistd.h>

int setuid(uid_t uid);

int setegid(gid_t egid);

int seteuid(uid_t euid);

int setgid(gid_t gid);

DESCRIPTION

The ***setuid()*** function sets the real user ID, effective user ID, and saved user ID of the calling process. The ***setgid()*** function sets the real group ID, effective group ID, and saved group ID of the calling process. The ***setegid()*** and ***seteuid()*** functions set the effective group and user ID's respectively for the calling process. At login time, the real user ID, effective user ID, and saved user ID of the login process are set to the login ID of the user responsible for the creation of the process. The same is true for the real, effective, and saved group IDs; they are set to the group ID of the user responsible for the creation of the process. When a process calls ***exec()*** to execute a file (program), the user and/or group identifiers associated with the process can change. If the file executed is a set-user-ID file, the effective and saved user IDs of the process are set to the owner of the file executed. If the file executed is a set-group-ID file, the effective and saved group IDs of the process are set to the group of the file executed.If the file executed is not a set-user-ID or set-group-ID file, the effective user ID, saved user ID, effective group ID, and saved group ID are not changed. The following subsections describe the behavior of ***setuid()*** and ***setgid()*** with respect to the three types of user and group IDs. If the effective user ID of the process calling ***setuid()*** is the super-user, the real, effective, and saved user IDs are set to the uid parameter. If the effective user ID of the calling process is not the super-user, but uid is either the real user ID or the saved user ID of the calling process, the effective user ID is set to uid. If the effective user ID of the process calling ***setgid()*** is the super-user, the real, effective, and saved group IDs are set to the gid parameter. If the effective user ID of the calling process is not the super-user, but gid is either the real group ID or the saved group ID of the calling process, the effective group ID is set to gid.

RETURN VALUES

Upon successful completion, a value of 0 is returned.Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

setuid() and ***setgid()*** fail if one or more of the following is true:

EINVAL The uid or gid is out of range.

EPERM For ***setuid()*** and ***seteuid()*** the effective user of the calling process is not super-user, and the *uid* parameter does not match either the real or saved user IDs.For ***setgid()*** and ***setegid()*** the effective user of the calling process is not the super-user, and the gid parameter does not match either the real or saved group IDs.

string, strcasecmp, strncasecmp, strcat strncat, strchr, strrchr, strcmp, strncmp strcpy, strncpy, strcspn, strspn, strdup strlen, strpbrk, strstr, strtok, strtok_r

NAME

string, strcasecmp, strncasecmp, strcat, strncat, strchr, strrchr, strcmp, strncmp, strcpy, strncpy, strcspn, strspn, strdup, strlen, strpbrk, strstr, strtok, strtok_r - string operations

SYNOPSIS

#include <strings.h>

*int strcasecmp(const char *s1, const char *s2);*

*int strncasecmp(const char *s1, const char *s2, int n);*

#include <string.h>

*char *strcat(char *dst, const char *src);*

*char *strncat(char *dst, const char *src, size_t n);*

*char *strchr(const char *s, int c);*

*char *strrchr(const char *s, int c);*

*int strcmp(const char *s1, const char *s2);*

*int strncmp(const char *s1, const char *s2, size_t n);*

*char *strcpy(char *dst, const char *src);*

*char *strncpy(char *dst, const char *src, size_t n);*

*size_t strcspn(const char *s1, const char *s2);*

*size_t strspn(const char *s1, const char *s2);*

*char *strdup(const char *s1);*

*size_t strlen(const char *s);*

*char *strpbrk(const char *s1, const char *s2);*

*char *strstr(const char *s1, const char *s2);*

*char *strtok(char *s1, const char *s2);*

*char *strtok_r(char *s1, const char *s2, char **lasts);*

DESCRIPTION

The arguments *s*, *s1*, *s2*, *src*, and *dst* point to strings (arrays of characters terminated by a null character). The ***functionsstrcat()***,***strncat()***,***strcpy()***, ***strncpy()***, ***strtok()***, and ***strtok_r()*** all alter their first argument. These functions do not check for overflow of the array pointed to by the first argument. ***strcasecmp()*** and ***strncasecmp()*** are case-insensitive versions of ***strcmp()*** and ***strncmp()*** respectively, described below.***strcasecmp()*** and ***strncasecmp()*** assume the ASCII character set and ignore differences in case when comparing lower and upper case characters.

strcat() appends a copy of string src, including the terminating null character, to the end of string dst. ***strncat()*** appends at most n characters .Each returns a pointer to the null-terminated result. The initial character of src overrides the null character at the end of dst. ***strchr()*** returns a pointer to the first occurrence of c (converted to a char) in string s, or a null pointer if c does not occur in the string. ***strrchr()*** returns a pointer to the last occurrence of c. The null character terminating a string is considered to be part of the string.

strcmp() compares two strings byte-by-byte, according to the ordering of your machine's character set. The function returns an integer greater than, equal to, or less than 0, if the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2 respectively. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes that differ in the strings being compared. ***strncmp()*** makes the same comparison but looks at a maximum of n bytes. Bytes following a null byte are not compared.***strcpy()*** copies string src to dst including the terminating null character, stopping after the null character has been copied. ***strncpy()*** copies exactly n bytes, truncating src or adding null characters to dst if necessary. The result will not be null-terminated if the length of src is n or more.

Each function returns *dst*. ***strcspn()*** returns the length of the initial segment of string s1 that consists entirely of characters not from string *s2*. ***strspn()*** returns the length of the initial segment of string s1 that consists entirely of characters from string *s2*. ***strdup()*** returns a pointer to a new string that is a duplicate of the string pointed to by s1. The space for the new string is obtained using ***malloc()***. If the new string cannot be created, a null pointer is returned. ***strlen()*** returns the number of bytes in s, not including the terminating null character. ***strpbrk()*** returns a pointer to the first occurrence in string s1 of any character from string s2, or a null pointer if no character from s2 exists in s1. ***strstr()*** locates the first occurrence of the string s2 (excluding the terminating null character) in string s1. ***strstr()*** returns a pointer to the located string, or a null pointer if the string is not found. If s2 points to a string with zero length (that is, the string ""), the function returns s1.

strtok() can be used to break the string pointed to by s1 into a sequence of tokens, each of which is delimited by one or more characters from the string pointed to by s2. ***strtok()*** considers the string s1 to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token, and will have written a null character into *s1* immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument being a null pointer) will work through the string s1 immediately following that token. In this way subsequent calls will work through the string *s1* until no tokens remain. The separator string s2 may be different from call to call. When no token remains in *s1*, a null pointer is returned. ***strtok_r()*** has the same functionality as ***strtok()*** except that a pointer to a string place holder lasts must be supplied by the caller. The lasts pointer is to keep track of the next sub-string in which to search for the next token.

SEE ALSO

malloc(), setlocale(), strxfrm()

NOTES

The ***strtok_r()*** interface is as proposed in the **POSIX.4a Draft #6** document, and is subject to change to be compliant to the standard when it is accepted. When compiling multi-thread applications, the **_REENTRANT** flag must be defined on the compile line. This flag should only be used in multi-thread applications. All of these functions assume the default locale ``C.'' For some locales, ***strxfrm()*** should be applied to the strings before they are passed to the functions. ***strtok()*** is unsafe in multi-thread applications. ***strtok_r()*** should be used instead. ***string()***, ***strcasecmp()***,***strcat()***,***strchr()***,***strcmp()***, ***strcpy()***, ***strcspn()***, ***strdup()***, ***strlen()***, ***strncasecmp()***, ***strncat()***, ***strncmp()***, ***strncpy()***, ***strpbrk()***, ***strrchr()***, ***strspn()***, and ***strstr()***, are MT-Safe in multi-thread applications.

strsignal

NAME

strsignal - get error message string

SYNOPSIS

#include <string.h>

char *strsignal(int sig);

DESCRIPTION

strsignal() maps the signal number in sig to a string describing the signal, and returns a pointer to that string. ***strsignal()*** uses the same set of the messages as ***psignal()***. The returned string should not be overwritten.

RETURN VALUES

strsignal() returns NULL if sig is not a valid signal number.

SEE ALSO

gettext(), psignal(), setlocale(), str2sig()

NOTES

If the application is linked with *-lintl*, then messages returned from this function are in the native language specified by the **LC_MESSAGES** locale category; see ***setlocale()***.

sysfs

NAME

sysfs - get file system type information

SYNOPSIS

#include <sys/fstyp.h>

#include <sys/fsid.h>

int sysfs(int opcode, const char *fsname);

int sysfs(int opcode, int fs_index, char *buf);

int sysfs(int opcode);

DESCRIPTION

sysfs() returns information about the file system types configured in the system. The number of arguments accepted by ***sysfs()*** varies and depends on the opcode.The currently recognized *opcodes* and their functions are:

GETFSIND Translate *fsname*, a null-terminated file system type identifier, into a file-system type index.

GETFSTYP Translate *fs_index*, a file-system type index, into a null-terminated file-system type identifier and write it into the buffer pointed to by buf; this buffer must be at least of size **FSTYPSZ** as defined in <*sys/fstyp.h*>.

GETNFSTYP Return the total number of file system types configured in the system.

RETURN VALUES

Upon successful completion, ***sysfs()*** returns the file-system type index if the *opcode* is **GETFSIND**, a value of 0 if the *opcode* is **GETFSTYP**, or the number of file system types configured if the *opcode* is **GETNFSTYP**. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

sysfs() fails if one or more of the following are true:

EFAULT buf or *fsname* points to an illegal address.

EINVAL *fsname* points to an invalid file-system identifier; *fs_index* is zero, or invalid; *opcode* is invalid.

ttyslot

NAME

ttyslot - find the slot in the utmp file of the current user

SYNOPSIS

#include <stdlib.h>

int ttyslot(void);

DESCRIPTION

ttyslot() returns the index of the current user's entry in the */var/adm/utmp* file. The returned index is accomplished by scanning files in /dev for the name of the terminal associated with the standard input, the standard output, or the standard error output (0, 1, or 2).

RETURN VALUES

A value of -1 is returned if an error was encountered while searching for the terminal name or if none of the above file descriptors are associated with a terminal device.

FILES

/var/adm/utmp

SEE ALSO

getutent(), ttyname(), attributes()

uadmin

NAME

uadmin - administrative control

SYNOPSIS

#include <sys/uadmin.h>

int uadmin(int cmd, int fcn, int mdep);

DESCRIPTION

uadmin() provides control for basic administrative functions.This function is tightly coupled to the system administrative procedures and is not intended for general use.The argument *mdep* is provided for machine-dependent use and is not defined here. As specified by cmd, the following commands are available:

A_SHUTDOWN	The system is shut down. All user processes are killed, the buffer cache is flushed, and the root file system is unmounted.The action to be taken after the system has been shut down is specified by *fcn*. The functions are generic; the hardware capabilities vary on specific machines.
AD_HALT	Halt the processor(s).
AD_POWEROFF	Halt the processor(s) and turn off the power.
AD_BOOT	Reboot the system, using the kernel file.
AD_IBOOT	Interactive reboot; user is prompted for bootable program name.
A_REBOOT	The system stops immediately without any further processing.The action to be taken next is specified by *fcn* as above.
A_REMOUNT	The root file system is mounted again after having been fixed. This should be used only during the startup process.
A_FREEZE	Suspend the whole system. The system state is preserved in the state file. The following three subcommands are available.
AD_COMPRESS	Save the system state to the state file with compression of data.
AD_CHECK	Check if your system supports suspend and resume. Without performing a system suspend/resume, this command checks if this feature is currently available on your system.
AD_FORCE	Force **AD_COMPRESS** even when threads of drivers are not suspendable.

RETURN VALUES

Upon successful completion, the value returned depends on cmd as follows:

A_SHUTDOWN	Never returns.
A_REBOOT	Never returns.
A_FREEZE	0 upon resume.
A_REMOUNT	0 Upon unsuccessful completion, -1 is returned and errno is set to indicate the error.

ERRORS

uadmin() fails if any of the following are true:

EPERM The effective user of the calling process is not super-user.

ENOMEM Suspend/resume ran out of physical memory.

ENOSPC Suspend/resume could not allocate enough space on the root file system to store system information.

ENOTSUP Suspend/resume not supported on this platform.

ENXIO Unable to successfully suspend system.

EBUSY Suspend already in progress.

SEE ALSO

kernel(), uadmin()

vfork

NAME

vfork - spawn new process in a virtual memory efficient way

SYNOPSIS

#include <unistd.h>

pid_t vfork(void);

DESCRIPTION

vfork() can be used to create new processes without fully copying the address space of the old process. It is useful when the purpose of ***fork()*** would have been to create a new system context for an ***execve()***. ***vfork()*** differs from ***fork()*** in that the child borrows the parent's memory and thread of control until a call to ***execve()*** or an exit (either by a call to ***_exit()*** (see ***exit()***) or abnormally).The parent process is suspended while the child is using its resources. In a multi-threaded application, ***vfork()*** borrows only the thread of control which called ***vfork()*** in the parent; that is, the child contains only one thread. In that sense, in a multi-threaded application ***vfork()*** behaves like ***fork()***. ***vfork()*** can normally be used just like ***fork()***. It does not work, however, to return while running in the child's context from the procedure which called ***vfork()*** since the eventual return from ***vfork()*** would then return to a no longer existent stack frame. Be careful, also, to call ***_exit()*** rather than ***exit()*** if you cannot ***execve()***, since ***exit()*** will flush and close standard I/O channels, and thereby corrupt the parent processes standard I/O data structures. Even with ***fork()*** it is wrong to call ***exit()*** since buffered data would then be flushed twice.

RETURN VALUES

Upon successful completion, ***vfork()*** returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and the global variable errno is set to indicate the error.

ERRORS

vfork() will fail and no child process will be created if one or more of the following are true:

EAGAIN	The system-imposed limit on the total number of processes under execution would be exceeded. This limit is determined when the system is generated.
EAGAIN	The system-imposed limit on the total number of processes under execution by a single user would be exceeded. This limit is determined when the system is generated.
ENOMEM	There is insufficient swap space for the new process.

SEE ALSO

exec(), exit(), fork(), ioctl(), wait()

NOTES

The use of ***vfork()*** for any purpose except as a prelude to an immediate call to a function from the exec family, or to ***_exit()***, is not advised. ***vfork()*** is unsafe in multi-thread applications. This function will be eliminated in a future release.The memory sharing semantics of ***vfork()*** can be obtained through other mechanisms. To avoid a possible deadlock situation, processes that are children in the middle of a ***vfork()*** are never sent **SIGTTOU** or **SIGTTIN** signals; rather, output or *ioctls* are allowed and input attempts result in an **EOF** indication. On some systems, the implementation of ***vfork()*** causes the parent to inherit register values from the child. This can create problems for certain optimizing compilers if <*unistd.h*> is not included in the source calling ***vfork()***.

vhangup

NAME

vhangup - virtually "hangup" the current controlling terminal

SYNOPSIS

void vhangup(void);

DESCRIPTION

vhangup() is used by the initialization process ***init()*** (among others) to arrange that users are given "clean" terminals at login, by revoking access of the previous users' processes to the terminal.To effect this, ***vhangup()*** searches the system tables for references to the controlling terminal of the invoking process, revoking access permissions on each instance of the terminal that it finds. Further attempts to access the terminal by the affected processes will yield I/O errors (**EBADF** or **EIO**). Finally, a **SIGHUP** (hangup signal) is sent to the process group of the controlling terminal.

SEE ALSO

init()

syslog

NAME

vsyslog - log message with a varargs argument list

SYNOPSIS

#include <syslog.h>

#include <varargs.h>

int vsyslog(int priority, const char *message, va_list ap);

DESCRIPTION

vsyslog() is the same as ***syslog()*** except that instead of being called with a variable number of arguments, it is called with an argument list as defined by ***varargs()***.

SEE ALSO

syslog(), varargs()

__div64

NAME

__div64 - 64 bit division function

SYNOPSIS

long long __div64(long long a, long long b);

DESCRIPTION

The function ***__div64***() computes the quotient of the division of the numerator "*a*" by the denominator "*b*", truncates any fractional part, and return the signed long long results.

This function returns 0 if "b" is 0.

Trap handling, when the divisor is zero, is intentionally not present in this specification, since it is considered **SPARC** architecture version dependent.

__dtoll

NAME

__dtoll - convert double to long long

SYNOPSIS

long long __dtoll (double d)

DESCRIPTION

This function converts the double precision value of “d” to a signed long long (integer result) by truncating (discarding) any fractional part and returns the signed long long value.

__dtoll() raises an invalid exception if the integer portion is outside of the range:

$-2^{63} \le d < 2^{63}$

and returns the negative number in the inequality expression above if “d” is negative, otherwise returning the positive number in the inequality.

Rounding, overflow and exceptions handling are intentionally not present in this specification, since they are considered **SPARC** architecture version dependent. There is no guarantee that their behavior is similar to **SPARC** Architecture Version 9 ***FdTOx*** instruction.

__dtoull

NAME

__dtoull - convert double to unsigned long long

SYNOPSIS

unsigned long long __dtoull (double d);

DESCRIPTION

This function converts the double precision value of “d” to an unsigned long long (integer result) by truncating (discarding) any fractional part and returns the unsigned long long value.

__dtoull raises an invalid exception if the integer portion of “d” is outside of the range:

$0 \leq \boldsymbol{abs}(\mathrm{d}) < 2^{64.}$

Rounding, overflow and exceptions handling are intentionally not present in this specification, since they are considered **SPARC** architecture version dependent. There is no guarantee that their behavior is similar to *SPARC Architecture Version 9* ***FdTOx*** instruction.

__ftoll

NAME

__ftoll - convert float to long long

SYNOPSIS

long long __ftoll (float f);

DESCRIPTION

This function converts the single precision value of "*f*" to a signed long long (integer result) by truncating (discarding) any fractional part and returns the signed long long value.

__ftoll() raises an invalid exception if the integer portion is outside of the range:

$-2^{63} \le f < 2^{63}$

and returns the negative number in the inequality expression above if "*f*" is negative, otherwise returning the positive number in the inequality.

Rounding, overflow and exceptions handling are intentionally not present in this specification, since they are considered **SPARC** architecture version dependent. There is no guarantee that their behavior is similar to *SPARC Architecture Version 9* ***FsTOx*** instruction.

__ftoull

NAME

__ftoull - convert float to unsigned long long

SYNOPSIS

unsigned long long __ftoull(float f);

DESCRIPTION

This function converts the single precision value of "*f*" to an unsigned long long (integer result) by truncating (discarding) any fractional part and returns the unsigned long long value.

__ftoull raises an invalid exception if the integer portion of "*f*" is outside of the range:
$0 \leq \mathbf{abs}(f) < 2^{64}.$

Rounding, overflow and exceptions handling are intentionally not present in this specification, since they are considered **SPARC** architecture version dependent. There is no guarantee that their behavior is similar to *SPARC Architecture Version 9* ***FsTOx*** instruction.

__mul64

NAME

__mul64 - 64 bit multiplication function

SYNOPSIS

long long __mul64(long long a, long long b);

DESCRIPTION

This function implements the multiplication of "*a*" and "*b*" ("*a * b*").

This function returns $p - 2^{64}$ if $p \geq 2^{63}$; it returns "*p*" otherwise. Where "*p*" denote the mathematical product modulo 2^{64} of "a" and "*b*"; "*p*" is in the range:

$0 \leq p < 2^{64}$

Overflow handling is intentionally not present in this specification, since it is considered **SPARC** architecture version dependent.

__rem64

NAME

__rem64 - 64 bit remain function

SYNOPSIS

long long __rem64(long long a, long long b);

DESCRIPTION

The function ***__rem64***() computes the remainder of the division of the numerator “a” by the “denominator “b” and returns the signed long long result.

This function returns 0 if “*b*” is 0.

Trap handling, if the divisor is zero, is intentionally not present in this specification, since it is considered **SPARC** architecture version dependent.

__udiv64

NAME

__udiv64 - Unsigned 64 bit division function

SYNOPSIS

unsigned long long __udiv64(unsigned long long a, unsigned long long b);

DESCRIPTION

The function ***__udiv64***() computes the quotient of the division of the numerator "*a*" by the denominator "*b*", truncates any fractional part, and return the unsigned long long result.

This function returns 0 if "*b*" is 0.

Trap handling, if the divisor is zero, is intentionally not present in this specification, since it is considered **SPARC** architecture version dependent.

__umul64

NAME

__umul64 - Unsigned 64 bit multiplication function

SYNOPSIS

unsigned long long __umul64(unsigned long long a, unsigned long long b);

DESCRIPTION

This function implements the multiplication of "*a*" and "*b*" ("*a * b*").

It returns the product modulo 2^{64} of "*a*" and "*b*". The result is in unsigned long long.

Overflow handling is intentionally not present in this specification, since it is considered **SPARC** architecture version dependent.

__urem64

NAME

__urem64 - unsigned 64 bits remain function

SYNOPSIS

unsigned long long __urem64(unsigned long long a, unsigned long long b);

DESCRIPTION

The function ***__urem64***() computes the remainder of the division of the numerator "*a*" by the "denominator "*b*" and returns the unsigned long long result.

This function returns 0 if "*b*" is 0.

Trap handling is intentionally not present in this specification, since it is considered **SPARC** architecture version dependent.

_Q_lltoq

NAME

_Q_lltoq - Convert long long to long double

SYNOPSIS

long double _Q_lltoq (long long a);

DESCRIPTION

This function converts the long long value of "a" to quad precision (floating result) and returns the quad precision value.

Rounding, overflow and exceptions handling are intentionally not present in this specification, since they are considered **SPARC** architecture version dependent. There is no guarantee that their behavior is similar to **SPARC** Architecture Version 9 ***FxTOq*** instruction.

_Q_qtoll

NAME

_Q_qtoll - convert long double to long long

SYNOPSIS

long long _Q_qtoll(long double a);

DESCRIPTION

This function converts the quad precision value of "*a*" to a signed long long (integer result) by truncating (discarding) any fractional part and returns the signed long long value.

__Q_qtoll() raises an invalid exception if the integer portion is outside of the range:

$-2^{63} \leq a < 2^{63}$

and returns the negative number in the inequality expression above if "a" is negative, otherwise returning the positive number in the inequality.

Rounding, overflow and exceptions handling are intentionally not present in this specification, since they are considered **SPARC** architecture version dependent. There is no guarantee that their behavior is similar to **SPARC** Architecture Version 9 ***FqTOx*** instruction.

_Q_qtoull

NAME

_Q_qtoull - convert double to unsigned long long.

SYNOPSIS

unsigned long long _Q_qtoull (long double a);

DESCRIPTION

This function converts the quad precision value of "*a*" to an unsigned long long (integer result) by truncating (discarding) any fractional part and returns the unsigned long long value.

_Q_qtoull raises an invalid exception if the integer portion of "*a*" is outside of the range:

$0 \leq \mathbf{abs}(a) < 2^{64}.$

Rounding, overflow and exceptions handling are intentionally not present in this specification, since they are considered **SPARC** architecture version dependent. There is no guarantee that their behavior is similar to **SPARC** Architecture Version 9 ***FqTOx*** instruction.

_Q_ulltoq

NAME

_Q_ulltoq - convert unsigned long long to long double

SYNOPSIS

long double _Q_ulltoq (unsigned long long a);

DESCRIPTION

This function converts the unsigned long long value of "*a*" to quad precision (floating result) and returns the quad precision value.

Rounding, overflow and exceptions handling are intentionally not present in this specification, since they are considered **SPARC** architecture version dependent. There is no guarantee that their behavior is similar to *SPARC Architecture Version 9* ***FxTOq*** instruction.

fgetgrent_r

NAME

fgetgrent_r - get group entry

SYNOPSIS

#include <grp.h>

struct group *fgetgrent_r (FILE *f, struct group *result, char *buffer, int buflen);

DESCRIPTION

fgetgrent_r() reads and parses the next line from the stream “f”, which is assumed to have the format of the group file, where each entry is of the form:

groupname: password: gid: user-list

The function ***fgetgrent_r***() provides a reentrant interface for the ***fgetgrent***() function which uses static storage that is re-used in each call. The use of static storage makes ***fgetgrent***() unsafe for use in multithreaded applications. ***fgetgrent_r***() performs the same operation as ***fgetgrent***(). ***fgetgrent_r***(), however, uses buffers supplied by the caller to store returned results, and is safe for use in both single-threaded and multithreaded applications.

The parameter “result” must be a pointer to a “struct group” structure allocated by the caller. On successful completion, the function returns the group entry in this structure. The parameter “buffer” must be a pointer to a buffer supplied by the caller. This buffer is used as storage space for the group data. All of the pointers within the returned struct group result point to data stored within this buffer (see RETURN VALUES). The buffer must be large enough to hold all of the data associated with the group entry. The parameter “buflen” should give the size in bytes of the buffer indicated by buffer.

RETURN VALUES

Group entries are represented by the struct group structure defined in <*grp.h*>:

```
struct group {
        char        *gr_name;     /* the name of the group */
        char        *gr_passwd; /* the encrypted group password */
        gid_t       gr_gid;       /* the numerical group ID */
        char        **gr_mem;     /* vector of pointers to member names */
};
```

The function ***fgetgrent_r***() returns a pointer to a *struct group* if it successfully enumerates an entry; otherwise it returns NULL, indicating the end of the enumeration.

When the pointer returned by ***fgetgrent_r***() is non-NULL, it is always equal to the result pointer that was supplied by the caller.

ERRORS

ERANGE ***fgetgrent_r***() will return NULL and set errno to **ERANGE** if the length of the buffer supplied by caller is not large enough to store the result.

NOTES

Programs that use ***fgetgrent_r***() cannot be linked statically since the implementations of these functions employ dynamic loading and linking of shared objects at run time.

fgetpwent_r

NAME

fgetpwent_r - get password entry

SYNOPSIS

#include <pwd.h>

struct passwd *fgetpwent_r (FILE *f, struct passwd *result, char *buffer, int buflen);

DESCRIPTION

This function is used to obtain password entries. ***fgetpwent_r***() reads and parses the next line from the stream f, which is assumed to have the format of the passwd file, where each entry is of the form:

username: password: uid: gid: gcos-field: home-dir: login-shell

The function ***fgetpwent_r***() provides a reentrant interface for ***fgetpwent***(). ***fgetpwent_r***() performs the same operation as ***fgetpwent***(). ***fgetpwent_r***(), however, uses buffers supplied by the caller to store returned results, and is safe for use in both single-threaded and multithreaded applications. ***fgetpwent***() is not safe for use in multithreaded applications since it uses static storage that is re-used in each call to this routine.

Each reentrant interface takes the same parameters as its non-reentrant counterpart, as well as the following additional parameters. The parameter result must be a pointer to a struct passwd structure allocated by the caller. On successful completion, the function returns the password entry in this structure. The parameter buffer must be a pointer to a buffer supplied by the caller. This buffer is used as storage space for the password data. All of the pointers within the returned struct passwd result point to data stored within this buffer. See RETURN VALUES. The buffer must be large enough to hold all of the data associated with the password entry. The parameter buflen should give the size in bytes of the buffer indicated by buffer.

RETURN VALUES

Password entries are represented by the struct passwd structure defined in <*pwd.h*>:

```
struct passwd {
        char     *pw_name;        /* user's login name */
        char     *pw_passwd;      /* no longer used */
        uid_t     pw_uid;         /* user's uid */
        gid_t     pw_gid;         /* user's gid */
        char     *pw_age;         /* not used */
        char     *pw_comment;     /* not used */
        char     *pw_gecos;       /* typically user's full name */
        char     *pw_dir;         /* user's home dir */
        char     *pw_shell;       /* user's login shell */
};
```

The function ***fgetpwent_r***() returns a pointer to a struct passwd if it successfully enumerates an entry; otherwise it returns NULL, indicating the end of the enumeration.

When the pointer returned by the reentrant function ***fgetpwent_r***() is non-NULL, it is always equal to the result pointer that was supplied by the caller.

ERRORS

ERANGE ***fgetpwent_r***() will return NULL and set errno to **ERANGE** if the length of the buffer supplied by caller is not large enough to store the result.

NOTES

fgetpwent_r() cannot be linked statically since, the implementations of this function employ dynamic loading and linking of shared objects at run time.

If the shell field is empty, "*login*" automatically assigns the default shell.

fork

NAME

fork - create a new process

SYNOPSIS

#include <sys/types.h>

#include <unistd.h>

pid_t fork(void);

DESCRIPTION

fork() causes the creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means the child process inherits the following attributes from the parent process:

- real user ID, real group ID, effective user ID, effective group ID
- environment
- open file descriptors
- close-on-exec flags (see ***exec***(**BA_OS**))
- signal handling settings (that is **SIG_DFL**, **SIG_IGN**, **SIG_HOLD**, Function address)
- supplementary group IDs
- set-user-ID mode bit
- set-group-ID mode bit
- profiling on/off status
- nice value (see ***nice***(**KE_OS**))
- scheduler class (see ***priocntl***(**RT_OS**))
- all attached shared memory segments (see ***shmop***(**KE_OS**))
- process group ID -- memory mappings (see ***mmap***(**KE_OS**))
- session ID (see ***exit***(**BA_OS**))
- current working directory
- root directory
- file mode creation mask (see ***umask***(**BA_OS**))
- resource limits (see ***getrlimit***(**BA_OS**))
- controlling terminal
- saved user ID and group ID

Scheduling priority and any per-process scheduling parameters that are specific to a given scheduling class may or may not be inherited according to the policy of that particular class (see ***priocntl***(**RT_OS**)).

The child process differs from the parent process in the following ways:

- The child process has a unique process ID which does not match any active process group ID.
- The child process has a different parent process ID (that is, the process ID of the parent process).
- The child process has its own copy of the parent's file descriptors and directory streams. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.
- Each shared memory segment remains attached and *shm_nattach* is incremented by 1.
- All *semadj* values are cleared (see ***semop***(**KE_OS**))).
- Process locks, text locks, data locks, and other memory locks are not inherited by the child (see ***plock***(**KE_OS**) and ***memcntl***(**RT_OS**)).
- The child process's *tms* structure is cleared: *tms_utime*, *stime*, *cutime*, and *cstime* are set to 0 (see ***times***(**BA_OS**)).
- The child processes resource utilizations are set to 0; see ***getrlimit***(**BA_OS**). The it_value and *it_interval* values for the **ITIMER_REAL** timer are reset to 0; see ***getitimer***(**RT_OS**).
- The set of signals pending for the child process is initialized to the empty set.
- No asynchronous input or asynchronous output operations are inherited by the child.

Record locks set by the parent process are not inherited by the child process (see ***fcntl***(**BA_OS**)).

fork() duplicates all the threads (see ***thr_create***) and **LWPs** in the parent process in the child process.

RETURN VALUES

Upon successful completion, ***fork***() returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of *(pid_t) - 1* is returned to the parent process, no child process is created, and *errno* is set to indicate the error.

ERRORS

fork() will fail and no child process will be created if one or more of the following are true:

EAGAIN There are two conditions that will cause an **EAGAIN** error.

The system-imposed limit on the total number of processes under execution by a single user would be exceeded.

The total amount of system memory available is temporarily insufficient to duplicate this process.

ENOMEM There is not enough swap space.

SEE ALSO

alarm(BA_OS), exec(BA_OS), exit(BA_OS), fcntl(BA_OS), getitimer(RT_OS), getrlimit(BA_OS), mmap(KE_OS), nice(KE_OS), plock(KE_OS), priocntl(RT_OS), ptrace(KE_OS), semop(KE_OS), shmop(KE_OS), times(BA_OS), umask(BA_OS), wait(BA_OS), memcntl(RT_OS), signal(BA_OS), system(BA_OS), thr_create

NOTES

The semantics of ***fork***() in a multi-threaded application are designated as **EXPERIMENTAL**. The **SCD2.3** definition of the multi-threaded semantics for ***fork***() is that the entire process is duplicated (i.e. all its threads). This differs from the **POSIX 1003.1c** specification in which only the thread invoking ***fork***() is duplicated in an MT application. MT semantics equivalent to the **POSIX 1003.1c** of ***fork***() are offered by the **SCD2.3** ***fork1***() interface.

Be careful to call ***_exit***() rather than ***exit***(**BA_OS**) if you cannot ***execve***(), since exit(**BA_OS**) will flush and close standard I/O channels, and thereby corrupt the parent processes standard I/O data structures. Using ***exit***(**BA_OS**) will flush buffered data twice. See ***exit***(**BA_OS**).

In a multi-threaded process, ***fork***() can cause blocking system calls to be interrupted and return with an error of **EINTR**

getgrent_r

NAME

getgrent_r - get group entry

SYNOPSIS

#include <grp.h>

struct group *getgrent_r (struct group *result, char *buffer, int buflen);

DESCRIPTION

getgrent_r() is used to obtain entries from the system's groups database. The function ***getgrent_r***() provides a reentrant interface for the ***getgrent***() function which uses static storage that is re-used in each call. The use of static storage makes ***getgrent***() unsafe for use in multithreaded applications.

getgrent_r() performs the same operation as ***getgrent***(). ***getgrent_r***() uses a buffer supplied by the caller to store returned results, and is safe for use in both single-threaded and multithreaded applications.

The parameter result must be a pointer to a struct group structure allocated by the caller. On successful completion, the function returns the group entry in this structure. The parameter buffer must be a pointer to a buffer supplied by the caller. This buffer is used as storage space for the group data. All of the pointers within the returned struct group result point to data stored within this buffer (see RETURN VALUES). The buffer must be large enough to hold all of the data associated with the group entry. The parameter buflen should give the size in bytes of the buffer indicated by buffer.

RETURN VALUES

Group entries are represented by the struct group structuredefined in <*grp.h*>:

```
struct group {
        char     *gr_name;      /* the name of the group */
        char     *gr_passwd;    /* the encrypted group password */
        gid_t    gr_gid;        /* the numerical group ID */
        char     **gr_mem;      /* vector of pointers to member names */
};
```

The function ***getgrent_r***() returns a pointer to a struct group if it successfully enumerates an entry; otherwise it returns NULL, indicating the end of the enumeration.

When the pointer returned by the reentrant function ***getgrent_r***() is non-NULL, it is always equal to the result pointer that was supplied by the caller.

ERRORS

ERANGE ***getgrent_r***() will return NULL and seterrno to **ERANGE** if the length of the buffer supplied by caller is not large enough to store the result.

NOTES

Programs that use ***getgrent_r***() cannot be linked statically since the implementations of this function employ dynamic loading and linking of shared objects at run time.

getlogin_r

NAME

getlogin_r - get login name

SYNOPSIS

#include <stdlib.h>

char *getlogin_r (char *name, int namelen);

DESCRIPTION

getlogin_r() returns a pointer to the login name associated with the controlling terminal. It may be used in conjunction with ***getpwnam_r***() to locate the correct password file entry when the same user id is shared by several login names.

If ***getlogin_r***() is called within a process that is not attached to a terminal, it returns a null pointer. The correct procedure for determining the login *name* is to call ***cuserid***(), or to call ***getlogin_r***() and if it fails to call ***getpwuid_r***().

getlogin_r() has the same functionality as ***getlogin***() except that a buffer name with length namelen has to be supplied by the caller to store the result. name must be at least **LOGNAME_MAX** bytes in size (defined in *limits.h*).

RETURN VALUES

Returns a null pointer if the login name is not found.

ERRORS

getlogin_r() will fail if the following is true:

ERANGE The size of the buffer is smaller than the result to be returned.

NOTES

The ***getlogin_r***() interface is different from the **POSIX 1003.1c** interface. The function ***getlogin_r*** is defined in **POSIX** as following:

int ***getlogin_r****(char *name, size_t namelen);*

This function is designated as **EXPERIMENTAL**.

The return values point to static data whose content is overwritten by each call.

getlogin() is unsafe in multi-thread applications. ***getlogin_r***() should be used instead.

getpwent_r

NAME

getpwent_r - get password entry

SYNOPSIS

#include <pwd.h>

struct passwd *getpwent_r (struct passwd *result, char *buffer, int buflen);

DESCRIPTION

This function is used to obtain password entries. The function ***getpwent_r***() is used to enumerate password entries from the system's passwords database. Successive calls to ***getpwent_r***() return either successive entries or NULL, indicating the end of the enumeration.

The function ***getpwent_r***() provides a reentrant interface for ***getpwent***(). ***getpwent_r***() performs the same operation as ***getpwent***(). ***getpwent_r***() uses buffers supplied by the caller to store returned results, and is safe for use in both single-threaded and multithreaded applications. ***getpwent***() is not safe for use in multithreaded applications since it uses static storage that is re-used in each call to this routine.

The parameter result must be a pointer to a struct passwd structure allocated by the caller. On successful completion, ***getpwent_r***() returns the password entry in this structure. The parameter buffer must be a pointer to a buffer supplied by the caller. This buffer is used as storage space for the password data. All of the pointers within the returned *struct passwd* result point to data stored within this buffer. See RETURN VALUES. The buffer must be large enough to hold all of the data associated with the password entry. The parameter buflen should give the size in bytes of the buffer indicated by buffer.

For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. If multiple threads interleave calls to ***getpwent_r***(), the threads will enumerate disjoint subsets of the password database.

RETURN VALUES

Password entries are represented by the struct passwd structure defined in <*pwd.h*>:

```
struct passwd {
        char    *pw_name;       /* user's login name */
        char    *pw_passwd;     /* no longer used */
        uid_t   pw_uid;         /* user's uid */
        gid_t   pw_gid;         /* user's gid */
        char    *pw_age;        /* not used */
        char    *pw_comment;    /* not used */
        char    *pw_gecos;      /* typically user's full name */
```

```
        char    *pw_dir;            /* user's home dir */
        char    *pw_shell;          /* user's login shell */
};
```

The function ***getpwent_r***() returns a pointer to a struct passwd if it successfully enumerates an entry; otherwise it returns NULL, indicating the end of the enumeration.

The function ***getpwent***() uses static storage, so returned data must be copied before a subsequent call to this function if the data is to be saved. When the pointer returned by ***getpwnam_r***() is non-NULL, it is always equal to the result pointer that was supplied by the caller.

ERRORS

ERANGE ***getpwent_r***() will return NULL and set errno to **ERANGE** if the length of the buffer supplied by caller is not large enough to store the result.

NOTES

getpwent_r cannot be linked statically since, the implementations of this function employ dynamic loading and linking of shared objects at run time.

If the shell field is empty, “*login*” automatically assigns the default shell.

getgrgid_r
getgrnam_r
getpwnam_r
getpwuid_r
readdir_r

NAME

getgrgid_r, getgrnam_r, getpwnam_r, getpwuid_r, readdir_r - Support routines for multithreading added to libsys and libc.

SYNOPSIS

#include <grp.h>

struct group *getgrgid_r (gid_t gid, struct group *result, char *buffer, int buflen);

struct group *getgrnam_r(const char *name, struct group *result, char *buffer, int buflen);

#include <pwd.h>

struct passwd *getpwnam_r (const char *name, struct passwd *result, char *buffer, int buflen);

struct passwd *getpwuid_r (uid_t uid, struct passwd *result, char *buffer, int buflen);

#include <dirent.h>

struct dirent *readdir_r(DIR *dirp, struct dirent *res);

DESCRIPTION and RETURN VALUES

These functions are "reentrant" versions of existing functions. They exist as the definition of the existing functions prevents the transparent implementation of multithreading, usually because of the use of a static storage area. In general, these functions are exactly equivalent to the non-reentrant versions in terms of function and results, but differ in providing for the implementation the necessary storage for completion of the function.

getgrgid_r and ***getgrnam_r*** are equivalent to ***getgrgid*** and ***getgrnam***, respectively. When these functions succeed, they return the argument result as their value. Otherwise they return NULL. When successful, the contents of result have been updated to return the group entry associated with either name or *gid*, respectively. buf is provided in order to store the strings and pointers needed to describe a group entry, and is buflen in length. If buflen is not large enough to store the resulting strings, the functions return NULL with errno set to **ERANGE**.

getpwuid_r and ***getpwnam_r*** are equivalent to getgrgid and getgrnam, respectively. When these functions succeed, they return the argument result as their value. Otherwise they return NULL. When successful, the contents of result have been updated to return the password entry associated with either name or uid, respectively. buf is provided in order to store the strings and pointers needed to describe a password entry, and is buflen in length. If buflen is not large enough to store the resulting strings, the functions return NULL with errno set to **ERANGE**.

readdir_r is equivalent to readdir except that res must be supplied by the caller to store the result. To allocate res, a block of storage equivalent to ***sizeof*** *(struct dirent)* + **_POSIX_PATH_MAX** should be allocated.

NOTES

These functions are designated as **EXPERMIMENTAL** since they have interfaces which are different from the ones in **POSIX 1003.1c**. The interfaces of these functions are in **POSIX** as following:

int	***getgrgid_r*** *(*	*gid_t*	*gid,*
		struct group	**grp,*
		char	**buffer,*
		size_t	*bufsize,*
		struct group	***result);*
int	***getgrnam_r*** *(*	*const char*	**name,*
		struct group	**grp,*
		char	**buffer,*
		size_t	*bufsize,*
		struct group	***result);*
int	***getpwnam_r*** *(*	*const char*	**name,*
		struct passwd	**pwd,*
		char	**buffer,*
		size_t	*buflen,*
		struct passwd	***result);*
int	***getpwuid_r*** *(*	*uid_t*	*uid,*
		struct passwd	**pwd,*
		char	**buffer,*
		size_t	*bufsize,*
		struct passwd	***result);*
int	***readdir_r*** *(*	*DIR*	**dirp,*
		struct direct	**entry,*
		struct dirent	***result);*

makecontext
swapcontext

NAME

makecontext, swapcontext - manipulate user contexts

SYNOPSIS

#include <ucontext.h>

void makecontext (ucontext_t *ucp, void(*func)(), int argc,...);

int swapcontext (ucontext_t *oucp, ucontext_t *ucp);

DESCRIPTION

These functions are useful for implementing user-level context switching between multiple threads of control within a process.

makecontext() modifies the context specified by ucp, which has been initialized using getcontext(); when this context is resumed using ***swapcontext***() or ***setcontext***() (see ***getcontext***(**BA_OS**)), program execution continues by calling the function func, passing it the arguments that follow argc in the ***makecontext***() call. The integer value of argc must match the number of arguments that follow *argc*. Otherwise the behavior is undefined.

swapcontext() saves the current context in the context structure pointed to by oucp and sets the context to the context structure pointed to by *ucp*.

RETURN VALUES

On successful completion, swapcontext returns a value of zero. Otherwise, a value of -1 is returned and errno is set to indicate the error.

ERRORS

These functions will fail if either of the following is true:

EFAULT	ucp or oucp points to an invalid address.
ENOMEM	ucp does not have enough stack left to complete the operation.

SEE ALSO

exit(**BA_OS**), ***getcontext***(**BA_OS**), ***sigaction***(**BA_OS**), ***sigprocmask***(**BA_OS**)

NOTES

The size of the *ucontext_t* structure may change in future releases. To remain binary compatible, users of these features must always use ***makecontext***() or ***getcontext***() to create new instances of them.

sbrk

NAME

sbrk - query the current break value

SYNOPSIS

#include <unistd.h>

void *sbrk (const int 0);

DESCRIPTION

The function sbrk is used to query the amount of space allocated for the calling process's data segment [see ***exec***(**BA_OS**)].

STATUS

This function is **DEPPRECATED** effective November 1st, 1993. It may be removed from the SCD as early as November 1st, 1996.

DIAGNOSTICS

Upon successful completion, sbrk returns the current break value. Otherwise, a value of -1 is returned and errno is set to indicate the error. If sbrk is called with a non-zero value, the application is not portable.

RATIONALE

Calling ***sbrk***(0) yields a value that, at one time, had a predictable, well defined interpretation. It has not had this property for many years, since the wide-spread usage of sparse, demand-paged address spaces. Its use is deprecated because the interpretation of the value returned is so highly variable as to be non-portable. It is more appropriately regarded as a function yielding a value relevant to one of many attributes of memory occupancy. ***sbrk*** (non-zero) is not specified in any relevant standard, as its interactions with and dependencies upon other memory allocation mechanisms (e.g., ***malloc***) are undefined. The use of ***sbrk*** (non-zero) is non-conforming since the implementation of system supplied functions may freely use such memory allocation mechanisms.

NOTES

Applications desiring memory allocation functionality should use malloc for this purpose. Alternatively, applications may construct their own memory allocation arenas by building upon ***mmap*** and mappings to */dev/zero*

swapctl

NAME

swapctl - manage swap space

SYNOPSIS

#include <sys/stat.h>

#include <sys/swap.h>

int swapctl(int cmd, void *arg);

DESCRIPTION

The ***swapctl***() function adds, deletes, or returns information about swap resources. cmd specifies one of the following options contained in <sys/swap.h>:

SC_ADD	/* add a resource for swapping */
SC_LIST	/* list the resources for swapping */
SC_REMOVE	/* remove a resource for swapping */
SC_GETNSWP	/* return number of swap resources */

When **SC_ADD** or **SC_REMOVE** is specified, arg is a pointer to a swapres structure containing the following members:

char	*sr_name;*	*/* pathname of resource */*
off_t	*sr_start;*	*/* offset to start of swap area */*
off_t	*sr_length;*	*/* length of swap area */*

The *sr_start* and *sr_length* members are specified in 512-byte blocks. A swap resource can only be removed by specifying the same values for the *sr_start* and *sr_length* members as were specified when it was added. Swap resources need not be removed in the order in which they were added.

When **SC_LIST** is specified, arg is a pointer to a swaptable structure containing the following members:

int	*swt_n;*		*/* number of swapents following */*
struct	*swapent*	*swt_ent[];*	*/* array of swt_n swapents */*

A swapent structure contains the following members:

char	*ste_path;	/* name of the swap file */
off_t	ste_start;	/* starting block for swapping */
off_t	ste_length;	/* length of swap area */
long	ste_pages;	/* number of pages for swapping */
long	ste_free;	/* number of ste_pages free */
long	ste_flags;	/* **ST_INDEL** bit set if swap file is now being deleted */

The **SC_LIST** function causes ***swapctl***() to return at most ***swt_n*** entries. The return value of ***swapctl***() is the number actually returned. The **ST_INDEL** bit is turned on in ste_flags if the swap file is in the process of being deleted. When **SC_GETNSWP** is specified, ***swapctl***() returns as its value the number of swap resources in use. arg is ignored for this operation. The **SC_ADD** and **SC_REMOVE** functions will fail if calling process does not have appropriate privileges.

RETURN VALUES

Upon successful completion, the function ***swapctl***() returns a value of 0 for **SC_ADD** or **SC_REMOVE**, the number of struct swapent entries actually returned for **SC_LIST**, or the number of swap resources in use for **SC_GETNSWP**. Upon failure, the function ***swapctl***() returns a value of -1 and sets errno to indicate an error.

ERRORS

Under the following conditions, the function ***swapctl***() fails and sets errno to:

EEXIST Part of the range specified by *sr_start* and *sr_length* is already being used for swapping on the specified resource (**SC_ADD**).

EFAULT Either arg, *sr_name*, or *ste_path* points to an illegal address.

EINVAL The specified function value is not valid, the path specified is not a swap resource (**SC_REMOVE**), part of the range specified by sr_start and sr_length lies outside the resource specified (**SC_ADD**), or the specified swap area is less than one page (**SC_ADD**).

EISDIR The path specified for **SC_ADD** is a directory.

ELOOP Too many symbolic links were encountered in translating the pathname provided to **SC_ADD** or **SC_REMOVE**.

ENAMETOOLONG The length of a component of the path specified for **SC_ADD** or **SC_REMOVE** exceeds {**NAME_MAX**} characters or the length of the path exceeds {**PATH_MAX**} characters and {**_POSIX_NO_TRUNC**} is in effect.

ENOENT The pathname specified for **SC_ADD** or **SC_REMOVE** does not exist.

ENOMEM An insufficient number of struct swapent structures were provided to **SC_LIST**,

or there were insufficient system storage resources available during an **SC_ADD** or **SC_REMOVE**, or the system would not have enough swap space after an **SC_REMOVE**.

ENOSYS The pathname specified for **SC_ADD** or **SC_REMOVE** is not a file or block special device.

ENOTDIR Pathname provided to **SC_ADD** or **SC_REMOVE** contained a component in the path prefix that was not a directory.

EPERM The effective user of the calling process is not super-user.

EROFS The pathname specified for **SC_ADD** is a read only file system.

Additionally, the ***swapctl***() function will fail for 32-bit interfaces if:

EOVERFLOW The amount of swap space configured on the machine is too large to be represented by a 32-bit quantity.

ttyname
ttyname_r

NAME

ttyname, ttyname_r - find name of a terminal

SYNOPSIS

#include <stdlib.h>

char *ttyname (int fildes);

char *ttyname_r (int fildes, char *buf, int len);

DESCRIPTION

ttyname() returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor *fildes*.

ttyname_r() has the equivalent functionality to ***ttyname***() except that a buffer buf with length len must be supplied by the caller to store the result. buf must be at least **POSIX_PATH_MAX** in size (defined in <*limits.h*>).

RETURN VALUES

ttyname() and ***ttyname_r***() return a NULL pointer if fildes does not describe a terminal device in directory /dev.

ERRORS

ttyname_r() will fail if the following is true:

ERANGE The size of the buffer is smaller than the result to be returned.

NOTES

ttyname_r is designated as **EXPERMIMENTAL** since it has an interface which is different from the one in **POSIX 1003.1c.** ttyname_r interface in **POSIX** is as following:

```
int     ttyname_r (   int      fildes,
                      char     *name,
                      size_t   namesize);
```

sync_instruction_memory

NAME

***sync_instruction_memory* -** make modified instructions executable

SYNOPSIS

void sync_instruction_memory (caddr_t addr, size_t len);

DESCRIPTION

sync_instruction_memory() performs whatever steps are required to make instructions modified by a program executable.

Some processor architectures, including some SPARC processors, have separate and independent instruction and data caches which are not kept consistent by hardware. For example, if the instruction cache contains an instruction from some address and the program then stores a new instruction at that address, the new instruction may not be immediately visible to the instruction fetch mechanism. Software must explicitly invalidate the instruction cache entries for new or changed mappings of pages that might contain executable instructions. ***sync_instruction_memory***() performs this function, and/or any other functions needed to make modified instructions between *addr* and *addr+len* visible. A program should call ***sync_instruction_memory***() after modifying instructions and before executing them.

On processors with unified caches (one cache for both instructions and data) and pipelines which are flushed by a branch instruction, the function may do nothing and just return. The changes are immediately visible to the thread calling ***sync_instruction_memory***() when the call returns, even if the thread should migrate to another processor during or after the call. The changes become visible to other threads in the same manner that stores do; that is, they eventually become visible, but the latency is implementation-dependent.

RETURN VALUES

None

ERRORS

The result of executing ***sync_instruction_memory***() are unpredictable if *addr* through *addr+len-1* are not valid for the address space of the program making the call.

SPARC COMPLIANCE DEFINITION 2.4.1 IS

libc 64 psABI

__align_cpy_1, __align_cpy_2, __align_cpy_3 __align_cpy_8, __align_cpy_16

NAME

__align_cpy_* **-** copies n bytes

SYNOPSIS

void *__align_cpy_1(void *s1, const void *s2, size_t n)
void *__align_cpy_2(void *s1, const void *s2, size_t n)
void *__align_cpy_4(void *s1, const void *s2, size_t n)
void *__align_cpy_8(void *s1, const void *s2, size_t n)
void *__align_cpy_16(void *s1, const void *s2, size_t n)

DESCRIPTION

The ***__align_cpy_1*** function copies n bytes from memory area *s2* to *s1*. It returns s1. If the memory areas are partially overlapped, or n is zero, the result of calling this function is undefined.

The ***__align_cpy_2*** function copies n bytes from memory area *s2* to *s1*. It returns s1. If the lower-order bit of any of s1, s2, or n is non-zero, or the memory areas are partially overlapped, or n is zero, the result of calling this function is undefined.

The ***__align_cpy_4*** function copies n bytes from memory area *s2* to *s1*. It returns s1. If the lower-order two bits of any of s1, s2, or n are non-zero, or the memory areas are partially overlapped, or n is zero, the result of calling this function is undefined.

This ***__align_cpy_8*** function copies n bytes from memory area *s2* to *s1*. It returns s1. If the lower-order three bits of any of s1, s2, or n are non-zero, or the memory areas are partially overlapped, or n is zero, the result of calling this function is undefined.

The ***__align_cpy_16*** function copies n bytes from memory area *s2* to *s1*. It returns s1. If the lower-order four bits of any of s1, s2, or n are non-zero, or the memory areas are partially overlapped, or n is zero, the result of calling this function is undefined.

SEE ALSO

none

__sparc_utrap_install

NAME:

__sparc_utrap_install - establish new trap handler

SYNOPSIS

```
int     __sparc_utrap_install(
                utrap_entry_t      type,
                utrap_handler_t    new_precise,
                utrap_handler_t    new_deferred,
                utrap_handler_t    *old_precise,
                utrap_handler_t    *old_deferred
);
```

DESCRIPTION:

This function establishes new values for the user trap handlers for the specified trap type and return the existing trap handler values in a single atomic operation. A new handler address of NULL means no user handler of that type will be installed. A new handler address of **UTH_NOCHANGE** means that the user handler for that type should not be changed. An old handler pointer of NULL means that the user is not interested in the old handler address.

SEE ALSO

none

_Qp_add
_Qp_cmp, _Qp_cmpe
_Qp_div, _Qp_dtoq
_Qp_feq, _Qp_fge, _Qp_fgt, _Qp_fle, _Qp_flt, _Qp_fne
_Qp_itoq, _Qp_mul, _Qp_neg, _Qp_qtod,_Qp_qtoi
_Qp_qtos, _Qp_qtoui, _Qp_qtoux, _Qp_qtox, _Qp_sqr
_Qp_stoq _Qp_sub
_Qp_uitoq, _Qp_uxtoq, _Qp_xtoq
__dtoul, __ftoul

void _Qp_add(long double *c, const long double *a, const long double *b)

This function sets *c = *a + *b computed to quadruple precision. (Exceptions mimic ***faddq***.)

int _Qp_cmp(const long double *a, const long double *b)

This function compares the quadruple precision values *a and *b and returns an integer value that indicates their relative ordering as shown in the table below. (Exceptions mimic ***fcmpq***).

relation	Value
a equal to b	0
a less than b	1
a greater than b	2
a unordered with respect to b	3

int _Qp_cmpe(const long double *a, const long double *b)

This function compares the quadruple precision values *a and *b and returns an integer value that indicates their relative ordering according to the same convention as ***_Qp_cmp***. (Exceptions mimic ***fcmpeq***.)

void _Qp_div(long double *c, const long double *a, const long double *b)

This function sets *c = *a / *b computed to quadruple precision. (Exceptions mimic ***fdivq***.)

void _Qp_dtoq(long double *c, const double a)

Note: The "a" is passed in an FP register.

This function converts the double precision value of a to quadruple precision and sets *c to the quadruple precision value. (Exceptions mimic ***fdtoq***.)

int _Qp_feq(const long double *a, const long double *b)

This function compares the quadruple precision values *a and *b and returns a nonzero value if they are equal, zero otherwise. (Exceptions mimic ***fcmpq***.)

int _Qp_fge(const long double *a, const long double *b)

This function compares the quadruple precision values *a and *b and returns a nonzero value if *a is greater than or equal to *b, zero otherwise. (Exceptions mimic ***fcmpeq***.)

int _Qp_fgt(const long double *a, const long double *b)

This function compares the quadruple precision values *a and *b and returns a nonzero value if *a is greater than *b, zero otherwise. (Exceptions mimic ***fcmpeq***.)

int _Qp_fle(const long double *a, const long double *b)

This function compares the quadruple precision values *a and *b and returns a nonzero value if *a is less than or equal to *b, zero otherwise. (Exceptions mimic ***fcmpeq***.)

int _Qp_flt(const long double *a, const long double *b)

This function compares the quadruple precision values *a and *b and returns a nonzero value if *a is less than *b, zero otherwise. (Exceptions mimic ***fcmpeq***.)

int _Qp_fne(const long double *a, const long double *b)

This function compares the quadruple precision values *a and *b and returns a nonzero value if they are unordered or not equal, zero otherwise. (Exceptions mimic ***fcmpq***.)

void _Qp_itoq(long double *c, const int a)

Note: The second argument is passed in an integer register.

This function converts the integer value of a to quadruple precision and sets *c to the quadruple precision value.

void _Qp_mul(long double *c, const long double *a, const long double *b)

This function sets *c = *a * *b computed to quadruple precision. (Exceptions mimic ***fmulq***.)

void _Qp_neg(long double *c, const long double *a)

This function sets *c = -*a without raising any exceptions.

double _Qp_qtod(const long double *a)

Note: The result of this function is returned in an FP register.

This function converts the quadruple precision value of *a to double precision and returns the double precision value. (Exceptions mimic ***fqtod***.)

int _Qp_qtoi(const long double *a)

Note: The result of this function is returned in an integer register.

This function converts the quadruple precision value of *a to a signed integer by truncating any fractional part and returns the signed integer value. (Exceptions mimic ***fqtoi***.)

float _Qp_qtos(const long double *a)

Note: The result of this function is returned in an FP register.

This function converts the quadruple precision value of *a to single precision and returns the single precision value. (Exceptions mimic ***fqtos***.)

unsigned int _Qp_qtoui(const long double *a)

Note: The result of this function is returned in an integer register.

This function converts the quadruple precision value of *a to an unsigned integer by truncating any fractional part and returns the unsigned integer value. ***_Qp_qtoui*** raises exceptions as follows: If -(2^31 + 1) < *a < 2^32, then the operation is successful. If *a is not a whole number, the inexact exception is raised. Note that negative values of *a are first converted to a signed integer and then cast to an unsigned integer. Otherwise, the value returned by ***_Qp_qtoui*** is unspecified, and the invalid exception is raised. (When any exceptions are raised, the behavior mimics ***fqtoi***.)

unsigned long _Qp_qtoux(const long double *a)

Note: The result of this function is returned in an integer register.

This function converts the quadruple precision value of *a to an unsigned extended word by truncating any fractional part and returns the unsigned extended word value. ***_Qp_qtoux*** raises exceptions as follows: If -(2^63 + 1) < *a < 2^64, then the operation is successful. If *a is not a whole number, the inexact exception is raised. Note that negative values of *a are first converted to a signed extended word and then cast to an unsigned extended word. Otherwise, the value returned by ***_Qp_qtoui*** is unspecified, and the invalid exception is raised. When any exceptions are raised, the behavior mimics ***fqtox***.)

long _Qp_qtox(const long double *a)

Note: The result of this function is returned in an integer register.

This function converts the quadruple precision value of *a to a signed extended word by truncating any fractional part and returns the signed extended word value. (Exceptions mimic ***fqtox***.)

void _Qp_sqrt(long double *c, const long double *a)

This function sets *c to the square root of *a computed to quadruple precision. (Exceptions mimic ***fsqrtq***.)

void _Qp_stoq(long double *c, const float a)

Note: "a" is not passed with a pointer, but in an FP register.

This function converts the single precision value of a to quadruple precision and sets *c to the quadruple precision value. (Exceptions mimic ***fstoq***.)

void _Qp_sub(long double *c, const long double *a, const long double *b)

This function sets *c = *a - *b computed to quadruple precision. (Exceptions mimic ***fsubq***.)

void _Qp_uitoq(long double *c, const unsigned int a)

Note: the second argument “a” is passed in an integer register.

This function converts the unsigned word value of a to quadruple precision and sets *c to the quadruple precision value.

void _Qp_uxtoq(long double *c, const unsigned long a)

Note: the second argument “a” is passed in an integer register.

This function converts the unsigned extended word value of a to quadruple precision and sets *c to the quadruple precision value.

void _Qp_xtoq(long double *c, const long a)

Note: the second argument “a” is passed in an integer register.

This function converts the extended word value of a to quadruple precision and sets *c to the quadruple precision value.

unsigned long __dtoul(const double a)

This function converts the double precision value of a to an unsigned extended word by truncating any fractional part and returns the unsigned extended word value. ***__dtoul*** raises exceptions as follows: If -(2^63+1) < a < 2^64, then the operation is successful. If a is not a whole number, the inexact exception is raised. Note that negative values of a are first converted to a signed extended word and then cast to an unsigned extended word. Otherwise, the value returned by ***__dtoul*** is unspecified, and the invalid exception is raised. (When any exceptions are raised, the behavior mimics ***fdtoi***.)

unsigned long __ftoul(const float a)

This function converts the single precision value of a to an unsigned extended word by truncating any fractional part and returns the unsigned extended word value. ***__ftoul*** raises exceptions as follows: If -(2^63+1) < a < 2^64, then the operation is successful. If a is not a whole number, the inexact exception is raised. Note that negative values of a are first converted to a signed extended word and then cast to an unsigned extended word. Otherwise, the value returned by ***__ftoul*** is unspecified, and the invalid exception is raised. (When any exceptions are raised, the behavior mimics ***fstoi***.)

NOTE:

The following restrictions apply to all of the functions listed above:

When a function computes a floating point result, that result is rounded in accordance with the setting of the rounding control (RM) field of the **FSR** register. If any floating point exceptions occur, the resulting behavior is identical to that which would be observed as a result of executing a floating point instruction with the same operands, to the extent that such behavior is defined. The particular instruction is listed in the description of each function that can incur floating point exceptions.

SPARC COMPLIANCE DEFINITION 2.4.1 IS

libdl

Introduction

The following terms are used in this specification:

For a program to reference a symbol means for the program to use the storage value associated with that symbol. To reference a data symbol means (a) to retrieve the value stored in the location associated with that symbol, or (b) to store a value into the location associated with that symbol. To reference a function symbol means to (a) use the value directly by calling that function, or (b) to obtain its value via a call to ***dlsym***, presumably in order to call the function later.

For a program to contain a reference to a symbol means that the program has been constructed in such a way that it will reference a symbol that is not defined within it. In the C language, this is done by declaring a data or function to have the extern attribute. The reference that the program contains is an indication to the linker and loader of what the name of the symbol is, and the fact that it will be found in some other program. For details on how this is implemented in a **SPARC** executable file, see the *System V Application Binary Interface* and the *System V Application Binary Interface*, ***SPARC*** *Processor Supplement*.

Two kinds of objects are mentioned in these specifications. A data object is the storage location associated with a symbol in an application program. A shared object is (a) a file on disk that was created by linking a program as a shared object, or (b) such a file that has been loaded into memory and prepared for execution. When the word “object” is used without qualification in this specification, it means shared object, and usually the shared object in memory.

For an object to reference another object means that the first object has been link-edited with the second object in such a way as to create **DT_NEEDED** entries that cause the second object to be loaded automatically with the first object. (See Chapter 5 of the **SCD** 2.4 document.)

dladdr

NAME

dladdr - translate address to symbolic information.

SYNOPSIS

#include <dlfcn.h>

int dladdr(void * address, dl_info * dlip);

DESCRIPTION

dladdr() is one of a family of routines that give the user direct access to the dynamic linking facilities. These routines are available to dynamically linked processes ONLY. ***Dladdr***() determines if the specified address is located within one of the mapped objects that make up the current applications address space. An address is deemed to fall within a mapped object when it is between the base address, and the *_end* address of that object. If a mapped object fits this criteria, the symbol table made available to the runtime linker is searched to locate the nearest symbol to the specified address. The nearest symbol is one that has a value less than or equal to the required address. The *dl_info* structure must be pre-allocated by the user. The structure members are filled in by ***dladdr***() based on the specified address. The *dl_info* includes the following members:

*const char *dli_fname;*

*void *dli_fbase;*

*const char *dli_sname;*

*void * dli_saddr;*

Descriptions of these members appear below.

dli_fname	contains a pointer to the filename of the containing object.
dli_fbase	base contains the base address of the containing object.
dli_sname	contains a pointer to the nearest symbol name to the specified address. This symbol either has the same address, or is the nearest symbol with a lower address.
dli_saddr	contains the actual address of the above symbol.

RETURN VALUES

If the specified address cannot be matched to a mapped object, a 0 is returned. Otherwise a nonzero return is made and the associated *Dl_info* elements are filled.

SEE ALSO

led, dlclose(), dlerror(), dlopen(), dlsym()

NOTES

The *Dl_info* pointer elements point to addresses within the mapped objects, these may become invalid if objects are removed prior to these elements being used (see ***dlclose***()).

If no symbol is found to describe the specified address, both the *dli_sname* and *dli_saddr* members are set to 0.

dlclose

NAME

dlclose - close a shared object

SYNOPSIS

#include <dlfcn.h>

int dlclose (void *handle);

DESCRIPTION

The function ***dlclose*** disassociates from the current process a shared object previously opened by dlopen.

handle is a value that was returned from a previous call to ***dlopen***. It designates the shared object whose pathname was specified in that previous call to dlopen.

Once an object has been dissasociated from the process using ***dlclose***, its symbols and those of any objects that were loaded automatically as a result of opening the object designated by handle are no longer available to ***dlsym*** via handle.

In order for ***dlclose*** to dissasociate an object from a process, there must have been exactly one ***dlclose*** executed for each ***dlopen*** that was executed. Thus if a dlopen was executed once for a pathname, ***dlclose*** would have to be executed once with the handle that was returned for pathname. If a dlopen were executed twice for the same pathname, the disassociation would occur only after the second ***dlclose***.

A successful invocation of ***dlclose*** does not guarantee that the objects associated with *handle* will actually be removed from the address space of the process, even if the object has been disassociated from the process and its symbols are no longer available through handle. Objects loaded by one invocation of ***dlopen*** may also be loaded by another invocation of ***dlopen***. The same object may also be opened multiple times. An object may be removed from the address space by the system only after all references to that object through an explicit ***dlopen*** invocation have been closed and all other objects that reference that object have also been closed. Even then, however, it is unspecified in this standard whether the object will actually be removed from the address space.

When the system removes an object from the process address space, the object's termination function is executed. The termination function for each object is specified by the **DT_FINI** entry in that object's dynamic section. The exact timing of the execution of termination function relative to the timing of the ***dlclose*** that release the object is unspecified in this standard.

An **SCD**-conforming application will not have any processing dependencies upon the system's removal or non-removal of an object from the process address space following dlclose.

DIAGNOSTICS

If the referenced object was successfully closed, ***dlclose*** returns 0. If the object could not be closed, or if *handle* does not refer to an open object, dlclose returns a non-0 value. More detailed diagnostic information will be available through ***dlerror***.

NOTES

The following notes are a consequence of that fact that this standard does not specify whether an object ever is actually removed from a process address space:

Once a program has executed a sequence of ***dlclose*** operations that would permit the system to remove an object from the process address space, the result of the program's executing any reference to symbols defined in that object are unspecified in this standard.

Once a program has executed a sequence of ***dlclose*** operations that would permit the system to remove an object from the process address space, if the program executes another ***dlopen*** for that object, it is unspecified in this standard whether the object is actually loaded again and whether the object's data will be in its initial state.

dlerror

NAME

dlerror - get diagnostic information

SYNOPSIS

#include <dlfcn.h>

char *dlerror (void);

DESCRIPTION

The function ***dlerror*** returns a null-terminated character string (with no trailing newline) that describes the last error that occurred during dynamic linking processing. If no dynamic linking errors have occurred since the last invocation of dlerror, dlerror returns **NULL**. Thus, invoking dlerror a second time, immediately following a prior invocation, will result in **NULL** being returned.

NOTES

The messages returned by ***dlerror*** may reside in a static buffer that is overwritten on each call to ***dlerror***. Application code should not write to this buffer. Programs wishing to preserve an error message should make their own copies of that message.

dlopen

NAME

dlopen - open a shared object

SYNOPSIS

#include <dlfcn.h>

void *dlopen (char *pathname, int mode);

DESCRIPTION

The function ***dlopen*** is one of a family of routines that give the user direct access to the dynamic linking facilities.

The function ***dlopen*** makes a shared object available to a running process. ***dlopen*** returns to the process a *handle* the process must use to identify the object on subsequent calls to ***dlsym*** and ***dlclose***. This value must not be interpreted in any way by the process. (See Rationale)

pathname is the *path* name of the object to be opened; it may be an absolute *path* or relative to the current directory. If the value of *pathname* is 0, ***dlopen*** will make the symbols contained in the original a.out, and all of the objects that were loaded at program startup with the *a.out*, available through ***dlsym***.

If the value of pathname is not zero, and no file specified by *pathname* has already been loaded into the address space, the file specified by pathname will be loaded. If the file specified by *pathname* contains **DT_NEEDED** entries for other shared objects, those objects will automatically be loaded by ***dlopen***.

Objects whose names resolve to the same absolute or relative *path* name may be opened any number of times either using dlopen or automatically as a result of executing ***dlopen*** for an object that uses them. However, the object referenced is loaded only once into the address space of the current process. This means that the object only takes up space once; there is only one copy of its static data; and the static data are initialized only once, when the initial load takes place.

When a shared object is brought into the address space of a process, it may contain references to symbols whose addresses are not known until the object is loaded. These references must be relocated before the symbols can be accessed. The *mode* parameter governs when these relocations take place and may have the following values:

RTLD_LAZY Under this *mode*, only references to data symbols are relocated when the object is loaded. References to functions are not relocated until a given function is referenced for the first time by the executing program. This *mode* should result in better performance, since a process may not reference all of the functions in any given shared object.

RTLD_NOW Under this *mode*, all necessary relocations are performed when the object is first loaded. This may result in some wasted effort, if relocations are performed for functions that are never referenced, but is useful for applications that need to know as soon as an object is loaded that all symbols referenced during execution will be available.

The mode parameter only takes effect when an object is initially loaded. If **RTLD_LAZY** is specified in the first dlopen for an object, and **RTLD_NOW** is specified for the second ***dlopen*** of the same object, the second ***dlopen*** will not cause any relocations to be performed.

The *mode* parameter is required, and always overrides the value of the **LD_BIND_NOW** environment variable.

When the system loads an object for the first time, the object's initialization function is executed. The initialization function for each object is specified by the **DT_INIT** entry in that object's dynamic section. If multiple objects are loaded as a result of ***dlopen***, the order initialization functions are called is unspecified.

Objects loaded by a single invocation of ***dlopen*** may import symbols from one another or from any object loaded automatically with a.out during program startup, but objects loaded by one ***dlopen*** invocation may not directly reference symbols from objects loaded by a different ***dlopen*** invocation. Those symbols may, however, be referenced indirectly using ***dlsym***.

RATIONALE

The functions ***dlopen*** and dlclose may not work in a manner consistent with the way the functions open and close work. For example, if the same file is opened twice, the open function will return unique file descriptors for each open operation. Using ***dlopen*** to open the same file multiple times may return the same file handle every time. The result is that if the first file handle for a ***dlopen*** call is used more than once as a parameter to ***dlclose***, there may be unexpected side effects.

DIAGNOSTICS

If the file specified by *pathname* cannot be found, cannot be opened for reading, is not a shared object, or if an error occurs during the process of loading the file specified by *pathname* or relocating its symbolic references, dlopen will return **NULL**. More detailed diagnostic information will be available through ***dlerror***.

NOTES

The same object referenced by different path names may be loaded multiple times. For example, given the object */usr/home/me/mylibs/mylib.so*, and assuming the current working directory is */usr/home/me/workdir*,

...

*void *handle1;*

*void *handle2;*

handle1 = dlopen ("/mylibs/mylib.so", RTLD_LAZY);

handle2 = dlopen ("/usr/home/me/mylibs/mylib.so", RTLD_LAZY);

...

results in mylibs.so being loaded twice for the current process. On the other hand, given the same object and current working directory, if **LD_LIBRARY_PATH**=/usr/home/me/mylibs, then

...

*void *handle1;*

*void *handle2;*

*handle1 = dlopen ("mylib.so", **RTLD_LAZY**);*

*handle2 = dlopen ("/usr/home/me/mylibs/mylib.so", **RTLD_LAZY**);*

...

results in *mylibs.so* being loaded only once.

Users who wish to gain access to the symbol table of the *a.out* itself using ***dlopen***(0, *mode*) should be aware that some symbols defined in the a.out may not be available to the dynamic linker. The symbol table created by ld for use by the dynamic linker might contain only a subset of the symbols originally defined in the a.out: specifically, those referenced by the shared objects with which the *a.out* is linked.

dlsym

NAME

dlsym - get the address of a symbol in a shared object

SYNOPSIS

#include <dlfcn.h>

void *dlsym (void *handle, char *name);

DESCRIPTION

The function ***dlsym*** allows a process to obtain the address of a symbol defined within a shared object previously opened by ***dlopen***.

handle is a value returned by a call to ***dlopen***; the corresponding shared object must not have been disassociated from the executing process using dlclose. *name* is the symbol's name as a character string.

dlsym searches for the named symbol in the shared object designated by handle and in all shared objects loaded automatically as a result of loading the object referenced by *handle* [see ***dlopen***(3X)].

EXAMPLES

The following example shows how one can use ***dlopen*** and ***dlsym*** to access either function or data objects. For simplicity, error checking has been omitted.

*void *handle;*

*int i, *iptr;*

*int (*fptr) (int);*

/ open the needed object */*

handle = ***dlopen*** *("/usr/mydir/libx.so",* ***RTLD_LAZY****);*

/ find address of function and data objects */*

fptr = (int (*)(int)) ***dlsym*** *(handle, "some_function");*

*iptr = (int *) dlsym (handle, "int_object");*

/ invoke function, passing value of integer as a parameter */*

*i = (*fptr) (*iptr);*

DIAGNOSTICS

If *handle* does not refer to a valid object opened by ***dlopen***, or if the named symbol cannot be found within any of the objects associated with *handle*, ***dlsym*** will return **NULL**. More detailed diagnostic information will be available through ***dlerror***.

SPARC COMPLIANCE DEFINITION 2.4.1 IS

libelf

elf32_size

NAME

elf32_fsize

SYNOPSIS

#include <libelf.h>

size_t elf32_fsize(Elf_Type type, size_t count, unsigned ver);

DESCRIPTION

elf32_fsize gives the size in bytes of the 32-bit file representation of count data objects with the given type. The library uses version ver to calculate the size. Constant values are available for the sizes of fundamental types.

ELF_T_ADDR	File Size	Memory Size
ELF_T_ADDR	**ELF32_FSZ**	sizeof(*Elf32_Addr*)
ELF_T_BYTE	**1**	sizeof(unsigned char)
ELF_T_HALF	**ELF32_FSZ_HALF**	sizeof(*Elf32_Half*)
ELF_T_OFF	**ELF32_FSZ_OFF**	sizeof(*Elf32_Off*)
ELF_T_SWORD	**ELF32_FSZ_SWORD**	sizeof(*Elf32_Sword*)
ELF_T_WORD	**ELF32_FSZ_WORD**	sizeof(*ELF32_Word*)

elf32_fsize returns zero if the value of type or ver is unknown.

elf32_getehdr
elf32_newehdr

NAME

elf32_getehdr, *elf32_newehdr*

SYNOPSIS

#include <libelf.h>

Elf32_Ehdr *elf32_getehdr(Elf *elf);

Elf32_Ehdr *elf32_newehdr(Elf *elf);

DESCRIPTION

For a 32-bit class file, ***elf32_getehdr*** returns a pointer to an **ELF** header, if one is available for the ELF descriptor elf. If no header exists for the descriptor, ***elf32_newehdr*** allocates a ``clean'' one, but it otherwise behaves the same as ***elf32_getehdr***. It does not allocate a new header if one exists already. If no header exists (for ***elf_getehdr***), one cannot be created (for ***elf_newehdr***), a system error occurs, the file is not a 32-bit class file, or elf is null, both functions return a null pointer.

The header includes the following members.

unsigned char	*e_ident[***EI_NIDENT***];*
Elf32_Half	*e_type;*
Elf32_Half	*e_machine;*
Elf32_Word	*e_version;*
Elf32_Addr	*e_entry;*
Elf32_Off	*e_phoff;*
Elf32_Off	*e_shoff;*
Elf32_Word	*e_flags;*
Elf32_Half	*e_ehsize;*
Elf32_Half	*e_phentsize;*
Elf32_Half	*e_phnum;*
Elf32_Half	*e_shentsize;*
Elf32_Half	*e_shnum;*
Elf32_Half	*e_shstrndx;*

elf32_newehdr automatically sets the **ELF_F_DIRTY** bit. A program may use ***elf_getident*** to inspect the identification bytes from a file.

elf32_getphdr
elf32_newphdr

NAME

elf32_getphdr, elf32_newphdr

SYNOPSIS

#include <libelf.h>

Elf32_Phdr *elf32_getphdr(Elf *elf);

Elf32_Phdr *elf32_newphdr(Elf *elf, size_t count);

DESCRIPTION

For a 32-bit class file, ***elf32_getphdr*** returns a pointer to the program execution header table, if one is available for the ELF descriptor elf.

elf32_newphdr allocates a new table with count entries, regardless of whether one existed previously, and sets the **ELF_F_DIRTY** bit for the table. Specifying a zero count deletes an existing table. Note this behavior differs from that of ***elf32_newehdr***, allowing a program to replace or delete the program header table, changing its size if necessary.

If no program header table exists, the file is not a 32-bit class file, an error occurs, or elf is null, both functions return a null pointer. Additionally, *elf32_newphdr* returns a null pointer if count is zero.

The table is an array of *Elf32_Phdr* structures, each of which includes the following members.

```
Elf32_Word    p_type;
Elf32_Off     p_offset;
Elf32_Addr    p_vaddr;
Elf32_Addr    p_paddr;
Elf32_Word    p_filesz;
Elf32_Word    p_memsz;
Elf32_Word    p_flags;
Elf32_Word    p_align;
```

The ELF header's *e_phnum* member tells how many entries the program header table has. A program may inspect this value to determine the size of an existing table; ***elf32_newphdr*** automatically sets the member's value to count. If the program is building a new file, it is responsible for creating the file's ELF header before creating the program header table.

elf32_getshdr

NAME

elf32_getshdr

SYNOPSIS

#include <libelf.h>

Elf32_Shdr *elf32_getshdr(Elf_Scn *scn);

DESCRIPTION

For a 32-bit class file, ***elf32_getshdr*** returns a pointer to a section header for the section descriptor scn. Otherwise, the file is not a 32-bit class file, scn was null, or an error occurred; ***elf32_getshdr*** then returns NULL.

The header includes the following members.

Elf32_Word	*sh_name;*
Elf32_Word	*sh_type;*
Elf32_Word	*sh_flags;*
Elf32_Addr	*sh_addr;*
Elf32_Off	*sh_offset;*
Elf32_Word	*sh_size;*
Elf32_Word	*sh_link;*
Elf32_Word	*sh_info;*
Elf32_Word	*sh_addralign;*
Elf32_Word	*sh_entsize;*

If the program is building a new file, it is responsible for creating the file's ELF header before creating sections.

elf32_xlatetof
elf32_xlatetom

NAME

elf32_xlatetof, elf32_xlatetom

SYNOPSIS

Elf_Data *elf32_xlatetof(Elf_Data *dst, const Elf_Data *src, unsigned ecode);

Elf_Data *elf32_xlatetom(Elf_Data *dst, const Elf_Data *src, unsigned en code);

DESCRIPTION

elf32_xlatetom translates various data structures from their 32-bit class file representations to their memory representations; ***elf32_xlatetof*** provides the inverse. This conversion is particularly important for cross development environments. *src* is a pointer to the source buffer that holds the original data; dst is a pointer to a destination buffer that will hold the translated copy. encode gives the byte encoding in which the file objects are (to be) represented and must have one of the encoding values defined for the ELF header's *e_ident[***EI_DATA***]* entry. If the data can be translated, the functions return dst. Otherwise, they return null because an error occurred, such as incompatible types, destination buffer overflow, and so forth. ***elf_getdata*** describes the *Elf_Data* descriptor, which the translation routines use as follows.

d_buf Both the source and destination must have valid buffer pointers.

d_type This member's value specifies the type of the data to which

d_buf points and the type of data to be created in the destination. The program supplies a *d_type* value in the source; the library sets the destination's *d_type* to the same value. These values are summarized below.

d_size This member holds the total size, in bytes, of the memory occupied by the source data and the size allocated for the destination data. If the destination buffer is not large enough, the routines do not change its original contents. The translation routines reset the destination's d_size member to the actual size required, after the translation occurs. The source and destination sizes may differ.

d_version This member holds version number of the objects (desired) in the buffer. The source and destination versions are independent.

Translation routines allow the source and destination buffers to coincide. That is, *dst*->*d_buf* may equal *src*->*d_buf*. Other cases where the source and destination buffers overlap give undefined behavior.

ELF_Type	32-Bit Memory Type
ELF_T_ADDR	*Elf32_Addr*
ELF_T_BYTE	*unsigned char*
ELF_T_DYN	*Elf32_Dyn*
ELF_T_EHDR	*Elf32_Ehdr*
ELF_T_HALF	*Elf32_Half*

ELF_Type	32-Bit Memory Type
ELF_T_OFF	*Elf32_Off*
ELF_T_PHDR	*Elf32_Phdr*
ELF_T_REL	*Elf32_Rel*
ELF_T_RELA	*Elf32_Rela*
ELF_T_SHDR	*Elf32_Shdr*
ELF_T_SWORD	*Elf32_Sword*
ELF_T_SYM	*Elf32_Sym*
ELF_T_WORD	*Elf32_Word*

“Translating” buffers of type **ELF_T_BYTE** does not change the byte order.

elf_begin

NAME

elf_begin

SYNOPSIS

#include <libelf.h>

Elf *elf_begin(int fildes, Elf_Cmd cmd, Elf *ref);

DESCRIPTION

elf_begin, ***elf_next***, ***elf_rand***, and ***elf_end*** work together to process **ELF** object files, either individually or as members of archives. After obtaining an ELF descriptor from elf_begin, the program may read an existing file, update an existing file, or create a new file. fildes is an open file descriptor that ***elf_begin*** uses for reading or writing. The initial file offset [see ***lseek***()] is unconstrained, and the resulting file offset is undefined. *cmd* may have the following values.

ELF_C_NULL When a program sets *cmd* to this value, ***elf_begin*** returns a null pointer, without opening a new descriptor. *ref* is ignored for this command.

ELF_C_READ When a program wishes to examine the contents of an existing file, it should set cmd to this value. Depending on the value of ref, this command examines archive members or entire files. Three cases can occur. First, if ref is a null pointer, ***elf_begin*** allocates a new **ELF** descriptor and prepares to process the entire file. If the file being read is an archive, ***elf_begin*** also prepares the resulting descriptor to examine the initial archive member on the next call to elf_begin, as if the program had used ***elf_next*** or ***elf_rand*** to “move” to the initial member. Second, if *ref* is a non-null descriptor associated with an archive file, ***elf_begin*** lets a program obtain a separate ELF descriptor associated with an individual member. The program should have used ***elf_next*** or ***elf_rand*** to position *ref* appropriately (except for the initial member, which ***elf_begin*** prepares;). In this case, fildes should be the same file descriptor used for the parent archive. Finally, if ref is a non-null **ELF** descriptor that is not an archive, ***elf_begin*** increments the number of activations for the descriptor and returns ref, without allocating a new descriptor and without changing the descriptor's read/write permissions. To terminate the descriptor for ref, the program must call ***elf_end*** once for each activation. See ***elf_next*** and the examples below for more information.

ELF_C_RDWR This command duplicates the actions of **ELF_C_READ** and additionally allows the program to update the file image. That is, using **ELF_C_READ** gives a read-only view of the file, while **ELF_C_RDWR** lets the program read and write the file. **ELF_C_RDWR** is not valid for archive members. If ref is non-null, it must have been created with the **ELF_C_RDWR** command.

ELF_C_WRITE If the program wishes to ignore previous file contents, presumably to create a new file, it should set cmd to this value. *ref* is ignored for this command. ***elf_begin*** “works” on all files (including files with zero bytes), providing it can allocate memory for its internal structures and read any necessary information from the file. Programs reading object files thus may call ***elf_kind*** or ***elf_getehdr*** to determine the file type (only object files have an **ELF** header). If the file is an archive with no more members to process, or an error occurs, ***elf_begin*** returns a null pointer. Otherwise, the return value is a non-null **ELF** descriptor. Before the first call to ***elf_begin***, a program must call ***elf_version*** to coordinate versions.

elf_cntl

NAME

elf_cntl

SYNOPSIS

#include <libelf.h>

int elf_cntl(Elf *elf, Elf_Cmd cmd);

DESCRIPTION

elf_cntl instructs the library to modify its behavior with respect to an **ELF** descriptor, elf. As ***elf_begin*** describes, an **ELF** descriptor can have multiple activations, and multiple **ELF** descriptors may share a single file descriptor. Generally, ***elf_cntl*** commands apply to all activations of elf. Moreover, if the **ELF** descriptor is associated with an archive file, descriptors for members within the archive will also be affected as described below. Unless stated otherwise, operations on archive members do not affect the descriptor for the containing archive.

The *cmd* argument tells what actions to take and may have the following values.

ELF_C_FDDONE This value tells the library not to use the file descriptor associated with elf. A program should use this command when it has requested all the information it cares to use and wishes to avoid the overhead of reading the rest of the file. The memory for all completed operations remains valid, but later file operations, such as the initial ***elf_getdata*** for a section, will fail if the data is not in memory already.

ELF_C_FDREAD This command is similar to **ELF_C_FDDONE**, except it forces the library to read the rest of the file. A program should use this command when it must close the file descriptor but has not yet read everything it needs from the file. After ***elf_cntl*** completes the **ELF_C_FDREAD** command, future operations, such as ***elf_getdata***, will use the memory version of the file without needing to use the file descriptor.

RETURN VALUE

~~If ***elf_cntl*** succeeds, it returns zero. Otherwise elf was null or an error occurred, and the function returns -1.~~

If ***elf_cntl*** succeeds or if elf is null, it returns zero. Otherwise an error occurred and the funtion returns -1.

elf_end

NAME

elf_end

SYNOPSIS

#include <libelf.h>

***int elf_end(Elf *elf)**;*

DESCRIPTION

A program uses ***elf_end*** to terminate an **ELF** descriptor, elf, and to de-allocate data associated with the descriptor. Until the program terminates a descriptor, the data remain allocated. elf should be a value previously returned by ***elf_begin***; a null pointer is allowed as an argument, to simplify error handling. If the program wishes to write data associated with the **ELF** descriptor to the file, it must use ***elf_update*** before calling ***elf_end***. As ***elf_begin*** explains, a descriptor can have more than one activation. Calling ***elf_end*** removes one activation and returns the remaining activation count. The library does not terminate the descriptor until the activation count reaches zero. Consequently, a zero return value indicates the ELF descriptor is no longer valid.

elf_errmsg
elf_errno

NAME

elf_errmsg, elf_errno

SYNOPSIS

#include <libelf.h>

const char *elf_errmsg(int err);

int elf_errno(void);

DESCRIPTION

If an **ELF** library function fails, a program may call ***elf_errno*** to retrieve the library's internal error number. As a side effect, this function resets the internal error number to zero, which indicates no error.

elf_errmsg takes an error number, err, and returns a null-terminated error message (with no trailing newline) that describes the problem. A zero err retrieves a message for the most recent error. If no error has occurred, the return value is a null pointer (not a pointer to the null string). Using err of -1 also retrieves the most recent error, except it guarantees a non-null return value, even when no error has occurred. If no message is available for the given number, elf_errmsg returns a pointer to an appropriate message. This function does not have the side effect of clearing the internal error number.

elf_fill

NAME

elf_fill

SYNOPSIS

#include <libelf.h>

void elf_fill(int fill);

DESCRIPTION

Alignment constraints for **ELF** files sometimes require the presence of “holes.” For example, if the data for one section are required to begin on an eight-byte boundary, but the preceding section is too “short,” the library must fill the intervening bytes. These bytes are set to the fill character. The library uses zero bytes unless the application supplies a value. See ***elf_getdata*** for more information about these holes.

elf_flagdata
elf_flagehdr
elf_flagelf
elf_flagphdr
elf_flagscn
elf_flagshdr

NAME

elf_flagdata, elf_flagehdr, elf_flagelf, elf_flagphdr, elf_flagscn, elf_flagshdr

SYNOPSIS

#include <libelf.h>

*unsigned elf_flagdata(Elf_Data *data, Elf_Cmd cmd, unsigned flags);*

*unsigned elf_flagehdr(Elf *elf, Elf_Cmd cmd, unsigned flags);*

*unsigned elf_flagelf(Elf *elf, Elf_Cmd cmd, unsigned flags);*

*unsigned elf_flagphdr(Elf *elf, Elf_Cmd cmd, unsigned flags);*

*unsigned elf_flagscn(Elf_Scn *scn, Elf_Cmd cmd, unsigned flags);*

*unsigned elf_flagshdr(Elf_Scn *scn, Elf_Cmd cmd, unsigned flags);*

DESCRIPTION

These functions manipulate the flags associated with various structures of an **ELF** file. Given an **ELF** descriptor *elf*, a data descriptor *data*, or a section descriptor *scn*, the functions may set or clear the associated status bits, returning the updated bits. A null descriptor is allowed, to simplify error handling; all functions return zero for this degenerate case.

cmd may have the following values.

ELF_C_CLR The functions clear the bits that are asserted in flags. Only the nonzero bits in flags are cleared; zero bits do not change the status of the descriptor.

ELF_C_SET The functions set the bits that are asserted in flags. Only the nonzero bits in flags are set; zero bits do not change the status of the descriptor.

Descriptions of the defined flags bits appear below.

ELF_F_DIRTY When the program intends to write an **ELF** file, this flag asserts the associated information needs to be written to the file. Thus, for example, a program that wished to update the ELF header of an existing file would call ***elf_flagehdr*** with this bit set in flags and cmd equal to **ELF_C_SET**. A later call to ***elf_update*** would write the marked header to the file.

ELF_F_LAYOUT Normally, the library decides how to arrange an output file. That is, it automatically decides where to place sections, how to align them in the file, etc. If this bit is set for an ELF descriptor, the program assumes responsibility for determining all file positions. This bit is meaningful only for *elf_flagelf* and applies to the entire file associated with the descriptor.When a flag bit is set for an item, it affects all the sub-items as well. Thus, for example, if the program sets the **ELF_F_DIRTY** bit with *elf_flagelf*, the entire logical file is ``dirty."

elf_getarhdr

NAME

elf_getarhdr

SYNOPSIS

#include <libelf.h>

Elf_Arhdr *elf_getarhdr(Elf *elf);

DESCRIPTION

elf_getarhdr returns a pointer to an archive member header, if one is available for the **ELF** descriptor elf. Otherwise, no archive member header exists, an error occurred, or elf was null; ***elf_getarhdr*** then returns a null value. The header includes the following members.

*char**	*ar_name;*
time_t	*ar_date;*
long	*ar_uid;*
long	*ar_gid;*
long	*ar_gid;*
unsigned long	*ar_mode;*
off_t	*ar_size;*
*char**	*ar_rawname;*

An archive member name, available through *ar_name*, is a null-terminated string, with the *ar* format control characters removed. The *ar_rawname* member holds a null-terminated string that represents the original name bytes in the file, including the terminating slash and trailing blanks as specified in the archive format.

In addition to ``regular'' archive members, the archive format defines some special members. All special member names begin with a slash (/), distinguishing them from regular members (whose names may not contain a slash). These special members have the names (*ar_name*) defined below.

/ This is the archive symbol table. If present, it will be the first archive member. A program may access the archive symbol table through ***elf_getarsym***. The information in the symbol table is useful for random archive processing.

// This member, if present, holds a string table for long archive member names. An archive member's header contains a 16-byte area for the name, which may be exceeded in some file systems. The library automatically retrieves long member names from the string table, setting *ar_name* to the appropriate value.

Under some error conditions, a member's name might not be available. Although this causes the library to set *ar_name* to a null pointer, the *ar_rawname* member will be set as usual.

elf_getarsym

NAME

elf_getarsym

SYNOPSIS

#include <libelf.h>

Elf_Arsym *elf_getarsym(Elf *elf, size_t *ptr);

DESCRIPTION

elf_getarsym returns a pointer to the archive symbol table, if one is available for the **ELF** descriptor elf. Otherwise, the archive doesn't have a symbol table, an error occurred, or elf was null; ***elf_getarsym*** then returns a null value. The symbol table is an array of structures that include the following members.

*char**	*as_name;*
size_t	*as_off;*
unsigned long	*as_hash;*

These members have the following semantics:

as_name — A pointer to a null-terminated symbol name resides here.

as_off — This value is a byte offset from the beginning of the archive to the member's header. The archive member residing at the given offset defines the associated symbol. Values in *as_off* may be passed as arguments to ***elf_rand*** to access the desired archive member.

as_hash — This is a hash value for the name, as computed by ***elf_hash***. If *ptr* is non-null, the library stores the number of table entries in the location to which *ptr* points. This value is set to zero when the return value is null. The table's last entry, which is included in the count, has a null *as_name*, a zero value for as_off, and ~0UL for *as_hash*.

elf_getbase

NAME

elf_getbase

SYNOPSIS

#include <libelf.h>

*off32_t elf_getbase(Elf *elf);*

DESCRIPTION

elf_getbase returns the file offset of the first byte of the file or archive member associated with elf, if it is known or obtainable, and -1 otherwise. A null elf is allowed, to simplify error handling; the return value in this case is -1. The base offset of an archive member is the beginning of the member's information, not the beginning of the archive member header.

elf_getdata
elf_newdata
elf_rawdata

NAME

elf_getdata, elf_newdata, elf_rawdata

SYNOPSIS

#include <libelf.h>

*Elf_Data *elf_getdata(Elf_Scn *scn, Elf_Data *data);*

*Elf_Data *elf_newdata(Elf_Scn *scn);*

*Elf_Data *elf_rawdata(Elf_Scn *scn, Elf_Data *data);*

DESCRIPTION

These functions access and manipulate the data associated with a section descriptor, *scn*. When reading an existing file, a section will have a single data buffer associated with it. A program may build a new section in pieces, however, composing the new data from multiple data buffers. For this reason, ``the'' data for a section should be viewed as a list of buffers, each of which is available through a data descriptor. ***elf_getdata*** lets a program step through a section's data list. If the incoming data descriptor, data, is null, the function returns the first buffer associated with the section. Otherwise, data should be a data descriptor associated with scn, and the function gives the program access to the next data element for the section. If *scn* is null or an error occurs, ***elf_getdata*** returns a null pointer.

elf_getdata translates the data from file representations into memory representations and presents objects with memory data types to the program, based on the file's class. The working library version specifies what version of the memory structures the program wishes ***elf_getdata*** to present. ***elf_newdata*** creates a new data descriptor for a section, appending it to any data elements already associated with the section. As described below, the new data descriptor appears empty, indicating the element holds no data. For convenience, the descriptor's type (*d_type* below) is set to **ELF_T_BYTE**, and the version (*d_version* below) is set to the working version. The program is responsible for setting (or changing) the descriptor members as needed. This function implicitly sets the **ELF_F_DIRTY** bit for the section's data. If *scn* is null or an error occurs, ***elf_newdata*** returns a null pointer. ***elf_rawdata*** differs from ***elf_getdata*** by returning only uninterpreted bytes, regardless of the section type. This function typically should be used only to retrieve a section image from a file being read, and then only when a program must avoid the automatic data translation described below. Moreover, a program may not close or disable the file descriptor associated with elf before the initial raw operation, because ***elf_rawdata*** might read the data from the file to ensure it doesn't interfere with ***elf_getdata***. When ***elf_getdata*** provides the right translation, its use is recommended over ***elf_rawdata***. If *scn* is null or an error occurs, ***elf_rawdata*** returns a null pointer.

The *Elf_Data* structure includes the following members.

void	**d_buf;*
Elf_Type	*d_type;*
size_t	*d_size;*

```
off_t       d_off;
size_t      d_align;
unsigned    d_version;
```

These members are available for direct manipulation by the program. Descriptions appear below.

d_buf — A pointer to the data buffer resides here. A data element with no data has a null pointer.

d_type — This member's value specifies the type of the data to which *d_buf* points. A section's type determines how to interpret the section contents, as summarized below.

d_size — This member holds the total size, in bytes, of the memory occupied by the data. This may differ from the size as represented in the file. The size will be zero if no data exist. [See the discussion of **SHT_NOBITS** below for more information.]

d_off — This member gives the offset, within the section, at which the buffer resides. This offset is relative to the file's section, not the memory object's.

d_align — This member holds the buffer's required alignment, from the beginning of the section. That is, *d_off* will be a multiple of this member's value. For example, if this member's value is four, the beginning of the buffer will be four-byte aligned within the section. Moreover, the entire section will be aligned to the maximum of its constituents, thus ensuring appropriate alignment for a buffer within the section and within the file.

d_version — This member holds the version number of the objects in the buffer. When the library originally read the data from the object file, it used the working version to control the translation to memory objects.

DATA ALIGNMENT

As mentioned above, data buffers within a section have explicit alignment constraints. Consequently, adjacent buffers sometimes will not abut, causing “holes” within a section. Programs that create output files have two ways of dealing with these holes. First, the program can use ***elf_fill*** to tell the library how to set the intervening bytes. When the library must generate gaps in the file, it uses the fill byte to initialize the data there. The library's initial fill value is zero, and ***elf_fill*** lets the application change that. Second, the application can generate its own data buffers to occupy the gaps, filling the gaps with values appropriate for the section being created. A program might even use different fill values for different sections. For example, it could set text sections' bytes to no-operation instructions, while filling data section holes with zero. Using this technique, the library finds no holes to fill, because the application eliminated them.

SECTION AND MEMORY TYPES

elf_getdata interprets sections' data according to the section type, as noted in the section header available through ***elf_getshdr***. The following table shows the section types and how the library represents them with memory data types for the 32-bit file class. Other classes would have similar tables. By implication, the memory data types control translation by ***elf_xlate***.

Section Type	Elf_type	32-Bit Type
SHT_DYNAMIC	**ELF_T_DYN**	*Elf32_Dyn*
SHT_DYNSYM	**ELF_T_SYM**	*Elf32_Sym*
SHT_HASH	**ELF_T_WORD**	*Elf32_Word*
SHT_NOBITS	**ELF_T_BYTE**	*unsigned char*
SHT_NOTE	**ELF_T_BYTE**	*unsigned char*
SHT_Null	none	none
SHT_PROGBITS	**ELF_T_BYTE**	*unsigned char*
SHT_REL	**ELF_T_REL**	*Elf32_Rel*
SHT_RELA	**ELF_T_RELA**	*Elf32_Rela*
SHT_STRTAB	**ELF_T_BYTE**	*unsigned char*
SHT_SYMTAB	**ELF_T_SYM**	*Elf32_Sym*
other	**ELF_T_BYTE**	*unsigned char*

elf_rawdata creates a buffer with type **ELF_T_BYTE**.

As mentioned above, the program's working version controls what structures the library creates for the application. The library similarly interprets section types according to the versions. If a section type ``belongs'' to a version newer than the application's working version, the library does not translate the section data. Because the application cannot know the data format in this case, the library presents an untranslated buffer of type **ELF_T_BYTE**, just as it would for an unrecognized section type.

A section with a special type, **SHT_NOBITS**, occupies no space in an object file, even when the section header indicates a nonzero size. ***elf_getdata*** and ***elf_rawdata*** “work” on such a section, setting the data structure to have a null buffer pointer and the type indicated above. Although no data is present, the *d_size* value is set to the size from the section header. When a program is creating a new section of type **SHT_NOBITS**, it should use ***elf_newdata*** to add data buffers to the section. These “empty” data buffers should have the *d_size* members set to the desired size and the *d_buf* members set to null.

elf_getident

NAME

elf_getident

SYNOPSIS

#include <libelf.h>

char *elf_getident(Elf *elf, size_t *ptr);

DESCRIPTION

ELF provides a framework for various classes of files, where basic objects may have 32 bits, 64 bits, and so forth. To accommodate these differences, without forcing the larger sizes on smaller machines, the initial bytes in an ELF file hold identification information common to all file classes. Every ELF header's *e_ident* has **EI_NIDENT** bytes with the following interpretation.

e_ident Index	Value	Purpose
EI_MAG0 **EI_MAG1** **EI_MAG2** **EI_MAG3**	**ELFMAG0** **ELFMAG1** **ELFMAG2** **ELFMAG3**	File Identification
EI_CLASS	**ELFCLASSNONE** **ELFCLASS32** **ELFCLASS64**	File Class
EI_DATA	**ELFDATANONE** **ELFDATA2LSB** **ELFDATA2MSB**	Data encoding
EI_VERSION	**EV_CURRENT**	File Version
7-15	0	Unused, set to zero

Other kinds of files also may have identification data, though they would not conform to *e_ident*. ***elf_getident*** returns a pointer to the file's ``initial bytes.'' If the library recognizes the file, a conversion from the file image to the memory image may occur. In any case, the identification bytes are guaranteed not to have been modified, though the size of the unmodified area depends on the file type. If *ptr* is non-null, the library stores the number of identification bytes in the location to which ptr points. If no data is present, elf is null, or an error occurs, the return value is a null pointer, with zero optionally stored through *ptr*.

elf_hash

NAME

elf_hash

SYNOPSIS

#include <libelf.h>

unsigned long elf_hash(const char *name);

DESCRIPTION

elf_hash computes a hash value, given a null terminated string, name. The returned hash value, h, can be used as a bucket index, typically after computing *h mod x* to ensure appropriate bounds.

Hash tables may be built on one machine and used on another because ***elf_hash*** uses unsigned arithmetic to avoid possible differences in various machines' signed arithmetic. Although name is shown as *char** above, ***elf_hash*** treats it as *unsigned char** to avoid sign extension differences. Using *char** eliminates type conflicts with expressions such as ***elf_hash***("name").

ELF files' symbol hash tables are computed using this function. The hash value returned is guaranteed not to be the bit pattern of all ones (~0UL).

elf_kind

NAME

elf_kind

SYNOPSIS

#include <libelf.h>

*Elf_Kind elf_kind(Elf *elf);*

DESCRIPTION

This function returns a value identifying the kind of file associated with an ELF descriptor elf. Currently defined values appear below.

ELF_K_AR	The file is an archive. An ELF descriptor may also be associated with an archive member, not the archive itself, and then ***elf_kind*** identifies the member's type.
ELF_K_COFF	The file is a **COFF** object file. ***elf_begin*** describes the library's handling for **COFF** files.
ELF_K_ELF	The file is an ELF file. The program may use ***elf_getident*** to determine the class. Other functions, such as ***elf_getehdr***, are available to retrieve other file information.
ELF_K_NONE	This indicates a kind of file unknown to the library. Other values are reserved, to be assigned as needed to new kinds of files. elf should be a value previously returned by ***elf_begin***. A null pointer is allowed, to simplify error handling, and causes ***elf_kind*** to return **ELF_K_NONE**.

elf_getscn
elf_ndxscn
elf_newscn
elf_nextscn

NAME

elf_getscn, elf_ndxscn, elf_newscn, elf_nextscn

SYNOPSIS

#include <libelf.h>

Elf_Scn	**elf_getscn(Elf *elf, size_t index);*
size_t	*elf_ndxscn(Elf_Scn *scn);*
Elf_Scn	**elf_newscn(Elf *elf);*
Elf_Scn	**elf_nextscn(Elf *elf, Elf_Scn *scn);*

DESCRIPTION

These functions provide indexed and sequential access to the sections associated with the ELF descriptor elf. If the program is building a new file, it is responsible for creating the file's ELF header before creating sections. ***elf_getscn*** returns a section descriptor, given an index into the file's section header table. Note the first ``real'' section has index 1. Although a program can get a section descriptor for the section whose index is 0 (**SHN_UNDEF**, the undefined section), the section has no data and the section header is ``empty'' (though present). If the specified section does not exist, an error occurs, or elf is null, ***elf_getscn*** returns a null pointer.

elf_newscn creates a new section and appends it to the list for elf. Because the **SHN_UNDEF** section is required and not ``interesting'' to applications, the library creates it automatically. Thus the first call to ***elf_newscn*** for an ELF descriptor with no existing sections returns a descriptor for section 1. If an error occurs or elf is null, ***elf_newscn*** returns a null pointer.***elf_newscn*** creates a new section and appends it to the list for elf. Because the **SHN_UNDEF** section is required and not ``interesting'' to applications, the library creates it automatically. Thus the first call to elf_newscn for an ELF descriptor with no existing sections returns a descriptor for section 1. If an error occurs or elf is null, ***elf_newscn*** returns a null pointer. After creating a new section descriptor, the program can use ***elf_getshdr*** to retrieve the newly created, ``clean'' section header. The new section descriptor will have no associated data. When creating a new section in this way, the library updates the *e_shnum* member of the ELF header and sets the **ELF_F_DIRTY** bit for the section. If the program is building a new file, it is responsible for creating the file's ELF header before creating new sections. ***Elf_nextscn*** takes an existing section descriptor, *scn*, and returns a section descriptor for the next higher section. One may use a null *scn* to obtain a section descriptor for the section whose index is 1 (skipping the section whose index is **SHN_UNDEF**). If no further sections are present or an error occurs, ***elf_nextscn*** returns a null pointer.

elf_ndxscn takes an existing section descriptor, *scn*, and returns its section table index. If *scn* is null or an error occurs, ***elf_ndxscn*** returns **SHN_UNDEF**.

elf_next

NAME

elf_next

SYNOPSIS

#include <libelf.h>

*Elf_Cmd elf_next(Elf *elf);*

DESCRIPTION

elf_next, ***elf_rand***, and ***elf_begin*** manipulate simple object files and archives. elf is an ELF descriptor previously returned from ***elf_begin***.

elf_next provides sequential access to the next archive member. That is, having an ELF descriptor, elf, associated with an archive member, ***elf_next*** prepares the containing archive to access the following member when the program calls ***elf_begin***. After successfully positioning an archive for the next member, ***elf_next*** returns the value **ELF_C_READ**. Otherwise, the open file was not an archive, elf was null, or an error occurred, and the return value is **ELF_C_NULL**. In either case, the return value may be passed as an argument to ***elf_begin***, specifying the appropriate action.

elf_rand

NAME

elf_rand

SYNOPSIS

#include <libelf.h>

size_t elf_rand(Elf *elf, size_t offset);

DESCRIPTION

elf_rand provides random archive processing, preparing elf to access an arbitrary archive member. elf must be a descriptor for the archive itself, not a member within the archive. *offset* gives the byte offset from the beginning of the archive to the archive header of the desired member. See ***elf_getarsym*** for more information about archive member offsets. When ***elf_rand*** works, it returns offset. Otherwise it returns 0, because an error occurred, *elf* was null, or the file was not an archive (no archive member can have a zero offset). A program may mix random and sequential archive processing.

elf_rawfile

NAME

elf_rawfile

SYNOPSIS

#include <libelf.h>

char *elf_rawfile(Elf *elf, size_t *ptr);

DESCRIPTION

elf_rawfile returns a pointer to an uninterpreted byte image of the file. This function should be used only to retrieve a file being read. For example, a program might use ***elf_rawfile*** to retrieve the bytes for an archive member.

A program may not close or disable the file descriptor associated with elf before the initial call to ***elf_rawfile***, because ***elf_rawfile*** might have to read the data from the file if it does not already have the original bytes in memory. Generally, this function is more efficient for unknown file types than for object files. The library implicitly translates object files in memory, while it leaves unknown files unmodified. Thus asking for the uninterpreted image of an object file may create a duplicate copy in memory.

elf_rawdata is a related function, providing access to sections within a file.

If *ptr* is non-null, the library also stores the file's size, in bytes, in the location to which *ptr* points. If no data is present, *elf* is null, or an error occurs, the return value is a null pointer, with zero optionally stored through *ptr*.

elf_strptr

NAME

elf_strptr

SYNOPSIS

#include <libelf.h>

char *elf_strptr(Elf *elf, size_t section, size_t offset);

DESCRIPTION

This function converts a string section *offset* to a string pointer. elf identifies the file in which the string section resides, and section gives the section table index for the strings. ***elf_strptr*** normally returns a pointer to a string, but it returns a null pointer when elf is null, section is invalid or is not a section of type **SHT_STRTAB**, the section data cannot be obtained, offset is invalid, or an error occurs.

elf_update

NAME

elf_update

SYNOPSIS

#include <libelf.h>

off32_t elf_update(Elf *elf, Elf_Cmd cmd);

DESCRIPTION

elf_update causes the library to examine the information associated with an ELF descriptor, elf, and to recalculate the structural data needed to generate the file's image.

cmd may have the following values.

ELF_C_NULL This value tells ***elf_update*** to recalculate various values, updating only the ELF descriptor's memory structures. Any modified structures are flagged with the **ELF_F_DIRTY** bit. A program thus can update the structural information and then reexamine them without changing the file associated with the ELF descriptor. Because this does not change the file, the ELF descriptor may allow reading, writing, or both reading and writing.

ELF_C_WRITE If cmd has this value, ***elf_update*** duplicates its **ELF_C_NULL** actions and also writes any “dirty” information associated with the ELF descriptor to the file. That is, when a program has used ***elf_getdata*** or the *elf_flag* facilities to supply new (or update existing) information for an ELF descriptor, those data will be examined, coordinated, translated if necessary, and written to the file. When portions of the file are written, any **ELF_F_DIRTY** bits are reset, indicating those items no longer need to be written to the file. The sections' data is written in the order of their section header entries, and the section header table is written to the end of the file. When the ELF descriptor was created with ***elf_begin***, it must have allowed writing the file. That is, the ***elf_begin*** command must have been either **ELF_C_RDWR** or **ELF_C_WRITE**.

If ***elf_update*** succeeds, it returns the total size of the file image (not the memory image), in bytes. Otherwise an error occurred, and the function returns -1.

When updating the internal structures, ***elf_update*** sets some members itself. Members listed below are the application's responsibility and retain the values given by the program.

Table 1: ELF HEADER

Member	Notes
*e_ident[***EI_DATA***]*	Library controls other *e_ident* values
e_type	
e_machine	
e_version	
e_entry	
e_phoff	Only when **ELF_F_LAYOUT** asserted
e_shoff	Only when **ELF_F_LAYOUT** asserted
e_flags	
e_shstrndx	

Table 2: Section Header

Member	Notes
sh_name	
sh_type	
sh_flags	
sh_addr	
sh_offset	Only when **ELF_F_LAYOUT** asserted
sh_size	Only when **ELF_F_LAYOUT** asserted
sh_link	
sh_info	
sh_addralign	Only when **ELF_F_LAYOUT** asserted
sh_entsize	

Table 3: Section Header

Member	Notes
sh_name	
sh_type	
sh_flags	
sh_addr	
sh_offset	Only when **ELF_F_LAYOUT** asserted
sh_size	Only when **ELF_F_LAYOUT** asserted
sh_link	
sh_info	

Table 3: Section Header

Member	Notes
sh_addralign	Only when **ELF_F_LAYOUT** asserted
sh_entsize	

Table 4: Data Descriptor

Member	Notes
d_buf	
d_type	
d_size	
d_off	Only when **ELF_F_LAYOUT** asserted
d_align	
d_version	

Note: the program is responsible for two particularly important members (among others) in the ELF header. The *e_version* member controls the version of data structures written to the file. If the version is **EV_NONE**, the library uses its own internal version. The *e_ident*[**EI_DATA**] entry controls the data encoding used in the file. As a special case, the value may be **ELFDATANONE** to request the native data encoding for the host machine. An error occurs in this case if the native encoding doesn't match a file encoding known by the library. Further note that the program is responsible for the *sh_entsize* section header member. Although the library sets it for sections with known types, it cannot reliably know the correct value for all sections. Consequently, the library relies on the program to provide the values for unknown section type. If the entry size is unknown or not applicable, the value should be set to zero. When deciding how to build the output file, ***elf_update*** obeys the alignments of individual data buffers to create output sections. A section's most strictly aligned data buffer controls the section's alignment. The library also inserts padding between buffers, as necessary, to ensure the proper alignment of each buffer.

elf_version

NAME

elf_version

SYNOPSIS

#include <libelf.h>

unsigned elf_version(unsigned ver);

DESCRIPTION

The program, the library, and an object file have independent notions of the "latest" ELF version. ***elf_version*** lets a program determine the ELF library's internalversion.It further lets the program specify what memory types it uses by giving its own working version, *ver*, to the library. Every program that uses the ELF library must coordinate versions as described below.

The header file *libelf.h* supplies the version to the program with the macro **EV_CURRENT**. If the library's internal version (the highest version known to the library) is lower than that known by the program itself, the library may lack semantic knowledge assumed by the program. Accordingly, ***elf_version*** will not accept a working version unknown to the library.

Passing ver equal to **EV_NONE** causes ***elf_version*** to return the library's internal version, without altering the working version. If ver is a version known to the library, ***elf_version*** returns the previous (or initial) working version number. Otherwise, the working version remains unchanged and ***elf_version*** returns **EV_NONE**.

SPARC COMPLIANCE DEFINITION 2.4.1 IS

libintl

gettext, dgettext, dcgettext
textdomain, bindtextdomain

NAME

gettext, dgettext, dcgettext, textdomain, bindtextdomain - message handling functions

SYNOPSIS

#include <libintl.h>

#include <locale.h> /* needed for dcgettext() only */

char *gettext(const char *msgid);

char *dgettext(const char *domainname, const char *msgid);

char *dcgettext(const char *domainname, const char *msgid, int category);

char *textdomain(const char *domainname);

char *bindtextdomain(const char *domainname, const char *dirname);

DESCRIPTION

gettext(), ***dgettext***(), and ***dcgettext***() attempt to retrieve a target string based on the specified msgid argument within the context of a specific domain and the current locale. The length of strings returned by ***gettext***(), ***dgettext***(), and ***dcgettext***() is undetermined until the function is called. The msgid argument is a null-terminated string.

NLSPATH is searched first for the location of the **LC_MESSAGES** catalogue. The setting of the **LC_MESSAGES** category of the current locale determines the locale used by ***gettext***() and ***dgettext***() for string retrieval. *category* determines the locale used by ***dcgettext***(). If **NLSPATH** is not defined and the current locale is "C", ***gettext***(), ***dgettext***(), and ***dcgettext***() simply return the message string that was passed. In a locale other than "C", if **NLSPATH** is not defined or if a message catalogue is not found in any of the components specified by **NLSPATH**, the routines search for the message catalogue *dirname/locale/category/domainname.mo*, after querying ***bindtextdomain***() for dirname.

For ***gettext***(), the domain used is set by the last valid call to ***textdomain***(). If a valid call to ***textdomain***() has not been made, the default domain (called messages) is used. For ***dgettext***() and ***dcgettext***(), the domain used is specified by the domainname argument. The domainname argument is equivalent in syntax and meaning to the domainname argument to ***textdomain***(), except that the selection of the domain is valid only for the duration of the ***dgettext***() or ***dcgettext***() call.

textdomain() sets or queries the name of the current domain of the active **LC_MESSAGES** locale category. The domainname argument is a null-terminated string that can contain only the characters allowed in legal filenames.

The *domainname* argument is the unique name of a domain on the system. If there are multiple versions of the same domain on one system, namespace collisions can be avoided by using ***bindtextdomain***(). If ***textdomain***() is not called, a default domain is selected. The setting of domain made by the last valid call to ***textdomain***() remains valid across subsequent calls to ***setlocale***(), and ***gettext***().

The *domainname* argument is applied to the currently active **LC_MESSAGES** locale. The current setting of the domain can be queried without affecting the current state of the domain by calling ***textdomain***() with domainname set to the null pointer. Calling ***textdomain***() with a *domainname* argument of a null string sets the domain to the default domain (messages). ***bindtextdomain***() binds the path predicate for a message domain *domainname* to the value contained in dirname. If domainname is a non-empty string and has not been bound previously, ***bindtextdomain***() binds domainname with dirname.

If domainname is a non-empty string and has been bound previously, ***bindtextdomain***() replaces the old binding with dirname. *dirname* can be an absolute or relative pathname being resolved when ***gettext***(), ***dgettext***(), or ***dcgettext***() are called. If *domainname* is a null pointer or an empty string, ***bindtextdomain***() returns NULL. User defined domain names cannot begin with the string **SYS_**. Domain names beginning with this string are reserved for system use.

RETURN VALUES

The individual bytes of the string returned by ***gettext***(), ***dgettext***(), or ***dcgettext***() can contain any value other than null. If *msgid* is a null pointer, the return value is undefined. The string returned must not be modified by the program, and can be invalidated by a subsequent call to ***gettext***(), ***dgettext***(), ***dcgettext***(), or ***setlocale***(). If the *domainname* argument to ***dgettext***() or ***dcgettext***() is a null pointer, the results are undefined. If the target string cannot be found in the current locale and selected domain, ***gettext***(), ***dgettext***(), and ***dcgettext***() return *msgid*. The normal return value from ***textdomain***() is a pointer to a string containing the current setting of the domain. If domainname is a null pointer, ***textdomain***() returns a pointer to the string containing the current domain. If ***textdomain***() was not previously called and domainname is a null string, the name of the default domain is returned. The name of the default domain is messages.

The return value from ***bindtextdomain***() is a null-terminated string containing dirname or the directory binding associated with domainname if *dirname* is NULL. If no binding is found, the default return value is */usr/lib/locale*. If domainname is a null pointer or an empty string, ***bindtextdomain***() takes no action and returns a null pointer. The string returned must not be modified by the caller.

FILES

/usr/lib/locale	The default path predicate for message domain files.
/usr/lib/locale/locale/LC_MESSAGES/domainname.mo	system default location for file containing messages for language locale and domainname
/usr/lib/locale/locale/LC_XXX/domainname.mo	system default location for file containing messages for language locale and domainname for ***dcgettext***() calls where **LC_XXX** is **LC_CTYPE**, **LC_NUMERIC**, **LC_TIME**, **LC_COLLATE**, **LC_MONETARY**, or **LC_MESSAGES**
dirname/*locale/LC_MESSAGES/domainname.mo*	location for file containing messages for domain domainname and path predicate dirname after a successful call to ***bindtextdomain***()

dirname/*locale/LC_XXX/domainname.mo*	location for files containing messages for domain domainname, language locale, and path predicate dirname after a successful call to ***bindtextdomain***() for ***dcgettext***() calls where **LC_XXX** is one of **LC_CTYPE**, **LC_NUMERIC**, **LC_TIME**, **LC_COLLATE**, **LC_MONETARY**, or **LC_MESSAGES**.

SEE ALSO

msgfmt(), ***xgettext***(), ***setlocale***()

NOTES

These routines impose no limit on message length. However, a text domainname is limited to **TEXTDOMAINMAX** (256) bytes.

gettext, ***dgettext***, ***dcgettext***, ***textdomain*** and ***bindtextdomain*** can be used safely in a multithread application, as long as ***setlocale***() is not being called to change the locale.

SPARC COMPLIANCE DEFINITION 2.4.1 IS

libm

copysign

NAME

copysign - return magnitude of first argument and sign of second argument

SYNOPSIS

#include <math.h>

double copysign(double x, double y);

DESCRIPTION

The ***copysign***() function returns a value with the magnitude of x and the sign of *y*. It produces a **NaN** with the sign of y if x is a **NaN**.

RETURN VALUES

The ***copysign***() function returns a value with the magnitude of x and the sign of y.

expm1

NAME

expm1 - computes exponential functions

SYNOPSIS

#include <math.h>

double expm1(double x);

DESCRIPTION

The ***expm1***() function computes $e^{x-1.0}$.

RETURN VALUES

If x is **NaN**, then the function returns **NaN**.

If x is positive infinity, ***expm1***() returns positive infinity.

If x is negative infinity, ***expm1***() returns -1.0.

If the value overflows, ***expm1***() returns **HUGE_VAL**.

ERRORS

No errors will occur.

USAGE

The value of ***expm1***(x) may be more accurate than ***exp***(x)-1.0 for small values of x.

The ***expm1***() and ***log1p***() functions are useful for financial calculations of $(((1+x)^n)-1)/x$ namely:

expm1(n * log1p(x))/x

when x is very small (for example, when performing calculations with a small daily interest rate). These functions also simplify writing accurate inverse hyperbolic functions.

SEE ALSO

exp(), ilogb(), log1p()

ilogb

NAME

ilogb - returns an unbiased exponent

SYNOPSIS

#include <math.h>

int ilogb(double x);

DESCRIPTION

The ***ilogb***() function returns the exponent part of x. Formally, the return value is the integral part of ***log***(sub r) |x| as a signed integral value, for non-zero finite x, where r is the radix of the machine's floating point arithmetic.

RETURN VALUES

Upon successful completion, ***ilogb***() returns the exponent part of x.

If x is 0, ***ilogb***() returns -**INT_MAX**.

If x is **NaN** or +/-Inf, ***ilogb***() returns **INT_MAX**.

SEE ALSO

logb()

log1p

NAME

log1p - compute natural logarithm

SYNOPSIS

#include <math.h>

double log1p(double x);

DESCRIPTION

The ***log1p***() function computes $\log_e(1.0 + x)$. The value of x must be greater than -1.0.

RETURN VALUES

Upon successful completion, ***log1p***() returns the natural logarithm of 1.0 + x.

If x is **NaN**, ***log1p***() returns **NaN**.

If x is less than -1.0, ***log1p***() returns -**HUGE_VAL** or **NaN** and sets errno to **EDOM**.

If x is -1.0, ***log1p***() returns -**HUGE_VAL** and may set errno to **ERANGE**.

For exceptional cases, ***matherr***() tabulates the values to be returned as dictated by Standards other than **XPG4**.

ERRORS

The ***log1p***() function will fail if:

EDOM The value of x is less than -1.0.

The ***log1p***() function may fail and set errno to:

ERANGE The value of x is -1.0.

SEE ALSO

log(), matherr()

rint

NAME

rint - round-to-nearest integral value

SYNOPSIS

#include <math.h>

double rint(double x);

DESCRIPTION

The ***rint***() function returns the integral value (represented as a double) nearest x in the direction of the current **IEEE754** rounding mode.

If the current rounding mode rounds toward negative infinity, then ***rint***() is identical to ***floor***(). If the current rounding mode rounds toward positive infinity, then ***rint***() is identical to ***ceil***().

RETURN VALUES

Upon successful completion, the ***rint***() function returns the integer (represented as a double precision number) nearest x in the direction of the current **IEEE754** rounding mode.

When x is +/-Inf, ***rint***() returns x.

If the value of x is **NaN**, **NaN** is returned.

ERRORS

No errors will occur.

SEE ALSO

ceil(), floor(), isnan()

scalbn

NAME

scalbn - load exponent of a radix-independent floating-point number

SYNOPSIS

#include <math.h>

double scalbn(double x, int n);

DESCRIPTION

The ***scalbn***() function computes $x * r^n$, where r is the radix of the machine's floating point arithmetic.

RETURN VALUES

Upon successful completion, the ***scalbn***() function returns $x * r^n$.

If the correct value would overflow, ***scalbn***() returns **+/-HUGE_VAL** (according to the sign of x).

The ***scalbn***() function returns x when x is +/-Inf.

If x is **NaN**, then ***scalbn***() returns **NaN**.

significand

NAME

significand - significand function

SYNOPSIS

#include <math.h>

double significand(double x);

DESCRIPTION

The ***significand***() function, along with the ***logb***() and ***scalb***() functions, allows users to verify compliance to **ANSI/IEEE Std 754-1985** by running certain test vectors distributed by the University of California.

If x equals $sig * 2^n$ with $1 < sig < 2$, then ***significand***(x) returns *sig* for exercising the fraction-part(F) test vector. ***significand***(x) is not defined when x is either 0, +/-Inf or **NaN**.

RETURN VALUES

For exceptional cases, ***matherr***() tabulates the values to be returned as dictated by various Standards.

SEE ALSO

logb(), matherr(), scalb()

SPARC COMPLIANCE DEFINITION 2.4.1 IS

libnisdb

db_table_exists, db_unload_table, db_free_result

NAME

db_table_exists, db_unload_table, db_free_result - **NIS+** service functions

SYNOPSIS

#include <rpcsvc/nis.h>

#include <rpcsvc/nis_db.h>

```
db_status    db_table_exists(const char *table_name);
db_status    db_unload_table(const char *table_name);
void         db_free_result(db_result *);
```

DESCRIPTION

db_table_exists() provides an efficient way for the **NIS+** service to detect that a table exists. This increases response time to the client and lowers the load on the server.

db_unload_table() is used by the service to unload or deactivate tables that are not currently being used. The service internally keeps track of access patterns to tables and will unload those tables that have not been accessed for a while. By unloading infrequently accessed tables, the service can minimize the amount of system resources for efficient operation.

db_free_result() frees up the space allocated by various functions listed on this manual page that return a *db_result* structure.

db_initialize, db_create_table, db_destroy_table, db_first_entry db_next_entry, db_reset_next_entry, db_list_entries, db_remove_entry db_add_entry, db_table_exists, db_unload_table, db_checkpoint, db_standby, db_free_result

NAME

db_initialize, db_create_table, db_destroy_table, db_first_entry, db_next_entry, db_reset_next_entry, db_list_entries, db_remove_entry, db_add_entry, db_table_exists, db_unload_table, db_checkpoint, db_standby, db_free_result - NIS+ Database access functions

SYNOPSIS

```
#include <rpcsvc/nis.h>

#include <rpcsvc/nis_db.h>

bool        db_initialize(const char *dictionary_pathname);
db_status   db_create_table(const char *table_name, const table_obj *table);
db_status   db_destroy_table(const char *table_name);
db_result   *db_first_entry(const char *table_name, const int numattrs, const nis_attr *attrs);
db_result   *db_next_entry(const char *table_name,const db_next_desc *next_handle);
db_result   *db_reset_next_entry(const char *table_name,const db_next_desc *next_handle);
db_result   *db_list_entries(const char *table_name,const int numattrs,
                    const nis_attr *attrs);
db_result   *db_remove_entry(const char *table_name, const int numattrs,
                    const nis_attr *attrs);
db_result   *db_add_entry(const char *table_name, const int numattrs,
                    const nis_attr *attrs, const entry_obj *entry);
db_status   db_table_exists(const char *table_name);
db_status   db_unload_table(const char *table_name);
db_status   db_checkpoint(const char *table_name);
db_status   db_standby(const char *table_name);
void        db_free_result(db_result *);
```

DESCRIPTION

These functions describe the interface between the NIS+ server and the underlying database. They are defined in the shared library */usr/lib/libnisdb.so*.

The interface is a simple subset of a complete relational database and provides just those items that are needed by the **NIS+** server daemon. When you replace the database, your interface routines should match these exactly. Also note that the database is responsible for verifying that the objects passed do not exceed the internal limits of the database being used.

The database's performance will directly affect the performance of the server. The default information base that is provided with **NIS+** is the Structured Storage Manager (**SSM**). This is a memory based database that has been tuned for **NIS+**.

These routines should not be invoked by any NIS+ client. NIS+ clients should use the NIS+ tables API described in *nis_tables*().

These routines only use the *table_obj*, *entry_obj* and the *nis_attr* structures defined in <*rpcsvc/nis.h*>. The **NIS+** directory is itself stored in a table by the service daemon. This table has two columns, one searchable with the name of the object in it, the other non-searchable with binary XDRed data in it. The **NIS+** server converts directory lookup requests in the namespace into table searches. The table it searches in response to these requests will have the same name as the directory of the name it is searching for.

The structure returned by the DB access routines is defined as:

```
enum db_status {
        DB_SUCCESS,
        DB_NOTFOUND,
        DB_NOTUNIQUE,
        DB_BADTABLE,
        DB_BADQUERY,
        DB_BADOBJECT,
        DB_MEMORY_LIMIT,
        DB_STORAGE_LIMIT,
        DB_INTERNAL_ERROR
};
struct db_result {
        db_status       status;                         /* Result status */
        db_next_desc    nextinfo;                       /* descriptor */
        struct {
                        u_int           objects_len;
                        entry_obj       *objects_val;
        } objects;                                      /* A variable list   of objects */
        long            ticks;                          /* execution time in microseconds */
};
```

The structure ***db_next_desc*** should be used as an opaque handle for ***db_next_entry***() and ***db_reset_next_entry***().

The *nis_attr* structure used in ***db_first_entry*** and other related functions is defined as follows:

```
struct nis_attr {
        char            *zattr_ndx;
        struct {
                        u_int           zattr_val_len;
                        char            *zattr_val_val;
        } zattr_val;
};
```

zattr_ndx is the name of the attribute. *zattr_val_len* is the value of the attribute *zattr_val_val*.

In db_result, the objects array contains objects if and only if the result returned in the status variable is **DB_SUCCESS**. A null pointer, instead of a pointer to a *db_result* structure, is returned if there is insufficient memory to create the structure.

db_initialize() is called prior to any interaction with the database. It takes as argument the pathname of the file that contains, or will contain, catalog information associated with the database.

db_create_table() creates a new table using the given table name and the table object. It returns TRUE if the table was successfully created; FALSE otherwise.

db_destroy_table() destroys the table of the given name. It returns TRUE if the destruction was successful; FALSE otherwise.

db_first_entry() returns a copy of the first entry in the specified table that satisfies the given attributes. If no attributes are supplied, a copy of the first entry in the table is returned. *attrs* is an array of nis_attr structure with numattrs number of elements. The returned structure, *db_result*, contains a structure, *db_next_desc*, to be used as an argument to ***db_next_entry***() or ***db_reset_next_entry***(). ***db_next_desc*** should only be used only as an opaque handle. ***db_free_result***() can be used to free up the returned *db_result* structure.

db_next_entry() returns a copy of the next entry as indicated by the *next_handle*. An initial call to ***db_first_entry***(), followed by a sequence of calls to ***db_next_entry***(), can be used to successfully obtain entries of an entire table or entries that satisfy the attributes supplied to ***db_first_entry***(). ***db_free_result***() can be used to free up the returned *db_result* structure.

db_reset_next_entry() terminates the ***db_first_entry***()/***db_next_entry***() sequence as indicated by *next_handle*, freeing any resources that have been used to maintain the sequence. After a call to ***db_reset_next_entry***(), a call to ***db_next_entry***() using the same *next_handle* would fail, returning a **DB_BADQUERY** reply.

db_free_result() can be used to free up the returned *db_result* structure.

db_list_entries() returns copies of entries that satisfy the given attributes. ***db_free_result***() can be used to free up the returned *db_result* structure. *attrs* is an array of *nis_attr* structure with numattrs number of elements.

db_remove_entry() removes all entries that satisfy the given attributes. ***db_free_result***() can be used to free up the returned *db_result* structure. *attrs* is an array of *nis_attr* structure with numattrs number of elements.

db_add_entry() adds a copy of the given object to the specified table, replacing the one identified by the given attributes. If the given attributes identify more than one object, **DB_NOTUNIQUE** is returned. If no object is identified by the given attributes, the object is added. *attrs* is an array of nis_attr structure with numattrs number of elements. ***db_free_result***() can be used to free up the returned *db_result* structure.

db_table_exists() provides an efficient way for the NIS+ service to detect that a table exists. This increases response time to the client and lowers the load on the server.

db_unload_table() is used by the service to unload or deactivate tables that are not currently being used. The service internally keeps track of access patterns to tables and will unload those tables that have not been accessed for a while. By unloading infrequently accessed tables, the service can minimize the amount of system resources for efficient operation.

db_checkpoint() organizes the contents of the table in a more efficient manner. Checkpointing may mean different things to different types of databases. It does not affect the logical contents of the table - operations and queries should return the same result before and after a checkpoint. For example, in a log-based system, checkpointing may mean incorporating log entries of updates accumulated since the previous checkpoint into the table.

db_free_result() frees up the space allocated by various functions listed on this manual page that return a *db_result* structure.

db_standby() is an advisory call to the database manager. This call informs the database that activity has slowed down and it can free up unnecessary resources such as file descriptors.

PROGRAMMING

Most of the routines in this library use an NIS+ name to identify the object that the user desires. The name must be in canonical form before being passed to the database because one server may be serving several namespaces and discrimination of the requested objects is accomplished by comparing the domain names.

DIAGNOSTICS

DB_SUCCESS	The query or operation completed successfully and returned status.
DB_NOTFOUND	The name or entry that was named in the argument did not exist.
DB_NOTUNIQUE	An attempt was made to remove an entry from a table that is not uniquely specified.
DB_BADQUERY	The query that was submitted to the database was invalid (for example, it might name some nonexistent fields).
DB_BADTABLE	The table was corrupted.
DB_BADOBJECT	The fields of the object does not conform to the fields of the table to which it is being added.
DB_MEMORY_LIMIT	There is insufficient memory to complete the operation requested.
DB_STORAGE_LIMIT	There is insufficient file storage available to complete the operation requested.
DB_INTERNAL_ERROR	An internal error was encountered during the execution of the operation requested (either a programming error or an unrecoverable exception).

SPARC COMPLIANCE DEFINITION 2.4.1 IS

libnsl

inet_addr
inet_netof
inet_ntoa

NAME

inet_addr, inet_netof, inet_ntoa - Internet address manipulation

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

unsigned long	***inet_addr (char *cp***);
int	***inet_netof (struct in_addr in***);
char	****inet_ntoa (struct in_addr in***);

DESCRIPTION

The ***inet_addr*** routines interpret a character string, *cp*, representing numbers expressed in the Internet standard "." notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively. The routines ***inet_netof*** breaks apart an Internet host address, *in*, returning the network number and local network address part, respectively. The routine ***inet_ntoa*** returns a pointer to a string in the base 256 notation "d.d.d.d" described below. All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the '.' notation take one of the following forms: a.b.c.d, a.b.c, a.b, a. When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as "128.net.host". When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as "net.host". When only one part is given, the value is stored directly in the network address without any byte rearrangement. All numbers supplied as "parts" in a '.' notation may be decimal, octal, or hexadecimal, as specified in the C language (that is, a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

RETURN VALUES

The value -1 is returned by ***inet_addr*** for malformed requests. The routines ***inet_netof*** break apart Internet host addresses, returning the network number and local network address part, respectively. The routine ***inet_ntoa*** returns a pointer to a string in the base 256 notation "d.d.d.d" described below.

authdes_create
authunix_create, authunix_create_default
callrpc
clnt_broadcast
clntraw_create
clnttcp_create,clntudp_bufcreate, clntudp_create
get_myaddress
getrpcport
pmap_getmaps
pmap_getport
pmap_rmtcall
pmap_set, pmap_unset
registerrpc
rpc_soc
svc_fds
svc_getcaller, svc_getreq
svc_register, svc_unregister
svcfd_create, svcraw_create, svctcp_create
svcudp_bufcreate, svcudp_create
xdr_authunix_parms

NAME

rpc_soc, authdes_create, authunix_create, authunix_create_default, callrpc, clnt_broadcast, clntraw_create, clnttcp_create, clntudp_bufcreate, clntudp_create, get_myaddress, getrpcport, pmap_getmaps, pmap_getport, pmap_rmtcall, pmap_set, pmap_unset, registerrpc, svc_fds, svc_getcaller, svc_getreq, svc_register, svc_unregister, svcfd_create, svcraw_create, svctcp_create, svcudp_bufcreate, svcudp_create, xdr_authunix_parms - obsolete library routines for RPC

SYNOPSIS

#define PORTMAP
#include <rpc/rpc.h>

AUTH	**authdes_create*	*(char *name, unsigned window,* *struct sockaddr *syncaddr, des_block *ckey)*;
AUTH	**authunix_create*	*(char *host, int uid, int gid,* *int grouplen, int gidlistp);*
AUTH	**authunix_create_default*	*(void)*
int	*callrpc*	*(char *host, u_long prognum,* *u_long versnum, u_long procnum,* *xdrproc_t inproc, char *in,* *xdrproc_t outproc, char *out);*
enum clnt_stat	*clnt_broadcast*	*(u_long prognum, u_long versnum,* *u_long procnum, xdrproc_t inproc,* *char *in, xdrproc_t outproc,* *char *out, resultproc_t eachresult);*
CLIENT	**clntraw_create*	*(u_long prognum, u_long versnum);*
CLIENT	**clnttcp_create*	*(struct sockaddr_in *addr,* *u_long prognum, u_long versnum,* *int *fdp, u_int sendsz, u_int recvsz);*
CLIENT	**clntudp_bufcreate*	*(struct sockaddr_in *addr,* *u_long prognum, u_long versnum,* *struct timeval wait, int *fdp,*

```
                                        u_int sendsz, u_int recvsz);

CLIENT             *clntudp_create      (struct sockaddr_in *addr,
                                        u_long prognum, u_long versnum,
                                        struct timeval wait, int *fdp);
void               get_myaddress        (struct sockaddr_in *addr);
void               getrpcport           (char *host, int prognum, int versnum, int proto)
struct pmaplist    *pmap_getmaps        (struct sockaddr_in *addr);
u_short            pmap_getport         (struct sockaddr_in *addr,
                                        u_long prognum, u_long versnum,
                                        u_long protocol);

enum clnt_stat     pmap_rmtcall         (struct sockaddr_in *addr,
                                        u_long prognum, u_long versnum,
                                        u_long procnum, char *in,
                                        xdrproct_t inproc, char *out,
                                        xdrproct_t outproc, struct timeval tout,
                                        u_long *portp);
bool_t             pmap_set             (u_long prognum, u_long versnum,
                                        u_long protocol, u_short port);
bool_t             pmap_unset           (u_long prognum, u_long versnum);
int                svc_fds;
struct sockaddr_in      *svc_getcaller  (SVCXPRT *xprt);
void               svc_getreq           (int rdfds);
SVCXPRT            *svcfd_create        (int fd, u_int sendsz, u_int recvsz);
SVCXPRT            *svcraw_create       (void);
SVCXPRT            *svctcp_create       (int fd, u_int sendsz, u_int recvsz);
SVCXPRT            *svcudp_bufcreate    (int fd, u_int sendsz, u_int recvsz);
SVCXPRT            *svcudp_create       (int fd);
int                registerrpc          (u_long prognum, u_long versnum,
                                        u_long procnum, char *(*procname)(),
                                        xdrproc_t inproc, xdrproc_t outproc);
int                svc_register         (SVCXPRT *xprt, u_long prognum,
                                        u_long versnum, void (*dispatch)(),
                                        u_long protocol);
void               svc_unregister       (u_long prognum, u_long versnum);
int                xdr_authunix_parms   (XDR *xdrs, struct authunix_parms *aupp);
```

DESCRIPTION

RPC routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a request to the server. Upon receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client. The routines described in this manual page have been superseded by other routines. The preferred routine is given after the description of the routine. New programs should use the preferred routines, as support for the older interfaces may be dropped in future releases. Transport independent **RPC** uses **TLI** as its transport interface instead of sockets. Some of the routines described in this section (such as ***clnttcp_create***()) take a pointer to a file descriptor as one of the parameters. If the user wants the file descriptor to be a socket, then the application will have to be linked with both *librpcsoc* and libnsl. If the user passed **RPC_ANYSOCK** as the file descriptor, and the application is linked with libnsl only, then the routine will return a **TLI** file descriptor and not a socket. The following routines require that the header <*rpc/rpc.h*> be included. The symbol **PORTMAP** should be defined so that the appropriate function declarations for the old interfaces are included through the header files.

authdes_create():

authdes_create() is the first of two routines which interface to the **RPC** secure authentication system, known as **DES** authentication.The second is ***authdes_getucred***(), below. Note: the keyserver daemon ***keyserv***() must be running for the **DES** authentication system to work. ***authdes_create***(), used on the client side, returns an authentication handle that will enable the use of the secure authentication system. The first parameter name is the network name, or netname, of the owner of the server process. This field usually represents a hostname derived from the utility routine ***host2netname***(), but could also represent a user name using ***user2netname***() (see ***secure_rpc***()). The second field is window on the validity of the client credential, given in seconds. A small window is more secure than a large one, but choosing too small of a window will increase the frequency of re-synchronizations because of clock drift. The third parameter syncaddr is optional. If it is NULL, then the authentication system will assume that the local clock is always in sync with the server's clock, and will not attempt re-synchronizations. If an address is supplied, however, then the system will use the address for consulting the remote time service whenever re-synchronization is required. This parameter is usually the address of the **RPC** server itself. The final parameter *ckey* is also optional. If it is NULL, then the authentication system will generate a random **DES** key to be used for the encryption of credentials. If it is supplied, however, then it will be used instead. Warning: this routine exists for backward compatibility only, and is obsoleted by ***authdes_seccreate***() (see ***secure_rpc***()).

authunix_create():

Create and return an **RPC** authentication handle that contains.**UX** authentication information. The parameter host is the name of the machine on which the information was created; uid is the user's user **ID**; gid is the user's current group **ID**; grouplen and gidlistp refer to a counted array of groups to which the user belongs. Warning: it is not very difficult to impersonate a user.

authunix_create_default():

Call ***authunix_create***() with the appropriate parameters. Warning: this routine exists for backward compatibility only, and is obsoleted by ***authsys_create_default***() (see ***rpc_clnt_auth***()).

callrpc():

Call the remote procedure associated with prognum, versnum, and procnum on the machine, host. The parameter inproc is used to encode the procedure's parameters, and outproc is used to decode the procedure's results; in is the address of the procedure's argument, and out is the address of where to place the result(s). This routine returns 0 if it succeeds, or the value of enum clnt_stat cast to an integer if it fails. The routine ***clnt_perrno***() (see ***rpc_clnt_calls***()) is handy for translating failure statuses into messages. Warning: you do not have control of timeouts or authentication using this routine. This routine exists for backward compatibility only, and is obsoleted by ***rpc_call***() (see ***rpc_clnt_calls***()).

clnt_broadcast:

Like ***callrpc***(), except the call message is broadcast to all locally connected broadcast nets. Each time the caller receives a response, this routine calls ***eachresult***(), whose form is: ***eachresult***(*char *out, struct sockaddr_in *addr*); where out is the same as out passed to ***clnt_broadcast***(), except that the remote procedure's output is decoded there; *addr* points to the address of the machine that sent the results. If ***eachresult***() returns 0 ***clnt_broadcast***() waits for more replies; otherwise it returns with appropriate status. If ***eachresult***() is NULL, ***clnt_broadcast***() returns without waiting for any replies. Warning: broadcast packets are limited in size to the maximum transfer unit of the transports involved. For Ethernet, the callers argument size is approximately 1500 bytes. Since the call message is sent to all connected networks, it may potentially lead to broadcast storms. ***clnt_broadcast***() uses **SB AUTH_SYS** credentials by default (see ***rpc_clnt_auth***()). Warning: this routine exists for backward compatibility only, and is obsoleted by ***rpc_broadcast***() (see ***rpc_clnt_calls***()).

clntraw_create():

This routine creates an internal, memory-based **RPC** client for the remote program prognum, version versnum. The transport used to pass messages to the service is actually a buffer within the process's address space, so the corresponding **RPC** server should live in the same address space; see ***svcraw_create***(). This allows simulation of **RPC** and acquisition of **RPC** overheads, such as round trip times, without any kernel interference. This routine returns NULL if it fails. Warning: this routine exists for backward compatibility only, and has the same functionality as ***clnt_raw_create***() (see ***rpc_clnt_create***()), which obsoletes it.

clnttcp_create():

This routine creates an **RPC** client for the remote program prognum, version *versnum*; the client uses **TCP/IP** as a transport. The remote program is located at Internet address addr. If *addr->sin_port* is 0, then it is set to the actual port that the remote program is listening on (the remote rpcbind service is consulted for this information). The parameter **fdp* is a file descriptor, which may be open and bound; if it is **RPC_ANYSOCK**, then this routine opens a new one and sets **fdp*. Refer to the File Descriptor section for more information. Since **TCP**-based **RPC** uses buffered I/O, the user may specify the size of the send and receive buffers with the parameters sendsz and recvsz; values of 0 choose suitable defaults. This routine returns NULL if it fails.

clntudp_bufcreate():

Create a client handle for the remote program prognum, on versnum; the client uses **UDP/IP** as the transport. The remote program is located at the Internet address addr. If *addr->sin_port* is 0, it is set to port on which the remote program is listening on (the remote rpcbind service is consulted for this information). The parameter **fdp* is a file descriptor, which may be open and bound; if it is **RPC_ANYSOCK**, then this routine opens a new one and sets **fdp*. Refer to the File Descriptor section for more information. The **UDP** transport resends the call message in intervals of wait time until a response is received or until the call times out. The total time for the call to time out is specified by ***clnt_call***() (see ***rpc_clnt_calls***()). If successful it returns a client handle, otherwise it returns NULL. The error can be printed using the ***clnt_pcreateerror***() (see ***rpc_clnt_create***()) routine. The user can specify the maximum packet size for sending and receiving by using *sendsz* and *recvsz* arguments for **UDP**-based **RPC** messages. Warning: if *addr->sin_port* is 0 and the requested version number versnum is not registered with the remote portmap service, it returns a handle if at least a version number for the given program number is registered. The version mismatch is discovered by a ***clnt_call***() later (see ***rpc_clnt_calls***()). Warning: this routine exists for backward compatibility only. ***clnt_tli_create***() or ***clnt_dg_create***() (see ***rpc_clnt_create***()) should be used instead.

clntudp_create():

This routine creates an **RPC** client handle for the remote program prognum, version versnum; the client uses UDP/IP as a transport. The remote program is located at Internet address addr. If *addr->sin_port* is 0, then it is set to actual port that the remote program is listening on (the remote rpcbind service is consulted for this information). The parameter **fdp* is a file descriptor, which may be open and bound; if it is **RPC_ANYSOCK**, then this routine opens a new one and sets **fdp*. Refer to the File Descriptor section for more information. The **UDP** transport resends the call message in intervals of wait time until a response is received or until the call times out. The total time for the call to time out is specified by clnt_call() (see ***rpc_clnt_calls***()). ***clntudp_create***() returns a client handle on success, otherwise it returns NULL. The error can be printed using the ***clnt_pcreateerror***() (see ***rpc_clnt_create***()) routine. Warning: since **UDP**-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results. Warning: this routine exists for backward compatibility only.***clnt_create***(), ***clnt_tli_create***(), or ***clnt_dg_create***() (see ***rpc_clnt_create***()) should be used instead.

get_myaddress():
Places the local system's **IP** address into *addr, without consulting the library routines that deal with */etc/hosts*. The port number is always set to ***htons***(**PMAPPORT**). Warning: this routine is only intended for use with the **RPC** library. It returns the local system's address in a form compatible with the **RPC** library, and should not be taken as the system's actual **IP** address. In fact, the **addr* buffer's host address part is actually zeroed. This address may have only local significance and should NOT be assumed to be an address that can be used to connect to the local system by remote systems or processes. Warning: this routine remains for backward compatibility only.The routine ***netdir_getbyname***() (see ***netdir***()) should be used with the name **HOST_SELF** to retrieve the local system's network address as a netbuf structure.

getrpcport():
getrpcport() returns the port number for the version versnum of the **RPC** program prognum running on host and using protocol proto. ***getrpcport***() returns 0 if the **RPC** system failed to contact the remote portmap service, the program associated with prognum is not registered, or there is no mapping between the program and a port. Warning: This routine exists for backward compatibility only.Enhanced functionality is provided by ***rpcb_getaddr***() (see ***rpcbind***()).

pmap_getmaps():
A user interface to the portmap service, which returns a list of the current **RPC** program-to-port mappings on the host located at **IP** address addr. This routine can return NULL. The command `rpcinfo -p' uses this routine. Warning: this routine exists for backward compatibility only, enhanced functionality is provided by ***rpcb_getmaps***() (see ***rpcbind***()).

pmap_getport():
A user interface to the portmap service, which returns the port number on which waits a service that supports program prognum, version versnum, and speaks the transport protocol associated with protocol. The value of protocol is most likely **IPPROTO_UDP** or **IPPROTO_TCP**. A return value of 0 means that the mapping does not exist or that the **RPC** system failure to contact the remote portmap service. In the latter case, the global variable ***rpc_createerr*** contains the **RPC** status. Warning: this routine exists for backward compatibility only, enhanced functionality is provided by ***rpcb_getaddr***() (see ***rpcbind***()).

pmap_rmtcall():
Request that the portmap on the host at **IP** address **addr* make an **RPC** on the behalf of the caller to a procedure on that host. **portp* is modified to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in ***callrpc***() and ***clnt_call***() (see ***rpc_clnt_calls***()). Note: this procedure is only available for the **UDP** transport. Warning: if the requested remote procedure is not registered with the remote portmap then no error response is returned and the call times out. Also, no authentication is done. Warning: this routine exists for backward compatibility only, enhanced functionality is provided by ***rpcb_rmtcall***() (see ***rpcbind***()).

pmap_set():
A user interface to the portmap service, that establishes a mapping between the triple [prognum, versnum, protocol] and port on the machine's portmap service. The value of protocol may be **IPPROTO_UDP** or **IPPROTO_TCP**. Formerly, the routine failed if the requested port was found to be in use. Now, the routine only fails if it finds that port is still bound. If port is not bound, the routine completes the requested registration. This routine returns 1 if it succeeds, 0 otherwise.Automatically done by ***svc_register***(). Warning: this routine exists for backward compatibility only, enhanced functionality is provided by ***rpcb_set***() (see ***rpcbind***()).

pmap_unset():
A user interface to the portmap service, which destroys all mapping between the triple [*prognum*, *versnum*, all protocols] and port on the machine's portmap service. This routine returns one if it succeeds, 0 otherwise. Warning: this routine exists for backward compatibility only, enhanced functionality is provided by ***rpcb_unset***() (see ***rpcbind***()).

svc_fds:
A global variable reflecting the **RPC** service side's read file descriptor bit mask; it is suitable as a parameter to the ***select***() call. This is only of interest if a service implementor does not call ***svc_run***(), but rather does his own asynchronous event processing. This variable is read-only (do not pass its address to ***select***()!), yet it may change after calls to ***svc_getreq***() or any creation routines. Similar to svc_fdset, but limited to 32 descriptors. Warning: this interface is obsoleted by ***svc_fdset*** (see ***rpc_svc_calls***()).

svc_getcaller():
This routine returns the network address, represented as a struct *sockaddr_in*, of the caller of a procedure associated with the **RPC** service transport handle, *xprt*. Warning: this routine exists for backward compatibility only, and is obsolete. The preferred interface is ***svc_getrpccaller***() (see ***rpc_svc_reg***()), which returns the address as a struct netbuf.

svc_getreq():
This routine is only of interest if a service implementor does not call ***svc_run***(), but instead implements custom asynchronous event processing. It is called when the ***select***() call has determined that an **RPC** request has arrived on some **RPC** file descriptors; rdfds is the resultant read file descriptor bit mask. The routine returns when all file descriptors associated with the value of rdfds have been serviced. This routine is similar to ***svc_getreqset***() but is limited to 32 descriptors.

svcfd_create():
Create a service on top of any open and bound descriptor. Typically, this descriptor is a connected file descriptor for a stream protocol. Refer to the File Descriptor section for more information. sendsz and recvsz indicate sizes for the send and receive buffers. If they are 0, a reasonable default is chosen.

svcraw_create();
This routine creates an internal, memory-based **RPC** service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding **RPC** client should live in the same address space; see ***clntraw_create***(). This routine allows simulation of **RPC** and acquisition of **RPC** overheads (such as round trip times), without any kernel interference. This routine returns NULL if it fails. Warning: this routine exists for backward compatibility only, and has the same functionality of ***svc_raw_create***() (see ***rpc_svc_create***()), which obsoletes it.

svctcp_create():
This routine creates a **TCP/IP**-based **RPC** service transport, to which it returns a pointer. The transport is associated with the file descriptor fd, which may be **RPC_ANYSOCK**, in which case a new file descriptor is created. If the file descriptor is not bound to a local TCP port, then this routine binds it to an arbitrary port. Refer to the File Descriptor section for more information. Upon completion, *xprt->xp_fd* is the transport's file descriptor, and *xprt->xp_port* is the transport's port number. This routine returns NULL if it fails. Since TCP-based RPC uses buffered I/O, users may specify the size of buffers; values of 0 choose suitable defaults. Warning: this routine exists for backward compatibiltyonly.***svc_create***(), ***svc_tli_create***(), or ***svc_vc_create***() (see ***rpc_svc_create***()) should be used instead.

svcudp_bufcreate():
This routine creates a **UDP/IP**-based RPC service transport, to which it returns a pointer. The transport is associated with the file descriptor fd. If fd is **RPC_ANYSOCK**, then a new file descriptor is created. If the file descriptor is not bound to a local **UDP** port, then this routine binds it to an arbitrary port. Upon completion, *xprt->xp_fd* is the transport's file descriptor, and *xprt->xp_port* is the transport's port number. Refer to the File Descriptor section for more information. This routine returns NULL if it fails. The user specifies the maximum packet size for sending and receiving **UDP**-based RPC messages by using the

sendsz and recvsz parameters. Warning: this routine exists for backward compatibility only. ***svc_tli_create***(), or ***svc_dg_create***() (see ***rpc_svc_create***()) should be used instead.

svcudp_create():
This routine creates a **UDP/IP**-based **RPC** service transport, to which it returns a pointer. The transport is associated with the file descriptor *fd*, which may be **RPC_ANYSOCK**, in which case a new file descriptor is created. If the file descriptor is not bound to a local UDP port, then this routine binds it to an arbitrary port. Upon completion, *xprt->xp_fd* is the transport's file descriptor, and *xprt->xp_port* is the transport's port number. This routine returns NULL if it fails. Warning: since **UDP**-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results. Warning: this routine exists for backward compatibility only. ***svc_create***(), ***svc_tli_create***(), or ***svc_dg_create***() (see ***rpc_svc_create***()) should be used instead.

registerrrpc():
Register program prognum, procedure procname, and version *versnum* with the **RPC** service package. If a request arrives for program *prognum*, version *versnum*, and procedure procnum, procname is called with a pointer to its parameter(s); procname should return a pointer to its static result(s); inproc is used to decode the parameters while outproc is used to encode the results. This routine returns 0 if the registration succeeded, -1 otherwise. ***svc_run***() must be called after all the services are registered. Warning: this routine exists for backward compatibility only, and is obsoleted by ***rpc_reg***().

svc_register():
Associates prognum and *versnum* with the service dispatch procedure, dispatch. If protocol is 0, the service is not registered with the portmap service. If protocol is nonzero, then a mapping of the triple [prognum, versnum, protocol] to *xprt->xp_port* is established with the local portmap service (generally protocol is 0, **IPPROTO_UDP** or **IPPROTO_TCP**). The procedure dispatch has the following form:
dispatch(*struct svc_req *request*, **SVCXPRT** **xprt*); The ***svc_register***() routine returns one if it succeeds, and 0 otherwise. Warning: this routine exists for backward compatibility only; enhanced functionality is provided by ***svc_reg***().

svc_unregister():
Remove all mapping of the double [*prognum*, *versnum*] to dispatch routines, and of the triple [*prognum*, *versnum*, all-protocols] to port number from portmap. Warning: this routine exists for backward compatibility, enhanced functionality is provided by ***svc_unreg***().

xdr_authunix_parms():
Used for describing UNIX credentials. This routine is useful for users who wish to generate these credentials without using the RPC authentication package. Warning: this routine exists for backward compatibility only, and is obsoleted by ***xdr_authsys_parms***() (see ***rpc_xdr***()).

SEE ALSO

keyserv(), rpcbind(), rpcinfo(), rpc(), rpc_clnt_auth(), rpc_clnt_calls(), rpc_clnt_create(), rpc_svc_calls(), rpc_svc_create(), rpc_svc_err(), rpc_svc_reg(), rpcbind(), secure_rpc(), select()

NOTES

These interfaces are unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

clnt_control
clnt_create
clnt_create_timed
clnt_create_vers
clnt_create_vers_timed
clnt_destroy
clnt_dg_create
clnt_pcreateerror
clnt_raw_create
clnt_spcreateerror
clnt_tli_create
clnt_tp_create
clnt_tp_create_timed
clnt_vc_create
rpc_clnt_create
rpc_createerr

NAME

rpc_clnt_create, clnt_control, clnt_create, clnt_create_timed, clnt_create_vers, clnt_create_vers_timed, clnt_destroy, clnt_dg_create, clnt_pcreateerror, clnt_raw_create, clnt_spcreateerror, clnt_tli_create, clnt_tp_create, clnt_tp_create_timed, clnt_vc_create, rpc_createerr - library routines for dealing with creation and manipulation of **CLIENT** handles

SYNOPSIS

```
#include <rpc/rpc.h>
bool_t      clnt_control(CLIENT *clnt, const u_int req, char *info);
void        clnt_pcreateerror(const char *s);
void        clnt_destroy(CLIENT *clnt);
char        *clnt_spcreateerror(const char *s);
CLIENT      *clnt_create(const char *host, const u_long prognum,
                    const u_long versnum, const char *nettype);
CLIENT      *clnt_raw_create(const u_long prognum, const u_long versnum);
CLIENT      *clnt_dg_create(const int fildes, const struct netbuf *svcaddr,
                    const u_long prognum, const u_long versnum,
                    const u_int sendsz, const u_int recvsz);
CLIENT      *clnt_tp_create(const char *host, const u_long prognum,
                    const u_long versnum, const struct netconfig *netconf);
CLIENT      *clnt_tli_create(const int fildes, const struct netconfig *netconf,
                    const struct netbuf *svcaddr, const_long prognum,
                    const u_long versnum, const u_int sendsz, const u_int recvsz);
CLIENT      *clnt_create_vers(const char *host, const u_long prognum,
                    u_long *vers_outp, const u_long vers_low,
                    const u_long vers_high, char *nettype);
CLIENT      *clnt_create_vers_timed(const char *host, const u_long prognum,
                    u_long *vers_outp, const u_long vers_low,
                    const u_long vers_high,
                    char *nettype const struct timeval *timeout);
CLIENT      *clnt_create_timed(const char *host, const u_long prognum,
                    const u_long versnum, const char *nettype,
                    const struct timeval *timeout);
CLIENT      *clnt_vc_create(const int fildes, const struct netbuf *svcaddr,
```

const u_long prognum, const u_long versnum,
const u_int sendsz, const u_int recvsz);
*CLIENT *clnt_tp_create_timed(const char *host, const u_long prognum,*
*const u_long versnum, const struct netconfig *netconf,*
*const struct timeval *timeout);*
struct rpc_createerr rpc_createerr;

DESCRIPTION

RPC library routines allow C language programs to make procedure calls on other machines across the network. First a **CLIENT** handle is created and then the client calls a procedure to send a request to the server. On receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends a reply. These routines are MT-Safe. In the case of multithreaded applications, the **_REENTRANT** flag must be defined on the command line at compilation time (-**D_REENTRANT**). When the **_REENTRANT** flag is defined, ***rpc_createerr*** becomes a macro which enables each thread to have its own rpc_createerr. See ***rpc***() for the definition of the **CLIENT** data structure.

clnt_control():

A function macro to change or retrieve various information about a client object. *req* indicates the type of operation, and info is a pointer to the information. For both connectionless and connection-oriented transports, the supported values of req and their argument types and what they do are:

CLSET_TIMEOUT	*struct timeval **	set total timeout
CLGET_TIMEOUT	*struct timeval **	get total timeout

Note: if you set the timeout using clnt_control(), the timeout argument passed by ***clnt_call***() is ignored in all subsequent calls. Note: If you set the timeout value to 0 ***clnt_control***() immediately returns an error (**RPC_TIMEDOUT**). Set the timeout parameter to 0 for batching calls.

CLGET_FD	*int **	get the associated file descriptor
CLGET_SVC_ADDR	*struct netbuf **	get servers address
CLSET_FD_CLOSE	*void*	close the file descriptor when destroying the client handle (see ***clnt_destroy***())
CLSET_FD_NCLOSE	*void*	do not close the file descriptor when destroying the client handle
CLSET_VERS	*unsigned long **	set the **RPC** program's version number associated with the client handle.
CLGET_VERS	*unsigned long **	get the **RPC** program's version number associated with the client handle. This assumes that the **RPC** server for this new version is still listening at the address of the previous version.
CLGET_XID	*unsigned long **	get the **XID** of the previous remote procedure call
CLSET_XID	*unsigned long **	set the **XID** of the next remote procedure call

The following operations are valid for connectionless transports only:

CLSET_RETRY_TIMEOUT	struct timeval *	set the retry timeout
CLGET_RETRY_TIMEOUT	struct timeval *	get the retry timeout

The retry timeout is the time that RPC waits for the server to reply before retransmitting the request. ***clnt_control***() returns TRUE on success and FALSE on failure.

clnt_create():

Generic client creation routine for program prognum and version versnum. host identifies the name of the remote host where the server is located. nettype indicates the class of transport protocol to use. The transports are tried in left to right order in **NETPATH** variable or in top to bottom order in the netconfig database. ***clnt_create***() tries all the transports of the nettype class available from the **NETPATH** environment variable and the netconfig database, and chooses the first successful one. A default timeout is set and can be modified using ***clnt_control***(). This routine returns NULL if it fails. The ***clnt_pcreateerror***() routine can be used to print the reason for failure. Note: ***clnt_create***() returns a valid client handle even if the particular version number supplied to ***clnt_create***() is not registered with the rpcbind service. This mismatch will be discovered by a ***clnt_call*** later (see ***rpc_clnt_calls***()).

clnt_create_timed():

Generic client creation routine which is similar to ***clnt_create***() but which also has the additional parameter timeout that specifies the maximum amount of time allowed for each transport class tried. In all other
respects, the ***clnt_create_timed***() call behaves exactly like the ***clnt_create***() call.

clnt_create_vers():

Generic client creation routine which is similar to ***clnt_create***() but which also checks for the version availability. host identifies the name of the remote host where the server is located. nettype indicates the class transport protocols to be used. If the routine is successful it returns a client handle created for the highest version between *vers_low* and *vers_high* that is supported by the server. *vers_outp* is set to this value. That is, after a successful return *vers_low* <= **vers_outp* <= *vers_high*. If no version between *vers_low* and *vers_high* is supported by the server then the routine fails and returns NULL. A default timeout is set and can be modified using ***clnt_control***(). This routine returns NULL if it fails. The ***clnt_pcreateerror***() routine can be used to print the reason for failure. Note: ***clnt_create***() returns a valid client handle even if the particular version number supplied to ***clnt_create***() is not registered with the *rpcbind* service. This mismatch will be discovered by a ***clnt_call*** later (see ***rpc_clnt_calls***()). However, ***clnt_create_vers***() does this for you and returns a valid handle only if a version within the range supplied is supported by the server.

clnt_create_vers_timed():

Generic client creation routine similar to ***clnt_create_vers***() but with the additional parameter timeout, which specifies the maximum amount of time allowed for each transport class tried. In all other respects, the ***clnt_create_vers_timed***() call behaves exactly like the ***clnt_create_vers***() call.

clnt_destroy()

A function macro that destroys the client's **RPC** handle. Destruction usually involves deallocation of private data structures, including clnt itself. Use of clnt is undefined after calling ***clnt_destroy***(). If the **RPC** library opened the associated file descriptor, or **CLSET_FD_CLOSE** was set using ***clnt_control***(), the file descriptor will be closed. The caller should call ***auth_destroy*** (*clnt->cl_auth*) (before calling ***clnt_destroy***()) to destroy the associated **AUTH** structure (see ***rpc_clnt_auth***()).

clnt_dg_create():

This routine creates an **RPC** client for the remote program prognum and version versnum; the client uses a connectionless transport. The remote program is located at address *svcaddr*. The parameter fildes is an open and bound file descriptor. This routine will resend the call message in intervals of 15 seconds until a response is received or until the call times out. The total time for the call to time out is specified by ***clnt_call***() (see ***clnt_call***() in ***rpc_clnt_calls***()). The retry time out and the total time out periods can be changed using ***clnt_control***(). The user may set the size of the send and receive buffers with the parameters sendsz and recvsz; values of 0 choose suitable defaults. This routine returns NULL if it fails.

clnt_pcreateerror():

Print a message to standard error indicating why a client **RPC** handle could not be created. The message is prepended with the string s and a colon, and appended with a newline.

clnt_raw_create():

This routine creates an **RPC** client handle for the remote program prognum and version versnum. The transport used to pass messages to the service is a buffer within the process's address space, so the corresponding **RPC** server should live in the same address space; (see ***svc_raw_create***() in ***rpc_svc_create***()). This allows simulation of **RPC** and measurement of **RPC** overheads, such as round trip times, without any kernel or networking interference. This routine returns NULL if it fails. ***clnt_raw_create***() should be called after ***svc_raw_create***().

clnt_spcreateerror():
Like ***clnt_pcreateerror***(), except that it returns a string instead of printing to the standard error. A newline is not appended to the message in this case. Warning: returns a pointer to a buffer that is overwritten on each call. In multithread applications, this buffer is implemented as thread-specific data.

clnt_tli_create():
This routine creates an **RPC** client handle for the remote program prognum and version versnum. The remote program is located at address svcaddr. If svcaddr is NULL and it is connection-oriented, it is assumed that the file descriptor is connected. For connectionless transports, if svcaddr is NULL, **RPC_UNKNOWNADDR** error is set. fildes is a file descriptor which may be open, bound and connected. If it is **RPC_ANYFD**, it opens a file descriptor on the transport specified by ***netconf***. If *fildes* is **RPC_ANYFD** and ***netconf*** is NULL, a **RPC_UNKNOWNPROTO** error is set. If fildes is unbound, then it will attempt to bind the descriptor. The user may specify the size of the buffers with the parameters *sendsz* and *recvsz*; values of 0 choose suitable defaults. Depending upon the type of the transport (connection-oriented or connectionless), ***clnt_tli_create***() calls appropriate client creation routines. This routine returns NULL if it fails. The ***clnt_pcreateerror***() routine can be used to print the reason for failure. The remote rpcbind service (see ***rpcbind***()) is not consulted for the address of the remote service.

clnt_tp_create():
Like ***clnt_create***() except ***clnt_tp_create***() tries only one transport specified through netconf. ***clnt_tp_create***() creates a client handle for the program prognum, the version versnum, and for the transport specified by netconf. Default options are set, which can be changed using ***clnt_control***() calls. The remote ***rpcbind*** service on the host is consulted for the address of the remote service. This routine returns NULL if it fails. The ***clnt_pcreateerror***() routine can be used to print the reason for failure.

clnt_tp_create_timed():
Like ***clnt_tp_create***() except ***clnt_tp_create_timed***() has the extra parameter timeout which specifies the maximum time allowed for the creation attempt to succeed. In all other respects, the ***clnt_tp_create_timed***() call behaves exactly like the ***clnt_tp_create***() call.

clnt_vc_create():
This routine creates an **RPC** client for the remote program prognum and version versnum; the client uses a connection-oriented transport. The remote program is located at address svcaddr. The parameter fildes is an open and bound file descriptor. The user may specify the size of the send and receive buffers with the parameters sendsz and recvsz; values of 0 choose suitable defaults. This routine returns NULL if it fails. The address svcaddr should not be NULL and should point to the actual address of the remote program.
clnt_vc_create() does not consult the remote rpcbind service for this information.

rpc_createerr():
A global variable whose value is set by any **RPC** client handle creation routine that fails. It is used by the routine ***clnt_pcreateerror***() to print the reason for the failure. In multithreaded applications, ***rpc_createerr*** becomes a macro which enables each thread to have its own ***rpc_createerr***.

SEE ALSO

rpcbind(), rpc(), rpc_clnt_auth(), rpc_clnt_calls(), rpc_svc_create(), svc_raw_create()

dial
undial

NAME

dial - establish an outgoing terminal line connection

SYNOPSIS

```
#include <dial.h>
int     dial(CALL call);
void    undial(int fd);
```

DESCRIPTION

dial() returns a file-descriptor for a terminal line open for read/write. The argument to ***dial***() is a **CALL** structure (defined in the header <*dial.h*>).

When finished with the terminal line, the calling program must invoke ***undial***() to release the semaphore that has been set during the allocation of the terminal device.

CALL is defined in the header <*dial.h*> and has the following members:

```
struct termio    *attr;     /* pointer to termio attribute struct */
int              baud;      /* transmission data rate */
int              speed;     /* 212A modem: low=300, high=1200 */
char             *line;     /* device name for outgoing line */
char             *telno;    /* pointer to telno digits string */
int              modem;     /* specify modem control for direct lines */
char             *device;   /* unused */
int              dev_len;   /* unused */
```

The *CALL* element speed is intended only for use with an outgoing dialed call, in which case its value should be the desired transmission baud rate. The **CALL** element baud is no longer used.

If the desired terminal line is a direct line, a string pointer to its device-name should be placed in the line element in the **CALL** structure. Legal values for such terminal device names are kept in the *Devices* file. In this case, the value of the baud element should be set to -1. This value will cause dial to determine the correct value from the <*Devices*> file.

The telno element is for a pointer to a character string representing the telephone number to be dialed. Such numbers may consist only of these characters: 0-9 dial 0-9
**dial **
#dial #
=wait for secondary dial tone
-delay for approximately 4 seconds

The **CALL** element modem is used to specify modem control for direct lines. This element should be nonzero if modem control is required. The **CALL** element attr is a pointer to a termio structure, as defined in the header <*termio.h*>. A NULL value for this pointer element may be passed to the dial function, but if such a structure is included, the elements specified in it will be set for the outgoing terminal line before the connection is established. This setting is often important for certain attributes such as parity and baudrate.

The **CALL** elements device and *dev_len* are no longer used. They are retained in the **CALL** structure for compatibility reasons.

RETURN VALUES

On failure, a negative value indicating the reason for the failure will be returned. Mnemonics for these negative indices as listed here are defined in the header <*dial.h*>.

INTRPT	-1	/* interrupt occurred */
D_HUNG	-2	/* dialer hung (no return from write) */
NO_ANS	-3	/* no answer within 10 seconds */
ILL_BD	-4	/* illegal baudrate */
A_PROB	-5	/* acu problem (***open***() failure) */
L_PROB	-6	/* line problem (***open***() failure) */
NO_Ldv	-7	/* can't open Devices file */
DV_NT_A	-8	/* requested device not available */
DV_NT_K	-9	/* requested device not known */
NO_BD_A	-10	/* no device available at requested baud */
NO_BD_K	-11	/* no device known at requested baud */
DV_NT_E	-12	/* requested speed does not match */
BAD_SYS	-13	/* system not in Systems file*/

FILES

/etc/uucp/Devices
/etc/uucp/Systems
/var/spool/uucp/LCK..tty-device

SEE ALSO

uucp(1C), alarm(), read(), write(), termio()

NOTES

Including the header <*dial.h*> automatically includes the header <*termio.h*>. An ***alarm***() system call for 3600 seconds is made (and caught) within the dial module for the purpose of ``touching'' the **LCK..** file and constitutes the device allocation semaphore for the terminal device. Otherwise, ***uucp***() may simply delete the **LCK**.. entry on its 90-minute cleanup rounds. The alarm may go off while the user program is in a ***read***() or ***write***() function, causing an apparent error return. If the user program expects to be around for an hour or more, error returns from ***read***()s should be checked for (errno==**EINTR**), and the ***read***() possibly reissued. This interface is unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

doconfig

NAME

doconfig - execute a configuration script

SYNOPSIS

 # include <sac.h>
int doconfig(int fildes, char *script, long rflag);

DESCRIPTION

doconfig() is a Service Access Facility library function that interprets the configuration scripts contained in the files <*/etc/saf/pmtag/_config*>, <*/etc/saf/_sysconfig*>, and <*/etc/saf/pmtag/svctag*>, where pmtag specifies the tag associated with the port monitor, and svctag specifies the service tag associated with a given service. See ***pmadm***() and ***sacadm***().

script is the name of the configuration script; fildes is a file descriptor that designates the stream to which stream manipulation operations are to be applied; rflag is a bit-mask that indicates the mode in which script is to be interpreted. If rflag is zero, all commands in the configuration script are eligible to be interpreted. If rflag has the **NOASSIGN** bit set, the assign command is considered illegal and will generate an error return. If rflag has the **NORUN** bit set, the ***run*** and ***runwait*** commands are considered illegal and will generate error returns.

The configuration language in which script is written consists of a sequence of commands, each of which is interpreted separately. The following reserved keywords are defined: assign, push, pop, runwait, and run. The comment character is #; when a # occurs on a line, everything from that point to the end of the line is ignored. Blank lines are not significant. No line in a command script may exceed 1024 characters.

assign variable=value:

Used to define environment variables. variable is the name of the environment variable and value is the value to be assigned to it. The value assigned must be a string constant; no form of parameter substitution is available. value may be quoted. The quoting rules are those used by the shell for defining environment variables. assign will fail if space cannot be allocated for the new variable or if any part of the specification is invalid.

push module1[, module2, module3,...]:

Used to push **STREAMS** modules onto the stream designated by fildes. *module1* is the name of the first module to be pushed, *module2* is the name of the second module to be pushed, etc. The command will fail if any of the named modules cannot be pushed. If a module cannot be pushed, the subsequent modules on the same command line will be ignored and modules that have already been pushed will be popped.

pop [module]:

Used to pop **STREAMS** modules off the designated stream. If *pop* is invoked with no arguments, the top module on the stream is popped. If an argument is given, modules will be popped one at a time until the named module is at the top of the stream. If the named module is not on the designated stream, the stream is left as it was and the command fails. If module is the special keyword ALL, then all modules on the stream will be popped. Note that only modules above the topmost driver are affected.

runwait command:

The runwait command runs a command and waits for it to complete. command is the pathname of the command to be run. The command is run with */usr/bin/sh -c* prepended to it; shell scripts may thus be executed from configuration scripts. The runwait command will fail if command cannot be found or cannot be executed, or if command exits with a nonzero status. run command The run command is identical to runwait except that it does not wait for command to complete. command is the pathname of the command to be run. run will not fail unless it is unable to create a child process to execute the command.

Although they are syntactically indistinguishable, some of the commands available to run and runwait are interpreter built-in commands. Interpreter built-ins are used when it is necessary to alter the state of a process within the context of that process. The ***doconfig***() interpreter built-in commands are similar to the shell special commands and, like these, they do not spawn another process for execution. See ***sh***(). The built-in commands are:

cd
ulimit
umask

RETURN VALUES

doconfig() returns 0 if the script was interpreted successfully. If a command in the script fails, the interpretation of the script ceases at that point and a positive number is returned; this number indicates which line in the script failed. If a system error occurs, a value of -1 is returned.When a script fails, the process whose environment was being established should not be started.

SEE ALSO

sh(), pmadm(), sacadm()

NOTES

This interface is unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

getrpcbyname
getrpcbyname_r
getrpcbynumber
getrpcbynumber_r
getrpcent
getrpcent_r
setrpcent

NAME

getrpcbyname, getrpcbyname_r, getrpcbynumber, getrpcbynumber_r, getrpcent, getrpcent_r, setrpcent, endrpcent - get RPC entry

SYNOPSIS

```
#include <rpc/rpcent.h>
struct rpcent    *getrpcbyname(const char * name);
struct rpcent    *getrpcbyname_r(const char * name, struct rpcent *result,
                          char *buffer, int buflen);
struct rpcent    *getrpcbynumber(const int number);
struct rpcent    *getrpcbynumber_r(const int number, struct rpcent *result,
                          char *buffer, int buflen);
struct rpcent    *getrpcent(void);
struct rpcent    *getrpcent_r(struct rpcent *result, char *buffer, int buflen);
void             setrpcent(const int stayopen);
void             endrpcent(void);
```

DESCRIPTION

These functions are used to obtain entries for **RPC** (Remote Procedure Call) services. An entry may come from any of the sources for rpc specified in the */etc/nsswitch.conf* file (see ***nsswitch.conf***()).

getrpcbyname() searches for an entry with the **RPC** service name specified by the parameter name.

getrpcbynumber() searches for an entry with the **RPC** program number.

The functions ***setrpcent***(), ***getrpcent***(), and ***endrpcent***() are used to enumerate **RPC** entries from the database.

setrpcent() sets (or resets) the enumeration to the beginning of the set of **RPC** entries. This function should be called before the first call to ***getrpcent***(). Calls to ***getrpcbyname***() and ***getrpcbynumber***() leave the enumeration position in an indeterminate state. If the stayopen flag is nonzero, the system may keep allocated resources such as open file descriptors until a subsequent call to ***endrpcent***().

Successive calls to ***getrpcent***() return either successive entries or NULL, indicating the end of the enumeration.

endrpcent() may be called to indicate that the caller expects to do no further **RPC** entry retrieval operations; the system may then deallocate resources it was using. It is still allowed, but possibly less efficient, for the process to call more **RPC** entry retrieval functions after calling ***endrpcent***().

Reentrant Interfaces

The functions ***getrpcbyname***(), ***getrpcbynumber***(), and ***getrpcent***() use static storage that is re-used in each call, making these routines unsafe for use in multithreaded applications.

The functions: ***getrpcbyname_r***(), ***getrpcbynumber_r***(), and ***getrpcent_r***() provide reentrant interfaces for these operations.

Each reentrant interface performs the same operation as its non-reentrant counterpart, named by removing the ``_r'' suffix. The reentrant interfaces, however, use buffers supplied by the caller to store returned results, and are safe for use in both single-threaded and multithreaded applications. Each reentrant interface takes the same parameters as its non-reentrant counterpart, as well as the following additional parameters. The parameter result must be a pointer to a struct rpcent structure allocated by the caller. On successful completion, the function returns the **RPC** entry in this structure. The parameter buffer must be a pointer to a buffer supplied by the caller. This buffer is used as storage space for the **RPC** entry data. All of the pointers within the returned struct rpcent result point to data stored within this buffer (see RETURN VALUES). The buffer must be large enough to hold all of the data associated with the **RPC** entry. The parameter buflen should give the size in bytes of the buffer indicated by buffer. For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. ***setrpcent***() may be used in a multithreaded application but resets the enumeration position for all threads. If multiple threads interleave calls to ***getrpcent_r***(), the threads will enumerate disjoint subsets of the **RPC** entry database. Like their non-reentrant counterparts, ***getrpcbyname_r***() and ***getrpcbynumber_r***() leave the enumeration position in an indeterminate state.

RETURN VALUES

RPC entries are represented by the struct rpcent structure defined in <*rpc/rpcent*.h>:

```
struct rpcent {
        char    *r_name;        /* name of this rpc service */
        char    **r_aliases;    /* zero-terminated list of alternate names */
        long    r_number;       /* rpc program number */
};
```

The functions ***getrpcbyname***(), ***getrpcbyname_r***(), ***getrpcbynumber***(), and ***getrpcbynumber_r***() each return a pointer to a struct rpcent if they successfully locate the requested entry; otherwise they return NULL. The functions ***getrpcent***() and ***getrpcent_r***() each return a pointer to a struct rpcent if they successfully enumerate an entry; otherwise they return NULL, indicating the end of the enumeration. The functions ***getrpcbyname***(), ***getrpcbynumber***(), and ***getrpcent***() use static storage, so returned data must be copied before a subsequent call to any of these functions if the data is to be saved.

When the pointer returned by the reentrant functions ***getrpcbyname_r***(), ***getrpcbynumber_r***(), and ***getrpcent_r***() is non-NULL, it is always equal to the result pointer that was supplied by the caller.

ERRORS

The reentrant functions ***getrpcyname_r***(), ***getrpcbynumber_r***() and ***getrpcent_r***() will return NULL and set errno to **ERANGE** if the length of the buffer supplied by caller is not large enough to store the result. See ***intro***() for the proper usage and interpretation of errno in multithreaded applications.

FILES

/etc/rpc
/etc/nsswitch.conf

SEE ALSO

rpcinfo(), nsswitch.conf(), rpc(), attributes()

getnetconfig
setnetconfig
endnetconfig
getnetconfigent
freenetconfigent
nc_perror
nc_sperror

NAME

getnetconfig, setnetconfig, endnetconfig, getnetconfigent, freenetconfigent, nc_perror, nc_sperror - get network configuration database entry

SYNOPSIS

#include <netconfig.h>

struct netconfig	**getnetconfig*	*(void *handlep);*
void	**setnetconfig*	*(void);*
int	*endnetconfig*	*(void *handlep);*
struct netconfig	**getnetconfigent*	*(const char *netid);*
void	*freenetconfigent*	*(struct netconfig *netconfigp);*
void	*nc_perror*	*(const char *msg);*
char	**nc_sperror*	*(void);*

DESCRIPTION

The library routines described on this page are part of the Network Selection component. They provide the application access to the system network configuration database, */etc/netconfig*. In addition to the routines for accessing the netconfig database, Network Selection includes the environment variable **NETPATH** (see ***environ***()) and the **NETPATH** access routines described in ***getnetpath***().

getnetconfig() returns a pointer to the current entry in the netconfig database, formatted as a struct netconfig. Successive calls will return successive netconfig entries in the netconfig database. ***getnetconfig***() can be used to search the entire netconfig file. ***getnetconfig***() returns NULL at the end of the file. handlep is the handle obtained through ***setnetconfig***().

A call to ***setnetconfig***() has the effect of ``binding" to or ``rewinding" the netconfig database. ***setnetconfig***() must be called before the first call to ***getnetconfig***() and may be called at any other time. ***setnetconfig***() need not be called before a call to ***getnetconfigent***(). ***setnetconfig***() returns a unique handle to be used by ***getnetconfig***().

endnetconfig() should be called when processing is complete to release resources for reuse. handlep is the handle obtained through ***setnetconfig***(). Programmers should be aware, however, that the last call to ***endnetconfig***() frees all memory allocated by ***getnetconfig***() for the struct netconfig data structure. ***endnetconfig***() may not be called before ***setnetconfig***().

getnetconfigent() returns a pointer to the struct netconfig structure corresponding to netid. It returns NULL if netid is invalid (that is, does not name an entry in the netconfig database).

freenetconfigent() frees the netconfig structure pointed to by ***netconfigp*** (previously returned by ***getnetconfigent***()).

nc_perror() prints a message to the standard error indicating why any of the above routines failed. The message is prepended with the string msg and a colon. A **NEWLINE** is appended at the end of the message.

nc_sperror() is similar to *nc_perror*() but instead of sending the message to the standard error, will return a pointer to a string that contains the error message.

nc_perror() and *nc_sperror*() can also be used with the **NETPATH** access routines defined in *getnetpath*().

RETURN VALUES

setnetconfig() returns a unique handle to be used by *getnetconfig*(). In the case of an error, *setnetconfig*() returns NULL and *nc_perror*() or *nc_sperror*() can be used to print the reason for failure.

getnetconfig() returns a pointer to the current entry in the *netconfig*() database, formatted as a struct netconfig. *getnetconfig*() returns NULL at the end of the file, or upon failure.

endnetconfig() returns 0 on success and -1 on failure (for example, if *setnetconfig*() was not called previously).

On success, *getnetconfigent*() returns a pointer to the struct netconfig structure corresponding to netid; otherwise it returns NULL.

nc_sperror() returns a pointer to a buffer which contains the error message string. This buffer is overwritten on each call. In multithreaded applications, this buffer is implemented as thread-specific data.

SEE ALSO

getnetpath(), netconfig(), ONC+ Developer's Guide, Transport Interfaces Programming Guide

netdir_free
netdir_getbyaddr
netdir_getbyname
netdir_mergeaddr
netdir_options
netdir_perror
netdir_sperror
taddr2uaddr
uaddr2taddr

NAME

netdir, netdir_getbyname, netdir_getbyaddr, netdir_free, netdir_options, taddr2uaddr, uaddr2taddr, netdir_perror, netdir_sperror, netdir_mergeaddr - generic transport name-to-address translation

SYNOPSIS

```
#include <netdir.h>
int            netdir_getbyname     (const struct netconfig *config,
                                     const struct nd_hostserv *service,
                                     struct nd_addrlist **addrs);
int            netdir_getbyaddr     (const struct netconfig *config,
                                     struct nd_hostservlist **service,
                                     const struct netbuf *netaddr);
void           netdir_free          (void *ptr, const int struct_type);
int            netdir_options       (const struct netconfig *config,
                                     const int option, const int fildes,
                                     char *point_to_args);
char           *taddr2uaddr         (const struct netconfig *config,
                                     const struct netbuf *addr);
struct netbuf  *uaddr2taddr         (const struct netconfig *config, const char *uaddr);
void           netdir_perror        (char *s);
char           *netdir_sperror      (void);
```

DESCRIPTION

These routines provide a generic interface for name-to-address mapping that will work with all transport protocols. This interface provides a generic way for programs to convert transport specific addresses into common structures and back again. The netconfig structure, described on the ***netconfig***() manual page, identifies the transport.

The ***netdir_getbyname***() routine maps the machine name and service name in the *nd_hostserv* structure to a collection of addresses of the type understood by the transport identified in the netconfig structure. This routine returns all addresses that are valid for that transport in the *nd_addrlist* structure. The *nd_hostserv* structure contains the following members:

```
char            *h_host; /* host name */
char            *h_serv; /* service name */
```

The nd_addrlist structure contains the following members:

```
int             n_cnt; /* number of addresses */
struct netbuf   *n_addrs;
```

netdir_getbyname() accepts some special-case host names. The host names are defined in <*netdir.h*>. The currently defined host names are:

HOST_SELF Represents the address to which local programs will bind their end points. **HOST_SELF** differs from the host name provided by ***gethostname***(), which represents the address to which remote programs will bind their end points.

HOST_ANY Represents any host accessible by this transport provider. **HOST_ANY** allows applications to specify a required service without specifying a particular host name.

HOST_SELF_CONNECT Represents the host address that can be used to connect to the local host.

HOST_BROADCAST Represents the address for all hosts accessible by this transport provider. Network requests to this address will be received by all machines.

All fields of the *nd_hostserv* structure must be initialized.

To find the address of a given host and service on all available transports, call the ***netdir_getbyname***() routine with each struct netconfig structure returned by ***getnetconfig***().

The ***netdir_getbyaddr***() routine maps addresses to service names. This routine returns service, a list of host and service pairs that would yield this address. If more than one *tuple* of host and service name is returned, then the first tuple contains the preferred host and service names:

```
struct nd_hostservlist {
        int                 *h_cnt;        /* number of hostservs found */
        struct hostserv     *h_hostservs;
}
```

The ***netdir_free***() structure is used to free the structures allocated by the name to address translation routines. ptr points to the structure that has to be freed.The *struct_type* identifies the structure:

struct netbuf	**ND_ADDR**
struct nd_addrlist	**ND_ADDRLIST**
struct hostserv	**ND_HOSTSERV**
struct nd_hostservlist	**ND_HOSTSERVLIST**

The universal address returned by ***taddr2uaddr***() should be freed by ***free***().

The ***netdir_options***() routine is used to do all transport-specific setups and option management. fildes is the associated file descriptor. option, fildes, and *pointer_to_args* are passed to the ***netdir_options***() routine for the transport specified in config. Currently four values are defined for option:

ND_SET_BROADCAST
ND_SET_RESERVEDPORT
ND_CHECK_RESERVEDPORT
ND_MERGEADDR

The ***taddr2uaddr***() and ***uaddr2taddr***() routines support translation between universal addresses and **TLI** type *netbufs*. The ***taddr2uaddr***() routine takes a struct netbuf data structure and returns a pointer to a string that contains the universal address. It returns NULL if the conversion is not possible. This is not a fatal condition as some transports may not suppose a universal address form.

uaddr2taddr() is the reverse of ***taddr2uaddr***(). It returns the struct netbuf data structure for the given universal address.

If a transport provider does not support an option, *netdir_options* returns -1 and the error message can be printed through ***netdir_perror***() or ***netdir_sperror***(). The specific actions of each option follow.

ND_SET_BROADCAST
: Sets the transport provider up to allow broadcast, if the transport supports broadcast. fildes is a file descriptor into the transport (i.e., the result of a *t_open* of */dev/udp*). *pointer_to_args* is not used. If this completes, broadcast operations may be performed on file descriptor fildes.

ND_SET_RESERVEDPORT
: Allows the application to bind to a reserved port, if that concept exists for the transport provider. fildes is an unbound file descriptor into the transport. If *pointer_to_args* is NULL, fildes will be bound to a reserved port. If pointer_to_args is a pointer to a netbuf structure, an attempt will be made to bind to any reserved port on the specified address.

ND_CHECK_RESERVEDPORT
: Used to verify that the address corresponds to a reserved port, if that concept exists for the transport provider. fildes is not used. *pointer_to_args* is a pointer to a netbuf structure that contains the address. This option returns 0 only if the address specified in *pointer_to_args* is reserved.

ND_MERGEADDR
: Used to take a ``local address'' (like the 0.0.0.0 address that **TCP** uses) and return a ``real address'' that client machines can connect to.fildesisnotused. *pointer_to_args* is a pointer to a struct *nd_mergearg*, which has the following members:

```
char    s_uaddr; /* server's universal address */
char    c_uaddr; /* client's universal address */
char    m_uaddr; /* the result */
```

If *s_uaddr* is something like 0.0.0.0.1.12, and, if the call is successful, *m_uaddr* will be set to something like 192.11.109.89.1.12. For most transports, *m_uaddr* is exactly what *s_uaddr* is.

RETURN VALUES

The ***netdir_perror***() routine prints an error message on the standard output stating why one of the name-to-address mapping routines failed. The error message is preceded by the string given as an argument.

The ***netdir_sperror***() routine returns a string containing an error message stating why one of the name-to-address mapping routines failed.

netdir_sperror() returns a pointer to a buffer which contains the error message string. This buffer is overwritten on each call. In multithreaded applications, this buffer is implemented as thread-specific data.

SEE ALSO

gethostname(), getnetconfig(), getnetpath(), netconfig()

rpc_reg
rpc_svc_reg
svc_auth_reg
svc_reg
svc_unreg
xprt_register
xprt_unregister

NAME

rpc_svc_reg, rpc_reg, svc_reg, svc_unreg, svc_auth_reg, xprt_register, xprt_unregister - library routines for registering servers

SYNOPSIS

#include <rpc/rpc.h>		
bool_t	***rpc_reg***	***(u_long prognum, u_long versnum, u_long procnum, char * const(*procname) (char *arg), xdrproc_t inproc, xdrproc_t outproc, const char *nettype);***
int	***svc_reg***	***(const SVCXPRT *xprt, const u_long prognum, const u_long versnum, const void (*dispatch), const struct netconfig *netconf);***
void	***svc_unreg***	***(const u_long prognum, const u_long versnum);***
int	***svc_auth_reg***	***(const int cred_flavor, const enum auth_stat (*handler));***
void	***xprt_register***	***(const SVCXPRT *xprt);***
void	***xprt_unregister***	***(const SVCXPRT *xprt);***

DESCRIPTION

These routines are a part of the **RPC** library which allows the **RPC** servers to register themselves with ***rpcbind***() (see ***rpcbind***()), and associate the given program and version number with the dispatch function. When the **RPC** server receives a **RPC** request, the library invokes the dispatch routine with the appropriate arguments. See ***rpc***() for the definition of the **SVCXPRT** data structure.

rpc_reg():

Register program prognum, procedure procname, and version versnum with the **RPC** service package. If a request arrives for program prognum, version versnum, and procedure procnum, procname is called with a pointer to its parameter(s); procname should return a pointer to its static result(s). The arg parameter to procname is a pointer to the (decoded) procedure argument. inproc is the **XDR** function used to decode the parameters while outproc is the **XDR** function used to encode the results. Procedures are registered on all available transports of the class nettype.See ***rpc***(). This routine returns 0 if the registration succeeded, -1 otherwise.

svc_reg():

Associates prognum and versnum with the service dispatch procedure, dispatch. If netconf is NULL, the service is not registered with the rpcbind service. For example, if a service has already been registered using some other means, such as *inetd* (see ***inetd***()), it will not need to be registered again. If netconf is nonzero, then a mapping of the triple [*prognum*, *versnum*, *netconf->nc_netid*] to *xprt->xp_ltaddr* is established with the local *rpcbind* service. The ***svc_reg***() routine returns 1 if it succeeds, and 0 otherwise.

svc_unreg():

Remove from the *rpcbind* service, all mappings of the triple [*prognum*, *versnum*, all-transports] to network address and all mappings within the **RPC** service package of the double [*prognum*, *versnum*] to dispatch routines.

svc_auth_reg():

Registers the service authentication routine handler with the dispatch mechanism so that it can be invoked to authenticate **RPC** requests received with authentication type *cred_flavor*. This interface allows developers to add new authentication types to their **RPC** applications without needing to modify the libraries. Service implementors usually do not need this routine. Typical service application would call ***svc_auth_reg***() after registering the service and prior to calling ***svc_run***(). When needed to process an **RPC** credential of type *cred_flavor*, the handler procedure will be called with two parameters (*struct svc_req *rqst, struct rpc_msg *msg*) and is expected to return a valid enum auth_stat value. There is no provision to change or delete an authentication handler once registered. The ***svc_auth_reg***() routine returns 0 if the registration is successful, 1 if *cred_flavor* already has an authentication handler registered for it, and -1 otherwise.

xprt_register():

after **RPC** service transport handle *xprt* is created, it is registered with the **RPC** service package. This routine modifies the global variable ***svc_fdset*** (see ***rpc_svc_calls***()). Service implementors usually do not need this routine.

xprt_unregister():

before an **RPC** service transport handle *xprt* is destroyed, it unregisters itself with the **RPC** service package. This routine modifies the global variable ***svc_fdset*** (see ***rpc_svc_calls***()). Service implementors usually do not need this routine.

SEE ALSO

inetd(), rpcbind(), rpc(), rpc_svc_calls(), rpc_svc_create(), rpc_svc_err(), rpcbind(), select()

rpc_svc_calls
svc_dg_enablecache
svc_done
svc_exit
svc_fdset
svc_freeargs
svc_getargs
svc_getreq_common
svc_getreq_poll
svc_getreqset
svc_getrpccaller
svc_pollset
svc_run
svc_sendreply

NAME

rpc_svc_calls, svc_dg_enablecache, svc_done, svc_exit, svc_fdset, svc_freeargs, svc_getargs, svc_getreq_common, svc_getreq_poll, svc_getreqset, svc_getrpccaller, svc_pollset, svc_run, svc_sendreply - library routines for RPC servers

SYNOPSIS

```
#include <rpc/rpc.h>
int             svc_dg_enablecache(SVCXPRT *xprt, const unsigned long cache_size);
int             svc_done(SVCXPRT *xprt);
void            svc_exit(void);
fd_set          svc_fdset;
bool_t          svc_freeargs(const SVCXPRT *xprt, const xdrproc_t inproc, caddr_t in);
bool_t          svc_getargs(const SVCXPRT *xprt, const xdrproc_t inproc, caddr_t in);
void            svc_getreq_common(const int fd);
void            svc_getreqset(fd_set *rdfds);
void            svc_getreq_poll(struct pollfd *pfdp, const int pollretval);
struct netbuf   *svc_getrpccaller(const SVCXPRT *xprt);
void            svc_run(void);
bool_t          svc_sendreply(const SVCXPRT *xprt, const xdrproc_t outproc,
                        const caddr_t out);
```

DESCRIPTION

These routines are part of the **RPC** library which allows C language programs to make procedure calls on other machines across the network. These routines are associated with the server side of the **RPC** mechanism. Some of them are called by the server side dispatch function, while others (such as ***svc_run***()) are called when the server is initiated.

In the current implementation, the service transport handle **SVCXPRT** contains a single data area for decoding arguments and encoding results. Therefore, this structure cannot be freely shared between threads that call functions that do this. However, when a server is operating in the Automatic or User **MT** modes, a copy of this structure is passed to the service dispatch procedure in order to enable concurrent request processing. Under these circumstances, some routines which would otherwise be unsafe, become safe. These are marked as such. Also marked are routines that are unsafe for **MT** applications, and are not to be used by such applications.

svc_dg_enablecache():

This function allocates a duplicate request cache for the service endpoint *xprt*, large enough to hold *cache_size* entries. Once enabled, there is no way to disable caching. This routine returns 1 if space necessary for a cache of the given size was successfully allocated, and 0 otherwise. This function is safe in MT applications.

svc_done():

This function frees resources allocated to service a client request directed to the service endpoint xprt. This call pertains only to servers executing in the User **MT** mode. In the User **MT** mode, service procedures must invoke this call before returning, either after a client request has been serviced, or after an error or abnormal condition that prevents a reply from being sent. After ***svc_done***() is invoked, the service endpoint *xprt* should not be referenced by the service procedure. Server multithreading modes and parameters can be set using the ***rpc_control***() call.

svc_exit():

This function when called by any of the **RPC** server procedure or otherwise, destroys all services registered by the server and causes ***svc_run***() to return. If **RPC** server activity is to be resumed, services must be reregistered with the **RPC** library either through one of the ***rpc_svc_create***() functions, or using ***xprt_register***(). ***svc_exit***() has global scope and ends all RPC server activity.

svc_fdset:

A global variable reflecting the **RPC** server's read file descriptor bit mask. This is only of interest if service implementors do not call ***svc_run***(), but rather do their own asynchronous event processing. This variable is read-only, and it may change after calls to ***svc_getreqset***() or any creation routines. Do not pass its address to ***select***(), Instead, pass the address of a copy.

svc_freeargs():

A function macro that frees any data allocated by the **RPC/XDR** system when it decoded the arguments to a service procedure using ***svc_getargs***(). This routine returns TRUE if the results were successfully freed, and FALSE otherwise.

svc_getargs():

A function macro that decodes the arguments of an **RPC** request associated with the **RPC** service transport handle *xprt*. The parameter in is the address where the arguments will be placed; inproc is the **XDR** routine used to decode the arguments. This routine returns TRUE if decoding succeeds, and FALSE otherwise.

svc_getreq_common():

This routine is called to handle a request on the given file descriptor.

svc_getreq_poll():

This routine is only of interest if a service implementor does not call ***svc_run***(), but instead implements custom asynchronous event processing. It is called when ***poll***() has determined that an **RPC** request has arrived on some **RPC** file descriptors; *pollretval* is the return value from ***poll***() and *pfdp* is the array of pollfd structures on which the ***poll***() was done. It is assumed to be an array large enough to contain the maximal number of descriptors allowed. This function macro is unsafe in **MT** applications.

svc_getreqset():

This routine is only of interest if a service implementor does not call ***svc_run***(), but instead implements custom asynchronous event processing. It is called when ***select***() has determined that an **RPC** request has arrived on some **RPC** file descriptors; rdfds is the resultant read file descriptor bit mask. The routine returns when all file descriptors associated with the value of rdfds have been serviced. This function macro is unsafe in MT applications.

svc_getrpccaller():

The approved way of getting the network address of the caller of a procedure associated with the **RPC** service transport handle *xprt*.

svc_run():

This routine never returns. In single threaded mode, it waits for **RPC** requests to arrive, and calls the appropriate service procedure using ***svc_getreq_poll***() when one arrives. This procedure is usually waiting for the ***poll***() library call to return.

svc_sendreply():

Called by an **RPC** service's dispatch routine to send the results of a remote procedure call. The parameter xprt is the request's associated transport handle; outproc is the **XDR** routine which is used to encode the results; and out is the address of the results. This routine returns TRUE if it succeeds, FALSE otherwise.

SEE ALSO

rpcgen(), poll(), rpc(), rpc_control(), rpc_svc_create(), rpc_svc_err(), rpc_svc_reg(), select(), xprt_register()

t_strerror

NAME

t_strerror - get error message string

SYNOPSIS

#include <xti.h>
*const char *t_strerror(int errnum);*

DESCRIPTION

This routine is part of the **XTI** interfaces which evolved from the **TLI** interfaces. **XTI** represents the future evolution of these interfaces. However, **TLI** interfaces are supported for compatibility. When using a **TLI** routine that has the same name as an **XTI** routine, a different header file, *tiuser.h*, must be used. Refer to the section, **TLI COMPATIBILITY**, for a description of differences between the two interfaces.

The ***t_strerror***() function maps the supplied number (*errnum*) corresponding to a transport-level error to a language- specific error message string and returns a pointer to that string. The string pointed to will not be modified by the program, but may be overwritten by a subsequent call to the ***t_strerror***() function. The string is not terminated by a newline character. The language for the error message strings written by ***t_strerror***() is implementation-defined. If it is English, the error message string describing the value in *t_errno* is identical to the comments following the *t_errno* codes defined in *xti.h*. If an error code is unknown and the language is English, ***t_strerror***() returns the string: <*error*>: error unknown where <error> is the error number supplied as input. In other languages, an equivalent text is provided.

VALID STATES

Legitimate states (see ***t_getstate***()) for a call to this routine are every one except **T_UNINIT**.

RETURN VALUES

The function ***t_strerror***() returns a pointer to the generated message string.

SEE ALSO

gettext(), perror(), setlocale(), strerror(), t_error()

xdr_bool
xdr_char
xdr_double
xdr_enum
xdr_float
xdr_free
xdr_hyper
xdr_int
xdr_long, xdr_longlong_t
xdr_quadruple
xdr_short
xdr_simple
xdr_u_char
xdr_u_hyper
xdr_u_int
xdr_u_long, xdr_u_longlong_t
xdr_u_short
xdr_void

NAME

xdr_simple, xdr_bool, xdr_char, xdr_double, xdr_enum, xdr_float, xdr_free, xdr_hyper, xdr_int, xdr_long, xdr_longlong_t, xdr_quadruple, xdr_short, xdr_u_char, xdr_u_hyper, xdr_u_int, xdr_u_long, xdr_u_longlong_t, xdr_u_short, xdr_void - library routines for external data representation

SYNOPSIS

```
#include <rpc/xdr.h>
bool_t    xdr_bool          (XDR *xdrs, bool_t *bp);
bool_t    xdr_char          (XDR *xdrs, char *cp);
bool_t    xdr_double        (XDR *xdrs, double *dp);
bool_t    xdr_enum          (XDR *xdrs, enum_t *ep);
bool_t    xdr_float         (XDR *xdrs, float *fp);
void      xdr_free          (xdrproc_t proc, char *objp);
bool_t    xdr_hyper         (XDR *xdrs, longlong_t *llp);
bool_t    xdr_int           (XDR *xdrs, int *ip);
bool_t    xdr_long          (XDR *xdrs, long *lp);
bool_t    xdr_longlong_t    (XDR *xdrs, longlong_t *llp);
bool_t    xdr_quadruple     (XDR *xdrs, long double *pq);
bool_t    xdr_short         (XDR *xdrs, short *sp);
bool_t    xdr_u_char        (XDR *xdrs, unsigned char *ucp);
bool_t    xdr_u_hyper       (XDR *xdrs, u_longlong_t *ullp);
bool_t    xdr_u_int         (XDR *xdrs, unsigned *up);
bool_t    xdr_u_long        (XDR *xdrs, unsigned long *ulp);
bool_t    xdr_u_longlong_t  (XDR *xdrs, u_longlong_t *ullp);
bool_t    xdr_u_short       (XDR *xdrs, unsigned short *usp);
bool_t    xdr_void          (void);
```

DESCRIPTION

XDR library routines allow C programmers to describe simple data structures in a machine-independent fashion. Protocols such as remote procedure calls (**RPC**) use these routines to describe the format of the data. These routines require the creation of **XDR** streams (see ***xdr_create***()).

See *rpc*() for the definition of the **XDR** data structure. Note that any buffers passed to the **XDR** routines must be properly aligned. It is suggested that *malloc*() be used to allocate these buffers or that the programmer insure that the buffer address is divisible evenly by four.

xdr_bool() translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either 1 or 0. This routine returns TRUE if it succeeds, FALSE otherwise.

xdr_char() translates between C characters and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise. Note: encoded characters are not packed, and occupy 4 bytes each. For arrays of characters, it is worthwhile to consider ***xdr_bytes***(), ***xdr_opaque***(), or ***xdr_string***() (see ***xdr_complex***()).

xdr_double() translates between C double precision numbers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

xdr_enum() translates between C enums (actually integers) and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

xdr_float() translates between C floats and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

xdr_free(), is a Generic freeing routine. The first argument is the **XDR** routine for the object being freed. The second argument is a pointer to the object itself. Note: the pointer passed to this routine is not freed, but what it points to is freed (recursively, depending on the **XDR** routine).

xdr_hyper() translates between **ANSI C** long long integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

xdr_int() translates between C integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

xdr_long() translates between C long integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

xdr_longlong_t() translates between **ANSI C** long long integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise. This routine is identical to ***xdr_hyper***().

xdr_quadruple() translates between IEEE quadruple precision floating point numbers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

xdr_short() translates between C short integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

xdr_u_char() translates between unsigned C characters and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

xdr_u_hyper() translates between unsigned **ANSI C** long long integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

xdr_u_int() is a primitive filter that translates between a C unsigned integer and its external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

xdr_u_long() translates between C unsigned long integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

xdr_u_longlong_t() translates between unsigned **ANSI C** long long integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise. This routine is identical to ***xdr_u_hyper***().

xdr_u_short() translates between C unsigned short integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.

xdr_void() always returns TRUE. It may be passed to **RPC** routines that require a function parameter, where nothing is to be done.

SEE ALSO

malloc(), rpc(), xdr_admin(), xdr_complex(), xdr_create()

xdr_admin
xdr_control
xdr_getpos
xdr_inline
xdr_setpos
xdr_sizeof
xdrrec_endofrecord
xdrrec_eof
xdrrec_readbytes
xdrrec_skiprecord

NAME

xdr_admin, xdr_control, xdr_getpos, xdr_inline, xdrrec_endofrecord, xdrrec_eof, xdrrec_readbytes, xdrrec_skiprecord, xdr_setpos, xdr_sizeof - library routines for external data representation

SYNOPSIS

#include <rpc/xdr.h>

bool_t	*xdr_control*	*(XDR *xdrs, int req, void *info);*
u_int	*xdr_getpos*	*(const XDR *xdrs);*
long	**xdr_inline*	*(XDR *xdrs, const int len);*
bool_t	*xdrrec_endofrecord*	*(XDR * xdrs, int sendnow);*
bool_t	*xdrrec_eof*	*(XDR *xdrs);*
int	*xdrrec_readbytes*	*(XDR *xdrs, caddr_t addr, u_int nbytes);*
bool_t	*xdrrec_skiprecord*	*(XDR *xdrs);*
bool_t	*xdr_setpos*	*(XDR *xdrs, const u_int pos);*
unsigned long	*xdr_sizeof*	*(xdrproc_t func, void *data);*

DESCRIPTION

XDR library routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Protocols such as remote procedure calls (**RPC**) use these routines to describe the format of the data. These routines deal specifically with the management of the **XDR** stream. See *rpc*() for the definition of the **XDR** data structure. Note that any buffers passed to the **XDR** routines must be properly aligned. It is suggested that *malloc*() be used to allocate these buffers or that the programmer insure that the buffer address is divisible evenly by four.

xdr_control(): A function macro to change or retrieve various information about an **XDR** stream. *req* indicates the type of operation and info is a pointer to the information. The supported values of req, their argument types and what they do are: **XDR_GET_BYTES_AVAIL** *xdr_bytesrec* * return number of bytes left unconsumed in the stream and a flag indicating whether or not this is the last fragment.

xdr_getpos(): A macro that invokes the get-position routine associated with the **XDR** stream, xdrs. The routine returns an unsigned integer, which indicates the position of the **XDR** byte stream. A desirable feature of **XDR** streams is that simple arithmetic works with this number, although the **XDR** stream instances need not guarantee this. Therefore, applications written for portability should not depend on this feature.

xdr_inline(): A macro that invokes the in-line routine associated with the **XDR** stream, xdrs. The routine returns a pointer to a contiguous piece of the stream's buffer; len is the byte length of the desired buffer. Note: pointer is cast to long *. Warning: *xdr_inline*() may return NULL (0) if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency, and applications written for portability should not depend on this feature.

xdrrec_endofrecord(): This routine can be invoked only on streams created by ***xdrrec_create***() (see ***xdr_create***()). The data in the output buffer is marked as a completed record, and the output buffer is optionally written out if sendnow is nonzero. This routine returns TRUE if it succeeds, FALSE otherwise.

xdrrec_eof(): This routine can be invoked only on streams created by ***xdrrec_create***(). After consuming the rest of the current record in the stream, this routine returns TRUE if there is no more data in the stream's input buffer. It returns FALSE if there is additional data in the stream's input buffer.

xdrrec_readbytes(): This routine can be invoked only on streams created by ***xdrrec_create***(). It attempts to read nbytes bytes from the **XDR** stream into the buffer pointed to by addr. On success this routine returns the number of bytes read, -1 on failure. A return value of 0 indicates an end of record.

xdrrec_skiprecord(): This routine can be invoked only on streams created by ***xdrrec_create***() (see ***xdr_create***()). It tells the **XDR** implementation that the rest of the current record in the stream's input buffer should be discarded. This routine returns TRUE if it succeeds, FALSE otherwise.

xdr_setpos(): A macro that invokes the set position routine associated with the **XDR** stream xdrs. The parameter pos is a position value obtained from ***xdr_getpos***(). This routine returns TRUE if the **XDR** stream was repositioned, and FALSE otherwise. Warning: it is difficult to reposition some types of **XDR** streams, so this routine may fail with one type of stream and succeed with another. Therefore, applications written for portability should not depend on this feature.

xdr_sizeof(): This routine returns the number of bytes required to encode data using the **XDR** filter function func, excluding potential overhead such as **RPC** headers or record markers. O is returned on error. This information might be used to select between transport protocols, or to determine the buffer size for various lower levels of **RPC** client and server creation routines, or to allocate storage when **XDR** is used outside of the **RPC** subsystem.

SEE ALSO

malloc(), rpc(), xdr_complex(), xdr_create(), xdr_simple()

rpc_broadcast_exp

NAME

rpc_broadcast_exp - broadcast a call message specifying timeout

SYNOPSIS

#include <rpc/rpc.h>

```
enum clnt_stat  rpc_broadcast_exp (   const u_long          prognum,
                                      const u_long          versnum,
                                      const u_long          procnum,
                                      const xdrproc_t       xargs,
                                      caddr_t               argsp,
                                      const xdrproc_t       xresults,
                                      caddr_t               resultsp,
                                      const resultproc_t    eachresult,
                                      const int             inittime,
                                      const int             waittime,
                                      char                  *nettype);
```

DESCRIPTION

This function is like ***rpc_broadcast***(), except that the initial timeout, *inittime*, and the maximum timeout, *waittime*, are specified in milliseconds.

inittime is the initial time that ***rpc_broadcast_exp***() waits before re-sending the request. After the first resend, the retransmission interval increases exponentially until it exceeds *waittime*.

SPARC COMPLIANCE DEFINITION 2.4.1 IS

libposix4

aio_cancel

NAME

aio_cancel - cancel asynchronous I/O request

SYNOPSIS

#include <aio.h>

int aio_cancel(int fildes, struct aiocb *aiocbp);

DESCRIPTION

The ***aio_cancel***() function attempts to cancel either one or all outstanding asynchronous I/O requests pending on the file descriptor specified by fildes.If *aiocbp* is **NULL**, then all such outstanding cancelable requests are canceled; otherwise, the individual request referenced by *aiocbp* references will be canceled. Normal completion notification occurs even for asynchronous I/O operations that are successfully canceled. If there are requests which cannot be canceled, then the normal asynchronous completion process takes place for those requests, and their associated *aiocb* structures are not modified.

```
struct aiocb {
        int                 aio_fildes;
        volatile void       *aio_buf;
        size_t              aio_nbytes;
        off_t               aio_offset;
        int                 aio_reqprio;
        struct sigevent     aio_sigevent;
        int                 aio_lio_opcode;
};
struct sigevent {
        int             sigev_notify;
        int             sigev_signo;
        union sigval    sigev_value;
};
union sigval    {
        int             sival_int;      /* integer value */
        void            *sival_ptr;     /* pointer value */
};
```

RETURN VALUES

If the requested operation(s) were canceled, ***aio_cancel***() returns **AIO_CANCELED**. But if at least one of the requested operation(s) cannot be canceled because it is in progress, then **AIO_NOTCANCELED** is returned, and the application may determine the state of affairs for these operation(s) by using ***aio_error***(). If all of the operation(s) had already completed, **AIO_ALLDONE** is returned. Otherwise, ***aio_cancel***() returns -1, and sets errno to indicate the error condition.

ERRORS

EBADF *fildes* is not a valid file descriptor.

ENOSYS The ***aio_cancel***() function is not supported.

SEE ALSO

aio_read(), aio_return(),

aio_error
aio_return

NAME

aio_return, aio_error - retrieve return or error status of asynchronous I/O operation

SYNOPSIS

```
#include <aio.h>

ssize_t aio_return(struct aiocb * aiocbp);

int aio_error(const struct aiocb * aiocbp);

struct aiocb {
        int                 aio_fildes;
        volatile void       *aio_buf;
        size_t              aio_nbytes;
        off_t               aio_offset;
        int                 aio_reqprio;
        struct sigevent     aio_sigevent;
        int                 aio_lio_opcode;
};
struct sigevent {
        int             sigev_notify;
        int             sigev_signo;
        union sigval    sigev_value;
};
union sigval    {
        int             sival_int;      /* integer value */
        void            *sival_ptr;     /* pointer value */
};
```

DESCRIPTION

The ***aio_return***() function returns the return status of the asynchronous I/O request associated with the *aiocb* structure pointed to by *aiocbp*. ***aio_error***() returns the error status of the asynchronous I/O request associated with the aiocb structure pointed to by *aiocbp*. The ***aio_return***() function should be called only once to retrieve the valid return status of a given asynchronous operation, after ***aio_error***() has returned a value other than **EINPROGRESS**.

RETURN VALUES

If the asynchronous I/O operation has completed successfully, ***aio_return***() returns the return status, as described for ***read***(), ***write***(), and ***fsync***(). If the asynchronous I/O operation has completed successfully, ***aio_error***() returns 0. If the operation has not yet completed, then **EINPROGRESS** is returned. If the asynchronous I/O operation has completed unsuccessfully, then the error status, as described for ***read***(), ***write***(), and ***fsync***() is returned. If unsuccessful, ***aio_return***() or ***aio_error***() return -1, and set errno to indicate the error condition.

ERRORS

The ***aio_return***() and ***aio_error***() functions will fail if:

EINVAL *aiocbp* does not reference an asynchronous operation which has completed or failed.

ENOSYS The ***aio_return***() or ***aio_error***() function is not supported.

SEE ALSO

close(), exec(), exit(), fork(), lseek(), read(), write(), aio_cancel(), aio_fsync(), aio_read(), fsync(), lio_listio(),

aio_fsync

NAME

aio_fsync - asynchronous file synchronization

SYNOPSIS

#include <aio.h>

int aio_fsync(int op, aiocb *aiocbp);

DESCRIPTION

The ***aio_fsync***() function queues an asynchronous ***fsync***() or ***fdatasync***() request for all the currently queued I/O operations on the file referenced by *aiocbp->aio_fildes*, and returns control immediately. This request is serviced concurrently with other activity of the process.If *op* is **O_DSYNC**, all I/O operations are completed by a call to ***fdatasync***() (synchronized I/O data integrity completion). If *op* is **O_SYNC**, all I/O operations are completed by a call to **fsync**() (synchronized I/O file integrity completion). (see ***fcntl***() definitions of **O_DSYNC** and **O_SYNC**.) When the request is queued, the error status for the operation is **EINPROGRESS**.When all data has been successfully transferred, the error status is reset to reflect the success or failure of the operation.***aio_return***() and ***aio_error***() may be used with this *aiocbp* value to monitor both the return and the error status of the asynchronous operation while it is proceeding. *aiocbp->aio_sigevent* defines the signal to be generated upon I/O completion. If *aiocbp->aio_sigevent.sigev_signo* is non-zero, then a signal will be generated when all I/O operations have achieved synchronized I/O completion.

```
struct aiocb {
        int                     aio_fildes;
        volatile void           *aio_buf;
        size_t                  aio_nbytes;
        off_t                   aio_offset;
        int                     aio_reqprio;
        struct sigevent         aio_sigevent;
        int                     aio_lio_opcode;
};
struct sigevent {
        int             sigev_notify;
        int             sigev_signo;
        union sigval    sigev_value;
};
union sigval    {
        int             sival_int;      /* integer value */
        void            *sival_ptr;     /* pointer value */
};
```

RETURN VALUES

If the I/O operation is successfully queued, ***aio_fsync***() returns 0.Otherwise, it returns -1, and sets errno to indicate the error condition.

ERRORS

The ***aio_fsync***() function will fail if:

EAGAIN The requested asynchronous operation was not queued due to temporary resource limitations.

EBADF *aiocbp->aio_fildes* is not a valid file descriptor open for writing.

EINVAL Synchronized I/O is not supported for this file.A value of *op* other than **O_DSYNC** or **O_SYNC** was specified.

ENOSYS ***aio_fsync***() is not supported by this implementation.

SEE ALSO

fcntl(), open(), read(), write(), aio_error(), aio_return(), fdatasync(), fsync(), fcntl(),

aio_read
aio_write

NAME

aio_read, aio_write - asynchronous read and write operations

SYNOPSIS

#include <aio.h>

int aio_read(struct aiocb *aiocbp);

int aio_write(struct aiocb *aiocbp);

DESCRIPTION

The ***aio_read***() function queues an asynchronous read request and returns control immediately. Rather than blocking until completion, the read operation continues concurrently with other activity of the process. Upon enqueueing the request, the calling process reads *aiocbp->nbytes* from the file referred to by *aiocbp->fildes* into the buffer pointed to by *aiocbp->aio_buf. aiocbp->offset* marks the absolute position from the beginning of the file (in bytes) at which the read begins. The ***aio_write***() function queues an asynchronous write request, and returns control immediately. Rather than blocking until completion, the write operation continues concurrently with other activity of the process. Upon enqueueing the request, the calling process writes aiocbp->nbytes from the buffer pointed to by *aiocbp->aio_buf* into the file referred to by a*iocbp->fildes*. If **O_APPEND** is set for *aiocbp->fildes*, ***aio_write***() operations append to the file in the same order as the calls were made. If **O_APPEND** is not set for the file descriptor, then the write operation will occur at the absolute position from the beginning of the file plus *aiocbp->offset* (in bytes). These asynchronous operations are submitted at a priority equal to the calling process' scheduling priority minus *aiocbp->aio_reqprio*.

For regular files, no data transfer will occur past the offset maximum established in the open file description associated with *aiocbp->fildes*. *aiocb->aio_sigevent* defines both the signal to be generated and how the calling process will be notified upon I/O completion. If *aio_sigevent.sigev_notify* is **SIGEV_NONE**, then no signal will be posted upon I/O completion, but the error status and the return status for the operation will be set appropriately. If *aio_sigevent.sigev_notify* is **SIGEV_SIGNAL**, then the signal specified in *aio_sigevent.sigev_signo* will be sent to the process. If the **SA_SIGINFO** flag is set for that signal number, then the signal will be queued to the process and the value specified in *aio_sigevent.sigev_value* will be the *si_value* component of the generated signal (see ***siginfo***()).

RETURN VALUES

If the I/O operation is successfully queued, ***aio_read***() and ***aio_write***() return 0; otherwise, they return -1, and set errno to indicate the error condition. *aiocbp* may be used as an argument to ***aio_error***() and ***aio_return***() in order to determine the error status and the return status of the asynchronous operation while it is proceeding.

ERRORS

The ***aio_read***() and ***aio_write***() function will fail if:

EAGAIN The requested asynchronous I/O operation was not queued due to system resource limitations.

ENOSYS The ***aio_read***() or ***aio_write***() functions are not supported.

EBADF If the calling function is ***aio_read***(), and *aiocbp->fildes* is not a valid file descriptor open for reading. If the calling function is ***aio_write***(), and *aiocbp->fildes* is not a valid file descriptor open for writing.

EINVAL +o The file offset value implied by *aiocbp->aio_offset* would be invalid, +o *aiocbp->aio_reqprio* is not a valid value, or +o *aiocbp->aio_nbytes* is an invalid value.

ECANCELED The requested I/O was canceled before the I/O completed due to an explicit ***aio_cancel***() request.

EINVAL The file offset value implied by *aiocbp->aio_offset* would be invalid.

The following are additional conditions which maybe detected synchronously or asynchronously:

aio_read()

OVERFLOW The file is a regular file, *aiocbp->aio_nbytes* is greater than 0 and the starting offset in *aiocbp->aio_offset* is before the end-of-file and is at or beyond the offset maximum in the open file description associated with *aiocbp->fildes*.

aio_write()

EFBIG The file is a regular file, *aiocbp->aio_nbytes* is greater than 0 and the starting offset in *aiocbp->aio_offset* is at or beyond the offset maximum in the open file description associated with *aiocbp->fildes*.

SEE ALSO

close(), exec(), exit(), fork(), lseek(), read(), write(), aio_cancel(), aio_return(), lio_listio(), siginfo()

NOTES

For portability, the application should set *aiocb->aio_reqprio* to 0.

aio_suspend

NAME

aio_suspend - wait for asynchronous I/O request

SYNOPSIS

#include <aio.h>

int aio_suspend(const struct aiocb * const list[], int nent, const struct timespec *timeout);

DESCRIPTION

The ***aio_suspend***() function suspends the caller until at least one of the asynchronous I/O operations referenced by list has completed, until a signal interrupts the function, or, if timeout is not NULL, until the time interval specified by timeout has passed. If any of the *aiocb* structures in the list corresponds to a completed asynchronous I/O operation (that is, the error status for the operation is not equal to **EINPROGRESS**), at the time of the call, the function returns without suspending the caller. If the time interval indicated in the *timespec* structure pointed to by timeout passes before any of the I/O operations referenced by list are completed, then ***aio_suspend***() returns with an error. The list argument is an array of pointers to asynchronous I/O control blocks. The *nent* argument indicates the number of elements in this array. Each *aiocb* structure pointed to must have been used in initiating an asynchronous I/O request via ***aio_read***(), ***aio_write***(), ***aio_fsync***(), or ***lio_listio***().This array may contain null pointers which will be ignored.

```
struct timespec {
        time_t      tv_sec;
        long        tv_nsec;
};
```

RETURN VALUES

If ***aio_suspend***() returns after one or more asynchronous I/O operations have completed, it returns 0. Otherwise, it returns -1, and sets errno to indicate the error condition. The application may determine which asynchronous I/O had completed with both the associated error and return status of ***aio_return***(), and ***aio_error***().

ERRORS

The ***aio_suspend***() function will fail if:

EAGAIN No asynchronous I/O indicated in the list referenced by list completed in the time interval indicated by timeout.

EINTR A signal interrupted the ***aio_suspend***() function. Note that, since each asynchronous I/O operation may possibly provoke a signal when it completes, this error return may be caused by the completion of one (or more) of the very I/O operations being awaited.

ENOSYS The ***aio_suspend***() function is not supported.

SEE ALSO

aio_fsync(), aio_read(), aio_return(), aio_write(), lio_listio(),

clock_settime
clock_gettime
clock_getres

NAME

clock_settime, clock_gettime, clock_getres - high-resolution clock operations

SYNOPSIS

#include <time.h>

int clock_settime(clockid_t clock_id, const struct timespec *tp);

int clock_gettime(clockid_t clock_id, struct timespec *tp);

int clock_getres(clockid_t clock_id, struct timespec *res);

struct timespec {time_t tv_sec; long tv_nsec; };

DESCRIPTION

clock_settime() sets the specified clock, *clock_id*, to the value specified by *tp*. The calling process must have an effective user ID of 0. ***clock_gettime***() returns the current value *tp* for the specified clock, *clock_id*. The resolution of any clock can be obtained by calling ***clock_getres***(). If *res* is not NULL, the resolution of the specified clock is stored in res. The *clock_id* for the real-time clock for the system is **CLOCK_REALTIME**.The values returned by ***clock_gettime***() and specified by ***clock_settime***() represent the amount of time (in seconds and nanoseconds) since 00:00 Universal Coordinated Time, January 1, 1970.

RETURN VALUES

clock_settime(), ***clock_gettime***(), and ***clock_getres***() return 0 upon success, otherwise they return -1 and set errno to indicate the error condition.

ERRORS

EINVAL *clock_id* does not specify a known clock. The tp argument to ***clock_settime***() is outside the range for the given clock id. The tp argument to ***clock_settime***() specified a nanosecond value less than zero or greater than or equal to 1,000,000,000.

ENOSYS ***clock_settime***(), ***clock_gettime***(), or ***clock_getres***() is not supported by this implementation.

EPERM The requesting process does not have the appropriate privilege to set the specified clock.

SEE ALSO

time(), ctime(), timer_gettime()

NOTES

Clock resolutions are implementation defined and are not settable by a process.Time values that are between two consecutive non-negative integer multiples of the resolution of the specified clock are truncated down to the smaller multiple of the resolution.

fdatasync

NAME

fdatasync - synchronize a file's data

SYNOPSIS

#include <unistd.h>

int fdatasync(int fildes);

DESCRIPTION

fdatasync() forces all currently queued I/O operations associated with the file descriptor fildes to synchronized I/O data integrity completion.See ***fcntl***() definition of **O_DSYNC**.

RETURN VALUES

fdatasync() returns 0 upon success; otherwise, it returns -1 and sets errno to indicate the error condition.

ERRORS

EBADF *fildes* is not a valid file descriptor.

EINVAL This implementation does not support synchronized I/O for this file.

ENOSYS ***fdatasync***() is not supported by this implementation.

In the event that any of the queued I/O operations fail, ***fdatasync***() returns the error conditions defined for ***read***() and ***write***().

SEE ALSO

fcntl(), open(), read(), write(), fsync(), aio_fsync(), fcntl()

NOTES

If ***fdatasync***() fails, outstanding I/O operations are not guaranteed to have been completed.

lio_listio

NAME

lio_listio - list directed I/O

SYNOPSIS

```
#include <aio.h>
int lio_listio(int mode, struct aiocb * const list[], int nent, struct sigevent *sig);

struct aiocb {
        int              aio_fildes;      /* file descriptor */
        volatile void    *aio_buf;        /* buffer location */
        size_t           aio_nbytes;      /* length of transfer */
        off_t            aio_offset;      /* file offset */
        int              aio_reqprio;     /* request priority offset */
        struct sigevent  aio_sigevent;    /* signal number and offset */
        int              aio_lio_opcode;  /* listio operation */
};

struct sigevent {
        int     sigev_notify;             /* notification mode */
        int     sigev_signo;              /* signal number */
        union   sigval   sigev_value;     /* signal value */
};

union sigval {
        int     sival_int;      /* integer value */
        void    *sival_ptr;     /* pointer value */
};
```

DESCRIPTION

The ***lio_listio***() function allows the calling process, **LWP**, or thread, to initiate a list of I/O requests within a single function call.

If mode is set to **LIO_WAIT**, ***lio_listio***() behaves synchronously, waiting until all I/O is completed, and the sig argument is ignored. If mode is set to **LIO_NOWAIT**, ***lio_listio***() behaves asynchronously; returning immediately, and signal delivery will occur, according to the sig argument, when all the I/O operations from this function complete. If *sig* is NULL, or the *sigev_signo* member of the sigevent structure referenced by *sig* is zero, then no signal delivery will occur. Otherwise, the signal number indicated by *sigev_signo* will be delivered when all the requests in list have completed.

list is an array of pointers to *aiocb* structures.This array consists of *nent* elements. The array may contain null pointers, which will be ignored.

The aio_lio_opcode field of each aiocb structure in list specifies the operation to be performed (see ***/usr/include/aio.h***).

LIO_READ requests ***aio_read***().

LIO_WRITE requests ***aio_write***().

LIO_NOP causes the list entry to be ignored.

nent specifies the length of the array (number of members of the list).

When mode has the value **LIO_WAIT**, a pointer to a signal control structure, sig, is used to define both the signal to be generated and how the calling process will be notified upon I/O completion. If *sig->sigev_notify* is **SIGEV_NONE**, then no signal will be posted upon I/O completion, but the error status and the return status for the operation will be set appropriately. If *sig->sigev_notify* is **SIGEV_SIGNAL**, then the signal specified in *sig->sigev_signo* will be sent to the process. If the **SA_SIGINFO** flag is set for that signal number, then the signal will be queued to the process and the value specified in *sig->sigev_value* will be the *si_value* component of the generated signal (see ***siginfo***()).

For regular files, no data transfer will occur past the offset maximum established in the open file description associated with *aiocbp->aio_fildes*.

The behavior of this function is altered according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion if synchronized I/O is enabled on the file associated with ***aio_fildes***. (see ***fcntl***() definitions of **O_DSYNC** and **O_SYNC**.)

RETURN VALUES

If the mode argument has the value **LIO_NOWAIT**, and the I/O operations are successfully queued, ***lio_listio***() returns 0; otherwise, it returns -1, and sets errno to indicate the error condition.

If the mode argument has the value **LIO_WAIT**, and when all the indicated I/O has completed successfully, ***lio_listio***() returns 0; otherwise, it returns -1, and sets errno to indicate the error condition.

In either case, the return value only indicates the success or failure of the ***lio_listio***() call itself, not the status of the individual I/O requests. In some cases, one or more of the I/O requests contained in the list may fail. Failure of an individual request does not prevent completion of any other individual request. To determine the outcome of each I/O request, the application must examine the error status associated with each aiocb control block. Each error status so returned is identical to that returned as a result of an ***aio_read***() or ***aio_write***() function.

ERRORS

The ***lio_listio***() function will fail if:

EAGAIN The resources necessary to queue all the I/O requests were not available.The error status for each request is recorded in the ***aio_error*** member of the corresponding *aiocb* structure, and can be retrieved using ***aio_error***(). The value of nent entries exceed the system-wide limit, **AIO_MAX**.

EINVAL The mode argument is an improper value.The value of *nent* is greater than **AIO_LISTIO_MAX**.

EINTR A signal was delivered while waiting for all I/O requests to complete during an **LIO_WAIT** operation.However, the outstanding I/O requests are not canceled. Use ***aio_fsync***() to determine if any request was initiated; ***aio_return***() to determine if any request has completed; or ***aio_error***() to determine if any request was canceled.

EIO One or more of the individual I/O operations failed.Using ***aio_error***() with each aiocb structure will determine the individual request(s) that failed.

ENOSYS ***lio_listio***() is not supported by this implementation.

If either ***lio_listio***() succeeds in queuing all of its requests, or errno is set to **EAGAIN**, **EINTR**, or **EIO**, then some of the I/O specified from the list may have been initiated. In this event, each aiocb structure contains errors specific to the ***read***() or ***write***() function being performed:

EAGAIN The requested I/O operation was not queued due to resource limitations.

ECANCELED The requested I/O was canceled before the I/O completed due to an explicit ***aio_cancel***() request.

EINPROGRESS The requested I/O is in progress.

The following are additional error codes which may be set for each aiocb control block:

EOVERFLOW The *aiocbp->aio_lio_opcode* is **LIO_READ**, the file is a regular file, *aiocbp->aio_nbytes* is greater than 0, and the *aiocbp->aio_offset* is before the end-of-file and is greater than or equal to the offset maximum in the open file description associated with *aiocbp->aio_fildes.*

EFBIG The *aiocbp->aio_lio_opcode* is **LIO_WRITE**, the file is a regular file, *aiocbp->aio_nbytes* is greater than 0, and the *aiocbp->aio_offse*t is greater than or equal to the offset maximum in the open file description associated with *aiocbp->aio_fildes*.

SEE ALSO

close(), exec(), exit(), fork(), lseek(), read(), write(), aio_cancel(), aio_fsync(), aio_read(), aio_return(), fcntl(), siginfo()

mq_close

NAME

mq_close - close a message queue

SYNOPSIS

#include <mqueue.h>

int mq_close(mqd_t mqdes);

DESCRIPTION

mq_close() removes the association between the message queue descriptor, mqdes, and its message queue.

If the process (or thread) has registered a notification request to the message queue via this *mqdes*, this registration is removed and the message queue is available for another process to attach for notification.

RETURN VALUES

Upon successful completion, ***mq_close***() returns 0; otherwise, the function returns -1 and sets errno to indicate the error condition.

ERRORS

EBADF *mqdes* is an invalid message queue descriptor.

ENOSYS ***sem_open***() is not supported by this implementation.

SEE ALSO

mq_notify(), mq_open(), mq_unlink()

mq_getattr
mq_setattr

NAME

mq_setattr, mq_getattr - set/get message queue attributes

SYNOPSIS

#include <mqueue.h>

int mq_setattr(mqd_t mqdes, const struct mq_attr *mqstat, struct mq_attr* omqstat);

int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);

struct mq_attr {long mq_flags; longmq_maxmsg; longmq_msgsize; longmq_curmsgs;...};

DESCRIPTION

mq_setattr() is used to set attributes associated with the message queue specified by mqdes. The message queue attributes corresponding to the following members defined in the mq_attr structure are set to the specified values upon successful completion of ***mq_setattr***(): *mq_flags* The value of this member is either 0 or **O_NONBLOCK**. The values of *mq_maxmsg*, *mq_msgsize*, and *mq_curmsgs* are ignored by ***mq_setattr***(). If *omqstat* is non-NULL, ***mq_setattr***() stores, in the location referenced by omqstat, the previous message queue attributes and the current queue status. These values are the same as would be returned by a call to ***mq_getattr***() at that point. ***mq_getattr***() is used to get status information and attributes associated with the message queue specified in mqdes. Upon return, the *mq_flags* member of the *mq_attr* structure referenced by *mqstat* has the value that was set when the message queue was created but also with modifications made by subsequent ***mq_setattr***() calls. The following attributes were set at message queue creation: *mq_maxmsg, mq_msgsize.* Upon return, the *mq_curmsgs* (the number of messages currently on the queue) member of the *mq_attr* structure referenced by *mqstat* is set according to the current state of the message queue.

RETURN VALUES

Upon successful completion, these function(s) return 0; otherwise, they return -1, and set errno to indicate the error condition. ***mq_setattr***(), if successful, also changes the attributes of the message queue as specified.

ERRORS

EBADF	*mqdes* is not a valid message queue descriptor.
ENOSYS	***mq_setattr***() and ***mq_getattr***() are not supported by this implementation.

SEE ALSO

mq_open(), mq_receive(), mq_send()

mq_notify

NAME

mq_notify - notify process (or thread) that a message is available on a queue

SYNOPSIS

#include <mqueue.h>

int mq_notify(mqd_t mqdes, const struct sigevent *notification);

DESCRIPTION

mq_notify() provides an asynchronous mechanism for processes to receive notice that messages are available in a message queue, rather than synchronously blocking (waiting) in ***mq_receive***(). If notification is not NULL, this function registers the calling process to be notified of message arrival at an empty message queue associated with the message queue descriptor, *mqdes*. The notification specified by notification will be sent to the process when the message queue becomes non-empty.At any time, only one process may be registered for notification by a specific message queue. Also, if the calling process or any other process has already registered for notification of message arrival at the specified message queue, subsequent attempts to register for that message queue will fail. *notification* points to a structure that defines both the signal to be generated and how the calling process will be notified upon I/O completion. If *notification->sigev_notify* is **SIGEV_NONE**, then no signal will be posted upon I/O completion, but the error status and the return status for the operation will be set appropriately. If *notification->sigev_notify* is **SIGEV_SIGNAL**, then the signal specified in *notification->sigev_signo* will be sent to the process. If the **SA_SIGINFO** flag is set for that signal number, then the signal will be queued to the process and the value specified in *notification->sigev_value* will be the *si_value* component of the generated signal (see ***siginfo***()). If notification is NULL and the process is currently registered for notification by the specified message queue, the existing registration is removed. The message queue is then available for future registration. When the notification is sent to the registered process, its registration is removed.The message queue is then be available for registration. If a process has registered for notification of message arrival at a message queue and some processes is blocked in ***mq_receive***() waiting to receive a message when a message arrives at the queue, the arriving message will be received by the appropriate ***mq_receive***(), and no notification will be sent to the registered process. The resulting behavior is as if the message queue remains empty, and this notification will not be sent until the next arrival of a message at this queue. Any notification registration is removed if the calling process either closes the message queue or exits.

RETURN VALUES

Upon successful completion, ***mq_notify***() returns 0; otherwise, it returns a value of -1 and sets errno to indicate the error condition.

ERRORS

EBADF	*mqdes* is not a valid message queue descriptor.
EBUSY	A process is already registered for notification by the message queue.
ENOSYS	***mq_notify***() is not supported by this implementation.

SEE ALSO

mq_close(), mq_open(), mq_receive(), mq_send(), siginfo()

mq_open

NAME

mq_open - open a message queue

SYNOPSIS

#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag, /* unsigned long mode, mq_attr attr */...);

```
struct mq_attr {
        long    mq_flags;       /* message queue flags */
        long    mq_maxmsg;      /* maximum number of messages */
        long    mq_msgsize;     /* maximum message size */
        long    mq_curmsgs;     /* number of messages currently queued */
        ...
};
```

DESCRIPTION

mq_open() establishes a connection to a named message queue, name, returning the address of the message queue descriptor to the caller for subsequent calls to ***mq_send***() or ***mq_receive***().The message queue once opened remains usable by this process until the message queue is closed by a successful call to ***mq_close***(), ***exit***(), or ***exec***().

name points to a string naming a message queue.The name argument must conform to the construction rules for a pathname. If name is not the name of an existing message queue and its creation is not requested, ***mq_open***() fails and returns an error. The first character of name must be a slash (/) character and the remaining characters of name cannot include any slash characters. For maximum portability, name should include no more than 14 characters, but this limit is not enforced.

oflag requests the desired receive and/or send access to the message queue.The requested access permission to receive messages or send messages is granted if the calling process would be granted read or write access, respectively, to a file with the equivalent permissions.

The value of oflag is the bitwise inclusive OR of values from the following list. Applications must specify exactly one of the first three values (access modes) below in the value of *oflag*:

O_RDONLY Open the message queue for receiving messages.The process can use the returned message queue descriptor with ***mq_receive***(), but not ***mq_send***(). A message queue may be open multiple times in the same or different processes for receiving messages.

O_WRONLY Open the queue for sending messages. The process can use the returned message queue descriptor with ***mq_send***() but not ***mq_receive***().A message queue may be open multiple times in the same or different processes for sending messages.

O_RDWR Open the queue for both receiving and sending messages.The process can use any of the functions allowed for **O_RDONLY** and **O_WRONLY**. A message queue may be open multiple times in the same or different processes for sending messages.

Any combination of the remaining flags may additionally be specified in the value of *oflag*:

O_CREAT This option is used to create a message queue, and it requires two additional arguments: *mode*, which is of type *mode_t*, and *attr*, which is pointer to a *mq_attr* structure. If the pathname, name, has already been used to create a message queue that still exists, then this flag has no effect, unless combined with **O_EXCL** (see below). Otherwise, a message queue is created without any messages in it.

The message queue's user ID is set to the process's effective user ID, and the message queue's group ID is set to the process's effective group ID. The message queue's permission bits will be set to the value of mode, and modified by clearing all bits set in the file mode creation mask of the process (see ***umask***()). **AND-NOT** those already set in the file mode creation mask of the process.

If attr is NULL, the message queue is created with the default message queue attributes,(*mq_maxmsg*=128 and *mq_maxsize* = 1024). If *attr* is non-NULL, the message queue mq_maxmsg and *mq_msgsize* attributes are set to the values of the corresponding members in the *mq_attr* structure referred to by attr.

O_EXCL If both **O_EXCL** and **O_CREAT** are set, ***mq_open***() will fail if the message queue name exists. The check for the existence of the message queue and the creation of the message queue if it does not exist are atomic with respect to other processes executing ***mq_open***() naming the same name with both **O_EXCL** and **O_CREAT** set.

O_NONBLOCK The setting of this flag is associated with the open message queue descriptor and determines whether a calling ***mq_send***() waits for message buffer space or a calling ***mq_receive***() waits for messages that are not currently available; or whether the calling function fails, thereby setting *errno* to **EAGAIN**.

RETURN VALUES

Upon successful completion, ***mq_open***() returns a message queue descriptor; otherwise the function returns (*mqd_t*)(-1) and sets errno to indicate the error condition.

ERRORS

EACCESS The message queue exists and the permissions specified by *oflag* are denied, or the message queue does not exist and permission to create the message queue is denied.

EEXISTO_CREAT and **O_EXCL** are set and the named message queue already exists.

EINTR The ***mq_open***() operation was interrupted by a signal.

EINVAL name is not a valid name.

O_CREAT was specified in *oflag*, the value of attr is not NULL, and either *mq_maxmsg* or *mq_msgsize* was less than or equal to zero.

EMFILE The number of open message queue descriptors in this process exceeds **MQ_OPEN_MAX**.

The number of open file descriptors in this process exceeds **OPEN_MAX**.

ENAMETOOLONG The length of the name string exceeds **PATH_MAX**, or a pathname component is longer than **NAME_MAX** while **_POSIX_NO_TRUNC** is in effect.

ENFILE The system file table is full

ENOENTO_CREAT is not set and the named message queue, name, does not exist.

ENOSPC There is insufficient space for the creation of the new message queue.

ENOSYS ***mq_open***() is not supported by this implementation.

SEE ALSO

exec(), exit(), umask(), mq_close(), mq_receive(), mq_send(), mq_setattr(), mq_unlink(), sysconf()

NOTES

Message queues are based on shared memory. Although permissions to send and receive messages are checked by the ***mq_receive***() and ***mq_send***() interfaces, any application which can open the message queue can directly access the shared memory to examine and manipulate messages in the queue. Thus message queues should not be considered secure.

mq_receive

NAME

mq_receive - receive a message from a message queue

SYNOPSIS

#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio);

struct mq_attr {longmq_flags; long mq_maxmsg;longmq_msgsize;longmq_curmsgs; ...};

DESCRIPTION

The ***mq_receive***() function is used to receive the oldest of the highest priority message(s) from the message queue specified by *mqdes*. If the size of the buffer in bytes, specified by *msg_len*, is less than the *mq_msgsize* member of the message queue, the function fails and returns an error. Otherwise, the selected message is removed from the queue and copied to the buffer pointed to by *msg_ptr*. If *msg_prio* is not NULL, the priority of the selected message is stored in the location referenced by *msg_prio*. If the specified message queue is empty and **O_NONBLOCK** is not set in the message queue description associated with mqdes, (see ***mq_open***() and ***mq_setattr***()), ***mq_receive***() blocks, waiting until a message is enqueued on the message queue, or until ***mq_receive***() is interrupted by a signal. If more than one process (or thread) is waiting to receive a message when a message arrives at an empty queue, then the process of highest priority that has been waiting the longest is selected to receive the message.If the specified message queue is empty and **O_NONBLOCK** is set in the message queue description associated with mqdes, no message is removed from the queue, and ***mq_receive***() returns an error.

RETURN VALUES

Upon successful completion, ***mq_receive***() returns the length of the selected message in bytes and the message will have been removed from the queue.Otherwise, no message is removed from the queue, the function returns a value of -1, and sets errno to indicate the error condition.

ERRORS

The ***mq_receive***() function will fail if:

EAGAINO_NONBLOCK	was set in the message description associated with mqdes, and the specified message queue is empty.
EBADF	The mqdes argument is not a valid message queue descriptor open for reading.
EMSGSIZE	The *msg_len* argument is less than the message size member of the message queue.
EINTR	The ***mq_receive***() function operation was interrupted by a signal.
ENOSYS	The ***mq_receive***() function is not supported by this implementation.

SEE ALSO

mq_open(), mq_send(), mq_setattr()

mq_send

NAME

mq_send - send a message to a message queue

SYNOPSIS

#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio);

struct mq_attr {long mq_flags;long mq_maxmsg; long mq_msgsize;long mq_curmsgs;...};

DESCRIPTION

mq_send() adds the message pointed to by *msg_ptr* to the message queue specified by *mqdes*. *msg_len* specifies the length of the message in bytes pointed to by *msg_ptr*. The value of *msg_len* must be less than or equal to the *mq_msgsize* attribute of the message queue, or ***mq_send***() will fail. If the specified message queue is not full, ***mq_send***() behaves as if the message is inserted into the message queue at the position indicated by *msg_prio*.A message with a larger numeric value of *msg_prio* is inserted before messages with lower values of *msg_prio*. A message is inserted after other messages in the queue, if any, with equal msg_prio priority. The value of *msg_prio* must be greater than 0, and less than or equal to **MQ_PRIO_MAX**. If the specified message queue is full and if **O_NONBLOCK** is not set in the message queue description associated with mqdes (see ***mq_open***() and ***mq_setattr***()), ***mq_send***() blocks, waiting until space becomes available to enqueue the message, or until ***mq_send***() is interrupted by a signal.If more than one process (or thread) is waiting to send when space becomes available in the message queue, then the process of the highest priority which has been waiting the longest is unblocked to send its message. If the specified message queue is full and **O_NONBLOCK** is set in the message queue description associated with *mqdes*, the message is not queued, and ***mq_send***() returns an error.

RETURN VALUES

Upon successful completion, ***mq_send***() returns a value of 0; otherwise, no message is enqueued, the function returns -1, and sets errno to indicate the error condition.

ERRORS

EAGAIN	The **O_NONBLOCK** flag is set in the message queue description associated with *mqdes*, and the specified message queue is full.
EBADF	*mqdes* is not a valid message queue descriptor open for writing.
EINTR	A signal interrupted the call to ***mq_send***()
EMSGSIZE	The specified message length, *msg_len*, exceeds the message size attribute of the message queue.
ENOSYS	***mq_send***() is not supported by this implementation.

SEE ALSO

mq_open(), mq_receive(), mq_setattr(), sysconf()

mq_unlink

NAME

mq_unlink - remove a message queue

SYNOPSIS

#include <mqueue.h>

int mq_unlink(const char *name);

DESCRIPTION

mq_unlink() removes the message queue named by name.After a successful call to ***mq_unlink***() with name, a call to ***mq_open***() with the same name will fail if the flag **O_CREAT** is not set in flags. If one or more processes have the message queue open when ***mq_unlink***() is called, destruction of the message queue is postponed until all references to the message queue have been closed.Calls to ***mq_open***() to re-create the message queue may fail until the message queue is actually removed. However, ***mq_unlink***() does not block (wait) until all references have been closed; it returns immediately.

RETURN VALUES

Upon successful completion, ***mq_unlink***() returns a value of 0; otherwise, the named message queue is not changed by this function call, the function returns a value of -1 and sets errno to indicate the error condition.

ERRORS

EACCESS Permission is denied to unlink the named message queue.

ENAMETOOLONG The length of the name string exceeds **PATH_MAX**, or a pathname component is longer than **NAME_MAX** while **_POSIX_NO_TRUNC** is in effect.

ENOENT The named message queue, name, does not exist.

ENOSYS ***mq_unlink***() is not supported by this implementation.

SEE ALSO

mq_close(), mq_open()

nanosleep

NAME

nanosleep - high resolution sleep

SYNOPSIS

#include <time.h>

int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);

```
struct timespec {
        time_t      tv_sec;      /* seconds */
        long        tv_nsec;     /* and nanoseconds */
};
```

DESCRIPTION

nanosleep() suspends the current thread from execution until either the time interval specified by *rqtp* has elapsed, or a signal is delivered to the calling thread and its action is to invoke a signal-catching function or to terminate the thread. The suspension time may be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activity by the system. Except for the case of being interrupted by a signal, the suspension time will not be less than the time specified by *rqtp*, as measured by the system clock, **CLOCK_REALTIME**. ***nanosleep***() will not block nor effect the action of any signal.

RETURN VALUES

If ***nanosleep***() returns because the requested time has elapsed, it returns 0. Otherwise, if it returns because it has been interrupted by a signal: it returns -1 and sets errno to indicate the interruption. If *rmtp* is non-NULL, the timespec structure referenced by *rmtp* will be updated to contain the remaining amount of time between *rqtp* and the time actually slept. If any of the following error conditions occur, ***nanosleep***() returns -1 and sets errno to indicate the error condition.

ERRORS

EINTR	***nanosleep***() was interrupted by a signal.
EINVAL	*rqtp* specified a nanosecond value less than zero or greater than or equal to 1,000,000,000.
ENOSYS	***nanosleep***() is not supported by this implementation.

SEE ALSO

sleep()

sched_get_priority_max
sched_get_priority_min
sched_rr_get_interval

NAME

sched_get_priority_max,sched_get_priority_min, sched_rr_get_interval - get scheduling parameter limits

SYNOPSIS

#include <sched.h>

int sched_get_priority_max(int policy);

int sched_get_priority_min(int policy);

int sched_rr_get_interval(pid_t pid, struct timespec *interval);

struct timespec {time_t tv_sec; long tv_nsec;};

DESCRIPTION

sched_get_priority_max() and ***sched_get_priority_min***() return the appropriate maximum or minimum values, respectively, for the scheduling policy specified by policy. ***sched_rr_get_interval***() updates the *timespec* structure referenced by interval to contain the current execution time limit (i.e., time quantum) for the process specified by *pid* under the **SCHED_RR** policy. After that time limit expires, when another process at the same priority is ready to execute, a scheduling decision will be made. If *pid* is zero, the current execution time limit for the calling process is stored in interval. The value of policy must be one of the scheduling policy values defined in <*sched.h*>: **SCHED_FIFO**, **SCHED_RR**, or **SCHED_OTHER**.

RETURN VALUES

If successful, ***sched_get_priority_max***() or ***sched_get_priority_min***() returns the appropriate maximum or minimum values, respectively. If successful, ***sched_rr_get_interval***() returns 0. If unsuccessful, these functions return -1, and set errno to indicate the error condition.

ERRORS

EINVAL The value of policy does not represent a defined scheduling policy.

ENOSYS ***sched_get_priority_max***(), ***sched_get_priority_min***(), and ***sched_rr_get_interval***() are not supported by this implementation.

ESRCH No process can be found corresponding to that specified by *pid*.

SEE ALSO

sched_setparam(), ***sched_setscheduler***()

sched_getparam, sched_setparam

NAME

sched_setparam, sched_getparam - set/get scheduling parameters

SYNOPSIS

#include <sched.h>

int sched_setparam(pid_t pid, const struct sched_param *param);

int sched_getparam(pid_t pid, struct sched_param *param);

struct sched_param {int sched_priority; /* process execution scheduling priority */}

DESCRIPTION

sched_setparam() sets the scheduling parameters of the process specified by *pid* to the values specified by the *sched_param* structure referenced by *param*. ***sched_getparam***() stores the scheduling parameters of a process, specified by *pid*, in the *sched_param* structure pointed to by *param*. If the target process has as its scheduling policy, **SCHED_FIFO** or **SCHED_RR**: If *pid* is zero, the scheduling parameters are set/stored for the calling process. Otherwise, if a process specified by *pid* exists and if the calling process has permission, the scheduling parameters are set/stored for the process whose process ID is equal to *pid*. The real or effective user ID of the calling process must match the real or saved (from ***exec***()) user ID of the target process unless the effective user ID of the calling process is 0. The target process, *pid*, whether it is running or not running, resumes execution after all other runnable processes of equal or greater priority have been scheduled to run. If the priority of the process, *pid*, is set higher than that of the lowest priority running process, and if process *pid* is ready to run, then process *pid* preempts a lowest priority running process. Similarly, if the process calling ***sched_setparam***() sets its own priority lower than that of one or more other non-empty process lists, then the process that is the head of the highest priority list preempts the calling process. Thus, in either case, the originating process might not receive notification of the completion of the requested priority change until the higher priority process has executed. The value of *param->sched_priority* must be an integer within the inclusive priority range for the current scheduling policy of the process specified by *pid*. Higher numerical values for the priority represent higher priorities.

RETURN VALUES

If successful, ***sched_setparam***() and ***sched_getparam***() returns 0; otherwise, the priority remains unchanged, the function returns -1, and sets errno to indicate the error condition.

ERRORS

EINVAL One or more of ***sched_setparam***()'s requested scheduling parameters is outside the range defined for the specified pid's scheduling policy.

ENOSYS ***sched_setparam***() and ***sched_getparam***() are not supported by this implementation.

EPERM The requesting process does not have permission to set/get the scheduling parameters for the specified process, or does not have the appropriate privilege to invoke ***sched_setparam***().

ESRCH No process can be found corresponding to that specified by *pid*.

SEE ALSO

exec(), ***sched_setscheduler***()

sched_getscheduler, sched_setscheduler

NAME

sched_setscheduler, sched_getscheduler - set/get scheduling policy and scheduling parameters

SYNOPSIS

#include <sched.h>

int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);

int sched_getscheduler(pid_t pid);

DESCRIPTION

sched_setscheduler() sets the scheduling policy and scheduling parameters of the process specified by *pid* to policy and the parameters specified in the *sched_param* structure pointed to by param, respectively. The value of *param->sched_priority* must be any integer with in the inclusive priority range for the scheduling policy specified by policy. The possible values for the policy parameter are defined in the header file <*sched.h*>: **SCHED_FIFO**, **SCHED_RR**, or **SCHED_OTHER**. If pid is zero, the scheduling policy and scheduling parameters are set for the calling process. Otherwise, if a process specified by *pid* exists and if the calling process has permission, the scheduling policy and scheduling parameters are set for the process whose process ID is equal to pid. The real or effective user ID of the calling process must match the real or saved (from ***exec***()) user ID of the target process unless the effective user ID of the calling process is super-user. To change the policy of any process to either of the real time policies **SCHED_FIFO** or **SCHED_RR**, the calling process must either have the **SCHED_FIFO**, or **SCHED_RR** policy or have an effective user **ID** of 0. ***sched_getscheduler***() returns the scheduling policy of the process specified by *pid*. If pid is zero, the scheduling policy is returned for the calling process. Otherwise, if a process specified by pid exists and if the calling process has permission, the scheduling policy is returned for the process whose process **ID** is equal to *pid*.

RETURN VALUES

If successful, ***sched_setscheduler***() returns the former scheduling policy of the specified process (*pid*), which will be one of the following values: **SCHED_FIFO** (realtime): First-In-First-Out; processes scheduled to this policy, if not preempted by a higher priority or interrupted by a signal, will proceed until completion. **SCHED_RR** (realtime): Round-Robin; processes scheduled to this policy, if not preempted by a higher priority or interrupted by a signal, will execute for a time period, returned by ***sched_rr_get_interval***() or by the system. or **SCHED_OTHER** (time-sharing). Otherwise, the policy and scheduling parameters remain unchanged, ***sched_setscheduler***() returns -1, and sets errno to indicate the error condition. If successful, ***sched_getscheduler***() returns the scheduling policy of the specified process; otherwise, it returns -1, and sets errno to indicate the error condition.

ERRORS

EINVAL The value of policy is invalid, or one or more of the parameters contained in param is outside the valid range for the specified scheduling policy.

ENOSYS ***sched_setscheduler***() and ***sched_getscheduler***() are not supported by this implementation.

EPERM ***sched_setscheduler***() does not have permission to set either or both of the scheduling parameters or the scheduling policy of the specified process.

ESRCH No process can be found corresponding to that specified by *pid*.

sched_yield

NAME

sched_yield - yield processor

SYNOPSIS

#include <sched.h>

int sched_yield(void);

DESCRIPTION

sched_yield() forces the running process to relinquish the processor until the process again becomes the head of its process list.

RETURN VALUES

If successful, ***sched_yield***() returns 0, other wise, it returns -1, and sets errno to indicate the error condition.

ERRORS

ENOSYS	***sched_yield***() is not supported by this implementation.

sem_close

NAME

sem_close - close a named semaphore

SYNOPSIS

#include <semaphore.h>

int sem_close(sem_t *sem);

typedef struct {...} sem_t; ***/*opaque POSIX.4 semaphore*/***

DESCRIPTION

sem_close() is used to indicate that the calling process is finished using the named semaphore *sem*. ***sem_close***() de-allocates any system resources for use by this process for this semaphore.If the semaphore has not been removed with a successful call to ***sem_unlink***(), then ***sem_close***() has no effect on the state of the semaphore. If ***sem_unlink***() has been successfully invoked for name after the most recent call to ***sem_open***() with **O_CREAT** for this semaphore, then when all processes that have opened the semaphore close it, the semaphore will no longer be accessible.

sem_close() should not be called for an unnamed semaphore initialized by ***sem_init***().

RETURN VALUES

If successful, ***sem_close***() returns 0, otherwise it returns -1 and sets errno to indicate the error condition.

ERRORS

EINVAL *sem* is not a valid semaphore descriptor.

ENOSYS ***sem_close***() is not supported by this implementation.

SEE ALSO

sem_init(), sem_open(), sem_unlink()

sem_destroy

NAME

sem_destroy - destroy an unnamed semaphore

SYNOPSIS

#include <semaphore.h>

int sem_destroy(sem_t *sem);

typedef struct {...} sem_t; ***/*opaque POSIX.4 semaphore*/***

DESCRIPTION

sem_destroy() is used to destroy the unnamed semaphore, *sem*, which was initialized by ***sem_init***().

RETURN VALUES

If successful, ***sem_destroy***() returns 0, otherwise it returns -1 and sets errno to indicate the error condition.

ERRORS

EINVAL *sem* is not a valid semaphore.

ENOSYS ***sem_destroy***() is not supported by this implementation.

EBUSY Other processes (or **LWPs** or threads) are currently blocked on the semaphore.

SEE ALSO

sem_init(), sem_open()

sem_getvalue

NAME

sem_getvalue - get the value of a semaphore

SYNOPSIS

#include <semaphore.h>

int sem_getvalue(sem_t *sem, int *sval);

typedef struct {...} sem_t; ***/*opaque POSIX.4 semaphore*/***

DESCRIPTION

sem_getvalue() updates the location referenced by sval to have the value of the semaphore referenced by *sem* without affecting the state of the semaphore.The updated value represents an actual semaphore value that occurred at some unspecified time during the call to ***sem_getvalue***(), but may not be the actual value of the semaphore when ***sem_getvalue***() is returned to the caller.

The value set in *sval* may be zero or positive. If sval is zero, there may be other processes (or **LWPs** or threads) waiting for the semaphore; if *sval* is positive, no one is waiting.

RETURN VALUES

If successful, ***sem_getvalue***() returns 0, otherwise, it returns -1, and sets errno to indicate the error condition.

ERRORS

EINVAL	*sem* does not refer to a valid semaphore.
ENOSYS	***sem_getvalue***() is not supported by this implementation.

SEE ALSO

sem_post(), sem_wait()

sem_init

NAME

sem_init - initialize an unnamed semaphore

SYNOPSIS

#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);

typedef struct {...} sem_t; /*opaque POSIX.4 semaphore*/

DESCRIPTION

sem_init() is used to initialize the unnamed semaphore, referred to by *sem*, to value. This semaphore may be used in subsequent calls to ***sem_wait***(), ***sem_trywait***(), ***sem_post***(), and ***sem_destroy***(). This semaphore remains usable until the semaphore is destroyed.

If *pshared* is non-zero, then the semaphore is sharable between processes.If the semaphore is not being shared between processes, the application should set pshared to 0.

RETURN VALUES

If successful, ***sem_init***() returns 0 and initializes the semaphore in *sem*; otherwise it returns -1 and sets errno to indicate the error condition.

ERRORS

EINVAL value exceeds **SEM_VALUE_MAX**.

ENOSPC A resource required to initialize the semaphore has been exhausted.

The resources have reached the limit on semaphores, **SEM_NSEMS_MAX**.

ENOSYS ***sem_init***() is not supported by this implementation.

EPERM The calling process lacks the appropriate privileges to initialize the semaphore.

SEE ALSO

sem_destroy(), sem_post(), sem_wait()

sem_open

NAME

sem_open - initialize/open a named semaphore

SYNOPSIS

#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag,

/* unsigned long mode, unsigned int value */...);

typedef struct {...} sem_t; /*opaque POSIX.4 semaphore*/

DESCRIPTION

sem_open() establishes a connection to a semaphore, name, returning the address of the semaphore to the calling process (or **LWP** or thread) for subsequent calls to ***sem_wait***(), ***sem_trywait***(), ***sem_post***(), and ***sem_close***(). The semaphore remains usable by this process until the semaphore is closed.

name points to a string naming a semaphore object. The *name* argument should conform to the construction rules for a pathname. If a process makes multiple successful calls to ***sem_open***() with the same value for *name*, the same semaphore address will be returned for each such successful call, provided that there have been no calls to ***sem_unlink***() for this semaphore. The first character of *name* must be a slash (/) character and the remaining characters of *name* cannot include any slash characters. For maximum portability, *name* should include no more than 14 characters, but this limit is not enforced.

oflag determines whether the semaphore is created or merely accessed by the call to ***sem_open***(). The three valid values for *oflag* are 0, **O_CREAT**, or the bitwise inclusive OR of **O_CREAT** and **O_EXCL**. Setting the oflag bits to **O_CREAT** will create the semaphore if it does not already exist. Setting both **O_CREAT** and **O_EXCL** will fail if the semaphore already exists. The check for the existence of the semaphore and the creation of the semaphore if it does not exist is atomic with respect to other processes executing ***sem_open***(). After the semaphore named name has been created by ***sem_open***() with the **O_CREAT** flag, other processes can connect to this semaphore by calling ***sem_open***() with the same value of name, and no bits set in *oflag*.

Using the **O_CREAT** flag requires a third and a fourth argument: mode and value. The semaphore is created with an initial count of value. value must be less than or equal to **SEM_VALUE_MAX**. The semaphore's user ID acquires the effective user ID of the process; the semaphore's group ID is set to a system default group ID or to the effective group ID of the process. The semaphore's permission bits is set to the value of mode, modified by clearing all bits set in the file creation mask of the process (see ***umask***()).

RETURN VALUES

If successful, ***sem_open***() returns the address of the semaphore, otherwise it returns -1 and sets errno to indicate the error condition.

ERRORS

EACCES The named semaphore exists and the **O_RDWR** permissions are denied, or the named semaphore does not exist and permission to create the named semaphore is denied.

EEXISTO_CREAT and **O_EXCL** are set and the named semaphore already exists.

EINTR ***sem_open***() was interrupted by a signal.

EINVAL *name* is not a valid name.

O_CREAT was set in *oflag* and value is greater than **SEM_VALUE_MAX**.

EMFILE The number of open semaphore descriptors in this process exceeds **SEM_NSEMS_MAX**.

The number of open file descriptors in this process exceeds **OPEN_MAX**.

ENAMETOOLONG The string-length of name exceeds **PATH_MAX**, or a pathname component is longer than **NAME_MAX** while **_POSIX_NO_TRUNC** is in effect.

ENFILE The system file table is full.

ENOENTO_CREAT is not set and the named semaphore does not exist.

ENOSPC There is insufficient space for the creation of the new named semaphore.

ENOSYS ***sem_open***() is not supported by this implementation.

SEE ALSO

exec(), exit(), umask(), sem_close(), sem_post(), sem_unlink(), sem_wait(), sysconf()

sem_post

NAME

sem_post - increment the count of a semaphore

SYNOPSIS

#include <semaphore.h>

int sem_post(sem_t *sem);

typedef struct {...} sem_t /*opaque POSIX.4 semaphore*/

DESCRIPTION

If, prior to the call to ***sem_post***(), the value of sem was 0, and other processes (or **LWPs** or threads) were blocked waiting for the semaphore, then one of them will be allowed to return successfully from its call to ***sem_wait***(). The process to be unblocked will be chosen in a manner appropriate to the scheduling policies and parameters in effect for the blocked processes. In the case of the policies **SCHED_FIFO** and **SCHED_RR**, the highest priority waiting process is unblocked, and if there is more than one highest-priority process blocked waiting for the semaphore, then the highest priority process which has been waiting the longest is unblocked.

If, prior to the call to ***sem_post***(), no other processes (or **LWPs** or thread) were blocked for the semaphore, then its value is incremented by one.

sem_post() is reentrant with respect to signals (**ASYNC-SAFE**), and may be invoked from a signal-catching function. The semaphore functionality described on this man page is for the **POSIX** threads implementation.

RETURN VALUES

If successful, ***sem_post***() returns 0, otherwise it returns - 1, and sets errno to indicate the error condition.

ERRORS

EINVAL	*sem* does not refer to a valid semaphore.
ENOSYS	***sem_post***() is not supported by this implementation.

SEE ALSO

sched_setscheduler(), sem_wait(), semaphore()

NOTES

sem_wait() and ***sem_trywait***() decrement the semaphore upon their successful return.

sem_wait
sem_trywait

NAME

sem_wait, ***sem_trywait*** - acquire or wait for a semaphore

SYNOPSIS

#include <semaphore.h>

int sem_wait(sem_t *sem);

int sem_trywait(sem_t *sem);

typedef struct {...} sem_t; ***/*opaque POSIX.4 semaphore*/***

DESCRIPTION

sem_wait() and ***sem_trywait***() are the functions by which a calling thread waits or proceeds depending upon the state of a semaphore. A synchronizing process can proceed only if the value of the semaphore it accesses is currently greater than 0. If at the time of a call to either ***sem_wait***() or ***sem_trywait***(), the value of *sem* is positive, these functions decrement the value of the semaphore, return immediately, and allow the calling process to continue. If the semaphore's value is 0:

> ***sem_wait***() blocks, awaiting the semaphore to be released by another process (or **LWP** or thread).
>
> ***sem_trywait***() fails, returning immediately.

RETURN VALUES

If at the time of a call to either ***sem_wait***() or ***sem_trywait***(), the value of *sem* is positive, these functions return 0 on success. If the call was unsuccessful, the state of the semaphore is unchanged, the calling function returns -1, and sets errno to indicate the error condition.

ERRORS

EAGAIN	The value of sem was 0 when ***sem_trywait***() was called.
EINVAL	*sem* does not refer to a valid semaphore.
EINTR	***sem_wait***() was interrupted by a signal.
ENOSYS	***sem_wait***() and ***sem_trywait***() are not supported by this implementation.
EDEADLK	A deadlock condition was detected; i.e., two separate processes are waiting for an available resource to be released via a semaphore “held” by the other process.

SEE ALSO

sem_post()

NOTES

sem_wait() can be interrupted by a signal, which may result in its premature return.

sem_post() increments the semaphore upon its successful return.

sem_unlink

NAME

sem_unlink - remove a named semaphore

SYNOPSIS

#include <semaphore.h>

int sem_unlink(const char *name);

DESCRIPTION

sem_unlink() removes the semaphore named by the string name. If the semaphore, name, is currently referenced by other processes, then ***sem_unlink***() has no effect on the state of the semaphore. If one or more processes have the semaphore open when ***sem_unlink***() is called, destruction of the semaphore is postponed until all references to the semaphore have been destroyed by calls to ***sem_close***(), ***exit***(), or ***exec***().Calls to ***sem_open***() to re-create or re-connect to the semaphore will refer to a new semaphore after ***sem_unlink***() is called. ***sem_unlink***() does not block until all references have been destroyed; rather, it returns immediately.

RETURN VALUES

If successful, ***sem_unlink***() returns 0; otherwise, the function returns -1, sets errno to indicate the error condition, and the semaphore is left unchanged.

ERRORS

EACCES Permission is denied to unlink the named semaphore.

ENAMETOOLONG The string-length of name exceeds **PATH_MAX**, or a pathname component is longer than **NAME_MAX** while **_POSIX_NO_TRUNC** is in effect.

ENOENT The named semaphore does not exist.

ENOSYS ***sem_unlink***() is not supported by this implementation.

SEE ALSO

exec(), exit(), sem_close(), sem_open()

shm_open

NAME

shm_open - open a shared memory object

SYNOPSIS

#include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode);

DESCRIPTION

shm_open() either opens a file descriptor for the shared memory object with the name referenced by name. If successful, ***shm_open***() returns a file descriptor for the shared memory object that is the lowest numbered file descriptor not currently open for that process. Since the open file description is new, the new file descriptor is not as yet shared with any other processes.

name points to a string naming a shared memory object.The name argument should conform to the construction rules for a pathname. If a process makes multiple successful calls to ***shm_open***(), with the same value for name, the same semaphore address will be returned for each successful call, provided that there have been no calls to ***sem_unlink***() for this semaphore. The first character of name must be a slash (/) character and the remaining characters of name cannot include any slash characters. For maximum portability, name should include no more than 14 characters, but this limit is not enforced.

The file status flags and file access modes of the open file descriptor are set according to the value of oflag: the bitwise inclusive OR of the following flags, defined in the header <*fcntl.h*>. (Applications must specify exactly one of the first two values below in the value of oflag):

O_RDONLY Open for read access only.

O_RDWR Open for read or write access. Any combination of the remaining flags may be bitwise inclusive OR- ed with the value of *oflag*:

O_CREAT If name does not exist, the shared memory object is created, it's user ID is set to the effective user ID of the process, and it's group ID is set to a system default group ID or to the effective group ID of the process.The shared memory object's permission bits will be set to the value of mode, modified by clearing all bits set in the file mode creation mask of the process mode does not affect whether the shared memory object is opened for reading, for writing, or for both.The new shared memory object has a size of zero. If the shared memory object does exist, this flag will have no effect, except as specified under **O_EXCL** below.

O_EXCL If both **OEXCL** and **O_CREAT** are set, ***shm_open***() fails if the shared memory object, name, exists. The check for the existence of the shared memory object and the creation of the object if it does not exist is atomic with respect to other processes executing ***shm_open***() naming the same shared memory object with **OEXCL** and **O_CREAT** set.

O_TRUNC If the shared memory object exists, and it is successfully opened **O_RDWR**, the object is truncated to zero length and the mode and ownership are unchanged by this function call.

RETURN VALUES

If successful, ***shm_open***() returns a non negative integer representing the lowest numbered unused file descriptor, otherwise it returns -1 and sets errno to indicate the error condition.

ERRORS

EACCES The shared memory object exists and the permissions specified by *oflag* are denied, or the shared memory object does not exist and permission to create the shared memory object is denied, or **O_TRUNC** is specified and write per mission is denied.

EEXIST **O_CREAT** and **O_EXCL** are set and the named shared memory object already exists.

EINTR The ***shm_open***() operation was interrupted by a signal.

EINVAL *name* is an invalid file description.

EMFILE The number of open file descriptors in this process exceeds **OPEN_MAX**.

ENAMETOOLONG T he length of the name string exceeds **PATH_MAX**, or a pathname component is longer than **NAME_MAX** while **_POSIX_NO_TRUNC** is in effect.

ENFILE The system file table is full

ENOENT **O_CREAT** is not set and the named shared memory object does not exist.

ENOSPC There is insufficient space for the creation of the new shared memory object.

ENOSYS ***shm_open***() is not supported by this implementation.

FILES

/usr/include/fcntl.h

SEE ALSO

close(), dup(), exec(), fcntl(), mmap(), umask(), shm_unlink(), sysconf()

NOTES

When a shared memory object is created, the state of the shared memory object, including all data associated with the shared memory object, persists until the shared memory object is unlinked and all other references are gone.

shm_unlink

NAME

shm_unlink - remove a shared memory object

SYNOPSIS

int shm_unlink(const char *name);

DESCRIPTION

shm_unlink() removes the name of the shared memory object named by the string pointed to by name. If one or more references to the shared memory object exists when the object is unlinked, the name is removed before ***shm_unlink***() returns, but the removal of the memory object contents will be postponed until all open and mapped references to the shared memory object have been removed.

RETURN VALUES

If successful, ***shm_unlink***() returns 0, otherwise it returns -1 and sets errno to indicate the error condition, and the named shared memory object is not affected by this function.

ERRORS

EACCES Permission is denied to unlink the named shared memory object.

ENAMETOOLONG The length of the name string exceeds **PATH_MAX**, or a pathname component is longer than **NAME_MAX** while **_POSIX_NO_TRUNC** is in effect.

ENOENT The named shared memory object does not exist.

ENOSYS ***shm_unlink***() is not supported by this implementation.

SEE ALSO

close(), mmap(), mlock(), shm_open()

sigqueue

NAME

sigqueue - queue a signal to a process

SYNOPSIS

#include <signal.h>

int sigqueue(pid_t pid, int signo, const union sigval value);

union sigval {int sival_int;void*sival_ptr;};

DESCRIPTION

sigqueue() causes the signal, *signo* to be sent with the value, value to the process, pid. If signo is zero (the null signal), error checking is performed, but no signal is actually sent.The null signal can be used to check the validity of *pid*. The conditions required for a process to have permission to queue a signal to another process are the same as for ***kill***().

If resources were not available to queue the signal, ***sigqueue***() exits and returns immediately. If **SA_SIGINFO** is set for signo in the receiving process, and if the resources were available, the signal is left queued and pending. If **SA_SIGINFO** is not set for *signo*, then signo is sent at least once to the receiving process. If the value of *pid* causes *signo* to be generated for the sending process, and if *signo* is not blocked, either signo or at least the pending, unblocked signal with the lowest number will be delivered to the sending process before ***sigqueue***() returns.

RETURN VALUES

If successful, ***sigqueue***() returns 0, and queues the specified signal.Otherwise, ***sigqueue***() returns -1 and sets errno to indicate the error condition.

ERRORS

EAGAIN No resources are available to queue the signal.

The process has already queued {**SIGQUEUE_MAX**} signals that are still pending at the receiver(s), or a system wide resource limit has been exceeded.

EINVAL The value of *signo* is an invalid or unsupported signal number.

ENOSYS ***sigqueue***() is not supported by this implementation.

EPERM The process does not have the appropriate privilege to send the signal to the receiving process.

ESRCH The process *pid* does not exist.

SEE ALSO

kill(), sigwaitinfo(), siginfo(), signal()

sigwaitinfo
sigtimedwait

NAME

sigwaitinfo, sigtimedwait - wait for queued signals

SYNOPSIS

#include <signal.h>

int sigwaitinfo(const sigset_t *set, siginfo_t *info);

int sigtimedwait(const sigset_t *set, siginfo_t *info, const struct timespec *timeout);

typedef struct siginfo {int si_signo; int si_code;...; int si_value; ... } siginfo_t;

struct timespec {time_t tv_sec; long tv_nsec;};

DESCRIPTION

sigwaitinfo() and ***sigtimedwait***() select the pending signal from the set specified by set. When multiple signals are pending, the lowest numbered one will be selected.The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified. If no signal in set is pending at the time of the call, ***sigwaitinfo***() suspends the calling process until one or more signals in set become pending or until it is interrupted by an unblocked, caught signal. ***sigtimedwait***(), on the other hand, suspends itself for the time interval specified in the timespec structure referenced by timeout. If the timespec structure pointed to by timeout is zero-valued, and if none of the signals specified by set are pending, then ***sigtimedwait***() returns immediately with the error **EAGAIN**. If, while ***sigwaitinfo***() or ***sigtimedwait***() is waiting, a signal occurs which is eligible for delivery (i.e., not blocked by the process signal mask), that signal is handled asynchronously and the wait is interrupted. If info is non-NULL, the selected signal number is stored in *si_signo*, and the cause of the signal is stored in the *si_code*. If any value is queued to the selected signal, the first such queued value is dequeued and, if info is non-NULL, the value is stored in the *si_value* member of info. The system resource used to queue the signal is released and made available to queue other signals. If the value of the *si_code* member is **SI_NOINFO**, only the *si_signo* member of *siginfo_t* is meaningful, and the value of all other members is unspecified. If no further signals are queued for the selected signal, the pending indication for that signal is reset.

RETURN VALUES

If one of the signals specified by set is either pending or generated, ***sigwaitinfo***() or ***sigtimedwait***() returns the selected signal number. Otherwise, the function returns -1 and sets errno to indicate the error condition.

ERRORS

EINTR	The wait was interrupted by an unblocked, caught signal.
ENOSYS	***sigwaitinfo***() or ***sigtimedwait***() is not supported by this implementation.
EAGAIN	No signal specified by set was delivered within the specified timeout period.
EINVAL	timeout specified a *tv_nsec* value less than 0 or greater than 1,000,000,000.

SEE ALSO

time(), sigqueue(), siginfo(), signal()

timer_create

NAME

timer_create - create a timer

SYNOPSIS

#include <signal.h>

#include <time.h>

int timer_create(clockid_t clock_id, struct sigevent *evp, timer_t *timerid);

DESCRIPTION

timer_create() creates a timer using the specified clock, *clock_id*, as the timing base. This timer ID is unique and meaningful only within the calling **LWP** until the timer is deleted.This timer is initially disarmed upon return from ***timer_create***(). The timer may be created per-**LWP** or per-process. Expiration signals for a per-**LWP** timer will be sent to the creating **LWP**. Expiration signals for a per-process timer will be sent to the process.A per-**LWP** timer will be automatically deleted when the creating **LWP** exits. By default, timers are created per-**LWP**.If the symbol **_POSIX_PER_PROCESS_TIMER_SOURCE** is defined or the symbol **_POSIX_C_SOURCE** is defined to have a value greater than 199500L before the inclusion of <*time.h*>, timers will be created per-process. If *evp* is non-NULL: then *evp* points to a *sigevent* structure, allocated by the application, which defines the asynchronous notification that will occur when the timer expires. If the *sigev_notify* member of evp is **SIGEV_SIGNAL**, then the structure also contains the signal number and the application specific data value to be sent to the process. If **SA_SIGINFO** is set for the expiration signal, then the signal and application-defined value specified in the structure will be queued to the process on timer expiration. If **SA_SIGINFO** is not set for the expiration signal, then the signal specified in the structure will be sent upon the timer expiration. If the *sigev_notify* member is **SIGEV_NONE**, no notification will be sent. If *evp* is NULL, and **SA_SIGINFO** is set for the expiration signal, then the default signal, **SIGALRM**, will be queued to the process and the signal data value will be set to the timer ID.

RETURN VALUES

timer_create() returns 0 upon success and creates a *timer_t*, *timerid*, which can be passed to the timer calls; otherwise it returns -1 and sets errno to indicate the error condition.

ERRORS

EAGAIN	The system lacks sufficient signal queuing resources to honor the request.
EINVAL	The specified clock ID, *clock_id*, is not defined.
ENOSYS	***timer_create***() is not supported by this implementation.

SEE ALSO

exec(), fork(), time(), clock_settime(), signal(), timer_delete(), timer_settime()

timer_delete

NAME

timer_delete - delete a per-**LWP** timer

SYNOPSIS

#include <time.h>

int timer_delete(timer_t timerid);

DESCRIPTION

timer_delete() deletes the specified timer, timerid, previously created by ***timer_create***(). If the timer is armed when ***timer_delete***() is called, the behavior is as if the timer is automatically disarmed before removal.

RETURN VALUES

timer_delete() returns 0 upon success, otherwise it returns -1 and sets errno to indicate the error condition.

ERRORS

EINVAL *timerid* does not refer to a valid timer.

ENOSYS ***timer_delete***() is not supported by this implementation.

SEE ALSO

timer_create()

timer_gettime
timer_settime
timer_getoverrun

NAME

timer_settime, timer_gettime, timer_getoverrun-high-resolution timer operations

SYNOPSIS

```
#include <time.h>

int timer_settime(timer_t timerid, int flags, const struct itimerspec *value, struct itimerspec *ovalue);

int timer_gettime(timer_t timerid, struct itimerspec *value);

int timer_getoverrun(timer_t timerid);


struct itimerspec {
        struct timespec    it_interval;      /* timer period */
        struct timespec    it_value;         /* timer expiration */
};

struct timespec{
        time_t             tv_sec;           /* seconds */
        long               tv_nsec;          /* and nanoseconds */
};
```

DESCRIPTION

If *value->it_value* is non-zero, ***timer_settime***() arms the timer, timerid, to next expire after the time designated by *value->it_value*. Upon expiration, an application-specified notification (see ***timer_create***()) or the default signal, **SIGALRM**, is queued for the calling **LWP**. If timerid was already armed when ***timer_settime***() is called, this call resets the time until the next expiration to the value of *value->it_value*. If *value->it_value* is zero, then the timer is disarmed.

value->it_value may be expressed as either an absolute or relative time.If flags is set to **TIMER_RELTIME**, then the timer will initially expire relative to when the call is made. If flags is set to **TIMER_ABSTIME**, then the initial expiration will be relative to 00:00 Universal Coordinated Time, January 1, 1970. If the specified (absolute) time has already passed, ***timer_settime***() succeeds and the expiration notification is made.

If *value->it_interval* is non-zero, then timerid, will be a “periodic” timer, to be reloaded to expire every *value->it_interval* seconds (nanoseconds).Otherwise, if *value->it_interval* is zero and *value->it_value* is non-zero, then timerid is a “one-shot” timer, which will expire only at the time specified by *value->it_value*.

If *ovalue* is not NULL, and timer timerid had previously been used, then ***timer_settime***() will store the remaining time until the previous timer expires in *ovalue->it_value*, and the previous reload interval in *ovalue->it_interval.* (If the previous timer was disarmed, *ovalue->it_value* will be set to zero). The values store dino value by ***timer_settime***() are the same values that would have been returned by a call to ***timer_gettime***(*timerid*,...).

timer_gettime() stores the amount of time until the specified timer, timerid, expires into *value->it_value*, and the timer's reload value into *value->it_interval*.

Only a single signal can be queued to the **LWP** for a given timer at any point in time. When a timer, for which a signal is still pending expires, (from a previous interval), no signal will be queued, and a “timer overrun count” will be incremented. When a timer expiration signal is delivered to an **LWP**, ***timer_overrun***() may be used to determine the timer expiration overrun count for the specified timer. The overrun count returned contains the number of extra timer expirations which occurred between the time the signal was generated (queued) and when it was delivered, up to but not including a maximum of {**DELAYTIMER_MAX**}. If the number of such extra expirations is greater than or equal to {**DELAYTIMER_MAX**}, then the overrun count is set to {**DELAYTIMER_MAX**}.The value returned by ***timer_getoverrun***() applies to the most recent expiration signal delivery for the timer.

RETURN VALUES

timer_settime(), and ***timer_gettime***() return 0 upon success. If ***timer_getoverrun***() succeeds, the number of extra timer expirations which occurred between the time the signal was queued and when it was delivered is returned. If these functions fail, they return -1 and set errno to indicate the error condition.

ERRORS

EINVAL *timerid* does not correspond to a timer returned by ***timer_create***().

The timer, *timerid*, had already been deleted by ***timer_delete***().

A value structure specified a nanosecond value less than zero or greater than or equal to 1,000,000,000.

ENOSYS ***timer_settime***(), ***timer_gettime***(), or ***timer_getoverrun***() is not supported by this implementation.

SEE ALSO

clock_settime(), timer_create(), timer_delete()

SPARC COMPLIANCE DEFINITION 2.4.1 IS

libsocket

accept

NAME

accept - accept a connection on a socket

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

int accept (int s, struct sockaddr *addr, int *addrlen);

DESCRIPTION

The argument *s* is a socket that has been created with ***socket***() and bound to an address with ***bind***(), and that is listening for connections after a call to ***listen***(). accept extracts the first connection on the queue of pending connections, creates a new socket with the properties of *s*, and allocates a new file descriptor, *ns*, for the socket. If no pending connections are present on the queue and the socket is not marked as non-blocking, accept blocks the caller until a connection is present. If the socket is marked as non-blocking and no pending connections are present on the queue, accept returns an error as described below. accept uses the ***netconfig***() file to determine the **STREAMS** device file name associated with *s*. This is the device on which the connect indication will be accepted. The accepted socket, *ns*, is used to read and write data to and from the socket that connected to *ns*; it is not used to accept more connections. The original socket (s) remains open for accepting further connections.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity as it is known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication occurs.

addrlen is a value-result parameter. Initially, it contains the amount of space pointed to by *addr*; on return it contains the length in bytes of the address returned.

accept is used with connection-based socket types, currently with **SOCK_STREAM**.

It is possible to ***poll***(BA_OS) a socket for the purpose of an accept by polling it for a read. However, this will only indicate when a connect indication is pending; it is still necessary to call accept.

RETURN VALUES

accept returns -1 on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.

ERRORS

accept will fail if:

EBADF	The descriptor is invalid.
ENODEV	The protocol family and type corresponding to *s* could not be found in the ***netconfig*** file.
ENOMEM	There was insufficient user memory available to complete the operation.
ENOSR	There were insufficient **STREAMS** resources available to complete the operation.

ENOTSOCK	The descriptor does not reference a socket.
EOPNOTSUPP	The referenced socket is not of type **SOCK_STREAM**.
EPROTO	A protocol error has occurred; for example, the **STREAMS** protocol stack has not been initialized.
EWOULDBLOCK	The socket is marked as non-blocking and no connections are present to be accepted.

bind

NAME

bind - bind a name to a socket

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

int bind (int s, struct sockaddr *name, int namelen);

DESCRIPTION

bind assigns a name to an unnamed socket, *s*. When a socket is created with ***socket***(), it exists in a name space (address family) but has no *name* assigned. bind requests that the name pointed to by *name* be assigned to the socket. *namelen* specifies the size of *name*.

RETURN VALUES

If the ***bind*** is successful, a 0 value is returned. A return value of -1 indicates an error, which is further specified in the global errno.

ERRORS

The ***bind*** call will fail if:

EADDRINUSE	The specified address is already in use.
EADDRNOTAVAIL	The specified address is not available on the local machine.
EBADF	*s* is not a valid descriptor.
EINVAL	*namelen* is not the size of a valid address for the specified address family.
EINVAL	The socket is already bound to an address.
ENOSR	There were insufficient **STREAMS** resources for the operation to complete.
ENOTSOCK	*s* is a descriptor for a file, not a socket.

connect

NAME

connect - initiate a connection on a socket

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

int connect(int s, struct sockaddr *name, int namelen);

DESCRIPTION

The parameter *s* is a socket. If it is of type **SOCK_DGRAM**, connect specifies the peer with which the socket is to be associated; this address is the address to which datagrams are to be sent if a receiver is not explicitly designated; it is the only address from which datagrams are to be received. If the socket *s* is of type **SOCK_STREAM**, connect attempts to make a connection to another socket. The other socket is specified by *name*. *name* is an address in the communication space of the socket. *namelen* specifies the size of data structure pointed to by *name*. Each communication space interprets the *name* parameter in its own way. If *s* is not bound, then it will be bound to an address selected by the underlying transport provider. Generally, stream sockets may successfully connect only once; datagram sockets may use connect multiple times to change their association. Datagram sockets may dissolve the association by connecting to a null address.

RETURN VALUES

If the connection or binding succeeds, then 0 is returned. Otherwise a -1 is returned and sets errno to indicate the error.

ERRORS

The call fails if:

EADDRINUSE	The address is already in use.
EADDRNOTAVAIL	The specified address is not available on the remote machine.
EAFNOSUPPORT	Addresses in the specified address family cannot be used with this socket.
EALREADY	The socket is non-blocking and a previous connection attempt has not yet been completed.
EBADF	*s* is not a valid descriptor.
ECONNREFUSED	The attempt to connect was forcefully rejected. The calling program should ***close***(BA_OS) the socket descriptor, and issue another ***socket***() call to obtain a new descriptor before attempting another connect call.
EINPROGRESS	The socket is non-blocking and the connection cannot be completed immediately. It is possible to ***poll***(BA_OS) for completion by polling the socket for writing. However, this is only possible if the socket **STREAMS** module is the topmost module on the protocol stack with a write service procedure. This will be the normal case.
EINTR	The connection attempt was interrupted before any data arrived by the

delivery of a signal.

EINVAL *namelen* is not the size of a valid address for the specified address family.

EISCONN The socket is already connected.

ENETUNREACH The network is not reachable from this host.

ENOSR There were insufficient **STREAMS** resources available to complete the operation.

gethostbyname, gethostbyaddr

NAME

gethostbyname, gethostbyaddr - get network host entry

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

#include <netdb.h>

struct hostent *gethostbyname (char *name);

struct hostent *gethostbyaddr (struct in_addr *addr,
const sizeof (struct in_addr), const int AF_INET);

DESCRIPTION

gethostbyaddr, and ***gethostbyname*** each return a host entry. The entry comes from the system's hosts database. The lookup order is unspecified. ***gethostbyname*** searches for a host entry with a given hostname. ***gethostbyaddr*** searches for a host entry with a given hostaddress. The internal representation of a host entry is a structure defined in <*netdb.h*> with the following members:

char	**h_name;*
char	***h_aliases;*
int	*h_addrtype;*
int	*h_length;*
char	***h_addr_list;*

Host addresses are supplied in network byte order.

RETURN VALUES

gethostbyname and ***gethostbyaddr*** return a pointer to a struct hostent if they successfully locate the requested entry; otherwise they return NULL, and set an integer *h_errno* to indicate one of these errors: **HOST_NOT_FOUND**, **TRY_AGAIN**, **NO_RECOVERY**, **NO_DATA** and **NO_ADDRESS** (see */usr/include/netdb.h*).

NOTES

All information is contained in a static area so it must be copied if it is to be saved.

getpeername

NAME

getpeername - get name of connected peer

SYNOPSIS

int getpeername(int s, struct sockaddr *name, int *namelen);

DESCRIPTION

getpeername returns the name of the peer connected to socket *s*. The int pointed to by the *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the *name* returned (in bytes). The *name* is truncated if the buffer provided is too small.

RETURN VALUES

If successful, ***getpeername*** returns 0; otherwise it returns -1 and sets errno to indicate the error.

ERRORS

The call succeeds unless:

EBADF	The argument *s* is not a valid descriptor.
ENOMEM	There was insufficient user memory for the operation to complete.
ENOSR	There were insufficient **STREAMS** resources available for the operation to complete.
ENOTCONN	The socket is not connected.
ENOTSOCK	The argument *s* is not a socket.

getprotobyname, getprotobynumber, getprotoent

NAME

getprotobyname, getprotobynumber, getprotoent - get protocol entry

SYNOPSIS

#include <netdb.h>

struct protoent *getprotobyname (char *name);

struct protoent *getprotobynumber (int proto);

struct protoent *getprotoent (void);

DESCRIPTION

getprotoent, ***getprotobyname***, and ***getprotobynumber*** each return a protocol entry. The entry may comes from the system's protocols databas*e. name* is a pointer to one of the strings "tcp", "udp", or "icmp". *proto* is one of the values 6 (***tcp***), 17 (***udp***), 0 (***ip***), or 1 (***icmp***).

getprotoent enumerates protocol entries: successive calls to ***getprotoent*** will return either successive protocol entries or NULL. Enumeration may not be supported by some sources.

The internal representation of a protocol entry is a protoent structure defined in **<***netdb.h***>** with the following members:

char	**p_name;*
char	***p_aliases;*
int	*p_proto;*

RETURN VALUES

getprotobyname and ***getprotobynumber*** return a pointer to a struct protoent if they successfully locate the requested entry; otherwise they return NULL.

getprotoent returns a pointer to a *struct protoent* if it successfully enumerates an entry; otherwise it returns NULL, indicating the end of the enumeration.

NOTES

All information is contained in a static area so it must be copied if it is to be saved.

Use of ***getprotoent*** is deprecated.

getservbyname, getservbyport

NAME

getservbyname, getservbyport - get service entry

SYNOPSIS

#include <netdb.h>

struct servent *getservbyname (char *name, char *proto);

struct servent *getservbyport (int port, char *proto);

DESCRIPTION

getservbyname, and ***getservbyport*** each return a service entry. The entry come from the system's services database. ***getservbyname*** searches for a service entry with a given service name.

getservbyport searches for a service entry with a given port number and, if the protocol name is non-NULL, the protocol.

name is a pointer to one of the strings "tcp" or "udp". *port* is the number of a well-known port.

The internal representation of a service entry is a struct servent defined in <*netdb.h*> with the following members:

char	**s_name;*
char	***s_aliases;*
int	*s_port;*
char	**s_proto;*

RETURN VALUES

getservbyname and ***getservbyport*** return a pointer to a struct servent if they successfully locate the requested entry; otherwise they return NULL.

NOTES

All information is contained in a static area, so it must be copied if it is to be saved.

getsockname

NAME

getsockname - get socket name

SYNOPSIS

#include <sys/types.h>

#include <sys/sockets.h>

***int getsockname(int s, struct sockaddr *name, int *namelen)**;*

DESCRIPTION

getsockname returns the current name for socket *s*. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size in bytes of the *name* returned.

RETURN VALUES

If successful, ***getsockname*** returns 0; otherwise it returns -1 and sets errno to indicate the error.

ERRORS

The call succeeds unless:

EBADF	The argument *s* is not a valid file descriptor.
ENOMEM	There was insufficient memory available for the operation to complete.
ENOSR	There were insufficient **STREAMS** resources available for the operation to complete.
ENOTSOCK	The argument *s* is not a socket.

inet_lnaof. inet_makeaddr, inet_network

NAME

inet_network, inet_makeaddr, inet_lnaof - Internet address manipulation

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

```
int               inet_network (char *cp);
struct in_addr    inet_makeaddr (int net, int lna);
int               inet_lnaof (struct in_addr in);
```

DESCRIPTION

The ***inet_network*** routine interprets a character string, *cp*, representing numbers expressed in the Internet standard "." notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively. The routine ***inet_makeaddr*** takes an Internet network number, *net*, and a local network address, *lna*, and constructs an Internet address from it. The routine ***inet_lnaof*** break apart an Internet host address, *in*, returning the network number and local network address part, respectively.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the '.' notation take one of the following forms: a.b.c.d, a.b.c, a.b, a . When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as "*128.net.host*". When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as "net.host". When only one part is given, the value is stored directly in the network address without any byte rearrangement. All numbers supplied as "parts" in a '.' notation may be decimal, octal, or hexadecimal, as specified in the C language (that is, a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

RETURN VALUES

The value -1 is returned by ***inet_network*** for malformed requests.

The routine ***inet_lnaof*** break apart Internet host addresses, returning the network number and local network address part, respectively.

listen

NAME

listen - listen for connections on a socket

SYNOPSIS

#include <sys/types.h>

#include <sys/sockets.h>

int listen(int s, int backlog);

DESCRIPTION

To accept connections, a socket, *s*, is first created with ***socket***(), a *backlog* for incoming connections is specified with listen and then the connections are accepted with ***accept***(). The ***listen*** call applies only to sockets of type **SOCK_STREAM** or **SOCK_SEQPACKET**.

The *backlog* parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full, the client will receive an error with an indication of **ECONNREFUSED**.

RETURN VALUES

A 0 return value indicates success; -1 indicates an error.

ERRORS

The call fails if:

EBADF	The argument *s* is not a valid file descriptor.
ENOTSOCK	The argument *s* is not a socket.
EOPNOTSUPP	The socket is not of a type that supports the operation listen.

NOTES

There is currently no backlog limit.

recv , recvfrom , recvmsg

NAME

recv, *recvfrom*, *recvmsg* - receive a message from a socket

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

#include <sys/uio.h>

int recv (int s, char *buf, int len, int flags);

int recvfrom (int s, char *buf, int len, int flags, struct sockaddr *from, int *fromlen);

int recvmsg (int s, struct msghdr *msg, int flags);

DESCRIPTION

recv, recvfrom, and recvmsg are used to receive messages from another socket. recv may be used only on a connected socket (see ***connect***()), while recvfrom and recvmsg may be used to receive data on a socket whether it is in a connected state or not. *s* is a socket created with ***socket***(). *buf* is a pointer to the buffer to receive the data and *len* is its size in bytes.

If *from* is not a NULL pointer, the source address of the message is filled in. *fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there. The length of the message is returned. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see ***socket***()).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see ***fcntl***(BA_OS)) in which case -1 is returned with the external variable errno set to **EWOULDBLOCK**.

The poll call may be used to determine when more data arrives.

The *flags* parameter is formed by ORing one or more of the following:

MSG_OOB Read any out-of-band data present on the socket rather than the regular in-band data.

MSG_PEEK Peek at the data present on the socket; the data is returned, but not consumed, so that a subsequent receive operation will see the same data.

The recvmsg call uses a struct msghdr, *msg*, to minimize the number of directly supplied parameters. This structure is defined in <*sys/socket.h*> and includes the following members:

caddr_t	*msg_name;*	*/* optional address */*
int	*msg_namelen;*	*/* size of address */*
struct iovec	**msg_iov;*	*/* scatter/gather array */*
int	*msg_iovlen;*	*/* # elements in msg_iov */*
caddr_t	*msg_accrights;*	*/* access rights sent/received */*
int	*msg_accrightslen;*	

Here *msg_name* and *msg_namelen* specify the destination address if the socket is unconnected; *msg_name* may be given as a NULL pointer if no names are desired or required. The msg_iov and *msg_iovlen* describe the scatter-gather locations, as described in ***read***(BA_OS). A buffer to receive any access rights sent along with the message is specified in *msg_accrights*, which has length *msg_accrightslen*.

RETURN VALUES

These calls return the number of bytes received, or -1 if an error occurred.

ERRORS

The calls fail if:

EBADF — *s* is an invalid file descriptor.

EINTR — The operation was interrupted by delivery of a signal before any data was available to be received.

ENOMEM — There was insufficient user memory available for the operation to complete.

ENOSR — There were insufficient **STREAMS** resources available for the operation to complete.

ENOTSOCK — *s* is not a socket.

EWOULDBLOCK — The socket is marked non-blocking and the requested operation would block.

send, sendto, sendmsg

NAME

send, sendto, sendmsg - send a message from a socket

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

int send (int s, char *buf, int len, int flags);

int sendto (int s, char *buf, int len, int flags, struct sockaddr *to, int tolen);

int sendmsg (int s, struct msghdr *msg, int flags);

DESCRIPTION

send, ***sendto***, and ***sendmsg*** are used to transmit a message to another transport end-point. send may be used only when the socket is in a connected state, while sendto and sendmsg may be used at any time. *s* is a socket created with ***socket***(). *buf* points to a buffer containing the data to be sent. *len* is number of bytes to be sent. The address of the target is given by *to* with *tolen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, then the error **EMSGSIZE** is returned, and the message is not transmitted. A return value of -1 indicates locally detected errors only. It does not implicitly mean the message was not delivered. If the socket does not have enough buffer space available to hold the message being sent, send blocks, unless the socket has been placed in non-blocking I/O mode (see ***fcntl***(BA_OS)). The poll call may be used to determine when it is possible to send more data. The flags parameter is formed from the bit-wise OR of zero or more of the following:

MSG_OOB	Send out-of-band data on sockets that support this notion. The underlying protocol must also support out-of-band data. Only **SOCK_STREAM** sockets created in the **AF_INET** address family support out-of-band data.
MSG_DONTROUTE	The **SO_DONTROUTE** option is turned on for the duration of the operation. It is used only by diagnostic or routing programs.

See ***recv***() for a description of the msghdr structure.

RETURN VALUES

These calls return the number of bytes sent, or -1 if an error occurred.

ERRORS

The calls fail if:

EBADF	*s* is an invalid file descriptor.
EINTR	The operation was interrupted by delivery of a signal before any data could be buffered to be sent.
EINVAL	*tolen* is not the size of a valid address for the specified address family.
EMSGSIZE	The socket requires that message be sent atomically, and the message was too long.

ENOMEM There was insufficient memory available to complete the operation.

ENOSR There were insufficient **STREAMS** resources available for the operation to complete.

ENOTSOCK *s* is not a socket.

EWOULDBLOCK The socket is marked non-blocking and the requested operation would block.

getsockopt , setsockopt

NAME

getsockopt, setsockopt - get and set options on sockets

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

int getsockopt (int s, int level, int optname, void *optval, int *optlen);

int setsockopt (int s, int level, int optname, void *optval, int optlen);

DESCRIPTION

getsockopt and ***setsockopt*** manipulate options associated with a socket, *s*. Options may exist at multiple protocol levels; they are always present at the uppermost socket level.

When manipulating socket options, the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, *level* is specified as **SOL_SOCKET**. To manipulate options at any other level, *level* is the protocol number of the protocol that controls the option. For example, to indicate that an option is to be interpreted by the **TCP** protocol, *level* is set to the **TCP** protocol number (see ***getprotobyname***()).

The parameters *optval* and *optlen* are used to access option values for setsockopt. For ***getsockopt***, they identify a buffer in which the value(s) for the requested option(s) are to be returned. For ***getsockopt***, *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. Use a 0 *optval* if no option value is to be supplied or returned.

optname and any specified options are passed un-interpreted to the appropriate protocol module for interpretation. The include file <*sys/socket.h*> contains definitions for the socket-level options described below. Options at other protocol levels vary in format and name.

Most socket-level options take an int for *optval*. For ***setsockopt***, the *optval* parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled. **SO_LINGER** uses a struct linger parameter that specifies the desired state of the option and the linger interval (see below). struct linger is defined in <*sys/socket.h*>. *struct linger* contains the following members:

l_onoff	option on/off
l_linger	linger time

The following options are recognized at the socket level. Except as noted, each may be examined with ***getsockopt*** and set with setsockopt.

SO_DEBUG	toggle recording of debugging information
SO_REUSEADDR	toggle local address reuse
SO_KEEPALIVE	toggle keep connections alive
SO_DONTROUTE	toggle routing bypass for outgoing messages
SO_LINGER	linger on close if data is present
SO_BROADCAST	toggle permission to transmit broadcast messages
SO_OOBINLINE	toggle reception of out-of-band data in band

SO_SNDBUF	set buffer size for output
SO_RCVBUF	set buffer size for input
SO_TYPE	get the type of the socket (get only)
SO_ERROR	get and clear error on the socket (get only)

SO_DEBUG enables debugging in the underlying protocol modules. **SO_REUSEADDR** indicates that the rules used in validating addresses supplied in a ***bind***() call should allow reuse of local addresses. **SO_KEEPALIVE** enables the periodic transmission of messages on a connected socket. If the connected party fails to respond to these messages, the connection is considered broken and processes using the socket are notified using a **SIGPIPE** signal. **SO_DONTROUTE** indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when un-sent messages are queued on a socket and a ***close***(BA_OS) is performed. If the socket promises reliable delivery of data and **SO_LINGER** is set, the system will block the process on the close attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the setsockopt call when **SO_LINGER** is requested). If **SO_LINGER** is disabled and a close is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

The option **SO_BROADCAST** requests permission to send broadcast datagrams on the socket. With protocols that support out-of-band data, the **SO_OOBINLINE** option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with recv or read calls without the **MSG_OOB** flag.

SO_SNDBUF and **SO_RCVBUF** are options that adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections or may be decreased to limit the possible backlog of incoming data.

Finally, **SO_TYPE** and **SO_ERROR** are options used only with getsockopt. **SO_TYPE** returns the type of the socket (for example, **SOCK_STREAM**). It is useful for servers that inherit sockets on startup. **SO_ERROR** returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

RETURN VALUES

If successful, ***getsockopt*** returns 0; otherwise it returns -1 and sets errno to indicate the error.

ERRORS

The call succeeds unless:

EBADF	The argument *s* is not a valid file descriptor.
ENOMEM	There was insufficient memory available for the operation to complete.
ENOPROTOOPT	The option is unknown at the level indicated.
ENOSR	There were insufficient **STREAMS** resources available for the operation to complete.
ENOTSOCK	The argument *s* is not a socket.

shutdown

NAME

shutdown - shut down part of a full-duplex connection

SYNOPSIS

int shutdown (int s, int how);

DESCRIPTION

The ***shutdown*** call shuts down all or part of a full-duplex connection on the socket associated with *s*. If *how* is 0, then further receives will be disallowed. If *how* is 1, then further sends will be disallowed. If *how* is 2, then further sends and receives will be disallowed.

RETURN VALUES

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

The call succeeds unless:

EBADF *s* is not a valid file descriptor.

ENOMEM There was insufficient user memory available for the operation to complete.

ENOSR There were insufficient **STREAMS** resources available for the operation to complete.

ENOTCONN The specified socket is not connected.

ENOTSOCK *s* is not a socket.

NOTES

The *how* values should be defined constants.

socket

NAME

socket - create an endpoint for communication

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

int socket (int domain, int type, int protocol);

DESCRIPTION

socket creates an endpoint for communication and returns a descriptor. The *domain* parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file <*sys/socket.h*>. The only supported protocol family is **PF_INET**. The socket has the indicated *type*, which specifies the communication semantics. Currently defined *type*s are:

SOCK_STREAM: A **SOCK_STREAM** type provides sequenced, reliable, two-way connection-based byte streams. An out-of-band data transmission mechanism may be supported.

SOCK_STREAM A **SOCK_DGRAM** socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length).

SOCK_SEQPACKET A **SOCK_SEQPACKET** socket may provide a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer may be required to read an entire packet with each read system call. This facility is protocol specific, and presently not implemented for any protocol family.

protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, multiple protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the "communication domain" in which communication is to take place. If a protocol is specified by the caller, then it will be packaged into a socket level option request and sent to the underlying protocol layers.

Sockets of type **SOCK_STREAM** are full-duplex byte streams, similar to pipes. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a ***connect***() call. Once connected, data may be transferred using ***read***(BA_OS) and ***write***(BA_OS) calls or some variant of the ***send***() and ***recv***() calls. When a session has been completed, a ***close***(BA_OS) may be performed. Out-of-band data may also be transmitted as described on the ***send***() manual page and received as described on the recv() manual page.

The communications protocols used to implement a **SOCK_STREAM** insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with -1 returns and with **ETIMEDOUT** as the specific code in the global variable errno. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (for instance 5 minutes). A **SIGPIPE** signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

SOCK_SEQPACKET sockets employ the same system calls as **SOCK_STREAM** sockets. The only difference is that read calls will return only the amount of data requested, and any remaining in the arriving packet will be discarded.

SOCK_DGRAM sockets allow datagrams to be sent to correspondents named in sendto calls. Datagrams are generally received with recvfrom, which returns the next datagram with its return address.

An ***ioctl***(BA_OS) call can be used to specify a process group to receive a **SIGURG** signal when the out-of-band data arrives. It may also enable non-blocking I/O and asynchronous notification of I/O events with **SIGPOLL** signals.

The operation of sockets is controlled by socket level options. These options are defined in the file **<***sys/socket.h***>**. ***setsockopt***() and ***getsockopt***() are used to set and get options, respectively.

RETURN VALUES

A -1 is returned if an error occurs. Otherwise the return value is a descriptor referencing the socket.

ERRORS

The ***socket*** call fails if:

EACCES	Permission to create a socket of the specified type and/or protocol is denied.
EMFILE	The per-process descriptor table is full.
ENOMEM	Insufficient user memory is available.
ENOSR	There were insufficient **STREAMS** resources available to complete the operation.
EPROTONOSUPPORT	The protocol type or the specified protocol is not supported within this domain.

endnetent, getnetbyaddr, getnetbyaddr_r, getnetbyname getnetbyname_r, getnetent, getnetent_r, setnetent

NAME

getnetbyname, getnetbyname_r, getnetbyaddr, getnetbyaddr_r, getnetent, getnetent_r, setnetent, endnetent - get network entry

SYNOPSIS

#include <netdb.h>

struct netent	**getnetbyname(const char *name);*
struct netent	**getnetbyname_r(const char *name, struct netent *result, char *buffer, int buflen);*
struct netent	**getnetbyaddr(long net, int type);*
struct netent	**getnetbyaddr_r(long net, int type, struct netent *result, char *buffer, int buflen);*
struct netent	**getnetent(void);*
struct netent	**getnetent_r(struct netent *result, char *buffer, int buflen);*
int	*setnetent(int stayopen);*
int	*endnetent(void);*

DESCRIPTION

These functions are used to obtain entries for networks. An entry may come from any of the sources for networks specified in the */etc/nsswitch.conf* file (see ***nsswitch.conf***()). ***getnetbyname***() searches for a network entry with the network name specified by the character string parameter name. ***getnetbyaddr***() searches for a network entry with the network address specified by net. The parameter type specifies the family of the address. This should be one of the address families defined in <*sys/socket.h*>. The functions ***setnetent***(), ***getnetent***(), and ***endnetent***() are used to enumerate network entries from the database. ***setnetent***() sets (or resets) the enumeration to the beginning of the set of network entries. This function should be called before the first call to ***getnetent***(). Calls to ***getnetbyname***() and ***getnetbyaddr***() leave the enumeration position in an indeterminate state. If the stayopen flag is non-zero, the system may keep allocated resources such as open file descriptors until a subsequent call to ***endnetent***(). Successive calls to ***getnetent***() return either successive entries or NULL, indicating the end of the enumeration. ***endnetent***() may be called to indicate that the caller expects to do no further network entry retrieval operations; the system may then deallocate resources it was using. It is still allowed, but possibly less efficient, for the process to call more network entry retrieval functions after calling ***endnetent***().

Reentrant Interfaces

The functions ***getnetbyname***(), ***getnetbyaddr***(), and ***getnetent***() use static storage that is re-used in each call, making these routines unsafe for use in multithreaded applications. The functions: ***getnetbyname_r***(), ***getnetbyaddr_r***(), and ***getnetent_r***() provide reentrant interfaces for these operations. Each reentrant interface performs the same operation as its non-reentrant counterpart, named by removing the ``_r'' suffix. The reentrant interfaces, however, use buffers supplied by the caller to store returned results, and are safe for use in both single-threaded and multithreaded applications. Each reentrant interface takes the same parameters as its non-reentrant counterpart, as well as the following additional parameters. The parameter result must be a pointer to a struct netent structure allocated by the caller. On successful completion, the function returns the network entry in this structure. The parameter buffer must be a pointer to a buffer supplied by the caller. This buffer is used as storage space for the network entry data. All of the pointers within the returned struct netent result point to data stored within this buffer (see RETURN VALUES). The buffer must be large enough to hold all of the data associated with the network entry. The parameter buflen should give the size in bytes of the buffer indicated by buffer . For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. ***setnetent***() may be used in a multithreaded application but resets the enumeration position for all threads. If multiple threads interleave calls to ***getnetent_r***(), the threads will enumerate disjoint subsets of the network database. Like their non-reentrant counterparts, ***getnetbyname_r***() and ***getnetbyaddr_r***() leave the enumeration position in an indeterminate state.

RETURN VALUES

Network entries are represented by the struct netent structure defined in <*netdb*.h>:

```
struct netent {
        char        *n_name;
        char        **n_aliases;
        int         n_addrtype;
        long        n_net;
};
```

The functions ***getnetbyname***(), ***getnetbyname_r***(), ***getnetbyaddr***(), and ***getnetbyaddr_r***() each return a pointer to a struct netent if they successfully locate the requested entry; otherwise they return NULL. The functions ***getnetent***() and ***getnetent_r***() each return a pointer to a struct netent if they successfully enumerate an entry; otherwise they return NULL, indicating the end of the enumeration. The functions ***getnetbyname***(), ***getnetbyaddr***(), and ***getnetent***() use static storage, so returned data must be copied before a subsequent call to any of these functions if the data is to be saved. When the pointer returned by the reentrant functions ***getnetbyname_r***(), ***getnetbyaddr_r***(), and ***getnetent_r***() is non-NULL, it is always equal to the result pointer that was supplied by the caller. The functions ***setnetent***() and ***endnetent***() return 0 on success.

ERRORS

The reentrant functions ***getnetbyname_r***(), ***getnetbyaddr_r***() and ***getnetent_r***() will return NULL and set errno to **ERANGE** if the length of the buffer supplied by caller is not large enough to store the result. See ***intro***() for the proper usage and interpretation of errno in multithreaded applications.

FILES

/etc/networks, /etc/nsswitch.conf

SEE ALSO

inet(), networks(), nsswitch.conf()

NOTES

The current implementation of these functions only return or accept network numbers for the Internet address family (type **AF_INET**). The functions described in ***inet***() may be helpful in constructing and manipulating addresses and network numbers in this form. Programs that use the interfaces described in this manual page cannot be linked statically since the implementations of these functions employ dynamic loading and linking of shared objects at run time. When compiling multithreaded applications, see ***Intro***(), Notes On Multithread Applications, for information about the use of the **_REENTRANT** flag.

endprotoent, getprotobyname, getprotobyname_r getprotobynumber, getprotobynumber_r, getprotoent getprotoent_r, setprotoent

NAME

getprotobyname,getprotobyname_r, getprotobynumber, getprotobynumber_r, getprotoent, getprotoent_r, setprotoent, endprotoent - get protocol entry

SYNOPSIS

#include <netdb.h>

```
struct protoent      *getprotobyname(const char *name);
struct protoent      *getprotobyname_r(const char *name,
                          struct protoent *result, char *buffer, int buflen);
struct protoent      *getprotobynumber(int proto);
struct protoent      *getprotobynumber_r(int proto, struct protoent *result,
                          char *buffer, int buflen);
struct protoent      *getprotoent(void);
struct protoent      *getprotoent_r(struct protoent *result, char *buffer, int buflen);
int                  setprotoent(int stayopen);
int                  endprotoent(void);
```

DESCRIPTION

These routines return a protocol entry. Two types of interfaces are supported: reentrant (***getprotobyname_r***(), ***getprotobynumber_r***(), and ***getprotoent_r***()) and non-reentrant (***getprotobyname***(), ***getprotobynumber***(), and ***getprotoent***()). The reentrant routines may be used in single-threaded applications and are safe for multi-threaded applications, making them the preferred interfaces. The reentrant routines require additional parameters which are used to return results data. result is a pointer to a struct protoent structure and will be where the returned results will be stored. buffer is used as storage space for elements of the returned results. buflen is the size of buffer and should be large enough to contain all returned data. *buflen* must be at least 1024 bytes. ***getprotobyname_r***(), ***getprotobynumber_r***(), and ***getprotoent_r***() each return a protocol entry. The entry may come from one of the following sources: the protocols file (see ***protocols***()), the **NIS** maps "*protocols.byname*" and "*protocols.bynumbe*r", and the **NIS+** table ``protocols". The sources and their lookup order are specified in the */etc/nsswitch.conf* file (see ***nsswitch.con***f() for details). Some name services such as **NIS** will return only one name for a host, whereas others such as **NIS+** or **DNS** will return all aliases. ***getprotobyname_r***() and ***getprotobynumber_r***() sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until an **EOF** is encountered. getprotobyname() and ***getprotobynumber***() have the same functionality as ***getprotobyname_r***() and ***getprotobynumber_r***() except that a static buffer is used to store returned results. These routines are unsafe in a multi-threaded application. ***getprotoent_r***() enumerates protocol entries: successive calls to ***getprotoent_r***() will return either successive protocol entries or NULL. Enumeration may not be supported by some sources. Note that if multiple threads call ***getprotoent_r***(), each will retrieve a subset of the protocol database.

getprotent() has the same functionality as ***getprotent_r***() except that a static buffer is used to store returned results. This routine is unsafe in a multi-threaded application. ***setprotoent***() "rewinds" to the beginning of the enumeration of protocol entries. If the stayopen flag is non-zero, resources such as open file descriptors are not de-allocated after each call to ***getprotobynumber_r***() and ***getprotobyname_r***(). Calls to ***getprotobyname_r***() , ***getprotobyname***() , ***getprotobynumber_r***() and ***getprotobynumber***() may leave the enumeration in an indeterminate state, so ***setprotoent***() should be called before the first ***getprotoent_r***() or ***getprotoent***(). Note that setprotoent() has process-wide scope, and ``rewinds'' the protocol entries for all threads calling ***getprotoent_r***() as well as main-thread calls to ***getprotoent***(). ***endprotoent***() may be called to indicate that protocol processing is complete; the system may then close any open protocols file, deallocate storage, and so forth. It is legitimate, but possibly less efficient, to call more protocol routines after ***endprotoent***(). The internal representation of a protocol entry is a protoent structure defined in <*netdb*.h> with the following members:

char	**p_name;*
char	***p_aliases;*
int	*p_proto;*

RETURN VALUES

getprotobyname_r(), ***getprotobyname***(), ***getprotobynumber_r***(), and ***getprotobynumber***() return a pointer to a struct protoent if they successfully locate the requested entry; otherwise they return NULL. ***getprotoent_r***() and ***getprotoent***() return a pointer to a struct protoent if they successfully enumerate an entry; otherwise they return NULL, indicating the end of the enumeration.

ERRORS

getprotobyname_r(), ***getprotobynumber_r***(), and ***getprotoent_r***() will fail if the following is true:

ERANGE length of the buffer supplied by caller is not large enough to store the result.

FILES

/etc/protocols, /etc/nsswitch.conf

SEE ALSO

intro(), nsswitch.conf(), protocols()

NOTES

Although ***getprotobyname_r***(), ***getprotobynumber_r***(), and ***getprotoent_r***() are not mentioned by **POSIX.4a Draft 6**, they were added to complete the functionality provided by similar thread-safe functions. These interfaces are subject to change to be compatible with the “spirit” of **POSIX.4a** when it is approved as a standard. When compiling multithreaded applications, see ***intro***(), Notes On Multithread Applications, for information about the use of the **_REENTRANT** flag. The routines ***getprotobyname_r***(), ***getprotobynumber_r***(), and ***getprotoent_r***() are reentrant and multi-thread safe. The reentrant interfaces can be used in single-threaded as well as multi-threaded applications and are therefore the preferred interfaces. The routines ***getprotobyname***(), ***getprotobyaddr***(), and ***getprontoent***() use static storage, so returned data must be copied if it is to be saved. Because of their use of static storage for returned data, these routines are not safe for multi-threaded applications. ***setprotoent***() and ***endprotoent***() have process-wide scope, and are therefore not safe in multi-threaded applications. Use of ***getprotoent_r***() and ***getprotoent***() is discouraged; enumeration is well-defined for the protocols file and is supported (albeit inefficiently) for **NIS** and **NIS+**, but in general may not be well-defined. The semantics of enumeration are discussed in ***nsswitch.conf***().

endservent, getservbyname, getservbyname_r getservbyport, getservbyport_r, getservent getservent_r, setservent

NAME

getservbyname, getservbyname_r, getservbyport, getservbyport_r, getservent, getservent_r, setservent, endservent **-** get service entry

SYNOPSIS

```
#include <netdb.h>
struct servent    *getservbyname(const char *name, const char *proto);
struct servent    *getservbyname_r(const char *name, const char *proto,
                          struct servent *result,   char *buffer, int buflen);
struct servent    *getservbyport(int port, const char *proto);
struct servent    *getservbyport_r(int port,   const   char *proto,
                          struct servent *result, char *buffer, int buflen);
struct servent    *getservent(void);
struct servent    *getservent_r(struct servent *result, char *buffer, int buflen);
int               setservent(int stayopen);
int               endservent(void);
```

DESCRIPTION

These functions are used to obtain entries for Internet services. An entry may come from any of the sources for services specified in the */etc/nsswitch.conf* file. See ***nsswitch.con***f(). ***getservbyname***() and ***getservbyport***() sequentially search from the beginning of the file until a matching protocol name or port number is found, or until end-of-file is encountered. If a protocol name is also supplied (non-NULL), searches must also match the protocol. ***getservbyname***() searches for an entry with the Internet service name specified by the parameter name. ***getservbyport***() searches for an entry with the Internet port number port.

The string *proto* is used by both ***getservbyname***() and ***getservbyport***() to restrict the search to entries with the specified protocol. If *proto* is NULL, entries with any protocol may be returned. The functions ***setservent***(), ***getservent***(), and ***endservent***() are used to enumerate entries from the services database. ***setservent***() sets (or resets) the enumeration to the beginning of the set of service entries. This function should be called before the first call to ***getservent***(). Calls to the functions ***getservbyname***() and ***getservbyport***() leave the enumeration position in an indeterminate state. If the stay open flag is non-zero, the system may keep allocated resources such as open file descriptors until a subsequent call to ***endservent***(). ***getservent***() reads the next line of the file, opening the file if necessary. ***getservent***() opens and rewinds the file. If the stayopen flag is non-zero, the net data base will not be closed after each call to ***getservent***() (either directly, or indirectly through one of the other “getserv” calls). Successive calls to ***getservent***() return either successive entries or NULL, indicating the end of the enumeration. ***endservent***() closes the file. ***endservent***() may be called to indicate that the caller expects to do no further service entry retrieval operations; the system may then deallocate resources it was using. It is still allowed, but possibly less efficient, for the process to call more service entry retrieval functions after calling ***endservent***().

Reentrant Interfaces

The functions ***getservbyname***(), ***getservbyport***(), and ***getservent***() use static storage that is re-used in each call, making these functions unsafe for use in multithreaded applications. The functions: ***getservbyname_r***(), ***getservbyport_r***(), and ***getservent_r***() provide reentrant interfaces for these operations. Each reentrant interface performs the same operation as its non-reentrant counterpart, named by removing the "_r" suffix. The reentrant interfaces, however, use buffers supplied by the caller to store returned results, and are safe for use in both single-threaded and multithreaded applications.

Each reentrant interface takes the same parameters as its non-reentrant counterpart, as well as the following additional parameters. The parameter result must be a pointer to a struct servent structure allocated by the caller. On successful completion, the function returns the service entry in this structure. The parameter buffer must be a pointer to a buffer supplied by the caller. This buffer is used as storage space for the service entry data. All of the pointers within the returned struct servent result point to data stored within this buffer. See the RETURN VALUES section of this man page. The buffer must be large enough to hold all of the data associated with the service entry. The parameter buflen should give the size in bytes of the buffer indicated by buffer. For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. ***setservent***() may be used in a multithreaded application but resets the enumeration position for all threads. If multiple threads interleave calls to ***getservent_r***(), the threads will enumerate disjoint subsets of the service database. Like their non-reentrant counterparts, ***getservbyname_r***() and ***getservbyport_r***() leave the enumeration position in an indeterminate state.

RETURN VALUES

Service entries are represented by the struct servent structure defined in <netdb.h>:

```
struct servent {
        char    *s_name;        /* official name of service */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port service resides at */
        char    *s_proto;       /* protocol to use */
};
```

The members of this structure are:

s_name	The official name of the service.
s_aliases	A zero terminated list of alternate names for the service.
s_port	The port number at which the service resides. Port numbers are returned in network byte order.
s_proto	The name of the protocol to use when contacting the service.

The functions ***getservbyname***(), ***getservbyname_r***(), ***getservbyport***(), and ***getservbyport_r***() each return a pointer to a struct servent if they successfully locate the requested entry; otherwise they return NULL. The functions ***getservent***() and ***getservent_r***() each return a pointer to a struct servent if they successfully enumerate an entry; otherwise they return NULL, indicating the end of the enumeration. The functions ***getservbyname***(), ***getservbyport***(), and ***getservent***() use static storage, so returned data must be copied before a subsequent call to any of these functions if the data is to be saved. When the pointer returned by the reentrant functions ***getservbyname_r***(), ***getservbyport_r***(), and ***getservent_r***() is non-null, it is always equal to the result pointer that was supplied by the caller.

ERRORS

The reentrant functions ***getservbyname_r***(), ***getservbyport_r***() and ***getservent_r***() will return NULL and set errno to **ERANGE** if the length of the buffer supplied by caller is not large enough to store the result.

FILES

/etc/services	Internet network services
/etc/netconfig	network configuration file
/etc/nsswitch.conf	configuration file for the name-service switch

SEE ALSO

intro(), intro(), netdir(), netconfig(), nsswitch.conf(), services()

NOTES

The functions that return struct servent return the least significant 16-bits of the s_port field in network byte order. ***getservbyport***() and ***getservbyport_r***() also expect the input parameter port in the network byte order . See ***htons***() for more details on converting between host and network byte orders. Programs that use the interfaces described in this manual page cannot be linked statically since the implementations of these functions employ dynamic loading and linking of shared objects at run time.

In order to ensure that they all return consistent results, ***getservbyname***(), ***getservbyname_r***(), and ***netdir_getbyname***() are implemented in terms of the same internal library function. This function obtains the system-wide source lookup policy based on the inet family entries in ***netconfig***() and the services: entry in ***nsswitch.conf***(). Similarly, ***getservbyport***(), ***getservbyport_r***(), and ***netdir_getbyaddr***() are implemented in terms of the same internal library function. If the inet family entries in ***netconfig***() have a "-" in the last column for *nametoaddr* libraries, then the entry for services in nsswitch.conf will be used; otherwise the *nametoaddr* libraries in that column will be used, and nsswitch.conf will not be consulted.

There is no analogue of ***getservent***() and ***getservent_r***() in the netdir functions, so these enumeration functions go straight to the services entry in nsswitch.conf. Thus enumeration may return results from a different source than that used by ***getservbyname***(), ***getservbyname_r***(), ***getservbyport***(), and ***getservbyport_r***().

When compiling multithreaded applications, see ***intro***(), Notes On Multithread Applications, for information about the use of the **_REENTRANT** flag.

Use of the enumeration interfaces ***getservent***() and ***getservent_r***() is discouraged; enumeration may not be supported for all database sources. The semantics of enumeration are discussed further in ***nsswitch.conf***().

ether_ntoa, ether_aton, ether_ntohost
ether_hostton, ether_line

NAME

ether_ntoa, ether_aton, ether_ntohost, ether_hostton, ether_line - Ethernet address mapping operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
char                 *ether_ntoa (struct ether_addr *e);
struct ether_addr    *ether_aton (char *s);
int                  ether_ntohost (char *hostname, struct ether_addr *e);
int                  ether_hostton (char *hostname, struct ether_addr *e);
int                  ether_line (char *l, struct ether_addr * e, char *hostname);
```

DESCRIPTION

These routines are useful for mapping 48 bit Ethernet numbers to their ASCII representations or their corresponding host names, and vice versa. The function ***ether_ntoa***() converts a 48 bit Ethernet number pointed to by e to its standard **ASCII** representation; it returns a pointer to the **ASCII** string. The representation is of the form x:x:x:x:x:x where x is a hexadecimal number between 0 and ff. The function ***ether_aton***() converts an **ASCII** string in the standard representation back to a 48 bit Ethernet number; the function returns NULL if the string cannot be scanned successfully. The function ***ether_ntohost***() maps an Ethernet number (pointed to by *e*) to its associated hostname. The string pointed to by hostname must be long enough to hold the hostname and a NULL character. The function returns zero upon success and non-zero upon failure. Inversely, the function ***ether_hostton***() maps a hostname string to its corresponding Ethernet number; the function modifies the Ethernet number pointed to by *e*. The function also returns zero upon success and non-zero upon failure. In order to do the mapping, both these functions may lookup one or more of the following sources: the ethers file, the NIS maps "ethers.byname"' and "ethers.byaddr" and the NIS+ table ``ethers''. The sources and their lookup order are specified in the */etc/nsswitch.conf* file (see ***nsswitch.conf***() for details). The function ***ether_line***() scans a line (pointed to by l) and sets the hostname and the Ethernet number (pointed to by *e*). The string pointed to by hostname must be long enough to hold the hostname and a NULL character. The function returns zero upon success and non-zero upon failure. The format of the scanned line is described by ***ethers***().

FILES

/etc/ethers, /etc/nsswitch.conf

SEE ALSO

ethers(), nsswitch.conf()

byteorder, htonl
htons, ntohl, ntohs

NAME

byteorder, ***htonl, htons, ntohl, ntoh***s - convert values between host and network byte order

SYNOPSIS

#include <sys/types.h>

#include <netinet/in.h>

u_long	***htonl(u_long hostlong);***
u_short	***htons(u_short hostshort);***
u_long	***ntohl(u_long netlong);***
u_short	***ntohs(u_short netshort);***

DESCRIPTION

These routines convert 16 and 32 bit quantities between network byte order and host byte order. On some architectures these routines are defined as NULL macros in the include file <*netinet/in.h*>. On other architectures, if their host byte order is different from network byte order, these routines are functional. These routines are most often used in conjunction with Internet addresses and ports as returned by ***gethostent***() and ***getservent***(). (See ***gethostbyname***() and ***getservbyname***() respectively.)

SEE ALSO

gethostbyname(), getservbyname()

rcmd, rresvport, ruserok

NAME

rcmd, rresvport, ruserok - routines for returning a stream to a remote command

SYNOPSIS

int rcmd(char **ahost, unsigned short inport, const char *user, const char *ruser, const char *cmd, int *fd2p);

int rresvport(int *port);

int ruserok(const char *rhost, int suser, const char *ruser, const char *luser);

DESCRIPTION

rcmd() is a routine used by the super-user to execute a command on a remote machine using an authentication scheme based on reserved port numbers. ***rresvport***() is a routine which returns a descriptor to a socket with an address in the privileged port space. ***ruserok***() is a routine used by servers to authenticate clients requesting service with ***rcmd***. All three functions are present in the same file and are used by the ***in.rshd***() server (among others).

rcmd() looks up the host **ahost* using ***gethostbyname***(), returning -1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host and a connection is established to a server residing at the well-known Internet port inport. If the connection succeeds, a socket in the Internet domain of type **SOCK_STREAM** is returned to the caller, and given to the remote command as its standard input (file descriptor 0) and standard output (file descriptor 1). If *fd2p* is non-zero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (file descriptor 2) on this channel, and will also accept bytes on this channel as signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the standard error (file descriptor 2) of the remote command will be made the same as its standard output and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data. The protocol is described in detail in ***in.rshd***().

The ***rresvport***() routine is used to obtain a socket bound to a privileged port number. This socket is suitable for use by ***rcmd***() and several other routines. Privileged Internet ports are those in the range 1 to 1023. Only the super-user is allowed to bind a socket to a privileged port number. The application must pass in port, which must be in the range 512 to 1023. The system first tries to bind to that port number. If it fails, it then tries to bind to port numbers less than port until either it succeeds or port number 512 is reached.

ruserok() takes a remote host's name, as returned by a ***gethostbyaddr***() (see ***gethostbyname***()) routine, two user names and a flag indicating whether the local user's name is that of the super-user. It then checks the files */etc/hosts.equiv* and possibly .rhosts in the local user's home directory to see if the request for service is allowed. 0 is returned if the machine name is listed in the */etc/hosts.equiv* file, or the host and remote user name are found in the .rhosts file; otherwise ruserok() returns -1. If the super-user flag is 1, the checking of the */etc/hosts.equiv* file is bypassed.

RETURN VALUES

rcmd() returns a valid socket descriptor on success. It returns -1 on error and prints a diagnostic message on the standard error.

rresvport() returns a valid, bound socket descriptor on success. It returns -1 on error with the global value errno set according to the reason for failure.

FILES

/etc/hosts.equiv	system trusted hosts and users
~/.rhosts	user's trusted hosts and users

SEE ALSO

rlogin, rsh, in.rexecd, in.rshd, gethostbyname(), rexec()

NOTES

The error code **EAGAIN** is overloaded to mean “All network ports in use “.

These interfaces are unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

rexec

NAME

rexec - return stream to a remote command

SYNOPSIS

int rexec (char **ahost, unsigned short inport, const char *user, const char *passwd, const char *cmd,int *fd2p);

DESCRIPTION

rexec() looks up the host *ahost using ***gethostbyname***(), returning -1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host. If a username and password are both specified, then these are used to authenticate to the foreign host; otherwise the user's .netrc file in his home directory is searched for appropriate information. If all this fails, the user is prompted for the information. The port inport specifies which well-known **DARPA** Internet port to use for the connection. The protocol for connection is described in detail in ***in.rexecd***().

If the call succeeds, a socket of type **SOCK_STREAM** is returned to the caller, and given to the remote command as its standard input and standard output. If *fd2p* is non-zero, then an auxiliary channel to a control process will be setup, and a file descriptor for it will be placed in **fd2p*. The control process will return diagnostic output (file descriptor 2, the standard error) from the command on this channel, and will also accept bytes on this channel as signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the standard error (file descriptor 2 of the remote command) will be made the same as its standard output and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

RETURN VALUES

If ***rexec***() succeeds, a file descriptor number, which is a socket of type **SOCK_STREAM**, is returned by the ***routine***. **ahost* is set to the standard name of the host, and if *fd2p* is not NULL, a file descriptor number is placed in **fd2p* which represents the command's standard error stream. If ***rexec***() fails, -1 is returned.

SEE ALSO

in.rexecd(), gethostbyname(), getservbyname()

NOTES

There is no way to specify options to the ***socket***() call that ***rexec***() makes. This interface is unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

getsockopt, setsockopt

NAME

getsockopt, setsockopt - get and set options on sockets

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

int getsockopt(int s, int level, int optname, char *optval,int *optlen);

int setsockopt(int s, int level, int optname, const char *optval, int optlen);

DESCRIPTION

getsockopt() and ***setsockopt***() manipulate options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level.

When manipulating socket options, the level at which the option resides and the name of the option must be specified. To manipulate options at the "socket" level, level is specified as **SOL_SOCKET**. To manipulate options at any other level, level is the protocol number of the protocol that controls the option. For example, to indicate that an option is to be interpreted by the **TCP** protocol, level is set to the **TCP** protocol number (see ***getprotobyname***()).

The parameters optval and optlen are used to access option values for ***setsockopt***(). For getsockopt(), they identify a buffer in which the value(s) for the requested option(s) are to be returned. For getsockopt(), optlen is a value-result parameter, initially containing the size of the buffer pointed to by optval, and modified on return to indicate the actual size of the value returned. Use a 0 optval if no option value is to be supplied or returned.

optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file <*sys/socket.h*> contains definitions for the socket-level options described below. Options at other protocol levels vary in format and name.

Most socket-level options take an int for optval. For ***setsockopt***(), the optval parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled. **SO_LINGER** uses a struct linger parameter that specifies the desired state of the option and the linger interval (see below). struct *linger* is defined in <*sys/socket.h*>. struct linger contains the following members:

l_onoff on = 1/off = 0

l_linger linger time, in seconds

The following options are recognized at the socket level. Except as noted, each may be examined with ***getsockopt***() and set with ***setsockopt***().

SO_DEBUG enable/disable recording of debugging information

SO_REUSEADDR enable/disable local address reuse

SO_KEEPALIVE enable/disable keep connections alive

SO_DONTROUTE enable/disable routing bypass for outgoing messages

SO_LINGER linger on close if data is present

SO_BROADCAST enable/disable permission to transmit broadcast messages

SO_OOBINLINE enable/disable reception of out-of-band data in band

SO_SNDBUF set buffer size for output

SO_RCVBUF set buffer size for input

SO_DGRAM_ERRIND application wants delayed error

SO_TYPE get the type of the socket (get only)

SO_ERROR get and clear error on the socket (get only)

SO_DEBUG enables debugging in the underlying protocol modules.

SO_REUSEADDR indicates that the rules used in validating addresses supplied in a bind() call should allow reuse of local addresses.

SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. If the connected party fails to respond to these messages, the connection is considered broken and processes using the socket are notified using a **SIGPIPE** signal.

SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when un-sent messages are queued on a socket and a ***close***() is performed. If the socket promises reliable delivery of data and **SO_LINGER** is set, the system will block the process on the ***close***() attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the ***setsockopt***() call when **SO_LINGER** is requested). If **SO_LINGER** is disabled and a ***close***() is issued, the system will process the ***close***() in a manner that allows the process to continue as quickly as possible.

SO_BRODCAST The option **SO_BROADCAST** requests permission to send broadcast datagrams on the socket. With protocols that support out-of-band data, the **SO_OOBINLINE** option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with ***recv***() or ***read***() calls without the **MSG_OOB** flag.

SO_SNDBUF/SO_RCVBUF **SO_SNDBUF** and **SO_RCVBUF** are options that adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections or may be decreased to limit the possible backlog of incoming data.

SO_DGRAM_ERRIND By default, delayed errors (such as **ICMP** port unreachable packets) are returned only for connected datagram sockets. **SO_DGRAM_ERRIND** makes it possible to receive errors for datagram sockets that are not connected. When this option is set, certain delayed errors received after completion of a ***sendto***() or ***sendmsg***() operation will cause a subsequent ***sendto***() or ***sendmsg***() operation using the same destination address (to parameter) to fail with the appropriate error. See ***send***().

SO_TYPE returns the type of the socket (for example, **SOCK_STREAM**). It is useful for servers that inherit sockets on startup. This option is used only with *getsockopt*().

SO_ERROR returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors. This option is used only with *getsockopt*().

RETURN VALUES

If successful, *getsockopt*() returns 0; otherwise, it returns -1 and sets errno to indicate the error.

ERRORS

The call succeeds unless:

EBADF The argument s is not a valid file descriptor.

ENOMEM There was insufficient memory available for the operation to complete.

ENOPROTOOPT The option is unknown at the level indicated.

ENOSR There were insufficient **STREAMS** resources available for the operation to complete.

ENOTSOCK The argument s is not a socket.

SEE ALSO

close(), ioctl(), bind(), getprotobyname(), send(), socket()

socketpair

NAME

socketpair - create a pair of connected sockets

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

int socketpair(int domain, int type, int protocol, intsv[2]);

DESCRIPTION

The ***socketpair***() library call creates an unnamed pair of connected sockets in the specified address family d, of the specified type , and using the optionally specified protocol. The descriptors used in referencing the new sockets are returned in *sv[0]* and *sv[1].* The two sockets are indistinguishable.

RETURN VALUES

socketpair() returns -1 on failure, and 0 on success.

ERRORS

The call succeeds unless:

EAFNOSUPPORT	The specified address family is not supported on this machine.
EMFILE	Too many descriptors are in use by this process.
ENOMEM	There was insufficient user memory for the operation to complete.
ENOSR	There were insufficient **STREAMS** resources for the operation to complete.
EOPNOSUPPORT	The specified protocol does not support creation of socket pairs.
EPROTONOSUPPORT	The specified protocol is not supported on this machine.

SEE ALSO

pipe(), read(), write()

NOTES

This call is currently implemented only for the **AF_UNIX** address family.

SPARC COMPLIANCE DEFINITION 2.4.1 IS

libthread

cond_broadcast, cond_destroy
cond_init, cond_timedwait
cond_signal, cond_wait

NAME

condition, cond_init, cond_destroy, cond_wait, cond_timedwait, cond_signal, cond_broadcast - condition variables

SYNOPSIS

#include <synch.h>

int cond_init (cond_t *cvp, int type, void *arg);

int cond_destroy (cond_t *cvp);

int cond_wait (cond_t *cvp, mutex_t *mp);

int cond_timedwait (cond_t *cvp, mutex_t *mp, timestruc_t *abstime);

int cond_signal (cond_t *cvp);

int cond_broadcast (cond_t *cvp);

DESCRIPTION

A condition variable enables threads to atomically block until a condition is satisfied. The condition is tested under the protection of a mutual exclusion lock (mutex). When the condition is false, a thread typically blocks on a condition variable and atomically releases the mutex waiting for the condition to change. When another thread changes the condition, it may signal the associated condition variable to cause one or more waiting threads to wake up, reacquire the mutex, and re-evaluate the condition.

Condition variables can be used to synchronize threads among processes if they are allocated in memory that is writable and shared by the cooperating processes (see ***mmap***(**KE_OS**)) and have been initialized for this behavior.

Condition variables must be initialized before use. ***cond_init***() initializes the condition variable pointed to by *cvp*. A condition variable can potentially have several different types of behavior, specified by *type*. No current *type* uses *arg* although a future *type* may specify additional behavior parameters via *arg*. *type* may be one of the following:

USYNC_PROCESS The condition variable can be used to synchronize threads in this process and other processes. Only one process should initialize the condition variable. *arg* is ignored.

USYNC_THREAD The condition variable can be used to synchronize threads in this process, only. *arg* is ignored.

Condition variables may also be initialized by allocation in zeroed memory. In this case a type of **USYNC_THREAD** is assumed. Multiple threads must not initialize the same condition variable simultaneously. A condition variable must not be re-initialized while other threads may be using it.

cond_destroy() destroys any state associated with the condition variable pointed to by *cvp*. The space for storing the condition variable is not freed. A condition variable must not be destroyed while other threads may be using it.

cond_wait() atomically releases the mutex pointed to by *mp* and causes the calling thread to block on the condition variable pointed to by *cvp*. The blocked thread may be awakened by ***cond_signal***(), ***cond_broadcast***(), or when interrupted by delivery of a signal or a ***fork***(). Any change in value of a condition associated with the condition variable cannot be inferred by the return of cond_wait() and any such condition must be re-evaluated.

cond_timedwait() is similar to ***cond_wait***(), except that the calling thread will not block past the time of day specified by *abstime*. If the time of day becomes greater than *abstime* then ***cond_timedwait***() returns with the error code **ETIME**.

cond_wait() and ***cond_timedwait***() always return with the mutex locked and owned by the calling thread even when returning an error.

cond_signal() unblocks one thread that is blocked on the condition variable pointed to by *cvp*.

cond_broadcast() unblocks all threads that are blocked on the condition variable pointed to by *cvp*.

If no threads are blocked on the condition variable then ***cond_signal***() and ***cond_broadcast***() have no effect.

Both functions should be called under the protection of the same mutex that is used with the condition variable being signaled. Otherwise the condition variable may be signaled between the test of the associated condition and blocking in ***cond_wait***(). This can cause an infinite wait.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, these functions fail and return the corresponding value:

EINVAL Invalid argument. For ***cond_init***(), *type* is not a recognized type. For ***cond_timedwait***(), the specified number of seconds, *abstime*, is greater than some implementation dependent time that is at least the start time of the application plus 50,000,000, or the number of nanoseconds is greater than or equal to 1,000,000,000.

If any of the following conditions are detected, ***cond_wait***() or ***cond_timedwait***() fails and returns the corresponding value:

EINTR The wait was interrupted by a signal or ***fork***().

If any of the following conditions are detected, ***cond_timedwait***() fails and returns the corresponding value:

ETIME The time specified by *abstime* has passed.

NOTES

These interfaces also available via: *#include* <*thread.h*>

By default, there is no defined order of unblocking if multiple threads are waiting for a condition variable.

fork1

NAME

fork1 - create a new process

SYNOPSIS

#include <sys/types.h>
#include <unistd.h>

pid_t fork1 (void);

DESCRIPTION

fork1() causes creation of a new process. It differs from ***fork***() in that ***fork***() duplicates all the threads in the parent process in the child process, while ***fork1***() duplicates only the calling thread in the child process.

RETURN VALUES

Upon successful completion, ***fork1***() returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of *(pid_t)-1* is returned to the parent process, no child process is created, and errno is set to indicate the error.

ERRORS

Same as ***fork***().

NOTES

When calling ***fork1***() the thread in the child must not depend on any resources that are held by threads that no longer exist in the child. In particular, locks held by these threads will not be released.

mutex_destroy, mutex_init, mutex_lock mutex_trylock, mutex_unlock

NAME

mutex, mutex_init, mutex_destroy, mutex_lock, mutex_trylock, mutex_unlock - mutual exclusion locks

SYNOPSIS

#include <synch.h>

int mutex_init (mutex_t *mp, int type, void *arg);

int mutex_destroy (mutex_t *mp);

int mutex_lock (mutex_t *mp);

int mutex_trylock (mutex_t *mp);

int mutex_unlock (mutex_t *mp);

DESCRIPTION

Mutual exclusion locks (mutexes) are used to serialize the execution of threads. They are typically used to ensure that only one thread executes a critical section of code at any one time (mutual exclusion).

Mutexes can be used to synchronize threads in this process and other processes if they are allocated in memory that is writable and shared among the cooperating processes (see ***mmap***(**KE_OS**)) and have been initialized for this behavior.

Mutexes must be initialized before use. ***mutex_init***() initializes the mutex pointed to by *mp*. A mutex can potentially have several different types of behavior, specified by *type*. No current *type* uses *arg* although a future type may specify additional behavior parameters via *arg*. *type* may be one of the following:

USYNC_PROCESS The mutex can be used to synchronize threads in this process and other processes. Only one process should initialize the mutex. *arg* is ignored.

USYNC_THREAD The mutex can be used to synchronize threads in this process, only. *arg* is ignored.

Mutexes may also be initialized by allocation in zeroed memory. In this case a type of **USYNC_THREAD** is assumed. Multiple threads must not initialize the same mutex simultaneously. A mutex lock must not be re-initialized while other threads may be using it.

mutex_destroy() destroys any state associated with the mutex pointed to by *mp*. The space for storing the mutex is not freed. A mutex lock must not be destroyed while other threads may be using it.

mutex_lock() locks the mutex pointed to by *mp*. If the mutex is already locked, the calling thread blocks until the mutex becomes available. When ***mutex_lock***() returns, the mutex is locked and the calling thread is the owner.

mutex_trylock() attempts to lock the mutex pointed to by *mp*. If the mutex is already locked it returns with an error. Otherwise the mutex is locked and the calling thread is the owner.

mutex_unlock() unlocks the mutex pointed to by *mp*. The mutex must be locked and the calling thread must be the one that last locked the mutex (the owner). If any other threads are waiting for the mutex to become available, one of them is unblocked. If the calling thread is not the owner of the lock, the behavior of the program is undefined.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, these functions fail and return the corresponding value:

EINVAL Invalid argument.

If any of the following conditions are detected, ***mutex_trylock***() fails and returns the corresponding value:

EBUSY The mutex pointed to by *mp* was already locked.

NOTES

In the current implementation, ***mutex_lock***(), ***mutex_unlock***(), and ***mutex_trylock***() do not validate the mutex type. Therefore, **EINVAL** is not returned for an uninitialized mutex or for a mutex with an invalid *type*. The behavior of these interfaces for mutexes containing an invalid *type* is unspecified. By default, there is no defined order of acquisition if multiple threads are waiting for a mutex. These interfaces are also available via: #*include* <*thread.h*>

rwlock_destroy, rwlock_init, rw_rdlock, rw_tryrdlock rw_trywrlock, rw_unlock, rw_wrlock

NAME

rwlock, rwlock_init, rwlock_destroy, rw_rdlock, rw_wrlock, rw_tryrdlock, rw_trywrlock, rw_unlock - multiple readers, single writer locks

SYNOPSIS

#include <synch.h>

int rwlock_init (rwlock_t *rwlp, int type, void *arg);

int rwlock_destroy (rwlock_t *rwlp);

int rw_rdlock (rwlock_t *rwlp);

int rw_wrlock (rwlock_t *rwlp);

int rw_unlock (rwlock_t *rwlp);

int rw_tryrdlock (rwlock_t *rwlp);

int rw_trywrlock (rwlock_t *rwlp);

DESCRIPTION

Multiple readers, single writer (readers/writer) locks allow many threads to have simultaneous read-only access to data while allowing only one thread to have write access at any given time. They are typically used to protect data that is searched more frequently than it is changed.

Readers/writer locks can be used to synchronize threads in this process and other processes if they are allocated in memory that is writable and shared among the cooperating processes (see ***mmap***(**KE_OS**)) and have been initialized for this behavior.

Readers/writer locks must be initialized before use. ***rwlock_init***() initializes the readers/writer lock pointed to by *rwlp*. A readers/writer lock can potentially have several different types of behavior, specified by *type*. No current type uses *arg* although a future type may specify additional behavior parameters via *arg*. *type* may be one of the following:

USYNC_PROCESS The readers/writer lock can be used to synchronize threads in this process and other processes. Only one process should initialize the readers/writer lock. *arg* is ignored.

USYNC_THREAD The readers/writer lock can be used to synchronize threads in this process, only. *arg* is ignored.

Readers/writer locks may also be initialized by allocation in zeroed memory. In this case a type of **USYNC_THREAD** is assumed. Multiple threads must not initialize the same readers/writer lock simultaneously. A readers/writer lock must not be re-initialized while other threads may be using it.

rwlock_destroy() destroys any state associated with the readers/writer lock pointed to by *rwlp*. The space for storing the readers/writer lock is not freed. A readers/writer lock must not be destroyed while other threads may be using it.

rw_rdlock() acquires a read lock on the readers/writer lock pointed to by *rwlp*. If the readers/writer lock is already locked for writing, the calling thread blocks until the write lock is released. More than one thread may hold a read lock on a readers/writer lock at any one time.

rw_tryrdlock() attempts to acquire a read lock on the readers/writer lock pointed to by *rwlp*. If the readers/writer lock is already locked for writing, it returns an error.

rw_wrlock() acquires a write lock on the readers/writer lock pointed to by *rwlp*. If the readers/writer lock is already locked for reading or writing, the calling thread blocks until all the read locks and write locks are released. Only one thread may hold a write lock on a readers/writer lock at any one time.

rw_trywrlock() attempts to acquire a write lock on the readers/writer lock pointed to by *rwlp*. If the readers/writer lock is already locked for reading or writing, it returns an error.

rw_unlock() unlocks a readers/writer lock pointed to by *rwlp*. The readers/writer lock must be locked and the calling thread must hold the lock either for reading or writing. If any other threads are waiting for the readers/writer lock to become available, one of them is unblocked. If the calling thread does not hold the lock for either reading or writing, the behavior of the program is undefined.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, these functions fail and return the corresponding value:

EINVAL Invalid argument.

If any of the following conditions are detected, ***rw_tryrdlock***() or ***rw_trywrlock***() fails and returns the corresponding value:

EBUSY The readers/writer lock pointed to by *rwlp* was already locked.

NOTES

These interfaces also available via: #*include* <*thread.h*>

By default, there is no defined order of acquisition if multiple threads are waiting for a readers/writer lock. However, implementations usually bias acquisition order in some way so as to avoid writer starvation.

sema_destroy, sema_init, sema_post
sema_trywait, sema_wait

NAME

semaphore, sema_init, sema_destroy, sema_wait, sema_trywait, sema_post - semaphores

SYNOPSIS

#include <synch.h>

int sema_init (sema_t *sp, unsigned int count, int type, void * arg);

int sema_destroy (sema_t *sp);

int sema_wait (sema_t *sp);

int sema_trywait (sema_t *sp);

***int sema_post (sema_t *sp)**;*

DESCRIPTION

Conceptually, a semaphore is a non-negative integer count. Semaphores are typically used to coordinate access to resources. The semaphore count is initialized to the number of free resources. Threads then atomically increment the count when resources are added and atomically decrement the count when resources are removed. When the semaphore count becomes zero, indicating no more resources are present, threads trying to decrement the semaphore will block until the count becomes greater than zero.

Semaphores can be used to synchronize threads in this process and other processes if they are allocated in memory that is writable and is shared among the cooperating processes (see ***mmap***(**KE_OS**)) and have been initialized for this behavior.

Semaphores must be initialized before use. ***sema_init***() initializes the semaphore pointed to by *sp* to *count*. A semaphore can potentially have several different types of behavior, specified by *type*. No current type uses *arg* although a future type may specify additional behavior parameters via *arg*. *type* may be one of the following:

USYNC_PROCESS The semaphore can be used to synchronize threads in this process and other processes. Only one process should initialize the semaphore. *arg* is ignored.

USYNC_THREAD The semaphore can be used to synchronize threads in this process, only. *arg* is ignored.

Multiple threads must not initialize the same semaphore simultaneously. A semaphore must not be re-initialized while other threads may be using it.

sema_destroy() destroys any state associated with the semaphore pointed to by *sp*. The space for storing the semaphore is not freed. A semaphore must not be destroyed while other threads may be using it.

sema_wait() blocks the calling thread until the count in the semaphore pointed to by *sp* becomes greater than zero and then atomically decrements it.

sema_trywait() atomically decrements the count in the semaphore pointed to by *sp* if the count is greater than zero. Otherwise it returns an error.

sema_post() atomically increments the count semaphore pointed to by *sp*. If there are any threads blocked on the semaphore, one is unblocked.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, these functions fail and return the corresponding value:

EINVAL Invalid argument.

If any of the following conditions are detected, ***sema_wait***() fails and returns the corresponding value:

~~**EINTR** The wait was interrupted by a signal.~~

EINTR The wait was interrupted by a signal or a fork

If any of the following conditions are detected, ***sema_trywait***() fails and returns the corresponding value:

EBUSY The semaphore pointed to by *sp* has a zero count.

NOTES

These interfaces also available via: *#include* <*thread.h*>

By default, there is no defined order of unblocking if multiple threads are waiting for a semaphore.

thr_continue, thr_suspend

NAME

thr_suspend, thr_continue - suspend or continue thread execution

SYNOPSIS

#include <thread.h>

int thr_suspend (thread_t target_thread);

int thr_continue (thread_t target_thread);

DESCRIPTION

thr_suspend() immediately suspends the execution of the thread specified by *target_thread*. On successful return from ***thr_suspend***(), the suspended thread is no longer executing. Once a thread is suspended, subsequent calls to ***thr_suspend***() have no effect.

thr_continue() resumes the execution of a suspended thread. Once a suspended thread is continued, subsequent calls to ***thr_continue***() have no effect.

A suspended thread will not be awakened by a signal. The signal stays pending until the execution of the thread is resumed by ***thr_continue***().

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, ***thr_suspend***() or ***thr_continue***() fails and returns the corresponding value:

ESRCH *target_thread* cannot be found in the current process.

thr_create

NAME

thr_create - create a new thread of control

SYNOPSIS

```
#include <thread.h>
int thr_create ( void        *stack_base,
                 size_t      stack_size,
                 void        *(*start_routine) (void *),
                 void        *arg,
                 long        flags,
                 thread_t    *new_thread
          );
```

DESCRIPTION

thr_create() adds a new thread of control to the current process. The new thread begins execution by calling the function specified by ***start_routine*** with a single argument, *arg*. If *start_routine* returns, the thread exits with the exit status set to the value returned by ***start_routine*** (see ***thr_exit***).

The new thread will use the stack starting at the address specified by *stack_base* and continuing for *stack_size* bytes. *stack_size* must be greater than the value returned by ***thr_min_stack***(). If *stack_base* is NULL then ***thr_create***() allocates a stack for the new thread with at least *stack_size* bytes. If *stack_size* is zero then a default size is used. If *stack_size* is not zero then it must be greater than the value returned by ***thr_min_stack***(). A stack of minimum size may not accommodate the stack frame for *start_function*. If a stack size is specified, it must take into account the requirements *start_function* and the functions that it may call in turn, in addition to the minimum requirement.

flags specifies additional attributes for the created thread. The value in *flags* is constructed from the bitwise inclusive OR of the following:

THR_SUSPENDED The new thread is created suspended and will not execute *start_routine* until it is started by ***thr_continue***().

THR_DETACHED The new thread is created detached. Its thread ID and other resources may be reused as soon as the thread terminates. A detached thread cannot be waited for via ***thr_join***().

THR_BOUND The new thread is created permanently bound to an **LWP** (i.e. it is a bound thread).

THR_NEW_LWP The desired concurrency level for unbound threads is increased by one. This is similar to incrementing concurrency by one via ***thr_setconcurrency***). Typically, this adds a new **LWP** to the pool of LWPs running unbound threads.

THR_DAEMON The thread is marked as a daemon. The process will exit when all non-daemon threads exit.

If both **THR_BOUND** and **THR_NEW_LWP** are specified then, typically, two **LWPs** are created, one for the bound thread and another for the pool of **LWPs** running unbound threads.

When *new_thread* is not NULL then it points to a location where the ID of the new thread is stored if ***thr_create***() is successful. The ID is only valid within the calling process.

The new thread inherits the calling thread's signal mask and priority. Pending signals are not inherited.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, ***thr_create***() fails and returns the corresponding value:

EAGAIN	A system limit is exceeded, e.g., too many **LWPs** were created.
ENOMEM	Not enough memory was available to create the new thread.
EINVAL	*stack_base* is not NULL and *stack_size* is less than the value returned by ***thr_min_stack***().
EINVAL	*stack_base* is NULL and *stack_size* is not zero and is less than the value returned by ***thr_min_stack***().

NOTES

Typically, thread stacks allocated by ***thr_create***() begin on page boundaries and any specified size is rounded up to the next page boundary. A page with no access permission is appended to the top of the stack so that most stack overflows will result in a **SIGSEGV** signal being sent to the offending thread. Thread stacks allocated by the caller are used as is.

Using a default stack size for the new thread, instead of passing a user-specified stack size, results in much better ***thr_create***() performance.

A thread has not terminated until ***thr_exit***() has finished. The only way to determine this is by ***thr_join***(). When ***thr_join***() returns a departed thread, it means that this thread has terminated and its resources are reclaimable. For instance, if a user specified a stack to ***thr_create***(), this stack can only be reclaimed after ***thr_join***() has reported this thread as a departed thread. It is not possible to determine when a detached thread has terminated. A detached thread disappears without leaving a trace.

If there is no explicit synchronization, an unsuspended, detached thread can die and have its thread ID re-assigned to another new thread before its creator returns from ***thr_create***().

thr_exit

NAME

thr_exit - thread termination

SYNOPSIS

#include <thread.h>

void thr_exit (void *status);

DESCRIPTION

thr_exit() terminates the calling thread. All thread-specific data bindings are released (see ***thr_keycreate***). If the calling thread is not detached, then the thread's ID and the exit status specified by *status* are retained until it is waited for (see ***thr_join***). Otherwise, *status* is ignored and the thread's ID may be reclaimed immediately.

If the calling thread is the last non-daemon thread in the process (see ***thr_create***), then the process terminates with a status of zero (see ***exit***(**BA_OS**)). If the initial thread returns from *main*() then the process exits with a status equal to the return value.

RETURN VALUE

thr_exit() does not return.

thr_getconcurrency, thr_setconcurrency

NAME

thr_setconcurrency, thr_getconcurrency - get/set thread concurrency level

SYNOPSIS

#include <thread.h>

int thr_setconcurrency (int new_level);

int thr_getconcurrency (void);

DESCRIPTION

Unbound threads in a process (see ***thr_create***) may or may not be required to be simultaneously active. By default, the threads system ensures that a sufficient number of threads are active so that the process can continue to make progress. While this conserves system resources, it may not produce the most effective level of concurrency. ***thr_setconcurrency***() permits the application to give the threads system a hint, specified by *new_level*, for the desired level of concurrency. The actual number of simultaneously active threads may be larger or smaller than this number. The value for the desired concurrency level may also be affected by creating threads with the **THR_NEW_LWP** flag set (see ***thr_create***).

If *new_level* is zero, the threads system will only ensure that a sufficient number of threads are active so that the process can continue to make progress.

thr_getconcurrency() returns the current value for the desired concurrency level. The actual number of simultaneously active threads may be larger or smaller than this number.

RETURN VALUE

thr_setconcurrency() returns zero when successful. A nonzero value indicates an error code.

thr_getconcurrency() always returns the current value for the desired concurrency level.

ERRORS

If any of the following conditions are detected, ***thr_setconcurrency***() fails and returns the corresponding value:

EAGAIN	the specified concurrency level would cause a system resource to be exceeded.
EINVAL	*new_level* is negative.

thr_getprio, thr_setprio

NAME

thr_setprio, thr_getprio - set/get a thread priority

SYNOPSIS

#include <thread.h>

int thr_setprio (thread_t target_thread, int pri);

int thr_getprio (thread_t target_thread, int *pri);

DESCRIPTION

Each thread has a priority which it inherits from its creator. ***thr_setprio***() changes the priority of the thread, specified by *target_thread*, within the current process to the priority specified by *pri*. By default, threads are scheduled based on fixed priorities that range from zero, the least significant, to the largest integer. The *target_thread* will preempt lower priority threads, and will yield to higher priority threads.

The function ***thr_getprio***() stores the current priority for the thread specified by *target_thread* in the location pointed to by *pri*.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates error.

ERRORS

If any of the following conditions are detected, ***thr_setprio***() or ***thr_getprio***() fails and returns the corresponding value:

ESRCH *target_thread* cannot be found in the current process.

If any of the following conditions are detected, ***thr_setprio***() fails and returns the corresponding value:

EINVAL The value of *pri* makes no sense for the scheduling class associated with the *target_thread*.

thr_getspecific, thr_keycreate, thr_setspecific

NAME

thr_keycreate, thr_setspecific, thr_getspecific - thread-specific data

SYNOPSIS

#include <thread.h>

int thr_keycreate(thread_key_t *keyp, void (*destructor) (void *value));

int thr_setspecific(thread_key_t key, void *value);

int thr_getspecific(thread_key_t key, void **valuep);

DESCRIPTION

thr_keycreate() allocates a key that is used to identify data that is specific to each thread in the process. The key is global to all threads in the process. Once a key has been created each thread may bind a value to the key. The values are specific to the binding thread and are maintained for each thread independently. All threads initially have the value NULL associated with the key when it is created. When ***thr_keycreate***() returns successfully the allocated key is stored in the location pointed to by *keyp*. The caller must ensure that storage and access to this key are properly synchronized.

An optional destructor function, specified by *destructor*, may be associated with each key. If a key has a non-NULL destructor function and the thread has a non-NULL value associated with that key, the destructor function is called with the current associated *value* when the thread exits. The order in which the destructor functions are called for all the allocated keys is unspecified.

thr_setspecific() binds *value* to the thread-specific data *key*, for the calling thread.

thr_getspecific() stores the current value bound to *key* for the calling thread into the location pointed to by *valuep*.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, ***thr_keycreate***() fails and returns the corresponding value:

EAGAIN The key name space is exhausted.

If any of the following conditions are detected, ***thr_keycreate***() or ***thr_setspecific***() fails and returns the corresponding value:

ENOMEM Not enough memory is available.

If any of the following conditions are detected, ***thr_setspecific***() or ***thr_getspecific***() fails and returns the corresponding value:

EINVAL *key* is invalid.

thr_join

NAME

thr_join - wait for thread termination

SYNOPSIS

#include <thread.h>

int thr_join (thread_t wait_for, thread_t *departed, void **status);

DESCRIPTION

thr_join() blocks the calling thread until the thread specified by *wait_for* terminates. The specified thread must be in the current process and must not be detached (see ***thr_create***). If *wait_for* is (*thread_t*)0, then ***thr_join***() waits for any undetached thread in the process to terminate.

If *departed* is not NULL, it points to a location that is set to the ID of the terminated thread if ***thr_join***() returns successfully. If *status* is not NULL, it points to a location that is set to the exit status of the terminated thread if ***thr_join***() returns successfully.

If ***thr_join***() is not successful, the value of the location pointed to by status is unchanged.

Multiple threads cannot wait for the same thread to terminate; one thread will return successfully and the others will fail with an error of **ESRCH**.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, ***thr_join***() fails and returns the corresponding value:

ESRCH	*wait_for* is not a valid, undetached thread in the current process.
EDEADLK	*wait_for* specifies the calling thread.
EDEADLCK	wait_for is (*thread_t*)0 and there is no valid, undetached thread in the current process which is not the calling thread.

thr_kill

NAME

thr_kill - send a signal to a thread

SYNOPSIS

#include <thread.h>
#include <signal.h>

int thr_kill (thread_t target_thread, int sig);

DESCRIPTION

thr_kill() sends the signal, *sig*, to the thread specified by *target_thread*. *target_thread* must be a thread within the same process as the calling thread. *sig* must be one from the list given in ***signal*** (**BA_ENV**)or zero. If *sig* is zero, error checking is performed but no signal is actually sent. This can be used to check the validity of *target_thread*.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions detected, ***thr_kill***() fails and returns the corresponding value:

EINVAL	*sig* is not a valid signal number.
ESRCH	*target_thread* cannot be found in the current process.

thr_min_stack

NAME

thr_min_stack - minimum stack space for a thread

SYNOPSIS

#include <thread.h>

size_t thr_min_stack(void);

DESCRIPTION

When a thread is created with a user-supplied stack, the user must reserve enough space to run this thread. In a dynamically linked execution environment, it is very hard to know what the minimum stack requirements are for a thread. The function ***thr_min_stack***() returns the amount of space needed to execute a null thread. This is a thread that was created to execute a null procedure. A thread that does something useful should have a stack size that is ***thr_min_stack***() + <some increment>.

Most users should not be creating threads with user-supplied stacks. This functionality was provided to support applications that wanted complete control over their execution environment.

Typically, users should let the threads library manage stack allocation. The threads library provides default stacks which should meet the requirements of any created thread.

RETURN VALUE

thr_min_stack returns the minimum stack size for a thread.

thr_self

NAME

thr_self - get thread identifier

SYNOPSIS

#include <thread.h>

thread_t thr_self(void)

DESCRIPTION

thr_self() returns the ID of the calling thread.

thr_sigsetmask

NAME

thr_sigsetmask - change and/or examine calling thread's signal mask

SYNOPSIS

#include <thread.h>

#include <signal.h>

int thr_sigsetmask (int how, const sigset_t *set, sigset_t *oset);

DESCRIPTION

thr_sigsetmask() examines and/or changes the calling thread's signal mask. If the value of the argument *set* is not NULL, it points to a set of signals to be used to change the currently blocked set. The value of the argument how determines the manner in which the set is changed. *how* may have one of the following values:

SIG_BLOCK	*set* represent a set of signals to block. They are added to the current signal mask.
SIG_UNBLOCK	*set* represents a set of signals to unblock. These signals are deleted from the current signal mask.
SIG_SETMASK	*set* represents the new signal mask. The current signal mask is replaced by set.

If the value of *oset* is not NULL, it points to space where the previous signal mask is stored. If the value of *set* is NULL, the value of *how* is not significant and the thread's signal mask is unchanged; thus, ***thr_sigsetmask***() can be used to enquire about the currently blocked signals.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, ***thr_sigsetmask***() fails and returns the corresponding value:

EINVAL	*set* is not NULL and the value of *how* is not defined.

NOTES

It is not possible to block those signals that cannot be ignored (see ***sigaction***(**BA_OS**)). In addition, if using the threads library, it is not possible to block the signal **SIGLWP**, reserved by the threads library, and it is not possible to unblock the signal **SIGWAITING**, which is always blocked on all threads. This restriction is silently imposed by the threads library.

thr_main

NAME

thr_main - identify the main thread

SYNOPSIS

#include <thread.h>

int thr_main(void);

DESCRIPTION

thr_main **—** identifies the calling thread as the main thread or not the main thread.

RETURN VALUES

thr_main() returns:

1	if the calling thread is the main thread.
0	if the calling thread is not the main thread.
-1	if *libthread* is not linked in or thread initialization has not completed.

thr_yield

NAME

thr_yield - yield execution to another thread

SYNOPSIS

#include <thread.h>

void thr_yield(void);

DESCRIPTION

thr_yield() causes the current thread to yield its execution in favor of another thread with the same or greater priority.

RETURN VALUE

No value is returned.

sigwait

NAME

sigwait - wait until a signal is posted

SYNOPSIS

#include <signal.h>

int sigwait (sigset_t *set);

DESCRIPTION

sigwait() selects a signal in set that is pending on the calling thread (see ***thr_create***()). If no signal in set is pending, then ***sigwait***() blocks until a signal in set becomes pending. The selected signal is cleared from the set of signals pending on the calling thread and the number of the signal is returned. The selection of a signal in set is independent of the signal mask of the calling thread. This means a thread can synchronously wait for signals that are being blocked by the signal mask of the calling thread.

If more than one thread waits for the same signal, only one is unblocked when the signal arrives.

RETURN VALUES

Upon successful completion, a signal number is returned. Otherwise, a value of -1 is returned and errno is set to indicate error.

ERRORS

If any of the following conditions are detected, ***sigwait***() fails and returns the corresponding value:

EINVAL	set contains an unsupported signal number.
EFAULT	set points to an invalid address.

NOTES

sigwait() cannot be used to wait for signals that cannot be caught (see ***sigaction***(**BA_OS**)). This restriction is silently imposed by the system.

sigwait() is designated as **EXPERMIMENTAL** since it has an interface which is different from the one in **POSIX 1003.1c**. sigwait interface in **POSIX** is as following:

int ***sigwait*** *(const sigset_t *setp, int *signo);*

SPARC COMPLIANCE DEFINITION 2.4.1 IS

libucb

nice

NAME

nice - change priority of a process

SYNOPSIS

#include <unistd.h>

int nice(int incr);

DESCRIPTION

The ***nice()*** function allows a process to change its priority. The invoking process must be in a scheduling class that supports the ***nice()***. The ***priocntl()*** function is a more general interface to scheduler functions.

nice() adds the value of incr to the ***nice*** value of the calling process. A process' ***nice*** value is a non-negative number for which a greater positive value results in lower CPU priority.

A maximum ***nice*** value of 2 * **NZERO** -1 and a minimum ***nice*** value of 0 are imposed by the system. **NZERO** is defined in <limits.h> with a default value of 20. Requests for values above or below these limits result in the ***nice*** value being set to the corresponding limit. A ***nice*** value of 40 is treated as 39. Only a process with super-user privileges can lower the ***nice*** value.

RETURN VALUES

Upon successful completion, ***nice()*** returns the new ***nice*** value minus **NZERO** . Otherwise, a value of -1 is returned, the process' ***nice*** value is not changed, and errno is set to indicate the error.

ERRORS

nice() fails if one or more of the following are true:

EINVAL ***nice()*** is called by a process in a scheduling class other than time-sharing.

EPERM *inc* is negative or greater than 40 and the effective user ID of the calling process is not superuser.

USAGE

As -1 is a permissible return value in a successful situation, an application wishing to check for error situations should set errno to 0, then call ***nice()***, and if it returns -1, check to see if errno is non-zero.

SEE ALSO

nice, ***exec()***, ***priocntl()***

setjmp
longjmp
_setjmp
_longjmp

NAME

setjmp, longjmp, _setjmp, _longjmp - non-local goto

SYNOPSIS

#include <setjmp.h>

int	***setjmp(jmp_buf env);***
void	***longjmp(jmp_buf env,int val);***
int	***_setjmp(jmp_buf env);***
void	***_longjmp(jmp_buf env,int val);***

DESCRIPTION

setjmp() and ***longjmp()*** are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program. ***setjmp()*** saves its stack environment in env for later use by ***longjmp()***. A normal call to ***setjmp()*** returns zero.***setjmp()*** also saves the register environment. If a ***longjmp()*** call will be made, the routine which called ***setjmp()*** should not return until after the ***longjmp()*** has returned control (see below).

longjmp() restores the environment saved by the last call of ***setjmp()***, and then returns in such a way that execution continues as if the call of ***setjmp()*** had just returned the value val to the function that invoked ***setjmp()***; however, if val were zero, execution would continue as if the call of ***setjmp()*** had returned one. This ensures that a ``return'' from ***setjmp()*** caused by a call to ***longjmp()*** can be distinguished from a regular return from ***setjmp()***. The calling function must not itself have returned in the interim, otherwise ***longjmp()*** will be returning control to a possibly non-existent environment. All memory-bound data have values as of the time ***longjmp()*** was called. The CPU and floating point data registers are restored to the values they had at the time that ***setjmp()*** was called. But, because the register storage class is only a hint to the C compiler, variables declared as register variables may not necessarily be assigned to machine registers, so their values are unpredictable after a ***longjmp()***. This is especially a problem for programmers trying to write machine-independent C routines.

setjmp() and ***longjmp()*** save and restore the signal mask while ***_setjmp()*** and ***_longjmp()*** manipulate only the C stack and registers. None of these functions save or restore any floating-point status or control registers.

SEE ALSO

sigvec()-BSD, ***setjmp()***, ***signal()***

NOTES

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.

scandir
alphasort

NAME

scandir, alphasort - scan a directory

SYNOPSIS

#include <sys/types.h>

#include <sys/dir.h>

int scandir(char *dirname, struct direct *(*namelist[]), int (*select)(.), (*dcomp)());

int alphasort(struct direct **d1, **d2);

DESCRIPTION

The ***scandir()*** function reads the directory dirname and builds an array of pointers to directory entries using ***malloc()***. The second parameter is a pointer to an array of structure pointers. The third parameter is a pointer to a routine which is called with a pointer to a directory entry and should return a non zero value if the directory entry should be included in the array. If this pointer is NULL, then all the directory entries will be included. The last argument is a pointer to a routine which is passed to ***qsort()***, which sorts the completed array. If this pointer is NULL, the array is not sorted.

The ***alphasort()*** function is a routine that sorts the array alphabetically.

RETURN VALUES

The ***scandir()*** function returns the number of entries in the array and a pointer to the array through the parameter namelist. The ***scandir()*** function returns -1 if the directory cannot be opened for reading or if ***malloc()*** cannot allocate enough memory to hold all the data structures.

SEE ALSO

getdents(),***malloc()***,***qsort()***,***readdir()***-BSD, ***readdir(***

NOTES

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.

fopen

NAME

fopen, freopen - open a stream

SYNOPSIS

#include <stdio.h>

FILE *fopen(const char *file, *mode);

FILE *freopen(const char *file, *mode, register FILE *iop);

DESCRIPTION

fopen() opens the file named by file and associates a stream with it. If the open succeeds, ***fopen()*** returns a pointer to be used to identify the stream in subsequent operations. file points to a character string that contains the name of the file to be opened. *mode* is a character string having one of the following values:

r	open for reading
w	truncate or create for writing
a	append: open for writing at end of file, or create for writing
r+	open for update (reading and writing)
w+	truncate or create for update
a+	append; open or create for update at **EOF**

freopen() opens the file named by file and associates the stream pointed to by iop with it. The mode argument is used just as in ***fopen()***. The original stream is closed, regardless of whether the open ultimately succeeds. If the open succeeds, ***freopen()*** returns the original value of iop. ***freopen()*** is typically used to attach the preopened streams associated with *stdin*, *stdout*, and *stderr* to other files.When a file is opened for update, both input and output may be done on the resulting stream. However, output may not be directly followed by input without an intervening ***fseek()*** or ***rewind()***, and input may not be directly followed by output without an intervening ***fseek()*** or ***rewind()***. An input operation which encounters **EOF** will fail.

RETURN VALUES

fopen() and ***freopen()*** return a NULL pointer on failure.

SEE ALSO

open(), ***fclose()***, ***fopen()***, ***freopen()***, ***fseek()***, ***malloc()***, ***rewind()***

NOTES

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported. In order to support the same number of open files that the system does, ***fopen()*** must allocate additional memory for data structures using ***malloc()*** after 64 files have been opened. This confuses some programs which use their own memory allocators. The interfaces of ***fopen()*** and ***freopen()*** differ from the Standard I/O Functions ***fopen()*** and ***freopen()***. The Standard I/O Functions distinguish binary from text files with an additional use of 'b' as part of the mode.

gettimeofday
settimeofday

NAME

gettimeofday, settimeofday - get or set the date and time

SYNOPSIS

#include <sys/time.h>

int gettimeofday(struct timeval *tzp, struct timezone *tzp);

int settimeofday(struct timeval *tzp, struct timezone *tzp);

DESCRIPTION

The system's notion of the current Greenwich time is obtained with the ***gettimeofday()*** call, and set with the ***settimeofday()*** call. The current time is expressed in elapsed seconds and microseconds since 00:00 GMT, January 1, 1970 (zero hour). The resolution of the system clock is hardware dependent; the time may be updated continuously, or in clock ticks. tp points to a timeval structure, which includes the following members:

long	*tv_sec;*	*/* seconds since Jan. 1, 1970 */*
long	*tv_usec;*	*/* and microseconds */*

If tp is a NULL pointer, the current time information is not returned or set. tzp is an obsolete pointer formerly used to get and set timezone information. tzp is now ignored. Timezone information is now handled using the **TZ** environment variable; see **TIMEZONE**. Only the privileged user may set the time of day.

RETURN VALUES

A -1 return value indicates an error occurred; in this case an error code is stored in the global variable errno.

ERRORS

The following error codes may be set in errno:

EINVAL	tp specifies an invalid time.
EPERM	A user other than the privileged user attempted to set the time.

SEE ALSO

adjtime(), ***ctime()***, ***gettimeofday()***, **TIMEZONE**

NOTES

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.tv_usec is always 0.

mctl

NAME

mctl - memory management control

SYNOPSIS

#include <sys/types.h>

#include <sys/mman.h>

int mctl(caddr_t addr, size_t le n, int function, int arg);

DESCRIPTION

mctl() applies a variety of control functions over pages identified by the mappings established for the address range [*addr*, *addr* + *len*). The function to be performed is identified by the argument function. Valid functions are defined in mman.h as follows:

MC_LOCK	Lock the pages in the range in memory. This function is used to support ***mlock()***. See ***mlock()*** for semantics and usage. arg is ignored.
MC_LOCKAS	Lock the pages in the address space in memory. This function is used to support ***mlockall()***. See ***mlockall()*** for semantics and usage. addr and len are ignored. arg is an integer built from the flags:
MCL_CURRENT	Lock current mappings
MCL_FUTURE	Lock future mappings
MC_SYNC	Synchronize the pages in the range with their backing storage. Optionally invalidate cache copies. This function is used to support ***msync()***. See ***msync()*** for semantics and usage. arg is used to represent the flags argument to ***msync()***. It is constructed from an OR of the following values:
MS_SYNC	Synchronized write
MS_ASYNC	Return immediately
MS_INVALIDATE	Invalidate mappings
MS_ASYNC	returns after all I/O operations are scheduled. **MS_SYNC** does not return until all I/O operations are complete. Specify exactly one of **MS_ASYNC** or **MS_SYNC**. **MS_INVALIDATE** invalidates all cached copies of data from memory, requiring them to be re-obtained from the object's permanent storage location upon the next reference.
MC_UNLOCK	Unlock the pages in the range. This function is used to support ***munlock()***. arg is ignored.
MC_UNLOCKAS	Remove address space memory lock, and locks on all current mappings. This function is used to support ***munlockall()***. addr and len must have the value 0. arg is ignored.

RETURN VALUES

mctl() returns 0 on success, -1 on failure.

ERRORS

mctl() fails if:

EAGAIN	Some or all of the memory identified by the operation could not be locked due to insufficient system resources.
EBUSYMS_INVALIDATE	was specified and one or more of the pages is locked in memory.
EINVAL	addr is not a multiple of the page size as returned by ***getpagesize()***.
EINVAL	addr and/or len do not have the value 0 when **MC_LOCKAS** or **MC_UNLOCKAS** are specified.
EINVAL	arg is not valid for the function specified.
EIO	An I/O error occurred while reading from or writing to the file system.
ENOMEM	Addresses in the range [addr, addr + len) are invalid for the address space of a process, or specify one or more pages which are not mapped.
EPERM	The process's effective user ID is not super-user and one of **MC_LOCK** **MC_LOCKAS**, **MC_UNLOCK**, or **MC_UNLOCKAS** was specified.

SEE ALSO

mmap(),***memcntl()***,***getpagesize()***,***mlock()***, ***mlockall()***, ***msync()***

NOTES

Use of these interfaces should be restricted to only applications written on **BSD** platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.

psignal
sys_siglist

NAME

psignal, sys_siglist - system signal messages

SYNOPSIS

void psignal (unsigned sig, char *s);

char *sys_siglist[];

DESCRIPTION

psignal() produces a short message on the standard error file describing the indicated signal. First the argument string s is printed, then a colon, then the name of the signal and a **NEWLINE**. Most usefully, the argument string is the name of the program which incurred the signal. The signal number should be from among those found in <*signal.h*>.

To simplify variant formatting of signal names, the vector of message strings *sys_siglist* is provided; the signal number can be used as an index in this table to get the signal name without the newline. The define **NSIG** defined in <*signal.*h> is the number of messages provided for in the table; it should be checked because new signals may be added to the system before they are added to the table.

SEE ALSO

perror(), ***signal()***

NOTES

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.

rand
srand

NAME

rand, srand - simple random number generator

SYNOPSIS

int rand()

int srand(unsigned seed);

DESCRIPTION

rand() uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo-random numbers in the range from 0 to "2^{31} -1."

srand() can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

SEE ALSO

drand48(), ***rand()***, ***random()***

NOTES

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.The spectral properties of ***rand()*** leave a great deal to be desired. ***drand48()*** and ***random()*** provide much better, though more elaborate, random-number generators. The low bits of the numbers generated are not very random; use the middle bits. In particular the lowest bit alternates between 0 and 1.

sigblock
sigmask
sigpause
sigsetmask

NAME

sigblock, sigmask, sigpause, sigsetmask - block signals

SYNOPSIS

#include <signal.h>

int sigblock(int mask);

int sigmask(int signum);

int sigpause(int mask);

int sigsetmask(int mask);

DESCRIPTION

sigblock() adds the signals specified in mask to the set of signals currently being blocked from delivery. Signals are blocked if the appropriate bit in mask is a 1; the macro sigmask is provided to construct the mask for a given signum. ***sigblock()*** returns the previous mask. The previous mask may be restored using ***sigsetmask()***.

sigpause() assigns mask to the set of masked signals and then waits for a signal to arrive; on return the set of masked signals is restored. mask is usually 0 to indicate that no signals are now to be blocked. ***sigpause()*** always terminates by being interrupted, returning -1 and setting errno to **EINTR**.

sigsetmask() sets the current signal mask (those signals that are blocked from delivery). Signals are blocked if the corresponding bit in mask is a 1; the macro sigmask is provided to construct the mask for a given signum. In normal usage, a signal is blocked using ***sigblock()***. To begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using ***sigpause()*** with the mask returned by ***sigblock()***. It is not possible to block **SIGKILL**, **SIGSTOP**, or **SIGCONT**, this restriction is silently imposed by the system.

RETURN VALUES

sigblock() and ***sigsetmask()*** return the previous set of masked signals. ***sigpause()*** returns -1 and sets errno to **EINTR**.

SEE ALSO

kill(), ***sigaction()***, ***signal()***-BSD, ***sigvec()***-BSD

NOTES

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.

siginterrupt

NAME

siginterrupt - allow signals to interrupt functions

SYNOPSIS

int siginterrupt(int sig, flag);

DESCRIPTION

siginterrupt() is used to change the function restart behavior when a function is interrupted by the specified signal. If the flag is false (0), then functions will be restarted if they are interrupted by the specified signal and no data has been transferred yet. System call restart is the default behavior when the ***signal()*** routine is used.

If the flag is true , then restarting of functions is disabled. If a function is interrupted by the specified signal and no data has been transferred, the function will return -1 with errno set to **EINTR**. Interrupted functions that have started transferring data will return the amount of data actually transferred. Issuing a ***siginterrupt()*** call during the execution of a signal handler will cause the new action to take place on the next signal to be caught.

NOTES

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.

This library routine uses an extension of the ***sigvec()***-BSD function that is not available in **4.2 BSD**, hence it should not be used if backward compatibility is needed.

RETURN VALUES

A 0 value indicates that the call succeeded. A -1 value indicates that the call failed and errno is set to indicate the error.

ERRORS

siginterrupt() may return the following error:

EINVAL sig is not a valid signal.

SEE ALSO

sigblock()-BSD, ***sigvec()***-BSD, ***signal()***

signal

NAME

signal - simplified software signal facilities

SYNOPSIS

#include <signal.h>

void (*signal(int sig, void (*func)()))()

DESCRIPTION

signal() is a simplified interface to the more general ***sigvec()***-BSD facility. Programs that use ***signal()*** in preference to ***sigvec()*** are more likely to be portable to all systems. A signal is generated by some abnormal event, initiated by a user at a terminal (quit, interrupt, stop), by a program error (bus error, etc.), by request of another program (**kill**), or when a process is stopped because it wishes to access its control terminal while in the background (see **termio**). Signals are optionally generated when a process resumes after being stopped, when the status of child processes changes, or when input is ready at the control terminal. Most signals cause termination of the receiving process if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the **SIGKILL** and **SIGSTOP** signals, the ***signal()*** call allows signals either to be ignored or to interrupt to a specified location. See ***sigvec()***-BSD for a complete list of the signals. If func is **SIG_DFL**, the default action for signal sig is reinstated; this default is termination (with a core image for starred signals) except for signals marked with @ or |+. Signals marked with @ are discarded if the action is **SIG_DFL**; signals marked with |+ cause the process to stop. If func is **SIG_IGN** the signal is subsequently ignored and pending instances of the signal are discarded. Otherwise, when the signal occurs further occurrences of the signal are automatically blocked and func is called. A return from the function unblocks the handled signal and continues the process at the point it was interrupted. If a caught signal occurs during certain functions, terminating the call prematurely, the call is automatically restarted. In particular this can occur during a ***read()*** or ***write()*** on a slow device (such as a terminal; but not a file) and during a ***wait()***. The value of ***signal()*** is the previous (or initial) value of func for the particular signal. After a ***fork()*** or ***vfork()*** the child inherits all signals. An ***exec()*** resets all caught signals to the default action; ignored signals remain ignored.

RETURN VALUES

The previous action is returned on a successful call. Otherwise, -1 is returned and errno is set to indicate the error.

ERRORS

signal() will fail and no action will take place if the following occurs:

EINVAL *sig* is not a valid signal number, or is **SIGKILL** or **SIGSTOP**.

SEE ALSO

kill, ***exec()***, ***fcntl()***, ***fork()***, ***getitimer()***, ***getrlimit()***, ***kill()***, ***ptrace()***, ***read()***, ***sigaction()***, ***wait()***, ***write()***, ***abort()***, ***setjmp()***-BSD, ***sigblock()***-BSD, ***sigstack()***-BSD, ***sigvec()***-BSD, ***wait()***-BSD, ***setjmp()***, ***signal()***, signal, termio

sigstack

NAME

sigstack - set and/or get signal stack context

SYNOPSIS

#include <signal.h>

int sigstack(struct sigstack *nss, *oss);

DESCRIPTION

The ***sigstack()*** function allows users to define an alternate stack, called the “signal stack”, on which signals are to be processed. When a signal's action indicates its handler should execute on the signal stack (specified with a ***sigvec()***-BSD call), the system checks to see if the process is currently executing on that stack. If the process is not currently executing on the signal stack, the system arranges a switch to the signal stack for the duration of the signal handler's execution. A signal stack is specified by a ***sigstack()*** structure, which includes the following members:

char	**ss_sp;*	*/* signal stack pointer */*
int	*ss_onstack;*	*/* current status */*

The *ss_sp* member is the initial value to be assigned to the stack pointer when the system switches the process to the signal stack. Note that, on machines where the stack grows downwards in memory, this is not the address of the beginning of the signal stack area. The *ss_onstack* member is zero or non-zero depending on whether the process is currently executing on the signal stack or not. If *nss* is not a null pointer, ***sigstack()*** sets the signal stack state to the value in the ***sigstack()*** structure pointed to by *nss*. If nss is a null pointer, the signal stack state will be unchanged. If *oss* is not a null pointer, the current signal stack state is stored in the ***sigstack()*** structure pointed to by *oss*.

RETURN VALUES

Upon successful completion, 0 is returned. Otherwise, -1 is returned and errno is set to indicate the error.

ERRORS

The ***sigstack()*** function will fail and the signal stack context will remain unchanged if one of the following occurs.

EFAULT Either *nss* or *oss* points to memory that is not a valid part of the process address space.

SEE ALSO

sigaltstack(), ***sigvec()***-BSD, ***signal()***

NOTES

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.

sigvec

NAME

sigvec - software signal facilities

SYNOPSIS

#include <signal.h>

int sigvec(int sig, struct sigvec *nvec, *ovec);

DESCRIPTION

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify a handler to which a signal is delivered, or specify that a signal is to be blocked or ignored. A process may also specify that a default action is to be taken by the system when a signal occurs. Normally, signal handlers execute on the current stack of the process. This may be changed, on a per-handler basis, so that signals are taken on a special signal stack.

All signals have the same priority. Signal routines execute with the signal that caused their invocation to be blocked, but other signals may yet occur. A global signal mask defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally 0). It may be changed with a ***sigblock()*** or ***sigsetmask()*** call, or when a signal is delivered to the process. A process may also specify a set of flags for a signal that affect the delivery of that signal.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently blocked by the process then it is delivered to the process. When a signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally the process will resume execution in the context from before the signal's delivery. If the process wishes to resume in a different context, then it must arrange to restore the previous context itself.

When a signal is delivered to a process a new signal mask is installed for the duration of the process' signal handler (or until a ***sigblock()*** or ***sigsetmask()*** call is made). This mask is formed by taking the current signal mask, adding the signal to be delivered, and ORing in the signal mask associated with the handler to be invoked. The action to be taken when the signal is delivered is specified by a ***sigvec()*** structure, which includes the following members:

*void(*sv_handler)();*	*/* signal handler */*
int sv_mask;	*/* signal mask to apply */*
int sv_flags;	*/* see signal options */*
#define **SV_ONSTACK**	/* take signal on signal stack */
#define **SV_INTERRUPT**	/* do not restart system on signal return */
#define **SV_RESETHAND**	/* reset handler to **SIG_DFL** when signal taken*/

If the **SV_ONSTACK** bit is set in the flags for that signal, the system will deliver the signal to the process on the signal stack specified with ***sigstack()***-BSD rather than delivering the signal on the current stack.

If *nvec* is not a NULL pointer, ***sigvec()*** assigns the handler specified by ***sv_handler()***, the mask specified by ***sv_mask()***, and the flags specified by ***sv_flags()*** to the specified signal. If nvec is a NULL pointer, ***sigvec()*** does not change the handler, mask, or flags for the specified signal.

The mask specified in nvec is not allowed to block **SIGKILL**, **SIGSTOP**, or **SIGCONT**. The system enforces this restriction silently.

If ovec is not a NULL pointer, the handler, mask, and flags in effect for the signal before the call to ***sigvec()*** are returned to the user. A call to ***sigvec()*** with nvec a NULL pointer and ovec not a NULL pointer can be used to determine the handling information currently in effect for a signal without changing that information.

The following is a list of all signals with names as in the include file <*signal.h*>:

SIGHUP	hangup
SIGINT	interrupt
SIGQUIT*	quit
SIGILL*	illegal instruction
SIGTRAP*	trace trap
SIGABRT*	abort (generated by ***abort()*** routine)
SIGEMT*	emulator trap
SIGFPE*	arithmetic exception
SIGKILL	kill (cannot be caught, blocked, or ignored)
SIGBUS*	bus error
SIGSEGV*	segmentation violation
SIGSYS*	bad argument to function
SIGPIPE	write on a pipe or other socket with no one to read it
SIGALRM	alarm clock
SIGTERM	software termination signal
SIGURG@	urgent condition present on socket
SIGSTOP|+	stop (cannot be caught, blocked, or ignored)
SIGTSTP|+	stop signal generated from keyboard
SIGCONT@	continue after stop (cannot be blocked)
SIGCHLD@	child status has changed
SIGTTIN|+	background read attempted from control terminal
SIGTTOU|+	background write attempted to control terminal
SIGIO@	I/O is possible on a descriptor (see ***fcntl()***)
SIGXCPU	cpu time limit exceeded (see ***getrlimit()***)

SIGXFSZ	file size limit exceeded (see ***getrlimit()***)
SIGVTALRM	virtual time alarm; see ***setitimer()*** on ***getitimer()***
SIGPROF	profiling timer alarm; see ***setitimer()*** on ***getitimer()***
SIGWINCH@	window changed (see ***termio***)
SIGLOST	resource lost (see ***lockd***)
SIGUSR1	user-defined signal 1
SIGUSR2	user-defined signal 2

The starred signals in the list above cause a core image if not caught or ignored.

Once a signal handler is installed, it remains installed until another ***sigvec()*** call is made, or an ***execve()*** is performed, unless the **SV_RESETHAND** bit is set in the flags for that signal. In that case, the value of the handler for the caught signal will be set to **SIG_DFL** before entering the signal-catching function, unless the signal is **SIGILL**, **SIGPWR**, or **SIGTRAP**. Also, if this bit is set, the bit for that signal in the signal mask will not be set; unless the signal mask associated with that signal blocks that signal, further occurrences of that signal will not be blocked. The **SV_RESETHAND** flag is not available in **4.2 BSD**, hence it should not be used if backward compatibility is needed.

The default action for a signal may be reinstated by setting the signal's handler to **SIG_DFL**; this default is termination except for signals marked with @ or |+. Signals marked with @ are discarded if the action is **SIG_DFL**; signals marked with |+ cause the process to stop. If the process is terminated, a “core image” will be made in the current working directory of the receiving process if the signal is one for which an asterisk appears in the above list. If the handler for that signal is **SIG_IGN**, the signal is subsequently ignored, and pending instances of the signal are discarded.

If a caught signal occurs during certain functions, the call is normally restarted. The call can be forced to terminate prematurely with an **EINTR** error return by setting the **SV_INTERRUPT** bit in the flags for that signal. The **SV_INTERRUPT** flag is not available in *4.2 BSD*, hence it should not be used if backward compatibility is needed. The affected functions are ***read()*** or ***write()*** on a slow device (such as a terminal or pipe or other socket, but not a file) and during a ***wait()***. After a ***fork()*** or ***vfork()*** the child inherits all signals, the signal mask, the signal stack, and the restart/interrupt and reset-signal-handler flags.

The ***execve()*** call resets all caught signals to d**efault** action and resets all signals to be caught on the user stack. Ignored signals remain ignored; the signal mask remains the same; signals that interrupt functions continue to do so.

The accuracy of addr is machine dependent. For example, certain machines may supply an address that is on the same page as the address that caused the fault. If an appropriate addr cannot be computed it will be set to **SIG_NOADDR**.

RETURN VALUES

A 0 value indicates that the call succeeded. A -1 return value indicates that an error occurred and errno is set to indicate the reason.

ERRORS

sigvec() will fail and no new signal handler will be installed if one of the following occurs:

EFAULT Either nvec or ovec is not a NULL pointer and points to memory that is not a valid part of the process address space.

EINVAL sig is not a valid signal number, or, **SIGKILL**, or **SIGSTOP**.

SEE ALSO

intro(), ***exec()***, ***fcntl()***, ***fork()***, ***getitimer()***, ***getrlimit()***, ***ioctl()***, ***kill()***, ***ptrace()***, ***read()***, ***umask()***, ***vfork()***, ***wait()***, ***write()***, ***setjmp()***, ***sigblock()***-BSD, ***sigstack()***-BSD, ***signal()***-BSD, ***wait()***-BSD, ***signal()***

NOTES

Use of these interfaces should be restricted to only applications written on **BSD** platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.

SIGPOLL is a synonym for **SIGIO**. A **SIGIO** will be issued when a file descriptor corresponding to a **STREAM** file has a "selectable" event pending. Unless that descriptor has been put into asynchronous mode (see ***fcntl()***), a process may specifically request that this signal be sent using the **I_SETSIG** ***ioctl()*** call. Otherwise, the process will never receive **SIGPOLL** *s0*.

The handler routine can be declared:

*void handler(int sig, int code, struct sigcontext *scp, char *addr);*

Here sig is the signal number; code is a parameter of certain signals that provides additional detail; scp is a pointer to the *sigcontext* structure (defined in *signal.h*), used to restore the context from before the signal; and addr is additional address information. The signals **SIGKILL**, **SIGSTOP**, and **SIGCONT** cannot be ignored.

sleep

NAME

sleep - suspend execution for interval

SYNOPSIS

int sleep(unsigned seconds);

DESCRIPTION

sleep() suspends the current process from execution for the number of seconds specified by the argument. The actual suspension time may be up to 1 second less than that requested, because scheduled wakeups occur at fixed 1-second intervals, and may be an arbitrary amount longer because of other activity in the system.

sleep() is implemented by setting an interval timer and pausing until it expires. The previous state of this timer is saved and restored. If the sleep time exceeds the time to the expiration of the previous value of the timer, the process sleeps only until the timer would have expired, and the signal which occurs with the expiration of the timer is sent one second later.

SEE ALSO

alarm(), ***getitimer()***, ***longjmp()***, ***siglongjmp()***, ***sleep()***, ***usleep()***

NOTES

Use of these interfaces should be restricted to only applications written on **BSD** platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.

SIGALRM should not be blocked or ignored during a call to ***sleep()***. Only a prior call to ***alarm()*** should generate **SIGALRM** for the calling process during a call to ***sleep()***. A signal-catching function should not interrupt a call to ***sleep()*** to call ***siglongjmp()*** or ***longjmp()*** to restore an environment saved prior to the ***sleep()*** call.

WARNINGS

sleep() is slightly incompatible with ***alarm()***. Programs that do not execute for at least one second of clock time between successive calls to ***sleep()*** indefinitely delay the alarm signal. Use ***sleep()***. Each ***sleep()*** call postpones the alarm signal that would have been sent during the requested sleep period to occur one second later.

printf
fprintf
sprintf
vprintf
vfprintf
vsprintf

NAME

printf, fprintf, sprintf, vprintf, vfprintf, vsprintf - formatted output conversion

SYNOPSIS

```
#include <stdio.h>
int     printf(const char *format);
int     fprintf( FILE *stream, char *format, va_dcl);
char    *sprintf( char *s, *format, va_dcl);
int     vprintf(char *format, va_list ap);
int     vfprintf(FILE *stream, char *format, va_list ap);
char    *vsprintf(char *s, *format, va_list ap);
```

DESCRIPTION

printf() places output on the standard output stream *stdout*. ***fprintf()*** places output on the named output stream. ***sprintf()*** places "output," followed by the NULL character (\0), in consecutive bytes starting at *s; it is the user's responsibility to ensure that enough storage is available. ***vprintf()***, ***vfprintf()***, and ***vsprintf()*** are the same as ***printf()***, ***fprintf()***, and ***sprintf()*** respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by varargs.

Each of these functions converts, formats, and prints its args under control of the format. The format is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of zero or more args. The results are undefined if there are insufficient args for the format. If the format is exhausted while args remain, the excess args are simply ignored.

Each conversion specification is introduced by the character

%.   After the %, the following appear in sequence:

Zero or more flags, which modify the meaning of the conversion specification.

An optional decimal digit string specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag `-', described below, has been given) to the field width. The padding is with blanks unless the field width digit string starts with a zero, in which case the padding is with zeros.

A precision that gives the minimum number of digits to appear for the d, i, o, u, x, or X conversions, the number of digits to appear after the decimal point for the e, E, and f conversions, the maximum number of significant digits for the g and G conversion, or the maximum number of characters to be printed from a string in s conversion. The precision takes the form of a period (.) followed by a decimal digit string; a NULL digit string is treated as zero. Padding specified by the precision overrides the padding specified by the field width.

An optional l (ell) specifying that a following d, i, o, u, x, or X conversion character applies to a long integer arg. An l before any other conversion character is ignored. A character that indicates the type of conversion to be applied. A field width or precision or both may be indicated by an asterisk (*) instead of a digit string. In this case, an integer arg supplies the field width or precision. The arg that is actually converted is not fetched until the conversion letter is seen, so the args specifying field width or precision must appear before the arg (if any) to be converted. A negative field width argument is taken as a `-' flag followed by a positive field width. If the precision argument is negative, it will be changed to zero.

The flag characters and their meanings are:

-	The result of the conversion will be left justified within the field.
+	The result of a signed conversion will always begin with a sign (+ or -).
blank	If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.
#	This flag specifies that the value is to be converted to an “alternate form.”For c, d, i, s, and u conversions, the flag has no effect. For o conversion, it increases the precision to force the first digit of the result to be a zero. For x or X conversion, a non-zero result will have 0x or 0X prefixed to it. For e, E, f, g, and G conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For g and G conversions, trailing zeroes will not be removed from the result (which they normally are).

The conversion characters and their meanings are: d,i,o,u,x,X

The integer *arg* is converted to signed decimal (d or i), unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x and X), respectively; the letters abcdef are used for x conversion and the letters ABCDEF for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. (For compatibility with older versions, padding with leading zeroes may alternatively be specified by prepending a zero to the field width. This does not imply an octal value for the field width.) The default precision is 1. The result of converting a zero value with a precision of zero is a NULL string. f The float or double arg is converted to decimal notation in the style [-]ddd.ddd where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed.

e,E The float or double arg is converted in the style [-]d.ddde+_ddd, where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, 6 digits are produced; if the precision is zero, no decimal point appears. The E format code will produce a number with E instead of e introducing the exponent. The exponent always contains at least two digits. g,G The float or double arg is printed in style f or e (or in style E in the case of a G format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style e or E will be used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.

The e, E f, g, and G formats print IEEE indeterminate values (infinity or not-a-number) as “Infinity” or “NaN” respectively.

c The character arg is printed.

s The arg is taken to be a string (character pointer) and characters from the string are printed until a NULL character (\0) is encountered or until the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first NULL character are printed. A NULL value for arg will yield undefined results. % Print a %; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Padding takes place only if the specified field width exceeds the actual width. Characters generated by ***printf()*** and ***fprintf()*** are printed as if ***putc()*** had been called.

RETURN VALUES

Upon success, ***printf()*** and ***fprintf()*** return the number of characters transmitted, excluding the null character. ***vprintf()*** and ***vfprintf()*** return the number of characters transmitted. ***sprintf()*** and ***vsprintf()*** always return s. If an output error is encountered, ***printf()***, ***fprint()***, ***vprintf()***, and ***vfprintf()*** return EOF.

SEE ALSO

econvert(), ***putc()***, ***scanf()***, ***vprintf()***, ***varargs***

NOTES

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported. Very wide fields (>128 characters) fail.

times

NAME

times - get process times

SYNOPSIS

#include <sys/param.h>

#include <sys/types.h>

#include <sys/times.h>

int times(register struct tms *tmsp);

DESCRIPTION

times() returns time-accounting information for the current process and for the terminated child processes of the current process. All times are reported in clock ticks. The number of clock ticks per second is defined by the variable **CLK_TCK**, found in the header *<limits.h>*. A structure with the following members is returned by ***times()***:

```
time_t      tms_utime;     /* user time */
time_t      tms_stime;     /* system time */
time_t      tms_cutime;    /* user time, children */
time_t      tms_cstime;    /* system time, children */
```

The children's times are the sum of the children's process times and their children's times.

RETURN VALUES

times() returns

0 on success.

-1 on failure.

SEE ALSO

time(), ***wait()***, ***getrusage()***

NOTES

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.

times() has been superseded by ***getrusage()***.

wait

NAME

wait, wait3, wait4, waitpid, WIFSTOPPED, **WIFSIGNALED*****, WIFEXITED*** - wait for process to terminate or stop

SYNOPSIS

#include <sys/wait.h>

int wait(int *statusp);

int waitpid(int pid, int *statusp, int options);

#include <sys/time.h>

#include <sys/resource.h>

int wait3(int *statusp, int options, struct rusage *rusage);

int wait4(int pid, int *statusp, int options, struct rusage *rusage);

WIFSTOPPED(int status);

WIFSIGNALED***(int status);***

WIFEXITED(int status);

DESCRIPTION

wait() delays its caller until a signal is received or one of its child processes terminates or stops due to tracing. If any child process has died or stopped due to tracing and this has not been reported using ***wait()***, return is immediate, returning the process ID and exit status of one of those children. If that child process has died, it is discarded. If there are no children, return is immediate with the value -1 returned. If there are only running or stopped but reported children, the calling process is blocked.

If *status* is not a NULL pointer, then on return from a successful ***wait()*** call the status of the child process whose process ID is the return value of ***wait()*** is stored in the ***wait()*** union pointed to by status. The *w_status* member of that union is an int; it indicates the cause of termination and other information about the terminated process in the following manner:

* If the low-order 8 bits of *w_status* are equal to 0177, the child process has stopped; the 8 bits higher up from the low-order 8 bits of *w_status* contain the number of the signal that caused the process to stop. See ***ptrace()*** and ***sigvec()***-BSD.

* If the low-order 8 bits of *w_status* are non-zero and are not equal to 0177, the child process terminated due to a signal; the low-order 7 bits of *w_status* contain the number of the signal that terminated the process. In addition, if the low-order seventh bit of *w_status* (that is, bit 0200) is set, a ``core image'' of the process was produced; see ***sigvec()***-BSD.

* Otherwise, the child process terminated due to an ***exit()*** call; the 8 bits higher up from the low-order 8 bits of *w_status* contain the low-order 8 bits of the argument that the child process passed to ***exit()***; see ***exit()***.

waitpid() behaves identically to ***wait()*** if pid has a value of -1 and options has a value of zero. Otherwise, the behavior of ***waitpid()*** is modified by the values of pid and options as follows:

pid specifies a set of child processes for which status is requested. ***waitpid()*** only returns the status of a child process from this set.

* If pid is equal to -1, status is requested for any child process. In this respect, ***waitpid()*** is then equivalent to ***wait()***.

* If pid is greater than zero, it specifies the process ID of a single child process for which status is requested.

* If pid is equal to zero, status is requested for any child process whose process group ID is equal to that of the calling process.

* If pid is less than -1, status is requested for any child process whose process group ID is equal to the absolute value of pid.

options is constructed from the bitwise inclusive OR of zero or more of the following flags, defined in the header <*sys/wait.h*>:

WNOHANG

waitpid() does not suspend execution of the calling process if status is not immediately available for one of the child processes specified by pid.

WUNTRACED

The status of any child processes specified by pid that are stopped, and whose status has not yet been reported since they stopped, are also reported to the requesting process.

wait3() is an alternate interface that allows both nonblocking status collection and the collection of the status of children stopped by any means. The status parameter is defined as above. The options parameter is used to indicate the call should not block if there are no processes that have status to report (**WNOHANG**), and/or that children of the current process that are stopped due to a **SIGTTIN**, **SIGTTOU**, **SIGTSTP**, or **SIGSTOP** signal are eligible to have their status reported as well (**WUNTRACED**). A terminated child is discarded after it reports status, and a stopped process will not report its status more than once. If rusage is not a NULL pointer, a summary of the resources used by the terminated process and all its children is returned. Only the user time used and the system time used are currently available. They are returned in *rusage.ru_utime* and *rusage.ru_stime*, respectively.

When the **WNOHANG** option is specified and no processes have status to report, ***wait3()*** returns 0. The **WNOHANG** and **WUNTRACED** options may be combined by ORing the two values.

wait4() is another alternate interface. With a pid argument of 0, it is equivalent to ***wait3()***. If pid has a nonzero value, then ***wait4()*** returns status only for the indicated process ID, but not for any other child processes.

WIFSTOPPED, **WIFSIGNALED**, **WIFEXITED**, are macros that take an argument status, of type int, as returned by ***wait()***, or ***wait3()***, or ***wait4()***. **WIFSTOPPED** evaluates to true when the process for which the ***wait()*** call was made is stopped, or to false (0) otherwise. **WIFSIGNALED** evaluates to true when the process was terminated with a signal. **WIFEXITED** evaluates to true when the process exited by using an ***exit()*** call.

RETURN VALUES

If ***wait()*** or ***waitpid()*** returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of -1 is returned and errno is set to indicate the error.

If ***wait()*** or ***waitpid()*** return due to the delivery of a signal to the calling process, a value of -1 is returned and errno is set to **EINTR**. If ***waitpid()*** function was invoked with **WNOHANG** set in options, it has at least one child process specified by pid for which status is not available, and status is not available for any process specified by pid, a value of zero is returned. Otherwise, a value of -1 is returned, and errno is set to indicate the error.

wait3() and ***wait4()*** returns 0 if **WNOHANG** is specified and there are no stopped or exited children, and returns the process ID of the child process if it returns due to a stopped or terminated child process. Otherwise, they returns a value of -1 and sets errno to indicate the error.

ERRORS

wait(), ***wait3()*** or ***wait4()*** will fail and return immediately if one or more of the following are true:

ECHILD	The calling process has no existing unwaited-for child processes.
EFAULT	The status or rusage arguments point to an illegal address.

waitpid() may set errno to:

ECHILD	The process or process group specified by pid does not exist or is not a child of the calling process.
EINTR	The function was interrupted by a signal. The value of the location pointed to by statusp is undefined.
EINVAL	The value of options is not valid.

wait(), and ***wait3()***, and ***wait4()*** will terminate prematurely, return -1, and set errno to **EINTR** upon the arrival of a signal whose **SV_INTERRUPT** bit in its flags field is set (see ***sigvec()***-BSD and ***siginterrupt()***-BSD). ***signal()***-BSD, sets this bit for any signal it catches.

SEE ALSO

exit(), ***ptrace()***, ***wait()***, ***waitpid()***, ***getrusage()***, ***siginterrupt()***-BSD, ***signal()***-BSD, ***sigvec()***-BSD, ***signal()***

NOTES

Use of these interfaces should be restricted to only applications written on **BSD** platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.If a parent process terminates without waiting on its children, the initialization process (process ID = 1) inherits the children.

wait(), and ***wait3()***, and ***wait4()*** are automatically restarted when a process receives a signal while awaiting termination of a child process, unless the **SV_INTERRUPT** bit is set in the flags for that signal.

Calls to ***wait()*** with an argument of 0 should be cast to type `int *', as in:

wait*((int *)0)*

Other members of the wait union could be used to extract this information more conveniently:

* If the *w_stopval* member had the value **WSTOPPED**, the child process had stopped; the value of the *w_stopsig* member was the signal that stopped the process.

* If the *w_termsig* member was non-zero, the child process terminated due to a signal; the value of the *w_termsig* member was the number of the signal that terminated the process. If the *w_coredump* member was non-zero, a core dump was produced.

* Otherwise, the child process terminated due to a call to ***exit()***. The value of the *w_retcode* member was the low-order 8 bits of the argument that the child process passed to ***exit()***.

union wait is obsolete in light of the new specifications provided by **IEEE Std 1003.1-1988** and endorsed by **SVID89** and **XPG3**.

reboot

NAME

reboot - reboot system or halt processor

SYNOPSIS

#include <sys/reboot.h>

int reboot(int howto, char *bootargs);

DESCRIPTION

reboot() reboots the system. howto is an option passed to specify the behavior of the system while rebooting. The function interface permits only one of **RB_HALT**, **RB_ASKNAME** or **RB_AUTOBOOT** to be passed. **RB_AUTOBOOT** is the default.

The *howto* options are:

RE_AUTOBOOT	The machine is rebooted from the root filesystem on the default boot device. See boot and kernel.
RB_HALT	the processor is simply halted; no reboot takes place. **RB_HALT** should be used with caution.
RB_ASKNAME	Interpreted by the bootstrap program and kernel, causing the user to be asked for pathnames during the bootstrap.

The interpretation of the *bootargs* argument is platform dependent.

RETURN VALUES

If successful, this call never returns. Otherwise, a -1 is returned and an error is returned in the global variable errno.

ERRORS

EPERM	The caller is not the super-user.

SEE ALSO

boot, halt, reboot, uadmin()

NOTES

Any other howto argument causes the kernel file to boot. Only the super-user may ***reboot()*** a machine.

bcopy
bcmp
bzero

NAME

bcopy, bcmp, bzero - bit and byte string operations

SYNOPSIS

#include <strings.h>

void ***bcopy(const void *s1, void *s2, size_t n);***

int ***bcmp(const void *s1, const void *s2, size_t n);***

void ***bzero(void *s, size_t n);***

DESCRIPTION

The functions ***bcopy()***, ***bcmp()***, and ***bzero()*** operate on variable length strings of bytes. They do not check for null bytes as the routines in ***string()*** do.

bcopy() copies n bytes from string *s1* to the string *s2*. Overlapping strings are handled correctly.

bcmp() compares byte string s1 against byte string *s2*, returning zero if they are identical, 1 otherwise. Both strings are assumed to be n bytes long. ***bcmp()*** using n zero bytes always returns zero.

bzero() places n 0 bytes in the string s.

SEE ALSO

memory(), ***string()***

ftime

NAME

ftime - get date and time

SYNOPSIS

#include <sys/timeb.h>

int ftime(struct timeb *tp);

DESCRIPTION

The ***ftime()*** function sets the *time* and *millitm* members of the timeb structure pointed to by tp. The structure is defined in <*sys/timeb.h*> and contains the following members:

time_t	*time;*
unsigned short	*millitm;*
short	*timezone;*
short	*dstflag;*

The *time* and *millitm* members contain the seconds and milliseconds portions, respectively, of the current time in seconds since 00:00:00 UTC (Coordinated Universal Time), January 1, 1970. The *timezone* member contains the local time zone. The *dstflag* member contains a flag that, if non-zero, indicates that Daylight Saving time applies locally during the appropriate part of the year. The contents of the timezone and dstflag members of tp after a call to ***ftime()*** are unspecified.

RETURN VALUES

Upon successful completion, the ***ftime()*** function returns 0. Otherwise -1 is returned.

USAGE

For portability to implementations conforming to earlier versions of this document, ***time()*** is preferred over this function.

The millisecond value usually has a granularity greater than one due to the resolution of the system clock. Depending on any granularity (particularly a granularity of one) renders code non-portable.

SEE ALSO

date, ***time()***, ***ctime()***, ***gettimeofday()***, ***timezone***

getdtablesize

NAME

getdtablesize - get the file descriptor table size

SYNOPSIS

#include <unistd.h>

int getdtablesize(void);

DESCRIPTION

The ***getdtablesize()*** function is equivalent to ***getrlimit()*** with the **RLIMIT_NOFILE** option.

RETURN VALUES

The ***getdtablesize()*** function returns the current soft limit as if obtained from a call to ***getrlimit()*** with the **RLIMIT_NOFILE** option.

USAGE

There is no direct relationship between the value returned by ***getdtablesize()*** and {**OPEN_MAX**} defined in *<limits.h>*.

SEE ALSO

close(), ***getrlimit()***, ***open()***, ***setrlimit()***, ***select()***

NOTES

Each process has a file descriptor table which is guaranteed to have at least 20 slots. The entries in the descriptor table are numbered with small integers starting at 0. The ***getdtablesize()*** function returns the current maximum size of this table by calling the ***getrlimit()*** function.

gethostid

NAME

gethostid - get unique identifier of current host

SYNOPSIS

#include <unistd.h>

long gethostid(void);

DESCRIPTION

gethostid() returns the 32-bit identifier for the current host, which should be unique across all hosts. This number is usually taken from the CPU board's ID **PROM**.

SEE ALSO

hostid, sysinfo()

gethostname
sethostname

NAME

gethostname, sethostname - get/set name of current host

SYNOPSIS

int gethostname(char *name, int namelen);

int sethostname(char *name, int namelen);

DESCRIPTION

gethostname() returns the standard host name for the current processor, as previously set by sethostname. The parameter namelen specifies the size of the array pointed to by name. The returned name is null-terminated unless insufficient space is provided.

sethostname() sets the name of the host machine to be name, which has length namelen. This call is restricted to the privileged user and is normally used only when the system is bootstrapped.

RETURN VALUES

If the call succeeds a value of 0 is returned. If the call fails, then a value of -1 is returned and an error code is placed in the global location errno.

ERRORS

The following error may be returned by these calls:

EFAULT The name or namelen parameter gave an invalid address.

EPERM The caller was not the privileged user. Note: this error only applies to ***sethostname()***.

SEE ALSO

uname(), ***sysinfo()***, ***gethostid()***

NOTES

Host names are limited to **MAXHOSTNAMELEN** characters, currently 256. (See the <sys/param.h> header.)

getpagesize

NAME

getpagesize - get system page size

SYNOPSIS

#include <unistd.h>

int getpagesize(void);

DESCRIPTION

getpagesize() returns the number of bytes in a page. Page granularity is the granularity of many of the memory management calls. The page size is a system page size and need not be the same as the underlying hardware page size.

The ***getpagesize()*** function is equivalent to ***sysconf***(**_SC_PAGE_SIZE**) and sysconf(**_SC_PAGESIZE**).

RETURN VALUES

The ***getpagesize()*** function returns the current page size.

SEE ALSO

pagesize, brk(), getrlimit(), mmap(), mprotect(), munmap(), malloc(), msync(), sysconf()

getpriority
setpriority

NAME

getpriority, setpriority - get or set process scheduling priority

SYNOPSIS

#include <sys/resource.h>

int getpriority(int which, id_t who);

int setpriority(int which, id_t who, int priority);

DESCRIPTION

The ***getpriority()*** function obtains the current scheduling priority of a process, process group, or user. The ***setpriority()*** function sets the scheduling priority of a process, process group, or user.

Target processes are specified by the values of the which and who arguments. The which argument may be one of the following values: **PRIO_PROCESS**, **PRIO_PGRP**, or **PRIO_USER**, indicating that the who argument is to be interpreted as a process ID, a process group ID, or a user ID, respectively. A 0 value for the who argument specifies the current process, process group, or user.

If more than one process is specified, ***getpriority()*** returns the highest priority (lowest numerical value) pertaining to any of the specified processes, and ***setpriority()*** sets the priorities of all of the specified processes to the specified value.

The default priority is 0; negative priorities cause more favorable scheduling. While the range of valid priority values is [-20, 20], implementations may enforce more restrictive limits. If the value specified to ***setpriority()*** is less than the system's lowest supported priority value, the system's lowest supported value is used; if it is greater than the system's highest supported value, the system's highest supported value is used.

Only a process with appropriate privileges can raise its priority (that is, assign a lower numerical priority value).

RETURN VALUES

Upon successful completion, ***getpriority()*** returns an integer in the range from -20 to 20. Otherwise, -1 is returned and errno is set to indicate the error.

Upon successful completion, ***setpriority()*** returns 0. Otherwise, -1 is returned and errno is set to indicate the error.

ERRORS

The ***getpriority()*** and ***setpriority()*** functions will fail if:

ESRCH No process could be located using the which and who argument values specified.

EINVAL The value of the which argument was not recognized, or the value of the who argument is not a valid process ID, process group ID, or user ID.

In addition, ***setpriority()*** may fail if:

EPERM A process was located, but neither the real nor effective user ID of the executing process is the privileged user or match the effective user ID of the process whose priority is being changed.

EACCES A request was made to change the priority to a lower numeric value (that is, to a higher priority) and the current process does not have appropriate privileges.

USAGE

The effect of changing the scheduling priority may vary depending on the process-scheduling algorithm in effect.

Because ***getpriority()*** can return the value -1 on successful completion, it is necessary to set errno to 0 prior to a call to ***getpriority()***. If ***getpriority()*** returns the value -1, then errno can be checked to see if an error occurred or if the value is a legitimate priority.

SEE ALSO

nice, re***nice***, ***fork()***

getrusage

NAME

getrusage - get information about resource utilization

SYNOPSIS

#include <sys/resource.h>

int getrusage(int who, struct rusage *r_usage);

DESCRIPTION

The ***getrusage()*** function provides measures of the resources used by the current process or its terminated and waited-for child processes. If the value of the who argument is **RUSAGE_SELF**, information is returned about resources used by the current process. If the value of the who argument is **RUSAGE_CHILDREN**, information is returned about resources used by the terminated and waited-for children of the current process. If the child is never waited for (for instance, if the parent has **SA_NOCLDWAIT** set or sets **SIGCHLD** to **SIG_IGN**), the resource information for the child process is discarded and not included in the resource information provided by ***getrusage()***.

The *r_usage* argument is a pointer to an object of type struct rusage in which the returned information is stored. The members of rusage are as follows:

struct timeval	*ru_utime;*	*/* user time used */*
struct timeval	*ru_stime;*	*/* system time used */*
long	*ru_maxrss;*	*/* maximum resident set size */*
long	*ru_idrss;*	*/* integral resident set size */*
long	*ru_minflt;*	*/* page faults not requiring physical I/O */*
long	*ru_majflt;*	*/* page faults requiring physical I/O */*
long	*ru_nswap;*	*/* swaps */*
long	*ru_inblock;*	*/* block input operations */*
long	*ru_oublock;*	*/* block output operations */*
long	*ru_msgsnd;*	*/* messages sent */*
long	*ru_msgrcv;*	*/* messages received */*
long	*ru_nsignals;*	*/* signals received */*
long	*ru_nvcsw;*	*/* voluntary context switches */*
long	*ru_nivcsw;*	*/* involuntary context switches */*

The fields are interpreted as follows:

ru_utime The total amount of time spent executing in user mode. Time is given in seconds and microseconds.

ru_stime The total amount of time spent executing in system mode. Time is given in seconds and microseconds.

ru_maxrss	The maximum resident set size. Size is given in pages (the size of a page, in bytes, is given by the ***getpagesize()*** function).
ru_idrss	An "integral" value indicating the amount of memory in use by a process while the process is running. This value is the sum of the resident set sizes of the process running when a clock tick occurs. The value is given in pages times clock ticks. It does not take sharing into account.
ru_minflt	The number of page faults serviced which did not require any physical I/O activity.
ru_majflt	The number of page faults serviced which required physical I/O activity. This could include page ahead operations by the kernel.
ru_nswap	The number of times a process was swapped out of main memory.
ru_inblock	The number of times the file system had to perform input in servicing a read( ) request.
ru_oublock	The number of times the file system had to perform output in servicing a ***write()*** request.
ru_msgsnd	The number of messages sent over sockets.
ru_msgrcv	The number of messages received from sockets.
ru_nsignals	The number of signals delivered.
ru_nvcsw	The number of times a context switch resulted due to a process voluntarily giving up the processor before its time slice was completed (usually to await availability of a resource).
ru_nivcsw	The number of times a context switch resulted due to a higher priority process becoming runnable or because the current process exceeded its time slice.

RETURN VALUES

Upon successful completion, ***getrusage()*** returns 0. Otherwise, -1 is returned and errno is set to indicate the error.

ERRORS

getrusage() will fail if:

EFAULT	The address specified by the r_usage argument is not in a valid portion of the process' address space.
EINVAL	The who parameter is not a valid value.

SEE ALSO

read(),***times()***,***wait()***,***write()***, ***getpagesize()***, ***gettimeofday()***

NOTES

Only the timeval fields of struct rusage are supported in this implementation.

The numbers ru_inblock and ru_oublock account only for real I/O, and are approximate measures at best. Data supplied by the cache mechanism is charged only to the first process to read and the last process to write the data.

The way resident set size is calculated is an approximation, and could misrepresent the true resident set size.

Page faults can be generated from a variety of sources and for a variety of reasons. The customary cause for a page fault is a direct reference by the program to a page which is not in memory. Now, however, the kernel can generate page faults on behalf of the user, for example, servicing ***read()*** and ***write()*** functions. Also, a page fault can be caused by an absent hardware translation to a page, even though the page is in physical memory.

In addition to hardware detected page faults, the kernel may cause pseudo page faults in order to perform some housekeeping. For example, the kernel may generate page faults, even if the pages exist in physical memory, in order to lock down pages involved in a raw I/O request.

By definition, major page faults require physical I/O, while minor page faults do not require physical I/O. For example, reclaiming the page from the free list would avoid I/O and generate a minor page fault. More commonly, minor page faults occur during process startup as references to pages which are already in memory. For example, if an address space faults on some “hot” executable or shared library, this results in a minor page fault for the address space. Also, any one doing a ***read()*** or ***write()*** to something that is in the page cache will get a minor page fault(s) as well. There is no way to obtain information about a child process which has not yet terminated.

getwd

NAME

getwd - get current working directory pathname

SYNOPSIS

#include <unistd.h>

char *getwd(char *path_name);

DESCRIPTION

The ***getwd()*** function determines an absolute pathname of the current working directory of the calling process, and copies that pathname into the array pointed to by the *path_name* argument.

If the length of the pathname of the current working directory is greater than ({**PATH_MAX**} + 1) including the null byte, ***getwd()*** fails and returns a null pointer.

RETURN VALUES

Upon successful completion, a pointer to the string containing the absolute pathname of the current working directory is returned. Otherwise, ***getwd()*** returns a null pointer and the contents of the array pointed to by *path_name* are undefined.

SEE ALSO

getcwd(

index

NAME

index, rindex - string operations

SYNOPSIS

#include <strings.h>

char *index(const char *s, int c);

char *rindex(const char *s, int c);

DESCRIPTION

These functions operate on null-terminated strings.

index() returns a pointer to the first occurrence of character c in string s, and ***rindex()*** returns a pointer to the last occurrence of character c in string s. Both ***index()*** and ***rindex()*** return a null pointer if c does not occur in the string. The null character terminating a string is considered to be part of the string.

SEE ALSO

bstring(), ***malloc()***, ***string()***

NOTES

On most modern computer systems, you can not use a null pointer to indicate a null string. A null pointer is an error and results in an abort of the program. If you wish to indicate a null string, you must have a pointer that points to an explicit null string. On some implementations of the C language on some machines, a null pointer, if dereferenced, would yield a null string; this highly nonportable trick was used in some programs. Programmers using a null pointer to represent an empty string should be aware of this portability issue; even on machines where dereferencing a null pointer does not cause an abort of the program, it does not necessarily yield a null string.

random
srandom
initstate
setstate

NAME

random, srandom, initstate, setstate - pseudorandom number functions

SYNOPSIS

#include <stdlib.h>

long random(void);

void srandom(unsigned int seed);

char *initstate(unsigned int seed, char *state, size_t size);

char *setstate(const char *state);

DESCRIPTION

The ***random()*** function uses a nonlinear additive feedback random-number generator employing a default state array size of 31 long integers to return successive pseudo-random numbers in the range from 0 to 231-1. The period of this random-number generator is approximately 16 x (231-1). The size of the state array determines the period of the random-number generator. Increasing the state array size increases the period.

The ***srandom()*** function initializes the current state array using the value of seed.

The ***random()*** and ***srandom()*** functions have (almost) the same calling sequence and initialization properties as ***rand()*** and ***srand()*** (see ***rand()***). The difference is that ***rand()*** produces a much less random sequence-in fact, the low dozen bits generated by rand go through a cyclic pattern. All the bits generated by ***random()*** are usable. For example, ***random()***&01 will produce a random binary value.

Unlike ***srand()***, ***srandom()*** does not return the old seed because the amount of state information used is much more than a single word. Two other routines are provided to deal with restarting/changing random number generators. With 256 bytes of state information, the period of the random-number generator is greater than 269.

Like ***rand()***, ***random()*** produces by default a sequence of numbers that can be duplicated by calling ***srandom()*** with 1 as the seed.

The ***initstate()*** and ***setstate()*** functions handle restarting and changing random-number generators. The ***initstate()*** function allows a state array, pointed to by the state argument, to be initialized for future use. The size argument, which specifies the size in bytes of the state array, is used by ***initstate()*** to decide what type of random-number generator to use; the larger the state array, the more random the numbers. Values for the amount of state information are 8, 32, 64, 128, and 256 bytes. Other values greater than 8 bytes are rounded down to the nearest one of these values. For values smaller than 8, ***random()*** uses a simple linear congruential random number generator. The seed argument specifies a starting point for the random-number sequence and provides for restarting at the same point. The ***initstate()*** function returns a pointer to the previous state information array.

If ***initstate()*** has not been called, then ***random()*** behaves as though ***initstate()*** had been called with seed=1 and size=128.

If ***initstate()*** is called with size<8, then ***random()*** uses a simple linear congruential random number generator.

Once a state has been initialized, ***setstate()*** allows switching between state arrays. The array defined by the state argument is used for further random-number generation until ***initstate()*** is called or ***setstate()*** is called again. The ***setstate()*** function returns a pointer to the previous state array.

RETURN VALUES

The ***random()*** function returns the generated pseudo-random number.

The ***srandom()*** function returns no value.

Upon successful completion, ***initstate()*** and ***setstate()*** return a pointer to the previous state array. Otherwise, a null pointer is returned.

SEE ALSO

drand48(), ***rand()***

NOTES

random() and ***srandom()*** are unsafe in multi-thread applications.

Use of these interfaces in multi-thread applications is unsupported.

random() and ***srandom()*** function at about two-thirds the speed of ***rand()***.

killpg

NAME

killpg - send signal to a process group

SYNOPSIS

#include <signal.h>

int killpg(pid_t pgrp, int sig);

DESCRIPTION

killpg() sends the signal *sig* to the process group *pgrp*. See signal for a list of signals.

The real or effective user ID of the sending process must match the real or saved set-user ID of the receiving process, unless the effective user ID of the sending process is the privileged user. A single exception is the signal **SIGCONT**, which may always be sent to any descendant of the current process.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the global variable errno is set to indicate the error.

ERRORS

killpg() will fail and no signal will be sent if any of the following occur:

EINVAL	sig is not a valid signal number.
EPERM	The effective user ID of the sending process is not privileged user, and neither its real nor effective user ID matches the real or saved set-user ID of one or more of the target processes.
ESRCH	No processes were found in the specified process group.

SEE ALSO

kill(), ***setpgrp()***, ***sigaction()***, ***signal***

re_comp
re_exec

NAME

re_comp, re_exec - compile and execute regular expressions

SYNOPSIS

#include <re_comp.h>

char *re_comp(const char *string);

int re_exec(const char *string);

DESCRIPTION

The ***re_comp()*** function converts a regular expression string (RE) into an internal form suitable for pattern matching. The ***re_exec()*** function compares the string pointed to by the string argument with the last regular expression passed to ***re_comp()***. If ***re_comp()*** is called with a null pointer argument, the current regular expression remains unchanged. Strings passed to both ***re_comp()*** and ***re_exec()*** must be terminated by a null byte, and may include **NEWLINE** characters.

The ***re_comp()*** and ***re_exec()*** functions support simple regular expressions, which are defined on the regexp manual page. The regular expressions of the form \{m\}, \{m,\}, or \{m,n\} are not supported.

RETURN VALUES

The ***re_comp()*** function returns a null pointer when the string pointed to by the string argument is successfully converted. Otherwise, a pointer to one of the following error message strings is returned:

No previous regular expression

Regular expression too long

unmatched \(

missing]

too many \(\) pairs

unmatched \)

Upon successful completion, ***re_exec()*** returns 1 if string matches the last compiled regular expression. Otherwise, ***re_exec()*** returns 0 if string fails to match the last compiled regular expression, and -1 if the compiled regular expression is invalid (indicating an internal error).

USAGE

For portability to implementations conforming to X/Open standards prior to **XPG4v2**, ***regcomp()*** and ***regexec()*** are preferred to these functions.

SEE ALSO

grep, ***regcmp***, ***regcmp()***, ***regcomp()***, ***regexec()***, ***regexpr()***, ***regexp***, standards

setbuffer
setlinebuf

NAME

setbuffer, setlinebuf - assign buffering to a stream

SYNOPSIS

#include <stdio.h>

void setbuffer(FILE *iop, char *abuf, size_t asize);

void setlinebuf(FILE *iop);

DESCRIPTION

setbuffer, ***setlinebuf*** - assign buffering to a stream The three types of buffering available are unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is block buffered many characters are saved up and written as a block; when it is line buffered characters are saved up until a **NEWLINE** is encountered or input is read from stdin. ***fflush()*** may be used to force the block out early. Normally all files are block buffered. A buffer is obtained from ***malloc()*** upon the first ***getc()*** or ***putc()*** on the file. If the standard stream stdout refers to a terminal it is line buffered. The standard stream stderr is unbuffered by default.

setbuffer() can be used after a stream, iop, has been opened but before it is read or written. It uses the character array abuf whose size is determined by the asize argument instead of an automatically allocated buffer. If abuf is the NULL pointer, input/output will be completely unbuffered. A manifest constant **BUFSIZ** , defined in the <*stdio.h*> header, tells how big an array is needed:

char buf[BUFSIZ];

setlinebuf() is used to change the buffering on a stream from block buffered or unbuffered to line buffered. Unlike ***setbuffer()***, it can be used at any time that the stream, iop, is active.

A stream can be changed from unbuffered or line buffered to block buffered by using ***freopen()***. A stream can be changed from block buffered or line buffered to unbuffered by using ***freopen()*** followed by ***setbuf()*** with a buffer argument of NULL.

SEE ALSO

malloc(), ***fclose()***, ***fopen()***, ***fread()***, ***getc()***, ***printf()***, ***putc()***, ***puts()***, ***setbuf()***, ***setvbuf()***

NOTES

A common source of error is allocating buffer space as an “automatic” variable in a code block, and then failing to close the stream in the same block.

setregid

NAME

setregid - set real and effective group IDs

SYNOPSIS

#include <unistd.h>

int setregid(gid_t rgid, gid_t egid);

DESCRIPTION

setregid() is used to set the real and effective group IDs of the calling process. If *rgid* is -1, the real GID is not changed; if egid is -1, the effective GID is not changed. The real and effective GIDs may be set to different values in the same call.

If the effective user ID of the calling process is superuser, the real **GID** and the effective GID can be set to any legal value.

If the effective user ID of the calling process is not super-user, either the real GID can be set to the saved ***setGID*** from ***execve()***, or the effective GID can either be set to the saved setGID or the real GID. Note: if a ***setGID*** process sets its effective GID to its real GID, it can still set its effective **GID** back to the saved setGID.

In either case, if the real GID is being changed (that is, if rgid is not -1), or the effective GID is being changed to a value not equal to the real GID, the saved ***setGID*** is set equal to the new effective GID.

RETURN VALUES

setregid() returns:

0	on success.
-1	on failure and sets errno to indicate the error.

ERRORS

setregid() will fail and neither of the group IDs will be changed if:

EINVAL	The value of rgid or egid is less than 0 or greater than **UID_MAX** (defined in <limits.h>).
EPERM	The calling process' effective UID is not the super-user and a change other than changing the real GID to the saved ***setGID***, or changing the effective GID to the real GID or the saved GID, was specified.

SEE ALSO

execve(), ***getgid()***, ***setreuid()***, ***setuid()***

setreuid

NAME

setreuid - set real and effective user IDs

SYNOPSIS

#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);

DESCRIPTION

setreuid() is used to set the real and effective user IDs of the calling process. If ruid is -1, the real user ID is not changed; if euid is -1, the effective user ID is not changed. The real and effective user IDs may be set to different values in the same call. If the effective user ID of the calling process is superuser, the real user ID and the effective user ID can be set to any legal value.

If the effective user ID of the calling process is not super-user, either the real user ID can be set to the effective user ID, or the effective user ID can either be set to the saved set-user ID from ***execve()*** or the real user ID. Note: if a set-UID process sets its effective user ID to its real user ID, it can still set its effective user ID back to the saved set-user ID.

In either case, if the real user ID is being changed (that is, if ruid is not -1), or the effective user ID is being changed to a value not equal to the real user ID, the saved set-user ID is set equal to the new effective user ID.

RETURN VALUES

setreuid() returns:

0	on success.
-1	on failure and sets errno to indicate the error.

ERRORS

setreuid() will fail and neither of the user IDs will be changed if:

EINVAL	The value of ruid or euid is less than 0 or greater than **UID_MAX** (defined in <limits.h>).
EPERM	The calling process' effective user ID is not the super-user and a change other than changing the real user ID to the effective user ID, or changing the effective user ID to the real user ID or the saved set-user ID, was specified.

SEE ALSO

execve(), ***getuid()***, ***setregid()***, ***setuid()***

ualarm

NAME

ualarm - schedule signal after interval in microseconds

SYNOPSIS

#include <unistd.h>

useconds_t ualarm(useconds_t useconds, useconds_t interval);

DESCRIPTION

The ***ualarm()*** function causes the **SIGALRM** signal to be generated for the calling process after the number of real-time microseconds specified by the useconds argument has elapsed. When the interval argument is non-zero, repeated timeout notification occurs with a period in microseconds specified by the interval argument. If the notification signal, **SIGALRM**, is not caught or ignored, the calling process is terminated.

Because of scheduling delays, resumption of execution when the signal is caught may be delayed an arbitrary amount of time.

Interactions between ***ualarm()*** and either ***alarm()*** or ***sleep()*** are unspecified.

RETURN VALUES

The ***ualarm()*** function returns the number of microseconds remaining from the previous ***ualarm()*** call. If no timeouts are pending or if ***ualarm()*** has not previously been called, ***ualarm()*** returns 0.

SEE ALSO

alarm(), ***setitimer()***, ***sighold()***, ***signal()***, ***sleep()***, ***usleep()***

usleep

NAME

usleep - suspend execution for interval in microseconds

SYNOPSIS

#include <unistd.h>

int usleep(useconds_t useconds);

DESCRIPTION

The ***usleep()*** function suspends the current process from execution for the number of microseconds specified by the useconds argument. (A microsecond is .000001 seconds.) Because of other activity, or because of the time spent in processing the call, the actual suspension time may be longer than the amount of time specified. The *useconds* argument must be less than 1,000,000. If the value of useconds is 0, then the call has no effect. The ***usleep()*** function uses the process' real-time interval timer to indicate to the system when the process should be woken up.

There is one real-time interval timer for each process. The ***usleep()*** function will not interfere with a previous setting of this timer. If the process has set this timer prior to calling ***usleep()***, and if the time specified by useconds equals or exceeds the interval timer's prior setting, the process will be woken up shortly before the timer was set to expire.

Interactions between ***usleep()*** and either ***alarm()*** or ***sleep()*** are unspecified.

RETURN VALUES

On successful completion, ***usleep()*** returns 0. Otherwise, it returns -1 and sets errno to indicate the error.

ERRORS

The ***usleep()*** function may fail if:

EINVAL The time interval specified 1,000,000 or more microseconds.

SEE ALSO

alarm(), ***poll()***, ***setitimer()***, ***sigaction()***, ***sigprocmask()***, ***select()***, ***sleep()***, ***ualarm()***

SPARC COMPLIANCE DEFINITION 2.4.1 IS

libw

fgetwc

NAME

fgetwc - get a wide-character code from a stream

SYNOPSIS

#include <stdio.h>

#include <wchar.h>

wint_t fgetwc(FILE *stream);

DESCRIPTION

The ***fgetwc***() function obtains the next character (if present) from the input stream pointed to by stream, converts that to the corresponding wide-character code and advances the associated file position indicator for the stream (if defined). If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. The ***fgetwc***() function may mark the *st_atime* field of the file associated with stream for update. The st_atime field will be marked for update by the first successful execution of ***fgetwc***(), ***fgetc***(), ***fgets***(), ***fgetws***(), ***fread***(), ***fscanf***(), ***getc***(), ***getchar***(), ***gets***(), or ***scanf***() using stream that returns data not supplied by a prior call to ***ungetc***() or ***ungetwc***().

RETURN VALUES

Upon successful completion the ***fgetwc***() function returns the wide-character code of the character read from the input stream pointed to by stream converted to a type *wint_t*. If the stream is at end-of-file, the end-of-file indicator for the stream is set and ***fgetwc***() returns **WEOF**. If a read error occurs, the error indicator for the stream is set, ***fgetwc***() returns **WEOF** and sets errno to indicate the error.

ERRORS

EAGAIN The **O_NONBLOCK** flag is set for the file descriptor underlying stream and the process would be delayed in the ***fgetwc***() operation.

EBADF The file descriptor underlying stream is not a valid file descriptor open for reading.

EINTR The read operation was terminated due to the receipt of a signal, and no data was transferred.

EIO A physical I/O error has occurred, or the process is in a background process group attempting to read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group is orphaned.

EOVERFLOW The file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the corresponding stream.

ENOMEM Insufficient storage space is available.

ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the device.

EILSEQ The data obtained from the input stream does not form a valid character.

SEE ALSO

feof(), ferror(), fgetc(), fgets(), fgetws(), fopen(), fread(), fscanf(), getc(), getchar(), gets(), scanf(), setlocale(), ungetc(), ungetwc()

getws, fgetws

NAME

getws, fgetws - convert a string of EUC characters from the stream to Process Code

SYNOPSIS

#include <stdio.h>

#include <widec.h>

wchar_t *getws(wchar_t *s);

wchar_t *fgetws(wchar_t *s, int n, FILE *stream);

DESCRIPTION

The ***getws***() function reads a string of Extended Unix Code (**EUC**) characters from the standard input stream, stdin, converts it to process code, and writes it to the array pointed to by s, until a new-line character is read or an end-of- file condition is encountered. The new-line character is discarded and the string is terminated with a *wchar_t* NULL character. The ***getws***() function returns its argument. The ***fgetws***() function reads **EUC** characters from the stream, converts them to Process Code, and writes them to the array pointed to by s. It stops when either n-1 characters are read, a new-line character is read and transferred to s, or an end-of-file condition is encountered. The string is then terminated with a wchar_t NULL character. The ***fgetws***() function returns its first argument.

RETURN VALUES

If end-of-file is encountered and no characters have been read, no characters are transferred to s and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise s is returned.

ERRORS

The ***fgetws***() function will fail if data needs to be read and:

EOVERFLOW The file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the corresponding stream.

SEE ALSO

ferror(), fread(), getwc(), putws(), scanf()

fputwc

NAME

fputwc - put wide-character code on a stream

SYNOPSIS

#include <stdio.h>

#include <wchar.h>

wint_t fputwc(wint_t wc, FILE *stream);

DESCRIPTION

The ***fputwc***() function writes the character corresponding to the wide-character code wc to the output stream pointed to by stream, at the position indicated by the associated file-position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream. If an error occurs while writing the character, the shift state of the output file is left in an undefined state. The *st_ctime* and *st_mtime* fields of the file will be marked for update between the successful execution of ***fputwc***() and the next successful completion of a call to ***fflush***() or ***fclose***() on the same stream or a call to ***exit***(2) or ***abort***().

RETURN VALUES

Upon successful completion, ***fputwc***() returns wc. Otherwise, it returns **WEOF**, the error indicator for the stream is set, and errno is set to indicate the error.

ERRORS

The ***fputwc***() function will fail if either the stream is unbuffered or data in the stream's buffer needs to be written, and:

EAGAIN The **O_NONBLOCK** flag is set for the file descriptor underlying stream and the process would be delayed in the write operation.

EBADF The file descriptor underlying stream is not a valid file descriptor open for writing.

EFBIG An attempt was made to write to a file that exceeds the maximum file size or the process' file size limit.

EFBIG The file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.

EINTR The write operation was terminated due to the receipt of a signal, and no data was transferred.

EIO A physical I/O error has occurred, or the process is a member of a background process group attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking **SIGTTOU** and the process group of the process is orphaned.

ENOSPC There was no free space remaining on the device containing the file.

EPIPE An attempt is made to write to a pipe or **FIFO** that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

The ***fputwc***() function may fail if:

ENOMEM Insufficient storage space is available.

ENXIOA request was made of a non-existent device, or the request was outside the capabilities of the device.

EILSEQ The wide-character code wc does not correspond to a valid character.

SEE ALSO

exit(2), ulimit(2), abort(), fclose(), ferror(), fflush(), fopen(), setbuf()

fputws

NAME

fputws - put wide character string on a stream

SYNOPSIS

#include <stdio.h>

#include <wchar.h>

int fputws(const wchar_t *s, FILE *stream);

DESCRIPTION

The ***fputws***() function writes a character string corresponding to the (null-terminated) wide character string pointed to by *ws* to the stream pointed to by stream. No character corresponding to the terminating null wide-character code is written.

The *st_ctime* and *st_mtime* fields of the file will be marked for update between the successful execution of ***fputws***() and the next successful completion of a call to ***fflush***() or ***fclose***() on the same stream or a call to ***exit***(2) or ***abort***().

RETURN VALUES

Upon successful completion, ***fputws***() returns a non-negative number. Otherwise it returns -1, sets an error indicator for the stream and errno is set to indicate the error.

ERRORS

Refer to ***fputwc***().

USAGE

The ***fputws***() function does not append a **NEWLINE** character.

SEE ALSO

exit(2), abort(), fclose(), fflush(), fopen(), fputwc()

getwidth

NAME

getwidth - get codeset information

SYNOPSIS

#include <euc.h>

#include <getwidth.h>

void getwidth(eucwidth_t *ptr);

DESCRIPTION

The ***getwidth***() function reads the character class table for the current locale to get information on the supplementary codesets. ***getwidth***() sets this information into the struct *eucwidth_t*. This struct is defined in <*euc.h*> and has the following members:

short int	*_eucw1,_eucw2,_eucw3;*
short int	*_scrw1,_scrw2,_scrw3;*
short int	*_pcw;*
char	*_multibyte;*

Codeset width values for supplementary codesets 1, 2, and 3 are set in *_eucw1*, *_eucw2*, and *_eucw3*, respectively. Screen width values for supplementary codesets 1, 2, and 3 are set in *_scrw1*, *_scrw2*, and *_scrw3*, respectively. The width of Extended Unix Code (**EUC**) Process Code is set in *_pcw*. The *_multibyte* entry is set to 1 if multibyte characters are used, and set to 0 if only single-byte characters are used.

SEE ALSO

euclen(), setlocale()

NOTES

This function can be used safely in a multi-thread application, as long as ***setlocale***() is not being called to change the locale.

This function will only work with **EUC** locales.

isenglish, isideogram, isnumber
isphonogram, isspecial, iswalnum
iswalpha, iswascii, iswcntrl
iswdigit, iswgraph, iswlower, iswprint
iswpunct, iswspace, iswupper, iswxdigit

NAME

iswalpha, iswupper, iswlower, iswdigit, iswxdigit, iswalnum, iswspace, iswpunct, iswprint, iswcntrl, iswascii, iswgraph, isphonogram, isideogram, isenglish, isnumber, isspecial - wide-character code classification functions

SYNOPSIS

#include <wchar.h>

int iswalpha(wint_t wc);

DESCRIPTION

These functions test whether wc is a wide-character code representing a character of a particular class defined in the **LC_CTYPE** category of the current locale. In all cases, wc is a *wint_t*, the value of which must be a wide-character code corresponding to a valid character in the current locale or must equal the value of the macro **WEOF**. If the argument has any other values, the behavior is undefined.

iswalpha(*wc*) tests whether *wc* is a wide-character code representing a character of class “alpha” in the program's current locale.

iswupper(*wc*) tests whether *wc* is a wide-character code representing a character of class “upper” in the program's current locale.

iswlower(*wc*) tests whether wc is a wide-character code representing a character of class “lower” in the program's current locale.

iswdigit(*wc*) tests whether *wc* is a wide-character code representing a character of class “digit” in the program's current locale.

iswxdigit(*wc*) tests whether *wc* is a wide-character code representing a character of class “xdigit” in the program's current locale.

iswalnum(*wc*) tests whether *wc* is a wide-character code representing a character of class “alpha” or “digit” in the program's current locale.

iswspace(*wc*) tests whether *wc* is a wide-character code representing a character of class “space” in the program's current locale.

iswpunct(*wc*) tests whether *wc* is a wide-character code representing a character of class “punct” in the program's current locale.

iswprint(*wc*) tests whether *wc* is a wide-character code representing a character of class “print” in the program's current locale.

iswgraph(*wc*) tests whether *wc* is a wide-character code representing a character of class “graph” in the program's current locale.

iswcntrl(*wc*) tests whether *wc* is a wide-character code representing a character of class “cntrl” in the program's current locale.

iswascii(*wc*) tests whether *wc* is a wide-character code representing an ASCII character.

isphonogram(*wc*)tests whether *wc* is a wide-character code representing a phonetic language character, excluding ASCII characters.

isideogram(*wc*) tests whether *wc* is a wide-character code representing an ideographic language character, excluding ASCII characters.

isenglish(*wc*) tests whether *wc* is a wide-character code representing an English language character, excluding ASCII characters.

isnumber(*wc*) tests whether *wc* is a wide-character code representing digit [0-9], excluding ASCII characters.

isspecial(*wc*) tests whether wc is a wide-character code representing a special language character, excluding ASCII characters.

SEE ALSO

localedef(), setlocale(), stdio(), wconv()

putws

NAME

putws - convert a string of Process Code characters to EUC characters

SYNOPSIS

#include <stdio.h>

#include <widec.h>

int putws(wchar_t *s);

DESCRIPTION

The ***putws***() function converts the Process Code string (terminated by a (*wchar_t*)NULL) pointed to by s, to an Extended Unix Code (**EUC**) string followed by a **NEWLINE** character, and writes it to the standard output stream stdout. It does not write the terminal null character.

RETURN VALUES

the ***putws***() function returns the number of Process Code characters transformed and written. It returns EOF if it attempts to write to a file that has not been opened for writing.

SEE ALSO

ferror(), fopen(), fread(), getws(), printf(), putwc()

towlower

NAME

towlower - transliterate upper-case wide-character code to lower-case

SYNOPSIS

#include <wchar.h>

wint_t towlower(wint_t wc);

DESCRIPTION

The ***towlower***() function has as a domain a type *wint_t*, the value of which must be a character representable as a *wchar_t*, and must be a wide-character code corresponding to a valid character in the current locale or the value of **WEOF**. If the argument has any other value, the argument is returned unchanged. If the argument of ***towlower***() represents an upper-case wide-character code, and there exists a corresponding lower-case wide-character code (as defined by character type information in the program locale category **LC_CTYPE**), the result is the corresponding lower-case wide-character code. All other arguments in the domain are returned unchanged.

RETURN VALUES

On successful completion, ***towlower***() returns the lower-case letter corresponding to the argument passed. Otherwise, it returns the argument unchanged.

SEE ALSO

iswalpha(), setlocale(), towupper()

towupper

NAME

towupper - transliterate lowercase wide-character code to uppercase

SYNOPSIS

#include <wchar.h>

wint_t towupper(wint_t wc);

DESCRIPTION

The ***towupper***() function has as a domain a type *wint_t*, the value of which must be a character representable as a *wchar_t*, and must be a wide-character code corresponding to a valid character in the current locale or the value of **WEOF**. If the argument has any other value, the argument is returned unchanged. If the argument of ***towupper***() represents a lowercase wide-character code (as defined by character type information in the program locale category **LC_CTYPE**), the result is the corresponding uppercase wide-character code. All other arguments in the domain are returned unchanged.

RETURN VALUES

Upon successful completion, ***towupper***() returns the uppercase letter corresponding to the argument passed. Otherwise, it returns the argument unchanged.

SEE ALSO

iswalpha(), setlocale(), towlower()

ungetwc

NAME

ungetwc - push wide-character code back into input stream

SYNOPSIS

#include <stdio.h>

#include <wchar.h>

wint_t ungetwc(wint_t wc, FILE *stream);

DESCRIPTION

The ***ungetwc***() function pushes the character corresponding to the wide character code specified by wc back onto the input stream pointed to by stream. The pushed-back characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by stream) to a file-positioning function (***fseek***(), ***fsetpos***() or ***rewind***()) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged. One character of push-back is guaranteed. If ***ungetwc***() is called too many times on the same stream without an intervening read or file-positioning operation on that stream, the operation may fail. If the value of wc equals that of the macro **WEOF**, the operation fails and the input stream is unchanged. A successful call to ***ungetwc***() clears the end-of-file indicator for the stream. The value of the file-position indicator for the stream after reading or discarding all pushed-back characters will be the same as it was before the characters were pushed back. The file-position indicator is decremented (by one or more) by each successful call to ***ungetwc***(); if its value was 0 before a call, its value is indeterminate after the call.

RETURN VALUES

Upon successful completion, ***ungetwc***() returns the wide-character code corresponding to the pushed-back character. Otherwise it returns **WEOF**.

ERRORS

The ***ungetwc***() function may fail if:

EILSEQ An invalid character sequence is detected, or a wide-character code does not correspond to a valid character.

SEE ALSO

read(2), fseek(), fsetpos(), rewind(), setbuf()

wscasecmp, wscol, wsdup, wsncasecmp

NAME

wscasecmp, wsncasecmp, wsdup, wscol - Process Code string operations

SYNOPSIS

#include <widec.h>

int	***wscasecmp***	***(const wchar_t *s1, const wchar_t *s2);***
int	***wsncasecmp***	***(const wchar_t *s1, const wchar_t *s2, int n);***
wchar_t	****wsdup***	***(const wchar_t *s);***
int	***wscol***	***(const wchar_t *s);***

DESCRIPTION

These functions operate on Process Code strings terminated by *wchar_t* NULL characters. During appending or copying, these routines do not check for an overflow condition of the receiving string. In the following, *s*, *s1*, and *s2* point to Process Code strings terminated by a *wchar_t* NULL.

wscasecmp(), wsncasecmp()

The ***wscasecmp***() function compares its arguments, ignoring case, and returns an integer greater than, equal to, or less than 0, depending upon whether *s1* is lexicographically greater than, equal to, or less than s2. ***wsncasecmp***() makes the same comparison but compares at most n Process Code characters. The four Extended Unix Code (**EUC**) codesets are ordered from lowest to highest as 0, 2, 3, 1 when characters from different codesets are compared.

wsdup()

The ***wsdup***() function returns a pointer to a new Process Code string, which is a duplicate of the string pointed to by s. The space for the new string is obtained using ***malloc***(). If the new string cannot be created, a null pointer is returned.

wscol()

The ***wscol***() function returns the screen display width (in columns) of the Process Code string s.

SEE ALSO

malloc(), string(), wcstring()

wcstring, wcscat, wscat
wcsncat, wsncat, wcscmp, wscmp
wcsncmp, wsncmp, wcscpy, wscpy
wcsncpy, wsncpy, wcslen, wslen
wcschr, wschr, wcsrchr, wsrchr
windex, wrindex, wcspbrk, wspbrk
wcswcs, wcsspn, wsspn,
wcscspn, wscspn, wcstok, wstok

NAME

wcstring, wcscat, wscat, wcsncat, wsncat, wcscmp, wscmp, wcsncmp, wsncmp, wcscpy, wscpy, wcsncpy, wsncpy, wcslen, wslen, wcschr, wschr, wcsrchr, wsrchr, windex, wrindex, wcspbrk, wspbrk, wcswcs, wcsspn, wsspn, wcscspn, wscspn, wcstok, wstok - wide character string operations

SYNOPSIS

```
#include <wchar.h>
wchar_t     *wcscat     (wchar_t *ws1,         const wchar_t *ws2);
wchar_t     *wscat      (wchar_t *ws1,         const wchar_t *ws2);
wchar_t     *wcsncat    (wchar_t *ws1,         const wchar_t *ws2,     size_t n);
wchar_t     *wsncat     (wchar_t *ws1,         const wchar_t *ws2,     size_t n);
int         wcscmp      (const wchar_t *ws1,   const wchar_t *ws2);
int         wscmp       (const wchar_t *ws1,   const wchar_t *ws2);
int         wcsncmp     (const wchar_t *ws1,   const wchar_t *ws2,     size_t n);
int         wsncmp      (const wchar_t *ws1,   const wchar_t *ws2,     size_t n);
wchar_t     *wcscpy     (wchar_t *ws1,         const wchar_t *ws2);
wchar_t     *wscpy      (wchar_t *ws1,         const wchar_t *ws2);
wchar_t     *wcsncpy    (wchar_t *ws1,         const wchar_t *ws2,     size_t n);
wchar_t     *wsncpy     (wchar_t *ws1,         const wchar_t *ws2,     size_t n);
size_t      wcslen      (const wchar_t *ws);
size_t      wslen       (const wchar_t *ws);
wchar_t     *wcschr     (const wchar_t *ws,    wint_t wc);
wchar_t     *wschr      (const wchar_t *ws,    wint_t wc);
wchar_t     *wcsrchr    (const wchar_t *ws,    wchar_t wc);
wchar_t     *wsrchr     (const wchar_t *ws,    wint_t wc);
wchar_t     *windex     (const wchar_t *ws,    wchar_t wc);
wchar_t     *wrindex    (const wchar_t *ws,    wchar_t wc);
wchar_t     *wcspbrk    (const wchar_t *ws1,   const wchar_t *ws2);
wchar_t     *wspbrk     (const wchar_t *ws1,   const wchar_t *ws2);
wchar_t     *wcswcs     (const wchar_t *ws1,   const wchar_t *ws2);
size_t      wcsspn      (const wchar_t *ws1,   const wchar_t *ws2);
size_t      wsspn       (const wchar_t *ws1,   const wchar_t *ws2);
```

size_t	***wcscspn***	***(const wchar_t *ws1,***	***const wchar_t *ws2);***
size_t	***wscspn***	***(const wchar_t *ws1,***	***const wchar_t *ws2);***
wchar_t	****wcstok***	***(wchar_t *ws1,***	***const wchar_t *ws2);***
wchar_t	****wstok***	***(wchar_t *ws1,***	***const wchar_t *ws2);***

DESCRIPTION

These functions operate on wide character strings terminated by *wchar_t* NULL characters. During appending or copying, these routines do not check for an overflow condition of the receiving string. In the following, *ws*, *ws1*, and *ws2* point to wide character strings terminated by a *wchar_t* NULL.

wcscat(), wscat()

The ***wcscat***() and ***wscat***() functions append a copy of the wide character string pointed to by *ws2* (including the terminating null wide-character code) to the end of the wide character string pointed to by ws1. The initial wide-character code of *ws2* overwrites the null wide-character code at the end of ws1. If copying takes place between objects that overlap, the behavior is undefined. Both functions return s1; no return value is reserved to indicate an error.

wcsncat(), wsncat()

The ***wcsncat***() and ***wsncat***() functions append not more than n wide-character codes (a null wide-character code and wide character codes that follow it are not appended) from the array pointed to by *ws2* to the end of the wide character string pointed to by *ws1*. The initial wide-character code of *ws2* overwrites the null wide-character code at the end of ws1. A terminating null wide-character code is always appended to the result. Both functions return ws1; no return value is reserved to indicate an error.

wcscmp(), wscmp()

The ***wcscmp***() and ***wscmp***() functions compare the wide character string pointed to by ws1 to the wide character string pointed to by *ws2*. The sign of a nonzero return value is determined by the sign of the difference between the values of the first pair of wide-character codes that differ in the objects being compared. Upon completion, both functions return an integer greater than, equal to, or less than zero, if the wide character string pointed to by ws1 is greater than, equal to, or less than the wide character string pointed to by *ws2*.

wcsncmp(), wsncmp()

The ***wcsncmp***() and ***wsncmp***() functions compare not more than n wide-character codes (wide-character codes that follow a null wide character code are not compared) from the array pointed to by *ws1* to the array pointed to by *ws2*. The sign of a nonzero return value is determined by the sign of the difference between the values of the first pair of wide-character codes that differ in the objects being compared. Upon successful completion, both functions return an integer greater than, equal to, or less than zero, if the possibly null-terminated array pointed to by *ws1* is greater than, equal to, or less than the possibly null-terminated array pointed to by *ws2*.

wcscpy(), wscpy()

The ***wcscpy***() and ***wscpy***() functions copy the wide character string pointed to by *ws2* (including the terminating null wide-character code) into the array pointed to by *ws1*. If copying takes place between objects that overlap, the behavior is undefined. Both functions return ws1; no return value is reserved to indicate an error.

wcsncpy(), wsncpy()

The ***wcsncpy***() and ***wsncpy***() functions copy not more than n wide-character codes (wide-character codes that follow a null wide character code are not copied) from the array pointed to by *ws2* to the array pointed to by *ws1*. If copying takes place between objects that overlap, the behavior is undefined. If the array pointed to by *ws2* is a wide character string that is shorter than n wide-character codes, null wide-character codes are appended to the copy in the array pointed to by ws1, until a total n wide-character codes are written. Both functions return ws1; no return value is reserved to indicate an error.

wcslen(), wslen()

The ***wcslen***() and ***wslen***() functions compute the number of wide-character codes in the wide character string to which ws points, not including the terminating null wide-character code. Both functions return *ws*; no return value is reserved to indicate an error.

wcschr(), wschr()

The ***wcschr***() and ***wschr***() functions locate the first occurrence of wc in the wide character string pointed to by ws. The value of *wc* must be a character representable as a type *wchar_t* and must be a wide-character code corresponding to a valid character in the current locale. The terminating null wide-character code is considered to be part of the wide character string. Upon completion, both functions return a pointer to the wide-character code, or a null pointer if the wide-character code is not found.

wcsrchr(), wsrchr()

The ***wcsrchr***() and ***wsrchr***() functions locate the last occurrence of wc in the wide character string pointed to by ws. The value of wc must be a character representable as a type *wchar_t* and must be a wide-character code corresponding to a valid character in the current locale. The terminating null wide-character code is considered to be part of the wide character string. Upon successful completion, both functions return a pointer to the wide-character code, or a null pointer if wc does not occur in the wide character string.

windex(), wrindex()

The ***windex***() and ***wrindex***() functions behave the same as ***wschr***() and ***wsrchr***(), respectively.

wcspbrk(), wspbrk()

The ***wcspbrk***() and ***wspbrk***() functions locate the first occurrence in the wide character string pointed to by *ws1* of any wide-character code from the wide character string pointed to by *ws2*. Upon successful completion, the function returns a pointer to the wide-character code, or a null pointer if no wide-character code from *ws2* occurs in *ws1*.

wcswcs()

The ***wcswcs***() function locates the first occurrence in the wide character string pointed to by ws1 of the sequence of wide-character codes (excluding the terminating null wide-character code) in the wide character string pointed to by *ws2*. Upon successful completion, the function returns a pointer to the located wide character string, or a null pointer if the wide character string is not found. If *ws2* points to a wide character string with zero length, the function returns ws1.

wcsspn(), wsspn()

The ***wcsspn***() and ***wsspn***() functions compute the length of the maximum initial segment of the wide character string pointed to by ws1 which consists entirely of wide-character codes from the wide string pointed to by *ws2*. Both functions return *ws1*; no return value is reserved to indicate an error.

wcscspn(), wscspn()

The ***wcscspn***() and ***wscspn***() functions compute the length of the maximum initial segment of the wide character string pointed to by *ws1* which consists entirely of wide-character codes not from the wide character string pointed to by *ws2*. Both functions return ws1; no return value is reserved to indicate an error.

wcstok(), wstok()

A sequence of calls to the ***wcstok***() and ***wstok***() functions break the wide character string pointed to by ws1 into a sequence of tokens, each of which is delimited by a wide-character code from the wide character string pointed to by *ws2*. The first call in the sequence has *ws1* as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by *ws2* may be different from call to call.

The first call in the sequence searches the wide character string pointed to by ws1 for the first wide-character code that is not contained in the current separator string pointed to by *ws2*. If no such wide-character code is found, then there are no tokens in the wide character string pointed to by *ws1*, and ***wcstok***() and ***wstok***() return a null pointer. If such a wide-character code is found, it is the start of the first token.

wcstok() and ***wstok***() then search from that point for a wide-character code that is contained in the current separator string. If no such wide-character code is found, the current token extends to the end of the wide character string pointed to by *ws1*, and subsequent searches for a token will return a null pointer. If such a wide-character code is found, it is overwritten by a null wide character, which terminates the current token. ***wcstok***() and ***wstok***() save a pointer to the following wide-character code, from which the next search for a token will start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above. Upon successful completion, both functions return a pointer to the first wide-character code of a token. Otherwise, if there is no token, a null pointer is returned.

SEE ALSO

malloc(), string(), wcswidth(), wcwidth()

wcscoll, wscoll

NAME

wcscoll, wscoll - wide character string comparison using collating information

SYNOPSIS

#include <wchar.h>

int wcscoll(const wchar_t *ws1, const wchar_t *ws2);

int wscoll(const wchar_t *ws1, const wchar_t *ws2);

DESCRIPTION

The ***wcscoll***() and ***wscoll***() functions compare the wide character string pointed to by ws1 to the wide character string pointed to by *ws2*, both interpreted as appropriate to the **LC_COLLATE** category of the current locale.

RETURN VALUES

Upon successful completion, ***wcscoll***() and ***wscoll***() return an integer greater than, equal to, or less than 0, depending upon whether the wide character string pointed to by *ws1* is greater than, equal to, or less than the wide character string pointed to by *ws2*, when both are interpreted as appropriate to the current locale. On error, ***wcscoll***() and ***wscoll***() may set errno, but no return value is reserved to indicate an error.

ERRORS

wcscoll() and ***wscoll***() may fail if:

EINVAL The *ws1* or *ws2* arguments contain wide character codes outside the domain of the collating sequence.

ENOSYS The function is not supported.

SEE ALSO

setlocale(), wcscmp(), wcsxfrm()

NOTES

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set errno to 0, call either ***wcscoll***() or wscoll(), then check errno and if it is nonzero, assume an error has occurred. ***wcsxfrm***() and ***wcscmp***() should be used for sorting large lists. ***wcscoll***() and ***wscoll***() can be used safely in multithreaded applications as long as ***setlocale***() is not being called to change the locale.

wsprintf

NAME

wsprintf - formatted output conversion

SYNOPSIS

#include <stdio.h>

#include <widec.h>

int wsprintf(wchar_t *s, const char *format, /* arg */...);

DESCRIPTION

wsprintf() outputs a Process Code string ending with a Process Code (*wchar_t*) NULL character. It is the user's responsibility to allocate enough space for this *wchar_t* string.

This returns the number of Process Code characters (excluding the NULL terminator) that have been written. The conversion specifications and behavior of ***wsprintf***() are the same as the regular ***sprintf***() function except that the result is a Process Code string for ***wsprintf***(), and on Extended Unix Code (**EUC**) character string for ***sprintf***().

RETURN VALUES

Upon success, ***wsprintf***() returns the number of characters printed. When an error condition is encountered, a negative value is returned.

SEE ALSO

wsscanf(), printf(), scanf(), sprintf()

wsscanf

NAME

wsscanf - formatted input conversion

SYNOPSIS

#include <stdio.h>

#include <widec.h>

int wsscanf(wchar_t *s, const char *format, /* pointer */...);

DESCRIPTION

wsscanf() reads Process Code characters from the Process Code string s, interprets them according to the format, and stores the results in its arguments. ***wsscanf***() expects, as arguments, a control string format, and a set of pointer arguments indicating where the converted input should be stored. The results are undefined if there are insufficient args for the format. If the format is exhausted while args remain, the excess args are simply ignored.

The conversion specifications and behavior of ***wsscanf***() are the same as the regular ***sscanf***() function except that the source is a Process Code string for ***wsscanf***(), and on Extended Unix Code (EUC) character string for ***sscanf***().

RETURN VALUES

wsscanf() returns the number of characters matched.   On error ***wsscanf***() returns a negative value.

SEE ALSO

wsprintf(), printf(), scanf()

wcstod, wstod, watof

NAME

wcstod, wstod, watof - convert wide character string to double-precision number

SYNOPSIS

#include <wchar.h>

double wcstod(const wchar_t *nptr, wchar_t **endptr);

double wstod(const wchar_t *nptr, wchar_t **endptr);

double watof(wchar_t *nptr);

DESCRIPTION

The ***wcstod***() and ***wstod***() functions convert the initial portion of the wide character string pointed to by nptr to double representation. They first decompose the input wide character string into three parts: an initial, possibly empty, sequence of whitespace wide character codes (as specified by ***iswspace***()); a subject sequence interpreted as a floating-point constant; and a final wide-character string of one or more unrecognised wide-character codes, including the terminating null wide character code of the input wide character string. They then attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional `+' or `-' sign, then a non-empty sequence of digits optionally containing a radix, then an optional exponent part. An exponent part consists of `e' or `E', followed by an optional sign, followed by one or more decimal digits. The subject sequence is defined as the longest initial subsequence of the input wide character string, starting with the first non-white-space wide-character code, that is of the expected form. The subject sequence contains no wide-character codes if the input wide character string is empty or consists entirely of whitespace wide-character codes, or if the first wide-character code that is not white space other than a sign, a digit or a radix.

If the subject sequence has the expected form, the sequence of wide-character codes starting with the first digit or the radix (whichever occurs first) is interpreted as a floating constant as defined in the C language, except that the radix is used in place of a period, and that if neither an exponent part nor a radix appears, a radix is assumed to follow the last digit in the wide character string. If the subject sequence begins with a minus sign (-), the value resulting from the conversion is negated. A pointer to the final wide character string is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

The radix is defined in the program's locale (category **LC_NUMERIC**). In the **POSIX** locale, or in a locale where the radix is not defined, the radix defaults to a period (.). In other than the **POSIX** locale, other implementation-dependent subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of nptr is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

watof(str) is equivalent to wstod(str, (*wchar_t* **)NULL).

RETURN VALUES

wcstod() and ***wstod***() return the converted value, if any. If no conversion could be performed, 0 is returned, and errno may be set to **EINVAL**. If the correct value is outside the range of representable values, **+_HUGE_VAL** is returned (according to the sign of the value), and errno is set to **ERANGE**. If the correct value would cause underflow, 0 is returned, and errno is set to **ERANGE**.

ERRORS

wcstod() and ***wstod***() will fail if:

ERANGE The value to be returned would cause overflow or underflow.

wcstod() and ***wcstod***() may fail if:

EINVAL No conversion could be performed.

SEE ALSO

iswspace(), ***localeconv***(), ***scanf***(), ***setlocale***(), ***wcstol***()

NOTES

Because 0 is returned on error and is also a valid return on success, an application wishing to check for error situations should set errno to 0, call ***wcstod***() or ***wstod***(), then check errno and if it is nonzero, assume an error has occurred.

wcstol, wstol, watol, watoll, watoi

NAME

wcstol, wstol, watol, watoll, watoi - convert wide character string to long integer

SYNOPSIS

#include <wchar.h>

long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);

#include <widec.h>

long int wstol(const wchar_t *nptr, wchar_t **endptr, int base);

long watol(wchar_t *nptr);

long long watoll(wchar_t *nptr);

int watoi(wchar_t *nptr);

DESCRIPTION

The ***wcstol***() and ***wstol***() functions convert the initial portion of the wide character string pointed to by nptr to long int representation. They first decompose the input wide character string into three parts: an initial, possibly empty, sequence of whitespace wide-character codes (as specified by ***iswspace***()), a subject sequence interpreted as an integer represented in some radix determined by the value of base; and a final wide character string of one or more unrecognised wide character codes, including the terminating null wide-character code of the input wide character string. They then attempt to convert the subject sequence to an integer, and return the result.

If the value of base is 0, the expected form of the subject sequence is that of a decimal constant, octal constant or hexadecimal constant, any of which may be preceded by a `+' or `-' sign. A decimal constant begins with a nonzero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix `0' optionally followed by a sequence of the digits `0' to `7' only. A hexadecimal constant consists of the prefix `0x' or `0X' followed by a sequence of the decimal digits and letters `a' (or `A') to `f' (or `F') with values 10 to 15 respectively.

If the value of base is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by base, optionally preceded by a `+' or `-' sign, but not including an integer suffix. The letters from `a' (or `A') to ’z' (or `Z') inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of base are permitted. If the value of base is 16, the wide-character code representations of `0x' or `0X' may optionally proceed the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide character string, starting with the first non-white-space wide-character code, that is of the expected form. The subject sequence contains no wide-character codes if the input wide character string is empty or consists entirely of whitespace wide-character code, or if the first non-white-space wide-character code is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of base is 0, the sequence of wide-character codes starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of base is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign (-), the value resulting from the conversion is negated. A pointer to the final wide character string is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

In other than the **POSIX** locale, additional implementation-dependent subject sequence forms may be accepted. If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of nptr is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

The ***watol***() function is equivalent to ***wstol***(str, (*wchar_t* **)NULL, 10). The ***watoll***() function is the long-long (*double long)* version of ***watol***(). The ***watoi***() function is equivalent to (int)***watol***().

RETURN VALUES

Upon successful completion, ***wcstol***() and ***wstol***() return the converted value, if any. If no conversion could be performed, 0 is returned, and errno may be set to indicate the error. If the correct value is outside the range of representable values, {**LONG_MAX**} or {**LONG_MIN**} is returned (according to the sign of the value), and *errno* is set to **ERANGE**.

ERRORS

The ***wcstol***() and ***wstol***() functions will fail if:

EINVAL The value of base is not supported.

ERANGE The value to be returned is not representable.

The ***wcstol***() and ***wstol***() functions may fail if:

EINVAL No conversion could be performed.

SEE ALSO

iswalpha(), ***iswspace***(), ***scanf***(), ***wcstod***()

NOTES

Because 0, {**LONG_MIN**}, and {**LONG_MAX**} are returned on error and are also valid returns on success, an application wishing to check for error situations should set errno to 0, call ***wcstol***() or ***wstol***(), then check errno and if it is nonzero assume an error has occurred. Truncation from long long to long can take place upon assignment or by an explicit cast.

wcsxfrm, wsxfrm

NAME

wcsxfrm, ***wsxfrm*** - wide character string transformation

SYNOPSIS

#include <wchar.h>

size_t wcsxfrm(wchar_t *ws1, const wchar_t *ws2, size_t n);

size_t wsxfrm(wchar_t *ws1, const wchar_t *ws2, size_t n);

DESCRIPTION

The ***wcsxfrm***() and ***wcsxfrm***() functions transform the wide character string pointed to by *ws2* and place the resulting wide character string into the array pointed to by ws1. The transformation is such that if either the ***wcscmp***() or ***wscmp***() functions are applied to two transformed wide strings, they return a value greater than, equal to, or less than 0, corresponding to the result of the ***wcscoll***() or ***wscoll***() function applied to the same two original wide character strings. No more than n wide-character codes are placed into the resulting array pointed to by *ws1*, including the terminating null wide-character code. If n is 0, *ws1* is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUES

wcsxfrm() and ***wsxfrm***() return the length of the transformed wide character string (not including the terminating null wide-character code). If the value returned is n or more, the contents of the array pointed to by ws1 are indeterminate. On error, ***wcsxfrm***() and ***wsxfrm***() return (*size_t*)-1, and set errno to indicate the error.

ERRORS

wcsxfrm() and ***wsxfrm***() may fail if:

EINVAL The wide character string pointed to by *ws2* contains wide-character codes outside the domain of the collating sequence.

ENOSYS The function is not supported.

SEE ALSO

setlocale(), ***wcscmp***(), ***wcscoll***(),***wscmp***(), ***wscoll***()

NOTES

The transformation function is such that two transformed wide character strings can be ordered by the ***wcscmp***() or ***wscmp***() functions as appropriate to collating sequence information in the program's locale (category **LC_COLLATE**). The fact that when n is 0, *ws1* is permitted to be a null pointer, is useful to determine the size of the *ws1* array prior to making the transformation. Because no return value is reserved to indicate an error, an application wishing to check for error situations should set errno to 0, call ***wcsxfrm***() or ***wsxfrm***(), then check errno and if it is non-zero, assume an error has occurred. ***wcsxfrm***() and ***wsxfrm***() can be used safely in multi-threaded applications as long as ***setlocale***() is not being called to change the locale.

SPARC COMPLIANCE DEFINITION 2.4.1 IS

Large Files Interfaces

Large File Support Interfaces

Overview

This section of the SCD IS defines large file support interfaces for the 32-bit ABI--these interfaces do not apply to the 64-bit ABI. There is no large file support library. Instead, large file support interfaces are provided within other dynamic libraries as described in the SCD. The SCD also specifies those interface members which are REQUIRED and those which are EXPERIMENTAL.

Rationale:

Interface members required in the document: "*Adding Support for Arbitrary File Sizes to the Single UNIX Specification*" have been implemented by multiple system vendors and are REQUIRED in the SCD. All other interface members are EXPERIMENTAL.

On a 32-bit system, a large file is a regular file whose size is greater than or equal to 2 Gbyte (2^{31} bytes). A small file is a regular file whose size is less than 2 Gbyte.

System Interfaces

The following table summarizes the differences in function signature definitions between small file (standard, 32-bit) and their corresponding large file (64-bit) interfaces.

Small File Definition	**Large File Definition**	**Header**
*int aio_cancel(...,struct aiocb *);*	*int aio_cancel64(...,struct aiocb64 *);*	*<aio.h>*
*int aio_error(const struct aiocb *);*	*int aio_error64(const struct aiocb64 *);*	
*int aio_fsync(..., struct aiocb *);*	*int aio_fsync64(...,struct aiocb64 *);*	
*int aio_read(struct aiocb *);*	*int aio_read64(struct aiocb64 *);*	
*int aio_return(struct aiocb *);*	*int aio_return64(struct aiocb64 *);*	
*int aio_suspend(const struct aiocb *,...);*	*int aio_suspend64 (const struct aiocb64 *,...)*	
*int aio_write(struct aiocb *);*	*int aio_write64(struct aiocb64 *);*	
*int lio_listio(..., const struct aiocb *,...);*	*int lio_listio64(...,const struct aiocb64 *,...);*	
*struct dirent *readdir;*	*struct dirent64 *readdir64;*	*<dirent.h>*
*struct dirent *readdir_r;*	*struct dirent64 *readdir64_r;*	
int creat;	*int creat64;*	*<fcntl.h>*
int open;	*int open64;*	
*int ftw(..., const struct stat *,...);*	*int ftw64(..., const struct stat64 *,...);*	*<ftw.h>*
*int nftw(..., const struct stat *,...);*	*int nftw64(..., const struct stat64 *,...);*	

int fgetpos;	*int fgetpos64;*	*<stdio.h>*
*FILE *fopen;*	*FILE *fopen64;*	
*FILE *freopen;*	*FILE *freopen64;*	
int fseeko(...,off_t,...);	*int fseeko64(..., off64_t,...);*	
*int fsetpos(...,const fpos_t *);*	*int fsetpos64(...,const fpos64_t *);*	
off_t ftello;	*off64_t ftello64;*	
*FILE *tmpfile;*	*FILE *tmpfile64;*	
int mkstemp;	*int mkstemp64;*	*<stdlib.h>*
int aioread(...,off_t,...);	*int aioread64(..., off64_t,...);*	*<sys/async.h>*
int aiowrite(..., off_t,...);	*int aiowrite64(...,off64_t,...);*	
*int alphasort(struct direct **,..);*	*int alphasort64(struct direct64 **);*	*<sys/dir.h>*
*struct direct *readdir;*	*struct direct64 *readdir64;*	
*int scandir(..., struct direct *(*[]),...);*	*int scandir64(...,struct direct64 *(*[]),...);*	
int getdents(..., dirent);	*int getdents64(..., dirent64);*	*<sys/dirent.h>*
void mmap(..., off_t);	*void mmap64(..., off64_t);*	*<sys/mman.h>*
*int getrlimit(...,struct rlimit *);*	*int getrlimit64(..., struct rlimit64 *);*	*<sys/resource.h>*
*int setrlimit(..., const struct rlimit *);*	*int setrlimit64(..., const struct rlimit64 *);*	
*int fstat(...,struct stat *);*	*int fstat64(...,struct stat64 *);*	*<sys/stat.h>*
*int lstat(...,struct stat *);*	*int lstat64(...,struct stat64 *);*	
*int stat(...,struct stat *);*	*int stat64(...,struct stat64 *);*	
*int statvfs(...,struct statvfs *);*	*int statvfs64(...,struct statvfs64 *);*	*<sys/statvfs.h>*
*int fstatvfs(...,struct statvfs *);*	*int fstatvfs64(...,struct statvfs64 *);*	
int lockf(..., off_t);	*int lockf64(...,off64_t);*	*<unistd.h>*
off_t lseek(...,off_t,...);	*off64_t lseek64(...,off64_t,...);*	
int ftruncate(...,off_t);	*int ftruncate64(...,off64_t);*	
ssize_t pread(...,off_t);	*ssize_t pread64(...,off64_t);*	
ssize_t pwrite(...,off_t);	*ssize_t pwrite64(...,off64_t);*	
int truncate(..., off_t);	*int truncate64(...,off64_t);*	

creat64 (libc, libthread)

NAME

creat64 - create a new file or rewrite an existing one in large file environment

SYNOPSIS

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int creat64 (const char *path, mode_t mode);

DESCRIPTION

The ***creat64*** function creates a new ordinary file or prepares to rewrite an existing file named by the path name pointed to by *path*.

If the file exists, the length is truncated to 0 and the mode and owner are unchanged.

If the file does not exist the file's owner ID is set to the effective user ID of the process. The group ID of the file is set to the effective group ID of the process, or if the **S_ISGID** bit is set in the parent directory then the group ID of the file is inherited from the parent directory. The access permission bits of the file mode are set to the value of *mode* modified as follows:

- If the group ID of the new file does not match the effective group ID or one of the supplementary group IDs, the **S_ISGID** bit is cleared.
- All bits set in the process's file mode creation mask are cleared (see ***umask***).
- The "save text image after execution bit" of the mode is cleared (see ***chmod*** for the values of mode).

Upon successful completion, a write-only file descriptor is returned and the file is open for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across ***exec*** functions (see ***fcntl***). A new file may be created with a mode that forbids writing.

The call ***creat64***(*path*, *mode*) is equivalent to:

> ***open64***(*path*, **O_WRONLY** | **O_CREAT** | **O_TRUNC**, *mode*)

RETURN VALUES

Upon successful completion a non-negative integer, namely the lowest numbered unused file descriptor, is returned. Otherwise, a value of -1 is returned, no files are created or modified, and **errno** is set to indicate the error.

ERRORS

The ***creat64*** function fails if one or more of the following are true:

EACCESS Search permission is denied on a component of the path prefix. The file doesn't exist and the directory in which the file is to be created does not permit writing.

EAGAIN The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file (see ***chmod***).

EDQUOT The directory where the new file entry is being placed cannot be extended because the user's quota of disk blocks on that file system has been exhausted.

The user's quota of *inodes* on the file system where the file is being created has been exhausted.

EFAULT *path* points to an illegal address.

EINTR A signal was caught during the ***creat64*** function.

EISDIR The named file is an existing directory.

ELOOP Too many symbolic links were encountered in translating *path*.

EMFILE The process has too many open files (see ***getrlimit64***).

EMULTIHOP Components of *path* require hopping to multiple remote machines.

ENAMETOOLONG The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a path component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.

ENFILE The system file table is full.

ENOENT A component of the path prefix does not exist.

ENOLINK *path* points to a remote machine and the link to that machine is no longer active.

ENOSPC The file system is out of inodes.

ENOTDIR A component of the path prefix is not a directory.

EROFS The named file resides or would reside on a ready-only file system.

SEE ALSO

chmod, close, dup, fcntl, getrlimit64, lseeko64, open64, read, umask, write, stat64.

fopen64 (libc), freopen64 (libc)

NAME

fopen64, freopen64 - open a stream file in large file environment

SYNOPSIS

#include <stdio.h>

FILE *fopen64(const char *filename, const char *type);

FILE *freopen64(const char *filename, const char *type, FILE *stream);

DESCRIPTION

fopen64 opens the file named by filename and associates a stream with it. ***fopen64*** returns a pointer to the FILE structure associated with the stream. *filename* points to a character string that contains the name of the file to be opened. type is a character string beginning with one of the following sequences:

"r" or *"rb"*	open for reading
"w" or "wb"	truncate to zero length or create for writing
"a" or "ab"	append; open for writing at end of file, or create for writing
"r+", "r+b" or "rb+"	open for update (reading and writing)
"w+", "w+b" or "wb+"	truncate or create for update
"a+", "a+b" or "ab+"	append; open or create for update at end-of-file

The "b" is ignored in the above types. The "b" exists to distinguish binary files from text files. However, there is no distinction between these types of files on a UNIX system. ***freopen64*** substitutes the named file in place of the open stream. A flush is first attempted, and then the original stream is closed, regardless of whether the open ultimately succeeds. Failure to flush or close stream successfully is ignored. ***freopen64*** returns a pointer to the FILE structure associated with stream. ***freopen64*** is typically used to attach the pre-opened streams associated with stdin, stdout, and stderr to other files. stderr is by default unbuffered, but the use of ***freopen64*** will cause it to become buffered or line-buffered. When a file is opened for update, both input and output may be done on the resulting stream. However, output may not be directly followed by input without an intervening ***fflush***, ***fseek***, ***fsetpos64***, or ***rewind***, and input may not be directly followed by output without an intervening ***fseek***, ***fsetpos64***, or ***rewind***, or an input operation that encounters end-of-file.

When a file is opened for append (that is, when type is "a", "ab", "a+", or "ab+"), it is impossible to overwrite information already in the file. ***fseek*** may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written. When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device.The error and end-of-file indicators are cleared for the stream.

RETURN VALUES

The functions ***fopen64*** and ***freopen64*** return a null pointer if path cannot be accessed, or if type is invalid, or if the file cannot be opened. The functions ***fopen64*** may fail and not set *errno* if there are no free *stdio* streams.

SEE ALSO

close, creat64, dup, open64, pipe, write, fclose, fseek, setbuf, stdio

fseeko64 (libc)

NAME

fseeko64 - reposition a file-position indicator in a stream in large file environment.

SYNOPSIS

#include <stdio.h>

int fseeko64 *(**FILE** *stream, **off64_t** offset, **int** whence);*

DESCRIPTION

fseeko64 sets the position of the next input or output operation on the stream. The new position is at the signed distance offset bytes from the beginning, from the current position, or from the end of the file, according to a ptrname value of **SEEK_SET**, **SEEK_CUR**, or **SEEK_END** (defined in **<***stdio.h***>**) as follows:

SEEK_SET	set position equal to offset bytes.
SEEK_CUR	set position to current location plus offset.
SEEK_END	set position to **EOF** plus offset.

fseeko64 allows the file position indicator to be set beyond the end of the existing data in the file. If data is later written at this point, subsequent reads of data in the gap will return zero until data is actually written into the gap. ***fseeko64,*** by itself, does not extend the size of the file.

RETURN VALUES

fseeko64 returns -1 for improper seeks, otherwise zero. An improper seek can be, for example, an *fseeko64* done on a file that has not been opened via ***fopen64***; in particular, ***fseeko64*** may not be used on a terminal or on a file opened via ***popen***. After a stream is closed, no further operations are defined on that stream.

fgetpos64 (libc), fsetpos64 (libc)

NAME

fsetpos64, fgetpos64 *- reposition a file pointer in a stream in large file environment*

SYNOPSIS

#include <stdio.h>

int fsetpos64(FILE *stream, const fpos64_t *pos);

int fgetpos64(FILE *stream, fpos64_t *pos);

DESCRIPTION

fsetpos64 sets the position of the next input or output operation on the stream according to the value of the object pointed to by *pos*. The object pointed to by *pos* must be a value returned by an earlier call to ***fgetpos64*** on the same stream.

fsetpos64 clears the end-of-file indicator for the stream and undoes any effects of the ***ungetc*** function on the same stream. After ***fsetpos64***, the next operation on a file opened for update may be either input or output.

fgetpos64 stores the current value of the file position indicator for stream in the object pointed to by pos. The value stored contains information usable by ***fsetpos64*** for repositioning the stream to its position at the time of the call to ***fgetpos64.***

RETURN VALUES

If successful, both ***fsetpos64*** and ***fgetpos64*** return zero. Otherwise, they both return nonzero.

stat64 (libc), lstat64 (libc), fstat64 (libc)

NAME

stat64, lstat64, fstat64 – get file status in large file environment

SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>
int stat64 (const char * path, struct stat64 * buf);

int lstat64 (const char * path, struct stat64 * buf);

int fstat64 (int fildes, struct stat64 * buf);

DESCRIPTION

The ***stat64*** function obtains information about the file pointed to by *path*. Read, write, or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

The ***lstat64*** function obtains file attributes similar to ***stat64***, except when the named file is a symbolic link; in that case ***lstat64*** returns information about the link, while ***stat64*** returns information about the file the link references.

The ***fstat64*** function obtains information about an open file known by the file descriptor *fildes*, obtained from a successful ***open64***, ***creat64***, ***dup***, ***fcntl***, or ***pipe*** function.

buf is a pointer to a **stat64** structure into which information is placed concerning the file.

The contents of the structure pointed to by *buf* include the following members:

```
dev_t        st_dev;          /* ID of device containing */
                              /* a directory entry for this file */
long         st_pad1[3];      /* reserve for dev expansion */

                              /* sysid definition */
ino64_t      st_ino;          /* Inode number */

mode_t       st_mode;         /* File mode (see mknod) */

dev_t        st_rdev;         /* ID of device */
                              /* This entry is defined only for */
long         st_pad2[2]       /* char special or block special files */
nlink_t      st_nlink;        /* Number of links */
uid_t        st_uid;          /* User ID of the file's owner */
gid_t        st_gid;          /* Group ID of the file's group */
off64_t      st_size;         /* File size in bytes */
time_t       st_atime;        /* Time of last access */
time_t       st_mtime;        /* Time of last data modification */
time_t       st_ctime;        /* Time of last file status change */
                              /* Times measured in seconds since */
                              /* 00:00:00 UTC, Jan. 1, 1970 */
long         st_blksize;      /* Preferred I/O block size */
blkcnt64_t   st_blocks;       /* large file support */

char         st_fstype[16];   /* file system type name */

long         st_pad4[8];      /* expansion area */
```

Descriptions of structure members are as follows:

st_mode The mode of the file as described in ***mknod***. In addition to the modes described in ***mknod***, the mode of a file may also be **S_IFLNK** if the file is a symbolic link. **S_IFLNK** may only be returned by ***lstat64***.

st_ino This field uniquely identifies the file in a given file system. The pair **st_ino** and **st_dev** uniquely identifies regular files.

st_dev This field uniquely identifies the file system that contains the file. Its value may be used as input to the ***ustat*** function to determine more information about this file system. No other meaning is associated with this value.

st_rdev This field should be used only by administrative commands. It is valid only for block special or character special files and only has meaning on the system where the file was configured.

st_nlink This field should be used only by administrative commands.

st_uid The user ID of the file's owner.

st_gid The group ID of the file's group.

st_size For large files, this is the address of the end of the file. For block special or character special, this is not defined. See also ***pipe***.

st_atime Time when file data was last accessed. Changed by the following functions: ***creat64***, ***mknod***, ***pipe***, ***utime***, and ***read***.

st_mtime Time when data was last modified. Changed by the following functions: ***creat64***, ***mknod***, ***pipe***, ***utime***, and ***write***.

st_ctime Time when file status was last changed. Changed by the following functions: ***chmod***, ***chown***, ***creat64***, ***link***, ***mknod***, ***pipe***, ***unlink***, ***utime***, and ***write***.

st_blksize A hint as to the "best" unit size for I/O operations. This field is not defined for block special or character special files.

st_blocks The total number of physical blocks of size 512 bytes actually allocated on disk. This field is not defined for block special or character special files.

st_fstype An array size 16 of buffer to store the file system type name.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and errno is set to indicate the error.

ERRORS

The ***stat64*** and ***lstat64*** functions will fail if one or more of the following are true:

EACCES Search permission is denied for a component of the path prefix.

EFAULT *buf* or *path* points to an illegal address.

EINTR A signal was caught during the ***stat64*** or ***lstat64*** function.

ELOOP Too many symbolic links were encountered in translating *path*.

EMULTIHOP Components of *path* require hopping to multiple remote machines and the file system does not allow it.

ENAMETOOLONG The length of the *path* argument exceeds **{PATH_MAX}**, or the length of a *path* component exceeds **{NAME_MAX}** while **{_POSIX_NO_TRUNC}** is in effect.

ENOENT The named file does not exist or is the null pathname.

ENOLINK *path* points to a remote machine and the link to that machine is no longer active.

ENOTDIR A component of the path prefix is not a directory.

EOVERFLOW A component is too large to store in the structure pointed to by *buf*.

The ***fstat64*** function will fail if one or more of the following are true:

EBADF *fildes* is not a valid open file descriptor.

EFAULT *buf* points to an illegal address.

EINTR A signal was caught during the ***fstat64*** function.

ENOLINK *fildes* points to a remote machine and the link to that machine is no longer active.

EOVERFLOW A component is too large to store in the structure pointed to by *buf*.

SEE ALSO

chmod**, **chown**, **creat64, link**, **mknod**, **pipe**, **read, time**, **unlink**, **utime**, **write**, **fattach

fstatvfs64 (libc), statvfs64 (libc)

NAME

statvfs64, fstatvfs64 – get file system information

SYNOPSIS

#include <sys/types.h>

#include <sys/statvfs.h>

int statvfs64 (const char *path, struct statvfs64 * buf);

int fstatvfs64 (int fildes, struct statvfs64 * buf);

DESCRIPTION

The ***statvfs64*** function returns a generic superblock describing a file system; it can be used to acquire information about mounted file systems. *buf* is a pointer to a structure (described below) that is filled by the function.

path should name a file that resides on that file system. The file system type is known to the operating system. Read, write, or execute permission for the named file is not required, but all directories listed in the path name leading to the file must be searchable.

The ***statvfs64*** structure pointed to by *buf* includes the following members:

u_long	*f_bsize;*	*/* preferred file system block size */*
u_long	*f_frsize;*	*/* fundamental filesystem block (size if supported) */*
fsblkcnt64_t	*f_blocks;*	*/* total # of blocks on file system in units of f_frsize */*
fsblkcnt64_t	*f_bfree;*	*/* total # of free blocks */*
fsblkcnt64_t	*f_bavail;*	*/* # of free blocks avail to non-super-user */*
fsfilcnt64_t	*f_files;*	*/* total # of file nodes (inodes) */*
fsfilcnt64_t	*f_ffree;*	*/* total # of free file nodes */*
fsfilcnt64_t	*f_favail;*	*/* # of inodes avail to non-super-user*/*
u_long	*f_fsid;*	*/* file system id (dev for now) */*
char	*f_basetype[16];*	*/* target fs type name, null-terminated */*
u_long	*f_flag;*	*/* bit mask of flags */*
u_long	*f_namemax;*	*/* maximum file name length */*
char	*f_fstr[32];*	*/* file system specific string */*
u_long	*f_filler[16];*	*/* reserved for future expansion */*

f_basetype contains a null-terminated file system type name of the mounted target.

The following flags can be returned in the *f_flag* field:

ST_RDONLY	*0x01*	*/* read-only file system */*
ST_NOSUID	*0x02*	*/* does not support setuid/setgid semantics */*
ST_NOTRUNC	*0x04*	*/* does not truncate file names longer than {***NAME_MAX***}*/*

The ***fstatvfs64*** function is similar to ***statvfs64***, except that the file named by *path* in ***statvfs64*** is instead identified by an open file descriptor *fildes* obtained from a successful ***open64, creat64, dup***, ***fcntl***, or ***pipe*** function.

RETURN VALUES

Upon successful completion **0** is returned. Otherwise, **–1** is returned and **errno** is set to indicate the error.

ERRORS

The ***statvfs64*** and ***fstatvfs64*** function will fail if one or more:

EOVERFLOW One of the values to be returned cannot be represented correctly in the structure pointed to by *buf*.

The ***statvfs64*** function will fail if one or more:

EACCES Search permission is denied on a component of the path prefix.

EFAULT *path* or *buf* points to an illegal address.

EINTRA signal was caught during ***statvfs64*** execution.

EIO An I/O error occurred while reading the file system.

ELOOP Too many symbolic links were encountered in translating *path*.

EMULTIHOP Components of *path* require hopping to multiple remote machines and file system type does not allow it.

ENAMETOOLONG The length of a *path* component exceeds **{NAME_MAX}** characters, or the length of *path* The exceeds **{PATH_MAX}** characters.

ENOENT Either a component of the path prefix or the file referred to by *path* does not exist.

ENOLINK *path* points to a remote machine and the link to that machine is no longer active.

ENOTDIR A component of the path prefix of *path* is not a directory.

The ***fstatvfs64*** function will fail if one or more:

EBADF *fildes* is not an open file descriptor.

EFAULT *buf* points to an illegal address.

EINTR A signal was caught during ***fstatvfs64*** execution.

EIO An I/O error occurred while reading the file system.

SEE ALSO

chmod, ***chown***, ***creat64, dup***, ***fcntl***, ***link***, ***mknod***, ***open64***, ***pipe***, ***read***, ***time***, ***unlink***, ***utime***, ***write***

ftello64 (libc)

NAME

ftello64 – return a file offset in a stream

SYNOPSIS

#include <stdio.h>

off64_t ftello64 (FILE * stream);

DESCRIPTION

The ***ftello64*** function obtains the current value of the file-position indicator for the stream pointed to by *stream*.

RETURN VALUES

Upon successful completion, ***ftello64*** returns the current value of the file-position indicator for the stream measured in bytes from the beginning of the file.

Otherwise, it returns –1 and sets *errno* to indicate the error.

ERRORS

The ***ftello64*** functions will fail if:

EBADF	The file descriptor underlying *stream* is not an open file descriptor.
ESPIPE	The file descriptor underlying *stream* is associated with a pipe or FIFO.

The ***ftello64*** function will fail if:

EOVERFLOW	The current file offset cannot be represented correctly in an object of type **off64_t**.

SEE ALSO

lseek64, fopen64, fseeko64

ftruncate64 (libc), truncate64 (libc)

NAME

truncate64, ftruncate64 – set a file to a specified length in large file environment

SYNOPSIS

#include <unistd.h>

*int truncate64 (const char *path, off64_t length);*

int ftruncate64(int fildes, off64_t length);

DESCRIPTION

The file whose name is given by path or referenced by the descriptor fildes has its size set to length bytes.

If the file was previously longer than length, bytes past length will no longer be accessible. If it was shorter, bytes from the EOF before the call to the EOF after the call will be read in as zeros. The effective user ID of the process must have write permission for the file, and for ***ftruncate64*** the file must be open for writing.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and errno is set to indicate the error.

ERRORS

truncate64 fails if one or more of the following are true:

EACCES	Search permission is denied on a component of the path prefix.
EACCES	Write permission is denied for the file referred to by *path*.
EFAULT	*path* points outside the process's allocated address space.
EINTR	A signal was caught during execution of the ***truncate64*** routine.
EINVAL	path is not an ordinary file.
EIO	An I/O error occurred while reading from or writing to the file system.
EISDIR	The file referred to by path is a directory.
ELOOP	Too many symbolic links were encountered in translating path.
EMFILE	The maximum number of file descriptors available to the process has been reached.
EMULTIHOP	Components of path require hopping to multiple remote machines and file system type does not allow it.
ENAMETOOLONG	The length of a path component exceeds {**NAME_MAX**} characters, or the length of path exceeds {**PATH_MAX**} characters.
ENFILE	Could not allocate any more space for the system file table.

ENOENT Either a component of the path prefix or the file referred to by path does not exist.

ENOLINK path points to a remote machine and the link to that machine is no longer active.

ENOTDIR A component of the path prefix of path is not a directory.

EROFS The file referred to by path resides on a read-only file system.

ftruncate64 fails if one or more of the following are true:

EAGAIN The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file (see ***chmod***).

EBADF fildes is not a file descriptor open for writing.

EINTR A signal was caught during execution of the ***ftruncate64*** routine.

EIO An I/O error occurred while reading from or writing to the file system.

ENOLINK *fildes* points to a remote machine and the link to that machine is no longer active.

EINVAL *fildes* does not correspond to an ordinary file.

SEE ALSO

chmod, fcntl, open64

ftw64 (libc), nftw64 (libc)

NAME

ftw64, nftw64 – walk a file tree

SYNOPSIS

#include <ftw.h>

int ftw64 (const char *path, int (*fn) (const char*, const struct stat64*,int),int depth);

int nftw64 (const char *path, int (*fn) (const char*, const struc t stat64*, int, struct FTW *), int depth, int flags);

DESCRIPTION

The ***ftw64*** function recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, ***ftw64*** calls the user-defined function *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a *stat64* structure (see ***stat64***) containing information about the object, and an integer. Possible values of the integer, defined in the **<***ftw.h***>** header, are:

FTW_F The object is a file.

FTW_D The object is a directory.

FTW_DNR The object is a directory that cannot be read. Descendants of the directory will not be processed.

FTW_NS ***stat64*** failed on the object because of lack of appropriate permission or the object is a symbolic link that points to a non-existent file. The stat buffer passed to *fn* is undefined. ***stat64*** failure other than lack of appropriate permission (**EACCES**) is considered an error and ***nftw64*** will return -1.

ftw64 visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a nonzero value, or some error is detected within ***ftw64*** (such as an I/O error). If the tree is exhausted, ***ftw64*** returns zero. If *fn* returns a nonzero value, ***ftw64*** stops its tree traversal and returns whatever value was returned by *fn*.

The function ***nftw64*** is similar to ***ftw64*** except that it takes an additional argument, *flags*. The *flags* field is used to specify:

FTW_PHYS Physical walk, does not follow symbolic links. Otherwise, ***nftw64*** will follow links but will not walk down any path that crosses itself.

FTW_MOUNT The walk will not cross a mount point.
FTW_DEPTH All subdirectories will be visited before the directory itself.

FTW_CHDIR The walk will change to each directory before reading it.

The function ***nftw64*** calls *fn* with four arguments at each file and directory. The first argument is the pathname of the object, the second is a pointer to the ***stat*** buffer, the third is an integer giving additional information, and the fourth is a struct *FTW* that contains the following members:

int base;

int level;

base is the offset into the pathname of the base name of the object. *level* indicates the depth relative to the rest of the walk, where the root level is zero.

The values of the third argument are as follows:

FTW_F The object is a file.

FTW_D The object is a directory.

FTW_DP The object is a directory and subdirectories have been visited.

FTW_SL The object is a symbolic link.

FTW_SLN The object is a symbolic link that points to a non-existent file.

FTW_DNR The object is a directory that cannot be read. *fn* will not be called for any of its descendants.

FTW_NS ***stat64*** failed on the object because of lack of appropriate permission. The stat buffer passed to *fn* is undefined. ***stat64*** failure other than lack of appropriate permission. **EACCES** is considered an error and ***nftw64*** will return –1.

Both ***ftw64*** and ***nftw64*** use one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors so used. If *depth* is zero or negative, the effect is the same as if it were 1. *depth* must not be greater than the number of file descriptors currently available for use. ***ftw64*** will run faster if *depth* is at least as large as the number of levels in the tree. When ***ftw64*** and ***nftw64*** return, they close any file descriptors they have opened; they do not close any file descriptors that may have been opened by *fn*.

RETURN VALUES

If successful, ***ftw64*** and ***nftw64*** return **0**. If either function detects an error other than **EACCES**, it returns –1, and sets the error type in **errno**.

NOTES

Because ***ftw64*** is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.

ftw64 uses ***malloc*** to allocate dynamic storage during its operation. If ***ftw64*** is forcibly terminated, such as by ***longjmp*** being executed by *fn* or an interrupt routine, ***ftw64*** will not have a chance to free that storage, so it will remain permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a nonzero value at its next invocation.

ftw64 is safe in multi-thread applications. ***nftw64*** is safe in multi-thread applications when the **FTW_CHDIR** flag is not set.

getdents64 (libc)

NAME

getdents64 – read directory entries and put in a file system independent format

SYNOPSIS

#include <sys/dirent.h>

int getdents64 (int fildes, struct dirent64 *buf, size_t nbyte);

DESCRIPTION

The ***getdents64*** function attempts to read *nbyte* bytes from the directory associated with the file descriptor *fildes* and to format them as file system independent directory entries in the buffer pointed to by *buf*. Since the file system independent directory entries are of variable length, in most cases the actual number of bytes returned will be strictly less than *nbyte.* See *dirent64* to calculate the number of bytes.

The file system independent directory entry is specified by the *dirent64* structure. For a description of this see *dirent64*.

On devices capable of seeking, ***getdents64*** starts at a position in the file given by the file pointer associated with *fildes*. Upon return from ***getdents64***, the file pointer is incremented to point to the next directory entry.

RETURN VALUES

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. A value of 0 indicates the end of the directory has been reached. If the function failed, **–1** is returned and **errno** is set to indicate the error.

ERRORS

The ***getdents64*** function will fail if one or more of the following are true:

EBADF	*fildes* is not a valid file descriptor open for reading.
EFAULT	*buf* points to an illegal address.
EINVAL	*nbyte* is not large enough for one directory entry.
EIO	An I/O error occurred while accessing the file system.
ENOENT	The current file pointer for the directory is not located at a valid entry.
ENOLINK	*fildes* points to a remote machine and the link to that machine is no longer active.
ENOTDIR	*fildes* is not a directory.
EOVERFLOW	The value of the *dirent64* structure member *d_ino* or *d_off* cannot be represented in an *ino64_t* or *off64_t*.

getrlimit64 (libc), setrlimit64 (libc)

NAME

getrlimit64, setrlimit64 **–** control maximum system resource consumption

SYNOPSIS

#include <sys/resource.h>
int getrlimit64 (int resource, struct rlimit64 *rlp);

int setrlimit64 (int resource, const struct rlimit64 *rlp);

DESCRIPTION

Limits on the consumption of a variety of system resources by a process and each process it creates may be obtained with the ***getrlimit64*** and set with ***setrlimit64*** functions.

Each call to either ***getrlimit64*** or ***setrlimit64*** identifies a specific resource to be operated upon as well as a resource limit. A resource limit is a pair of values: one specifying the current (soft) limit, the other a maximum (hard) limit. Soft limits may be changed by a process to any value that is less than or equal to the hard limit. A process may (irreversibly) lower its hard limit to any value that is greater than or equal to the soft limit. Only a process with an effective user ID of super-user can raise a hard limit. Both hard and soft limits can be changed in a single call to ***setrlimit64*** subject to the constraints described above. Limits may have an “infinite” value of **RLIM_INFINITY**. *rlp* is a pointer to struct *rlimit64* that includes the following members:

rlim64_t	***rlim_cur;***	***/* current (soft) limit */***
rlim64_t	***rlim_max;***	***/* hard limit */***

rlim64_t is an arithmetic data type to which objects of type *size_t* and *off64_t* can be cast without loss of information.

The possible resources, their descriptions, and the actions taken when the current limit is exceeded are summarized in the table below:

RLIMIT_CORE	The maximum size of a core file in bytes that may be created by a process. A limit of **0** will prevent the creation of a core file. The writing of a core file will terminate at this size.
RLIMIT_CPU	The maximum amount of CPU time in seconds used by a process. This is a soft limit only. **SIGXCPU** is sent to the process. If the process is holding or ignoring **SIGXCPU**, the behavior is scheduling class defined.
RLIMIT_DATA	The maximum size of a process’s heap in bytes. **brk** will fail with **errno** set to **ENOMEM**.
RLIMIT_FSIZE	The maximum size of a file in bytes that may be created by a process. A limit of **0** will prevent the creation of a file.
SIGXFSZ	is sent to the process. If the process is holding or ignoring **SIGXFSZ**, continued attempts to increase the size of a file beyond the limit will fail with **errno** set to **EFBIG**.
RLIMIT_NOFILE	One more than the maximum value that the system may assign to a newly

created descriptor. This limit constrains the number of file descriptors that a process may create.

RLIMIT_STACK The maximum size of a process's stack in bytes. The system will not automatically grow the stack beyond this limit.

Within a process, ***setrlimit64*** will increase the limit on the size of your stack, but will not move current memory segments to allow for that growth. To guarantee that the process stack can grow to the limit, the limit must be altered prior to the execution of the process in which the new stack size is to be used.

SIGSEGV is sent to the process. If the process is holding or ignoring **SIGSEGV**, or is catching **SIGSEGV** and has not made arrangements to use an alternate stack (see ***sigaltstack***), the disposition of **SIGSEGV** will be set to **SIG_DFL** before it is sent.

RLIMIT_VMEM The maximum size of a process's mapped address space in bytes. ***brk*** and ***setrlimit64()*** and ***getrlimit64()*** functions will fail with **errno** set to **ENOMEM**. In addition, the automatic stack growth will fail with the effects outlined above.

RLIMIT_AS This is the maximum size of a process' total available memory, in bytes. If this limit is exceeded, the ***brk***, ***malloc***, ***mmap64*** and ***sbrk*** functions will fail with **errno** set to **ENOMEM**. In addition, the automatic stack growth will fail with the effects outlined above.

Because limit information is stored in the per-process information, the shell built-in *ulimit* command must directly execute this system call if it is to affect all future processes created by the shell.

The value of the current limit of the following resources affect these implementation defined parameters:

Limit	Implementation Defined Constant
RLIMIT_FSIZE	**FCHR_MAX**
RLIMIT_NOFILE	**OPEN_MAX**

When using the ***getrlimit64*** function, if a resource limit can be represented correctly in an object of type *rlim64_t*, then its representation is returned; otherwise, if the value of the resource limit is equal to that of the corresponding saved hard limit, the value returned is **RLIM_SAVED_MAX**; otherwise the value returned is **RLIM_SAVED_CUR**.

When using the ***setrlimit64*** function, if the requested new limit is **RLIM_INFINITY**, the new limit will be no limit; otherwise if the requested new limit is **RLIM_SAVED_MAX**, the new limit will be the corresponding saved hard limit; otherwise, if the requested new limit is **RLIM_SAVED_CUR**, the new limit will be the corresponding saved soft limit; otherwise, the new limit will be the requested value. In addition, if the corresponding saved limit can be represented correctly in an object of type *rlim64_t*, then it will be overwritten with the new limit.

The result of setting a limit to **RLIM_SAVED_MAX** or **RLIM_SAVED_CUR** is unspecified unless a previous call to ***getrlimit64*** returned that value as the soft or hard limit for the corresponding resource limit.

A limit whose value is greater than **RLIM_INFINITY** is permitted.

The ***exec*** family of functions also cause resource limits to be saved. See ***exec***.

RETURN VALUES

Upon successful completion, ***getrlimit64*** and ***setrlimit64*** return **0**. Otherwise, these functions return **–1** and set **errno** to indicate the error.

ERRORS

The ***getrlimit64*** and ***setrlimit64*** functions will fail if:

EFAULT *rlp* points to an illegal address.

EINVAL An invalid *resource* was specified; or in a ***setrlimit64*** call, the new **rlim_cur** exceeds the new **rlim_max**.

EPERM The limit specified to ***setrlimit64*** would have raised the maximum limit value, and the effective user of the calling process is not super-user.

The ***setrlimit64*** function may fail if:

EINVAL The limit specified cannot be lowered because current usage is already higher than the limit.

SEE ALSO

brk, exec, fork, open64, sigaltstack, ulimit, getdtablesize, malloc, signal, sysconf, signal

lockf64 (libc)

NAME

lockf64 – record locking on files

SYNOPSIS

#include <unistd.h>

int lockf64(int fildes, int function, off64_t size);

DESCRIPTION

The ***lockf64*** function allows sections of a file to be locked; advisory or mandatory write locks depending on the mode bits of the file (see ***chmod***). Locking calls from other processes that attempt to lock the locked file section will either return an error value or be put to sleep until the resource becomes unlocked. All the locks for a process are removed when the process terminates. See ***fcntl*** for more information about record locking.

The *fildes* argument is an open file descriptor. The file descriptor must have **O_WRONLY** or **O_RDWR** permission in order to establish locks with this function call. *function* is a control value that specifies the action to be taken. The permissible values for *function* are defined in **<***unistd.h***>** as follows:

```
#define  F_ULOCK    0    /* unlock previously locked section */
#define  F_LOCK     1    /* lock section for exclusive use */
#define  F_TLOCK    2    /* test & lock section for exclusive use */
#define  F_TEST     3    /* test section for other locks */
```

All other values of *function* are reserved for future extensions and will result in an error return if not implemented.

F_TEST is used to detect if a lock by another process is present on the specified section. **F_LOCK** and **F_TLOCK** both lock a section of a file if the section is available. **F_ULOCK** removes locks from a section of the file.

The *size* argument is the number of contiguous bytes to be locked or unlocked. The resource to be locked or unlocked starts at the current offset in the file and extends forward for a positive size and backward for a negative size (the preceding bytes up to but not including the current offset). If *size* is zero, the section from the current offset through the largest file offset is locked (that is, from the current offset through the present or any future end-of-file). An area need not be allocated to the file in order to be locked as such locks may exist past the end-of-file.

The sections locked with **F_LOCK** or **F_TLOCK** may, in whole or in part, contain or be contained by previously locked section for the same process. Locked sections will be unlocked starting at the point of the offset through *size* bytes or to the end of file if *size* is (**off64_t**) 0. When this situation occurs, or if this situation occurs in adjacent sections, the sections are combined into a single section. If the request requires that a new element be added to the table of active locks and this table is already full, an error is returned, and the new section is not locked.

F_LOCK and **F_TLOCK** requests differ only by the action taken if the resource is not available. **F_LOCK** will cause the calling process to sleep until the resource is available. **F_TLOCK** will cause the function to return a –1 and set **errno** to **EAGAIN** if the section is already locked by another process.

File locks are released on first close by the locking process of any file descriptor for the file.

F_ULOCK requests may, in whole or in part, release one or more locked sections controlled by the process. When sections are not fully released, the remaining sections are still locked by the process. Releasing the center section of a locked section requires an additional element in the table of active locks. If this table is full, an **errno** is set to **EDEADLK** and the requested section is not released.

An **F_ULOCK** request in which *size* is nonzero and the offset of the last byte of the requested section is the maximum value for an object of type **off64_t**, when the process has an existing lock in which *size* is 0 and which includes the last byte of the requested section, will be treated as a request to unlock from the start of the requested section with a size equal to 0. Otherwise, an **F_ULOCK** request will attempt to unlock only the requested section.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by requesting another process's locked resource. Thus calls to ***lockf64*** or ***fcntl*** scan for a deadlock prior to sleeping on a locked resource. An error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. The ***alarm*** function may be used to provide a timeout facility in applications that require this facility.

RETURN VALUES

Upon successful completion, **0** is returned. Otherwise, **–1** is returned and **errno** is set to indicate the error.

ERRORS

The ***lockf64*** function will fail if:

EBADF The *fildes* argument is not a valid open file descriptor; or *function* is **F_LOCK** or **F_TLOCK** and *fildes* is not a valid file descriptor open for writing.

EACCES or **EAGAIN** The *function* argument is **F_TLOCK** or **F_TEST** and the section is already locked by another process.

EDEADLK The *function* argument is **F_LOCK** and a deadlock is detected.

EINTR A signal was caught during execution of the function.

ECOMM The *fildes* argument is on a remote machine and the link to that machine is no longer active.

EINVAL The *function* argument is not one of **F_LOCK**, **F_TLOCK**, **F_TEST**, or **F_ULOCK**; or *size* plus the current file offset is less than 0.

EOVERFLOW The offset of the first, or if *size* is not 0 then the last, byte in the requested section cannot be represented correctly in an object of type **off64_t**.

The ***lockf64*** function may fail if:

EAGAIN The *function* argument is **F_LOCK** or **F_TLOCK** and the file is mapped with ***mmap64***.

EDEADLK or **ENOLCK** The *function* argument is **F_LOCK**, **F_TLOCK**, or **F_ULOCK**, and the request would cause the number of locks to exceed a system-imposed limit.

EOPNOTSUPP or **EINVAL** The locking of files of the type indicated by the *fildes* argument is not supported.

USAGE

Record-locking should not be used in combination with the ***fopen64***, ***fread***, ***fwrite*** and other **stdio** functions. Instead, the more primitive, non-buffered functions (such as ***open64***) should be used. Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data which is/was locked. The ***stdio*** functions are the most common source of unexpected buffering.

The ***alarm*** function may be used to provide a timeout facility in applications requiring it.

SEE ALSO

alarm, chmod, close, creat64, fcntl, mmap64, open64, read, write

lseek64 (libc)

NAME

lseek64 – move read/write file pointer

SYNOPSIS

#include <sys/types.h>
#include <unistd.h>
off64_t lseek64 (int fildes, off64_t offset, int whence);

DESCRIPTION

The ***lseek64*** function sets the file pointer associated with the open file descriptor specified by *fildes* as follows:

- If *whence* is **SEEK_SET**, the pointer is set to *offset* bytes.
- If *whence* is **SEEK_CUR**, the pointer is set to its current location plus *offset*.
- If *whence* is **SEEK_END**, the pointer is set to the size of the file plus *offset*.

On success, ***lseek64*** returns the resulting pointer location, as measured in bytes from the beginning of the file. Note that if *fildes* is a remote file descriptor and *offset* is negative, ***lseek64*** returns the file pointer even if it is negative.

The ***lseek64*** function allows the file pointer to be set beyond the existing data in the file. If data are later written at this point, subsequent reads in the gap between the previous end of data and the newly written data will return bytes of value 0 until data are written into the gap.

RETURN VALUES

Upon successful completion, the resulting file pointer is returned. Remote file descriptors are the only ones that allow negative file pointers. Otherwise, **–1** is returned and **errno** is set to indicate the error.

ERRORS

The ***lseek64*** function fails and the file pointer remains unchanged if one or more of the following are true:

EBADF	The *fildes* argument is not an open file descriptor.
EINVAL	The *whence* argument is not **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**.
EINVAL	The *fildes* argument is not a remote file descriptor, and the resulting file pointer would be negative.
EOVERFLOW	The resulting file offset would be a value which cannot be represented correctly in an object of type **off64_t** for regular files.
ESPIPE	The *fildes* argument is associated with a pipe, a FIFO, or a socket.

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

SEE ALSO

creat64, open64, read, write

NOTES

In multithreaded programs, using ***lseek64*** in conjunction with a ***read*** or ***write*** on a file descriptor shared amongst more than one thread is not an atomic operation. To ensure atomicity, use ***pread64*** or ***pwrite*64.**

mmap64 (libc)

NAME

mmap64 – map pages of memory

SYNOPSIS

#include <sys/mman.h>

void*mmap64(void *addr, size_t len, int prot, int flags, int fildes, off64_t off);

DESCRIPTION

The ***mmap64*** function establishes a mapping between a process's address space and a virtual memory object. The format of the call is as follows:

pa = ***mmap64(****addr*, *len*, *prot*, *flags*, *fildes*, *off***);**

at an address *pa* for *len* bytes to the memory object represented by the file descriptor *fildes* at offset *off* for *len* bytes. The value of *pa* is an implementation-dependent function of the parameter *addr* and values of *flags*, further described below. A successful ***mmap64*** call returns *pa* as its result. The address ranges covered by [*pa, pa* + *len*) and [*off, off* + *len*) must be legitimate for the possible (not necessarily current) address space of a process and the object in question, respectively.

The ***mmap64*** function allows [*pa, pa* + *len*) to extend beyond the end of the object, both at the time of the ***mmap64*** and while the mapping persists, such as when the file was created just before the ***mmap64*** and has no contents, or if the file is truncated. Any reference to addresses beyond the end of the object, however, will result in the delivery of a "**SIGBUS**" signal. In other words, ***mmap64*** cannot be used to implicitly extend the length of files.

The mapping established by ***mmap64*** replaces any previous mappings for the process's pages in the range [*pa, pa* + *len*).

Mappings established from *fildes* are not removed upon a ***close*** of that descriptor. Use ***munmap*** to remove a mapping.

The parameter *prot* determines whether read, write, execute, or some combination of accesses are permitted to the pages being mapped. The protection options are defined in **<***sys/mman.h***>** as:

PROT_READ	Page can be read.
PROT_WRITE	Page can be written.
PROT_EXEC	Page can be executed.
PROT_NONE	Page can not be accessed.

Not all implementations literally provide all possible combinations. **PROT_WRITE** is often implemented as **PROT_READ | PROT_WRITE** and **PROT_EXEC** as **PROT_READ | PROT_EXEC**. However, no implementation will permit a write to succeed where **PROT_WRITE** has not been set. The behavior of **PROT_WRITE** can be influenced by setting **MAP_PRIVATE** in the *flags* parameter, described below.

The parameter *flags* provides other information about the handling of the mapped pages. The options are defined in **<***sys/mman.h***>** as:

MAP_SHARED	Share changes.

MAP_PRIVATE	Changes are private.
MAP_FIXED	Interpret addr exactly.
MAP_NORESERVE	Don't reserve swap space.

MAP_SHARED and **MAP_PRIVATE** describe the disposition of write references to the memory object. If **MAP_SHARED** is specified, write references will change the memory object. If **MAP_PRIVATE** is specified, the initial write reference will create a private copy of the memory object page and redirect the mapping to the copy. Either **MAP_SHARED** or **MAP_PRIVATE** must be specified, but not both. The mapping type is retained across a ***fork***.

Note that the private copy is not created until the first write; until then, other users who have the object mapped **MAP_SHARED** can change the object.

MAP_FIXED informs the system that the value of *pa* must be *addr*, exactly. The use of **MAP_FIXED** is discouraged, as it may prevent an implementation from making the most effective use of system resources.

When **MAP_FIXED** is not set, the system uses *addr* in an implementation-defined manner to arrive at *pa*. The *pa* so chosen will be an area of the address space which the system deems suitable for a mapping of *len* bytes to the specified object. All implementations interpret an *addr* value of zero as granting the system complete freedom in selecting *pa*, subject to constraints described below. A nonzero value of *addr* is taken to be a suggestion of a process address near which the mapping should be placed. When the system selects a value for *pa*, it will never place a mapping at address **0**, nor will it replace any extant mapping, nor map into areas considered part of the potential data or stack segments.

MAP_NORESERVE specifies that no swap space be reserved for a mapping. Without this flag, the creation of a writable **MAP_PRIVATE** mapping reserves swap space equal to the size of the mapping; when the mapping is written into, the reserved space is employed to hold private copies of the data. A write into a **MAP_NORESERVE** mapping produces results which depend on the current availability of swap space in the system. If space is available, the write succeeds and a private copy of the written page is created; if space is not available, the write fails and a **SIGBUS** signal is delivered to the writing process. **MAP_NORESERVE** mappings are inherited across ***fork***; at the time of the **fork** swap space is reserved in the child for all private pages that currently exist in the parent; thereafter the child's mapping behaves as described above.

The parameter *off* is constrained to be aligned and sized according to the value returned by ***sysconf***. When **MAP_FIXED** is specified, the parameter *addr* must also meet these constraints. The system performs mapping operations over whole pages. Thus, while the parameter *len* need not meet a size or alignment constraint, the system will include, in any mapping operation, any partial page specified by the range [*pa*, *pa* + *len*).

The system will always zero-fill any partial page at the end of an object. Further, the system will never write out any modified portions of the last page of an object which are beyond its end. References to whole pages following the end of an object will result in the delivery of a **SIGBUS** signal. **SIGBUS** signals may also be delivered on various file system conditions, including quota exceeded errors.

If the process calls ***mlockall*** with the **MCL_FUTURE** flag, the pages mapped by all future calls to ***mmap64*** will be locked in memory. In this case, if not enough memory could be locked, ***mmap64*** fails and sets **errno** to **EAGAIN**.

RETURN VALUES

On success, ***mmap64*** returns the address at which the mapping was placed (*pa*). On failure it returns **MAP_FAILED** and sets **errno** to indicate an error.

ERRORS

The ***mmap64*** function will fail if:

EACCES — *fildes* is not open for read, regardless of the protection specified, or *fildes* is not open for write and **PROT_WRITE** was specified for a **MAP_SHARED** type mapping.

EAGAIN — The mapping could not be locked in memory. There was insufficient room to reserve swap space for the mapping. The file to be mapped is already locked using advisory or mandatory record locking. See ***fcntl***.

EBADF — *fildes* is not open.

EINVAL — The arguments *addr* (if **MAP_FIXED** was specified) or *off* are not multiples of the page size as returned by ***sysconf***.

The field in *flags* is invalid (neither **MAP_PRIVATE** or **MAP_SHARED**).

The argument *len* has a value less than or equal to **0**.

EMFILE — The number of mapped regions would exceed an implementation-dependent limit (per process or per system).

ENODEV — *fildes* refers to an object for which ***mmap64*** is meaningless, such as a terminal.

ENOMEM — **MAP_FIXED** was specified and the range [*addr*, *addr* + *len*) exceeds that allowed for the address space of a process. **MAP_FIXED** was "not" specified and there is insufficient room in the address space to effect the mapping.

The composite size of *len* plus the lengths of all previous **mmap**pings exceeds **RLIMIT_VMEM** (see ***getrlimit64***).

ENXIO — The range [*off*, *off* + *len*) is illegal for **mmap**ping to this device.

EOVERFLOW — The file is a regular file and the value of *off* plus *len* exceeds the offset maximum establish in the open file description associated with *fildes*.

SEE ALSO

close, ***exec***, ***fcntl***, ***fork***, ***getrlimit64***, ***mprotect***, ***munmap***, ***shmat***, ***lockf64***, ***mlockall***, ***msync***, ***plock***, ***sysconf***

open64 (libc, libthread)

NAME

open64 **–** open a large file

SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open64(const char *path, int oflag, /*mode_t mode*/ ...);

DESCRIPTION

The ***open64*** function establishes the connection between a file and a file descriptor. It creates an open file description that refers to a file and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to refer to that file. The *path* argument points to a pathname naming the file.

The ***open64*** function will return a file descriptor for the named file that is the lowest file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor does not share it with any other process in the system. The **FD_CLOEXEC** file descriptor flag associated with the new file descriptor will be cleared.

The file offset used to mark the current position within the file is set to the beginning of the file.

The file status flags and file access modes of the open file description will be set according to the value of *oflag*.

Values for *oflag* are constructed by a bitwise-inclusive-OR of flags from the following list, defined in **<***fcntl.h***>**. Applications must specify exactly one of the first three values (file access modes) below in the value of *oflag*:

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing. The result is undefined if this flag is applied to a **FIFO**.

Any combination of the following may be used:

O_APPEND If set, the file offset will be set to the end of the file prior to each write.

O_CREAT If the file exists, this flag has no effect except as noted under **O_EXCL** below. Otherwise, the file is created with the user ID of the file set to the effective user ID of the process. The group ID of the file is set to the effective group IDs of the process, or if the **S_ISGID** bit is set in the directory in which the file is being created, the file's group ID is set to the group ID of its parent directory. If the group ID of the new file does not match the effective group ID or one of the supplementary groups IDs, the **S_ISGID**'s bit is cleared. The access permission bits (see **<***sys/stat.h***>**) of the file mode are set to the value of *mode*, modified as follows (see ***creat64()***): a bitwise-AND is performed on the file-mode bits and the corresponding bits in the complement of the process' file mode creation mask. Thus, all bits in the file mode whose corresponding bit in the file mode creation mask is set are cleared. The save text image after execution bit of the mode is cleared (see ***chmod***).

O_SYNC Write I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion (see ***fcntl*** definition of **O_SYNC**.) When bits other than the file permission bits are set, the effect is unspecified. The *mode* argument does not affect whether the file is open for reading, writing or for both.

O_DSYNC Write I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion.

O_EXCL If **O_CREAT** and **O_EXCL** are set, ***open64*** will fail if the file exists. The check for the existence of the file and the creation of the file if it does not exist will be atomic with respect to other processes executing ***open64*** naming the same filename in the same directory with **O_EXCL** and **O_CREAT** set. If **O_CREAT** is not set, the effect is undefined.

O_LARGEFILE If set, the offset maximum in the open file description will be the largest value that can be represented correctly in an object of type **off64_t**.

O_NOCTTY If set and *path* identifies a terminal device, ***open64*** will not cause the terminal device to become the controlling terminal for the process.

O_NONBLOCK or **O_NDELAY** These flags may affect subsequent reads and writes (see ***read*** and ***write***). If both **O_NDELAY** and **O_NONBLOCK** are set, **O_NONBLOCK** will take precedence.

When opening a FIFO with **O_RDONLY** or **O_WRONLY** set:

If **O_NONBLOCK** or **O_NDELAY** is set:

An ***open64*** for reading only will return without delay. An ***open64*** for writing only will return an error if no process currently has the file open for reading.

If **O_NONBLOCK** and **O_NDELAY** are clear:

An ***open64*** for reading only will block until a process opens the file for writing. An ***open64*** for writing only will block until a process opens the file for reading.

When opening a block special or character special file that supports non-blocking opens:

If **O_NONBLOCK** or **O_NDELAY** is set:

The ***open64*** function will return without blocking for the device to be ready or available. Subsequent behavior of the device is device-specific.

If **O_NONBLOCK** and **O_NDELAY** are clear:

The ***open64*** function will block until the device is ready or available before returning.

Otherwise, the behavior of **O_NONBLOCK** and **O_NDELAY** is unspecified.

O_RSYNC Read I/O operations on the file descriptor complete at the same level of integrity as specified by the **O_DSYNC** and **O_SYNC** flags. If both **O_DSYNC** and **O_RSYNC** are set in *oflag*, all I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion. If both **O_SYNC** and **O_RSYNC** are set in *oflag*, all I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion.

O_SYNC If **O_SYNC** is set on a regular file, writes to that file will cause the process to block until the data is delivered to the underlying hardware.

O_TRUNC If the file exists and is a regular file, and the file is successfully opened **O_RDWR** or **O_WRONLY**, its length is truncated to 0 and the mode and owner are unchanged. It will have no effect on **FIFO** special files or terminal device files. Its effect on other file types is implementation-dependent. The result of using **O_TRUNC** with **O_RDONLY** is undefined.

If **O_CREAT** is set and the file did not previously exist, upon successful completion, **open64** will mark for update the **st_atime**, **st_ctime**, and **st_mtime** fields of the file and the **st_ctime** and **st_mtime** fields of the parent directory.

If **O_TRUNC** is set and the file did previously exist, upon successful completion, **open64** will mark for update the **st_ctime** and **st_mtime** fields of the file.

If *path* refers to a **STREAMS** file, *oflag* may be constructed from **O_NONBLOCK** or **O_NODELAY** OR-ed with either **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. Other flag values are not applicable to STREAMS devices and have no effect on them. The values **O_NONBLOCK** and **O_NODELAY** affect the operation of STREAMS drivers and certain functions (see ***read***, ***getmsg***, ***putmsg***, and **write**) applied to file descriptors associated with **STREAMS** files. For **STREAMS** drivers, the implementation of **O_NONBLOCK** and **O_NODELAY** is device-specific.

When ***open64*** is invoked to open a named stream, and the ***connld*** module has been pushed on the pipe, ***open64()*** blocks until the server process has issued an **I_RECVFD ioctl** (see ***streamio***) to receive the file descriptor.

If *path* names the master side of a pseudo-terminal device, then it is unspecified whether ***open64*** locks the slave side so that it cannot be opened. Portable applications must call ***unlockpt*** before opening the slave side.

If *path* is a symbolic link and **O_CREAT** and **O_EXCL** are set, the link is not followed.

Certain flag values can be set following ***open64*** as described in ***fcntl***.

The largest value that can be represented correctly in an object of type *off64_t* will be established as the offset maximum in the open file description.

RETURN VALUES

Upon successful completion, the function will open the file and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, **–1** is returned and *errno* is set to indicate the error. No files will be created or modified if the function returns **–1**.

ERRORS

The ***open64*** function will fail if:

EACCES Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by *oflag* are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created, or **O_TRUNC** is specified and write permission is denied.

EDQUOT The file does not exist**,** **O_CREAT** is specified, and either the directory where the new file entry is being placed cannot be extended because the user's quota of disk blocks on that file system has been exhausted, or the user's quota of inodes on the file system where the file is being created has been exhausted.

EEXIST	**O_CREAT** and **O_EXCL** are set, and the named file exists.
EINTR	A signal was caught during ***open64***.
EFAULT	*path* points to an illegal address.
EIO	The *path* argument names a STREAMS file and a hang-up or error occurred during the **open64**.
EISDIR	The named file is a directory and *oflag* includes **O_WRONLY** or **O_RDWR**.
ELOOP	Too many symbolic links were encountered in resolving *path*.
EMFILE	**OPEN_MAX** file descriptors are currently open in the calling process.
EMULTIHOP	Components of *path* require hopping to multiple remote machines and the file system does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds **PATH_MAX** or a pathname component is longer than **NAME_MAX**.
ENFILE	The maximum allowable number of files is currently open in the system.
ENOENT	**O_CREAT** is not set and the named file does not exist; or **O_CREAT** is set and either the path prefix does not exist or the *path* argument points to an empty string.
ENOLINK	*path* points to a remote machine, and the link to that machine is no longer active.
ENOSR	The *path* argument names a STREAMS-based file and the system is unable to allocate a STREAM.
ENOSPC	The directory or file system that would contain the new file cannot be expanded, the file does not exist, and **O_CREAT** is specified.
ENOTDIR	A component of the path prefix is not a directory.
ENXIO	**O_NONBLOCK** is set, the named file is a **FIFO**, **O_WRONLY** is set and no process has the file open for reading.
ENXIO	The named file is a character special or block special file, and the device associated with this special file does not exist.
EOPNOTSUPP	An attempt was made to open a path that corresponds to a **AF_UNIX** socket.
EOVERFLOW	The named file is a regular file and either **O_LARGEFILE** is not set and the size of the file cannot be represented correctly in an object of type **off64_t** or **O_LARGEFILE** is set and the size of the file cannot be represented correctly in an object of type **off64_t**.
EROFS	The named file resides on a read-only file system and either **O_WRONLY**, **O_RDWR**, **O_CREAT** (if file does not exist), or **O_TRUNC** is set in the *oflag* argument.

The ***open64*** function may fail if:

EAGAIN	The *path* argument names the slave side of a pseudo-terminal device that is locked.
EINVAL	The value of the *oflag* argument is not valid.
ENAMETOOLONG	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds **PATH_MAX**.

ENOMEM The *path* argument names a STREAMS file and the system is unable to allocate resources.

ETXTBSY The file is a pure procedure (shared text) file that is being executed and *oflag* is **O_WRONLY** or **O_RDWR**.

SEE ALSO

chmod, close, creat64, dup, exec, fcntl, getmsg, getrlimit64, lseek64, putmsg, read, umask, write, unlockpt, fcntl, stat64, connld, streamio

pread64 (libc)

NAME

pread64 – read from file in large file environment

SYNOPSIS

#include <sys/types.h>

#include <sys/uio.h>

#include <unistd.h>

size_t pread64 (int fildes, void *buf, size_t nbyte, off64_t offset);

DESCRIPTION

pread64 performs the same action as ***read***, except that it reads from a given position in the file without changing the file pointer. The first three arguments to ***pread64*** are the same as ***read*** with the addition of a fourth argument offset for the desired position inside the file. An attempt to perform a ***pread64*** on a file that is incapable of seeking results in an error.

RETURN VALUES

On success a non-negative integer is returned indicating the number of bytes actually read. Otherwise, a -1 is returned and errno is set to indicate the error.

ERRORS

The ***pread64*** functions will fail if:

EAGAIN	Mandatory file/record locking was set, **O_NDELAY** or **O_NONBLOCK** was set, and there was a blocking record lock.
EAGAIN	Total amount of system memory available when reading using raw I/O is temporarily insufficient.
EAGAIN	No data is waiting to be read on a file associated with a tty device and **O_NONBLOCK** was set.
EAGAIN	No message is waiting to be read on a stream and **O_NDELAY** or **O_NONBLOCK** was set.
EBADF	*fildes* is not a valid file descriptor open for reading.
EBADMSG	Message waiting to be read on a stream is not a data message.
EDEADLK	The read was going to go to sleep and cause a deadlock to occur.

EFAULT *buf* points to an illegal address.

EINTR A signal was caught during the read operation and no data was transferred.

EINVAL Attempted to read from a stream linked to a multiplexor.

EIO: A physical I/O error has occurred, or the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group of the process is orphaned.

EISDIR *fildes* refers to a directory on a file system type that does not support read operations on directories.

ENOLCK The system record lock table was full, so the ***read*** or ***readv*** could not go to sleep until the blocking record lock was removed.

ENOLINK *fildes* is on a remote machine and the link to that machine is no longer active.

ENXIO The device associated with *fildes* is a block special or character special file and the value of the file pointer is out of range.

In addition, ***pread64*** fails and the file pointer remains unchanged if the following is true:

ESPIPE *fildes* is associated with a pipe or FIFO.

SEE ALSO

chmod, creat64, dup, fcntl, getmsg, ioctl, lseek64, open64, pipe, streamio, termio

pwrite64 (libc)

NAME

pwrite64 – write on a file in large file environment

SYNOPSIS

#include <sys/types.h>

#include <unistd.h>

size_t pwrite64 (int fildes, const void *buf, size_t nbyte, off64_t offset);

DESCRIPTION

pwrite64 performs the same action as ***write*** except that it writes into a given position without changing the file pointer. The first three arguments to ***pwrite64*** are the same as ***write*** with the addition of a fourth argument offset for the desired position inside the file.

RETURN VALUES

On success, ***pwrite64*** returns the number of bytes actually written. Otherwise, it returns -1 and sets errno to indicate the error.

ERRORS

The ***pwrite64*** function fails and the file pointer remains unchanged if one or more of the following are true:

EAGAIN Mandatory file/record locking is set, **O_NDELAY** or **O_NONBLOCK** is set, and there is a blocking record lock; Total amount of system memory available when reading using raw I/O is temporarily insufficient;

An attempt is made to write to a STREAM that can not accept data with the **O_NDELAY** or **O_NONBLOCK** flag set;

If a write to a pipe or **FIFO** of **{PIPE_BUF}** bytes or less is requested and less than *nbytes* of free space is available.

EBADF *fildes* is not a valid file descriptor open for writing.

EDEADLK The write was going to go to sleep and cause a deadlock situation to occur.

EDQUOT The user’s quota of disk blocks on the file system containing the file has been exhausted.

EFAULT *buf* points to an illegal address.

EFBIG An attempt is made to write a file that exceeds the process’ file size limit or the maximum file size (see ***getrlimit64*** and ***ulimit***).

EFBIG An attempt is made to write a file that exceeds the process's file size limit or the maximum file size (see ***getrlimit64*** and ***ulimit***).

EINTR A signal was caught during the write operation and no data was transferred.

EINVAL An attempt is made to write to a stream linked below a multiplexor.

EIO The process is in the background and is attempting to write to its controlling terminal whose "**TOSTOP**" flag is set, or the process is neither ignoring nor blocking "**SIGTTOU**" signals and the process group of the process is orphaned.

ENOLCK Enforced record locking was enabled and **{LOCK_MAX}** regions are already locked in the system

The system record lock table was full and the write could not go to sleep until the blocking record lock was removed.

ENOLINK *fildes* is on a remote machine and the link to that machine is no longer active.

ENOSPC During a write to an ordinary file, there is no free space left on the device.

ENOSR An attempt is made to write to a STREAMS with insufficient STREAMS memory resources available in the system.

ENXIO A hang-up occurred on the stream being written to.

EPIPE and **SIGPIPE** An attempt is made to write to a pipe that is not open for reading by any process (or to a file descriptor created by ***socket***, using type **SOCK_STREAM** that is no longer connected to a peer endpoint). Note: an attempted write of this kind also causes you to receive a **SIGPIPE** signal from the kernel. If you've not made a special provision to catch or ignore this signal, then your process dies.

EPIPE An attempt is made to write to a FIFO that is not open for reading by any process. An attempt is made to write to a pipe that has only one end open.

ERANGE An attempt is made to write to a stream with nbyte outside specified minimum and maximum write range, and the minimum value is nonzero.

In addition, ***pwrite64*** fails and the file pointer remains unchanged if the following is true:

ESPIPE *fildes* is associated with a pipe or **FIFO**.

SEE ALSO

chmod**, **creat64, dup**, **fcntl**, **getrlimit64**, **ioctl**, **lseek64, open64, pipe**, **ulimit**, **socket**, **streamio

readdir64 (libc), readdir64_r (libc)

NAME

readdir64, readdir64_r – read a directory entry in large file environment

SYNOPSIS

#include <dirent.h>
struct direct64 *readdir64 (DIR *dirp);

struct dirent64 *readdir64_r (DIR *dirp, struct dirent64 *entry);

DESCRIPTION

The ***readdir64*** function returns a pointer to a structure representing the directory entry at the current position in the directory stream to which *dirp* refers, and positions the directory stream at the next entry, except on read-only file systems. It returns a "NULL" pointer upon reaching the end of the directory stream, or upon detecting an invalid location in the directory.

The ***readdir64*** function shall not return directory entries containing empty names. It is unspecified whether entries are returned for dot **"**." or dot-dot "..". The pointer returned by ***readdir64*** points to data that may be overwritten by another call to ***readdir64*** on the same directory stream.

This data shall not be overwritten by another call to ***readdir64*** on a different directory stream. The ***readdir64*** function may buffer several directory entries per actual read operation. The ***readdir64*** function marks for update the ***st_atime*** field of the directory each time the directory is actually read.

readdir64_r has the equivalent functionality as ***readdir64*** except that a buffer result must be supplied by the caller to store the result. The size should be ***sizeof***(struct *dirent64*) + {**NAME_MAX**} (that is, ***pathconf***(**_PC_NAME_MAX**)) + 1. **_PC_NAME_MAX** is defined in **<***unistd.h***>**.

The POSIX ***readdir64_r*** function initializes the structure referenced by entry and stores a pointer to this structure in result.

RETURN VALUES

readdir64, and ***readdir64_r*** return NULL on failure and set errno to indicate the error. The POSIX ***readdir64_r*** returns zero if successful, or an error number to indicate failure.

ERRORS

The ***readdir64*** function will fail if one or more of the following are true:

EAGAIN	Mandatory file/record locking was set, **O_NDELAY** or **O_NONBLOCK** was set, and there was a blocking record lock.
EAGAIN	Total amount of system memory available when reading using raw I/O is temporarily insufficient.

EAGAIN	No data is waiting to be read on a file associated with a tty device and **O_NONBLOCK** was set.
EAGAIN	No message is waiting to be read on a stream and **O_NDELAY** or **O_NONBLOCK** was set.
EBADF	The file descriptor determined by the **DIR** stream is no longer valid. This results if the **DIR** stream has been closed.
EBADMSG	Message waiting to be read on a stream is not a data message.
EDEADLK	The ***read*** was going to go to sleep and cause a deadlock to occur.
EFAULT	*buf* points to an illegal address.
EINTR	A signal was caught during the ***read*** or ***readv*** function.
EINVAL	Attempted to read from a stream linked to a multiplexor.
EIO	A physical I/O error has occurred, or the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group of the process is orphaned.
ENOENT	The current file pointer for the directory is not located at a valid entry.
ENOLCK	The system record lock table was full, so the ***read*** or ***readv*** could not go to sleep until the blocking record lock was removed.
ENOLINK	*fildes* is on a remote machine and the link to that machine is no longer active.
ENXIO	The device associated with *fildes* is a block special or character special file and the value of the file pointer is out of range.

SEE ALSO

getdents64

NOTES

readdir64 is unsafe in multithread applications. ***readdir64_r*** is safe, and should be used instead.

tmpfile64 (libc)

NAME

tmpfile64 - create a temporary file in large file environment

SYNOPSIS

#include <stdio.h>

*FILE *tmpfile64(void);*

DESCRIPTION

tmpfile64() creates a temporary file using a name generated by the ***tmpnam*** routine and returns a corresponding FILE pointer. If the file cannot be opened, a NULL pointer is returned. The file is automatically deleted when the process using it terminates or when the file is closed. The file is opened for update (“w+”).

SEE ALSO

creat64, open64, unlink, fopen64, mktemp, perror, stdio, tmpnam

scandir64 (libucb), alphasort64 (libucb)

NAME

scandir64, alphasort64 - scan a directory in large file environment, alphabetically sort on BSD platform

SYNOPSIS

#include <sys/types.h>

#include <sys/dir.h>

int scandir64(char *dirname, struct direct64 *(*namelist[]), int (*select)(.), (*dcomp));

int alphasort64(struct direct64 **d1, struct direct64 **d2);

DESCRIPTION

scandir64 reads the directory dirname and builds an array of pointers to directory entries using ***malloc***. The second parameter is a pointer to an array of structure pointers. The third parameter is a pointer to a routine which is called with a pointer to a directory entry and should return a non zero value if the directory entry should be included in the array. If this pointer is NULL, then all the directory entries will be included. The last argument is a pointer to a routine which is passed to ***qsort***, which sorts the completed array. If this pointer is NULL, the array is not sorted.

alphasort64 is a routine that sorts the array alphabetically.

scandir64 returns the number of entries in the array and a pointer to the array through the parameter namelist.

RETURN VALUES

Returns -1 if the directory cannot be opened for reading or if ***malloc*** cannot allocate enough memory to hold all the data structures.

SEE ALSO

getdents64, readdir64(BSD), malloc, qsort

NOTES

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.

readdir64 (libucb)

NAME

readdir64 - read a directory entry in large file environment on BSD platform

SYNOPSIS

#include <sys/types.h>

#include <sys/dir.h>

struct direct64 *readdir64(DIR *dirp);

DESCRIPTION

readdir64 returns a pointer to a structure representing the directory entry at the current position in the directory stream to which *dirp* refers, and positions the directory stream at the next entry, except on read-only filesystems. It returns a NULL pointer upon reaching the end of the directory stream, or upon detecting an invalid location in the directory. ***readdir64*** shall not return directory entries containing empty names. It is unspecified whether entries are returned for dot or dot-dot. The pointer returned by ***readdir64*** points to data that may be overwritten by another call to ***readdir64*** on the same directory stream. This data shall not be overwritten by another call to ***readdir64*** on a different directory stream. ***readdir64*** may buffer several directory entries per actual read operation. ***readdir64*** marks for update the *st_atime* field of the directory each time the directory is actually read.

RETURN VALUES

readdir64 returns NULL on failure and sets errno to indicate the error.

ERRORS

readdir64 will fail if one or more of the following are true:

EAGAIN	Mandatory file/record locking was set, **O_NDELAY** or **O_NONBLOCK** was set, and there was a blocking record lock.
EAGAIN	Total amount of system memory available when reading using raw I/O is temporarily insufficient.
EAGAIN	No data is waiting to be read on a file associated with a tty device and **O_NONBLOCK** was set.
EAGAIN	No message is waiting to be read on a stream and **O_NDELAY** or **O_NONBLOCK** was set.
EBADF	The file descriptor determined by the **DIR** stream is no longer valid. This results if the DIR stream has been closed.
EBADMSG	Message waiting to be read on a stream is not a data message.
EDEADLK	The ***read*** was going to go to sleep and cause a deadlock to occur.
EFAULT	buf points to an illegal address.
EINTR	A signal was caught during the ***read*** or ***readv*** function.
EINVAL	Attempted to read from a stream linked to a multiplexor.

EIO A physical I/O error has occurred, or the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group of the process is orphaned.

ENOENT The current file pointer for the directory is not located at a valid entry.

ENOLCK The system record lock table was full, so the ***read*** or ***readv*** could not go to sleep until the blocking record lock was removed.

ENOLINK fildes is on a remote machine and the link to that machine is no longer active.

ENXIO The device associated with fildes is a block special or character special file and the value of the file pointer is out of range.

SEE ALSO

getdents64, scandir64

NOTES

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.

mkstemp64(libc)

NAME

mkstemp64 - make a unique file name in large file environment.

SYNOPSIS

*int mkstemp64(char *template);*

DESCRIPTION

mkstemp64 creates a unique file name, typically in a temporary filesystem, by replacing template with a unique file name, and returns a file descriptor for the template file open for reading and writing. The string in template should contain a file name with six trailing XXXXs; ***mkstemp64*** replaces the XXXXs with a letter and the current process ID. The letter will be chosen so that the resulting name does not duplicate an existing file. ***mkstemp64*** avoids the race between testing whether the file exists and opening it for use.

RETURN VALUES

mkstemp64 returns -1 if no suitable file could be created.

SEE ALSO

getpid, open64, mktemp, tmpfile64, tmpnam

NOTES

It is possible to run out of letters.

mkstemp64 actually changes the template string which you pass; this means that you cannot use the same template string more than once - you need a fresh template for every unique file you want to open.

When ***mkstemp64*** is creating a new unique filename it checks for the prior existence of a file with that name. This means that if you are creating more than one unique filename, it is bad practice to use the same root template for multiple invocations of ***mkstemp64***.

aio_cancel64 (libposix4)

NAME

aio_cancel64 - cancel asynchronous I/O request

SYNOPSIS

#include <aio.h>

int aio_cancel64(int fildes, struct aiocb64 *aiocbp);

DESCRIPTION

The ***aio_cancel64*** function attempts to cancel either one or all outstanding asynchronous I/O requests pending on the file descriptor specified by fildes.

If *aiocbp* is **NULL**, then all such outstanding cancelable requests are canceled; otherwise, the individual request referenced by *aiocbp* references will be canceled.

Normal completion notification occurs even for asynchronous I/O operations that are successfully canceled. If there are requests which cannot be canceled, then the normal asynchronous completion process takes place for those requests, and their associated *aiocb64* structures are not modified.

```
struct aiocb64 {
        int                 aio_fildes;      /* file descriptor */
        volatile void       *aio_buf;        /* buffer location */
        size_t              aio_nbytes;      /* length of transfer */
        off64_t             aio_offset;      /* file offset */
        int                 aio_reqprio;     /* request priority offset */
        struct sigevent     aio_sigevent;    /* signal number and offset */
        int                 aio_lio_opcode;  /* listio operation */
};
struct sigevent {
        int                 sigev_notify;    /* notification mode */
        int                 sigev_signo;     /* signal number */
        union sigval        sigev_value;     /* signal value */
};
union sigval {
        int                 sival_int;       /* integer value */
        void                *sival_ptr;      /* pointer value */
};
```

RETURN VALUES

If the requested operation(s) were canceled, ***aio_cancel64*** returns ***AIO_CANCEL*****ED**. But if at least one of the requested operation(s) cannot be canceled because it is in progress, then

AIO_NOTCANCELED is returned, and the application may determine the state of affairs for these operation(s) by using ***aio_error64()***. If all of the operation(s) had already completed, **AIO_ALLDONE** is returned. Otherwise, ***aio_cancel64()*** returns -1, and sets errno to indicate the error condition.

ERRORS

EBADF *fildes* is not a valid file descriptor.

ENOSYS The ***aio_cancel64***() function is not supported.

SEE ALSO

aio_read64(), aio_return64()

aio_fsync64 (libposix4)

NAME

aio_fsync64 - asynchronous file synchronization

SYNOPSIS

#include <aio.h>

int aio_fsync64(int op, aiocb64 *aiocbp);

DESCRIPTION

The ***aio_fsync64***() function queues an asynchronous ***fsync (libc)*** or ***fdatasync (libposix4)*** request for all the currently queued I/O operations on the file referenced by *aiocbp*-**>** *aio_fildes*, and returns control immediately. This request is serviced concurrently with other activity of the process.

If op is **O_DSYNC**, all I/O operations are completed by a call to ***fdatasync*** (synchronized I/O data integrity completion).

If op is **O_SYNC**, all I/O operations are completed by a call to ***fsync*** (synchronized I/O file integrity completion). (see ***fcntl*** definitions of **O_DSYNC** and **O_SYNC**.)

When the request is queued, the error status for the operation is **EINPROGRESS**. When all data has been successfully transferred, the error status is reset to reflect the success or failure of the operation.

aio_return64() and ***aio_error64()*** may be used with this *aiocbp* value to monitor both the return and the error status of the asynchronous operation while it is proceeding.

aiocbp-**>** *aio_sigevent* defines the signal to be generated upon I/O completion.

If *aiocbp*-**>** *aio_sigevent.sigev_signo* is nonzero, then a signal will be generated when all I/O operations have achieved synchronized I/O completion.

```
struct aiocb64 {
        int             aio_fildes;     /* file descriptor */
        volatile void   *aio_buf;       /* buffer location */
        size_t          aio_nbytes;     /* length of transfer */
        off64_t         aio_offset;     /* file offset */
        int             aio_reqprio;    /* request priority offset */
        struct sigevent aio_sigevent;   /* signal number and offset */
        int             aio_lio_opcode; /* listio operation */
};
struct sigevent {
        int             sigev_notify;   /* notification mode */
        int             sigev_signo;    /* signal number */
        union   sigval  sigev_value;    /* signal value */
};
union sigval {
```

```
    int         sival_int;     /* integer value */
    void        *sival_ptr;    /* pointer value */
};
```

RETURN VALUES

If the I/O operation is successfully queued, ***aio_fsync64()*** returns 0. Otherwise, it returns -1, and sets errno to indicate the error condition.

ERRORS

The ***aio_fsync64()*** function will fail if:

EAGAIN The requested asynchronous operation was not queued due to temporary resource limitations.

EBADF *aiocbp->* *aio_fildes* is not a valid file descriptor open for writing.

EINVAL Synchronized I/O is not supported for this file.

A value of op other than **O_DSYNC** or **O_SYNC** was specified.

ENOSYS ***aio_fsync64()*** is not supported by this implementation.

SEE ALSO

fcntl, open64, read, write, aio_error64, aio_return64, fdatasync, fsync, fcntl

NOTES

If ***aio_fsync64()*** fails, outstanding I/O operations are not guaranteed to have been completed.

aio_read64 (libposix4), aio_write64 (libposix4)

NAME

aio_read64, aio_write64 - asynchronous read and write operations

SYNOPSIS

```
#include <aio.h>
int aio_read64 (struct aiocb64 *aiocbp);
int aio_write64 (struct aiocb64 *aiocbp);
struct aiocb64 {
        int                 aio_fildes;     /* file descriptor */
        volatile void       *aio_buf;       /* buffer location */
        size_t              aio_nbytes;     /* length of transfer */
        off64_t             aio_offset;     /* file offset */
        int                 aio_reqprio;    /* request priority offset */
        struct sigevent     aio_sigevent;   /* signal number and offset */
        int                 aio_lio_opcode; /* listio operation */
};
struct sigevent {
        int             sigev_notify;   /* notification mode */
        int             sigev_signo;    /* signal number */
        union sigval    sigev_value;    /* signal value */
};
union sigval {
        int             sival_int;      /* integer value */
        void            *sival_ptr;     /* pointer value */
};
```

DESCRIPTION

aio_read64() is asynchronous read and ***aio_write64()*** is asynchronous write operations. The ***aio_read64()*** function queues an asynchronous read request and returns control immediately. Rather than blocking until completion, the read operation continues concurrently with other activity of the process.

Upon enqueuing the request, the calling process reads "*aiocbp->nbytes*" from the file referred to by "*aiocbp->fildes*" into the buffer pointed to by "*aiocbp->aio_buf*". "*aiocbp->offset*" marks the absolute position from the beginning of the file (in bytes) at which the read begins.

The ***aio_write64()*** function queues an asynchronous write request, and returns control immediately. Rather than blocking until completion, the write operation continues concurrently with other activity of the process.

Upon enqueuing the request, the calling process writes "*aiocbp->nbytes*" from the buffer pointed to by "*aiocbp->aio_buf*" into the file referred to by "*aiocbp->fildes*". If **O_APPEND** is set for "*aiocbp->fildes*", ***aio_write64()*** operations append to the file in the same order as the calls were made. If **O_APPEND** is not set for the file descriptor, then the write operation will occur at the absolute position from the beginning of the file plus "*aiocbp->offset*" (in bytes). These asynchronous operations are submitted at a priority equal to the calling process' scheduling priority minus "*aiocbp->aio_reqprio*". For regular files, no data transfer will occur past the offset maximum established in the open file description associated with "*aiocbp->fildes*". *aiocb-> aio_sigevent* defines both the signal to be generated and how the calling process will be notified upon I/O completion. If *aio_sigevent.sigev_notify* is **SIGEV_NONE**, then no signal will be posted upon I/O completion, but the error status and the return status for the operation will be set appropriately. If *aio_sigevent.sigev_notify* is **SIGEV_SIGNAL**, then the signal specified in *aio_sigevent.sigev_signo* will be sent to the process. If the **SA_SIGINFO** flag is set for that signal number, then the signal will be queued to the process and the value specified in *aio_sigevent.sigev_value* will be the *si_value* component of the generated signal (see ***siginfo***).

RETURN VALUES

If the I/O operation is successfully queued, ***aio_read64()*** and ***aio_write64()*** return 0; otherwise, they return -1, and set errno to indicate the error condition.*aiocbp* may be used as an argument to ***aio_error64()*** and ***aio_return64()*** in order to determine the error status and the return status of the asynchronous operation while it is proceeding.

ERRORS

The ***aio_read64()*** and ***aio_write64()*** function will fail if:

EAGAIN The requested asynchronous I/O operation was not queued due to system resource limitations.

ENOSYS The ***aio_read64()*** or ***aio_write64()*** functions are not supported.

EBADF If the calling function is ***aio_read64()***, and "*aiocbp->fildes*" is not a valid file descriptor open for reading. If the calling function is ***aio_write64()***, and "*aiocbp->fildes*" is not a valid file descriptor open for writing.

EINVAL The file offset value implied by "*aiocbp->aio_offset*" would be invalid, "*aiocbp->aio_reqprio*" is not a valid value, or "*aiocbp->aio_nbytes*" is an invalid value.

ECANCELED The requested I/O was canceled before the I/O completed due to an explicit ***aio_cancel64()*** request.

EINVAL The file offset value implied by "*aiocbp->aio_offset*" would be invalid.

The following are additional conditions which may be detected synchronously or asynchronously:

aio_read64()

EFBIG The file is a regular file, *aiocbp->aio_nbytes* is greater than 0 and the starting offset in *aiocbp->aio_offset* is at or beyond the offset maximum in the open file description associated with *aiocbp->fildes*.

SEE ALSO

close, exec, exit, fork, lseek, read, write, aio_cancel64, aio_return64, lio_listio64, siginfo

NOTES

For portability, the application should set *aiocb-> aio_reqprio* to 0.

aio_return64 (libposix4), aio_error64(libposix4)

NAME

aio_return64, ***aio_error64*** - retrieve return or error status of asynchronous I/O operation

SYNOPSIS

```
#include <aio.h>

ssize_t      aio_return64    (struct aiocb64 *aiocbp);

int          aio_error64     (const struct aiocb64 *aiocbp);


struct aiocb64 {
        int              aio_fildes;      /* file descriptor */
        volatile void    *aio_buf;        /* buffer location */
        size_t           aio_nbytes;      /* length of transfer */
        off64_t          aio_offset;      /* file offset */
        int              aio_reqprio;     /* request priority offset */
        struct sigevent  aio_sigevent;    /* signal number and offset */
        int              aio_lio_opcode;/* listio operation */
};
struct sigevent {
        int              sigev_notify;        /* notification mode */
        int              sigev_signo;         /* signal number */
        union   sigval   sigev_value;         /* signal value */
};
union sigval {
        int              sival_int;           /* integer value */
        void             *sival_ptr;          /* pointer value */
};
```

DESCRIPTION

The ***aio_return64()*** function returns the return status of the asynchronous I/O request associated with the *aiocb64* structure pointed to by *aiocbp*.

aio_error64() returns the error status of the asynchronous I/O request associated with the *aiocb64* structure pointed to by *aiocbp*.

The ***aio_return64()*** function should be called only once to retrieve the valid return status of a given asynchronous operation, after ***aio_error64()*** has returned a value other than **EINPROGRESS**.

RETURN VALUES

If the asynchronous I/O operation has completed successfully, ***aio_return64()*** returns the return status, as described for ***read***, ***write***, and ***fsync***.

If the asynchronous I/O operation has completed successfully, ***aio_error64()*** returns 0. If the operation has not yet completed, then **EINPROGRESS** is returned. If the asynchronous I/O operation has completed unsuccessfully, then the error status, as described for read, write, and ***fsync*** is returned.

If unsuccessful, ***aio_return64()*** or ***aio_error64()*** return -1, and set errno to indicate the error condition.

ERRORS

The ***aio_retur64***()n and ***aio_error64()*** functions will fail if:

EINVAL *aiocbp* does not reference an asynchronous operation which has completed or failed.

ENOSYS The ***aio_return64()*** or ***aio_error64()*** function is not supported.

SEE ALSO

close, exec, exit, fork, lseek, read, write, aio_cancel64, aio_fsync64, aio_read64, fsync, lio_listio64

aio_suspend64 (libposix4)

NAME

aio_suspend64 - wait for asynchronous I/O request

SYNOPSIS

#include <aio.h>

int aio_suspend64 (const struct aiocb64 *const list, int nent, const struct timespec *timeout);

DESCRIPTION

aio_suspend64() wait for asynchronous I/O request. The ***aio_suspend64()*** function suspends the caller until at least one of the asynchronous I/O operations referenced by list has completed, until a signal interrupts the function, or, if timeout is not **NULL**, until the time interval specified by timeout has passed. If any of the *aiocb64* structures in the list corresponds to a completed asynchronous I/O operation (that is, the error status for the operation is not equal to **EINPROGRESS**), at the time of the call, the function returns without suspending the caller.

If the time interval indicated in the timespec structure pointed to by timeout passes before any of the I/O operations referenced by list are completed, then ***aio_suspend64()*** returns with an error.

The list argument is an array of pointers to asynchronous I/O control blocks. The nent argument indicates the number of elements in this array. Each *aiocb64* structure pointed to must have been used in initiating an asynchronous I/O request via ***aio_read64()***, ***aio_write64()***, ***aio_fsync64()***, or ***lio_listio64()***. This array may contain null pointers which will be ignored.

```
struct aiocb64 {
        int                 aio_fildes;       /* file descriptor */
        volatile void       *aio_buf;         /* buffer location */
        size_t              aio_nbytes;       /* length of transfer */
        off64_t             aio_offset;       /* file offset */
        int                 aio_reqprio;      /* request priority offset */
        struct   sigevent   aio_sigevent;     /* signal number and offset */
        int                 aio_lio_opcode;   /* listio operation */
};
struct sigevent {
        int                 sigev_notify;     /* notification mode */
        int                 sigev_signo;      /* signal number */
        union sigval        sigev_value;      /* signal value */
};
union sigval {
        int                 sival_int;        /* integer value */
        void                *sival_ptr;       /* pointer value */
};
struct timespec {
```

```
        time_t          tv_sec;      /* seconds */
        long            tv_nsec;     /* and nanoseconds */
    };
```

RETURN VALUES

If ***aio_suspend64()*** returns after one or more asynchronous I/O operations have completed, it returns 0. Otherwise, it returns -1, and sets errno to indicate the error condition.

The application may determine which asynchronous I/O had completed with both the associated error and return status of ***aio_return64()***, and ***aio_error64().***

ERRORS

The ***aio_suspend64()*** function will fail if:

EAGAIN No asynchronous I/O indicated in the list referenced by list completed in the time interval indicated by timeout.

EINTR A signal interrupted the ***aio_suspend64()*** function. Note that, each asynchronous I/O operation may possibly provoke a signal when it completes, this error return may be caused by the completion of one (or more) of the very I/O operations being awaited.

ENOSYS The ***aio_suspend64()*** function is not supported.

SEE ALSO

aio_fsync64, aio_read64, aio_return64, aio_write64, lio_listio64

lio_listio64 (libposix4)

NAME

lio_listio64 - list directed I/O

SYNOPSIS

```
#include <aio.h>
int lio_listio64 (int mode, struct aiocb64 *const list, int nent, struct sigevent *sig);
struct aiocb64 {
        int                aio_fildes;     /* file descriptor */
        volatile void      *aio_buf;       /* buffer location */
        size_t             aio_nbytes;     /* length of transfer */
        off64_t            aio_offset;     /* file offset */
        int                aio_reqprio;    /* request priority offset */
        struct sigevent    aio_sigevent;   /* signal number and offset */
        int                aio_lio_opcode; /* listio operation */
};
struct sigevent {
        int                sigev_notify;   /* notification mode */
        int                sigev_signo;    /* signal number */
        union    sigval    sigev_value;    /* signal value */
};
union sigval {
        int                sival_int;      /* integer value */
        void               *sival_ptr;     /* pointer value */
};
```

DESCRIPTION

The ***lio_listio64()*** function allows the calling process, **LWP**, or thread, to initiate a list of I/O requests within a single function call.

If mode is set to **LIO_WAIT**, ***lio_listio64()*** behaves synchronously, waiting until all I/O is completed, and the sig argument is ignored. If mode is set to **LIO_NOWAIT**, ***lio_listio64()*** behaves asynchronously; returning immediately, and signal delivery will occur, according to the *sig* argument, when all the I/O operations from this function complete. If sig is **NULL**, or the *sigev_signo* member of the sigevent structure referenced by sig is zero, then no signal delivery will occur. Otherwise, the signal number indicated by *sigev_signo* will be delivered when all the requests in list have completed.

list is an array of pointers to *aiocb64* structures. This array consists of nent elements. The array may contain null pointers, which will be ignored.

The *aio_lio_opcode* field of each *aiocb64* structure in list specifies the operation to be performed (see */usr/include/aio.h*).

LIO_READ requests ***aio_read64().***

LIO_WRITE requests ***aio_write64()***.

LIO_NOP causes the list entry to be ignored. *nent* specifies the length of the array (number of members of the list).

LIO_WAIT When mode has the value **LIO_WAIT**, a pointer to a signal control structure, *sig*, is used to define both the signal to be generated and how the calling process will be notified upon I/O completion.

If *sig*-> *sigev_notify* is **SIGEV_NONE**, then no signal will be posted upon I/O completion, but the error status and the return status for the operation will be set appropriately.

If *sig*-> *sigev_notify* is **SIGEV_SIGNAL**, then the signal specified in *sig*-> *sigev_signo* will be sent to the process.

If the **SA_SIGINFO** flag is set for that signal number, then the signal will be queued to the process and the value specified in *sig*-> *sigev_value* will be the *si_value* component of the generated signal (see ***siginfo***).

For regular files, no data transfer will occur past the offset maximum established in the open file description associated with *aiocbp*-> *aio_fildes*.

The behavior of this function is altered according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion if synchronized I/O is enabled on the file associated with *aio_fildes*. (see ***fcntl*** definitions of **O_DSYNC** and **O_SYNC**.)

RETURN VALUES

If the mode argument has the value **LIO_NOWAIT**, and the I/O operations are successfully queued, ***lio_listio64()*** returns 0; otherwise, it returns -1, and sets errno to indicate the error condition.

If the mode argument has the value **LIO_WAIT**, and when all the indicated I/O has completed successfully, ***lio_listio64()*** returns 0; otherwise, it returns -1, and sets errno to indicate the error condition.

In either case, the return value only indicates the success or failure of the ***lio_listio64()*** call itself, not the status of the individual I/O requests. In some cases, one or more of the I/O requests contained in the list may fail. Failure of an individual request does not prevent completion of any other individual request. To determine the outcome of each I/O request, the application must examine the error status associated with each *aiocb64* control block. Each error status so returned is identical to that returned as a result of an ***aio_read64()*** or ***aio_write64()*** function.

ERRORS

The ***lio_listio64()*** function will fail if:

EAGAIN The resources necessary to queue all the I/O requests were not available.The error status for each request is recorded in the ***aio_error64()*** member of the corresponding *aiocb64* structure, and can be retrieved using ***aio_error64()*** *nent* entries exceed the system-wide limit, **AIO_MAX**.

EINVAL The mode argument is an improper value. The value of nent is greater than **AIO_LISTIO_MAX.**

EINTR A signal was delivered while waiting for all I/O requests to complete during an **LIO_WAIT** operation. However, the outstanding I/O requests are not canceled. Use ***aio_fsync64()*** to determine if any request was initiated; ***aio_return64()*** to determine if any request has completed; or

aio_error64() to determine if any request was canceled.

EIO One or more of the individual I/O operations failed. Using ***aio_error64()*** with each *aiocb64* structure will determine the individual request(s) that failed.

ENOSYS ***lio_listio64()*** is not supported by this implementation. If either ***lio_listio64()*** succeeds in queuing all of its requests, or errno is set to **EAGAIN, EINTR**, or **EIO**, then some of the I/O specified from the list may have been initiated. In this event, each *aiocb64* structure contains errors specific to the ***read*** or ***write*** function being performed:

EAGAIN The requested I/O operation was not queued due to resource limitations.

ECANCELED The requested I/O was canceled before the I/O completed due to an explicit ***aio_cancel64()*** request.

EINPROGRESS The requested I/O is in progress.

The following are additional error codes which may be set for each *aiocb64* control block:

EFBIG The *aiocbp*-> *aio_lio_opcode* is **LIO_WRITE**, the file is a regular file, *aiocbp*-> *aio_nbytes* is greater than 0, and the *aiocbp*-> *aio_offset* is greater than or equal to the offset maximum in the open file description associated with *aiocbp*-> *aio_fildes*.

SEE ALSO

close, exec, fork, lseek, read, write, aio_cancel64, aio_fsync64, aio_read64, aio_return64, fcntl, siginfo

aioread64 (libaio), aiorwrite64 (libaio)

NAME

aioread64, aiowrite64 - asynchronous I/O operations in large file environment

SYNOPSIS

#include <sys/types.h>

#include <sys/asynch.h>

int aioread64(int fildes, char *bufp, int bufs,off64_t offset, int whence,
aio_result_t *resultp);

int aiowrite64(int fildes, const char *bufp, int bufs, off64_t offset, int whence,
aio_result_t *resultp);

DESCRIPTION

aioread64 initiates one asynchronous ***read*** and returns control to the calling program. The ***read*** continues concurrently with other activity of the process. An attempt is made to read bufs bytes of data from the object referenced by the descriptor fildes into the buffer pointed to by *bufp*.

aiowrite64 initiates one asynchronous ***write*** and returns control to the calling program. The write continues concurrently with other activity of the process. An attempt is made to write bufs bytes of data from the buffer pointed to by bufp to the object referenced by the descriptor fildes.

On objects capable of seeking, the I/O operation starts at the position specified by whence and offset. These parameters have the same meaning as the corresponding parameters to the ***llseek*** function. On objects not capable of seeking the I/O operation always start from the current position and the parameters whence and offset are ignored. The seek pointer for objects capable of seeking is not updated by ***aioread64*** or ***aiowrite64***. Sequential asynchronous operations on these devices must be managed by the application using the whence and offset parameters.

The result of the asynchronous operation is stored in the structure pointed to by resultp:

int aio_return; /* return value of read or write */

int aio_errno; /* value of errno for read or write */

Upon completion of the operation both *aio_return* and *aio_errno* are set to reflect the result of the operation. **AIO_INPROGRESS** is not a value used by the system so the client may detect a change in state by initializing aio_return to this value.

The application supplied buffer bufp should not be referenced by the application until after the operation has completed. While the operation is in progress, this buffer is in use by the operating system.

Notification of the completion of an asynchronous I/O operation may be obtained synchronously through the ***aiowait*** function, or asynchronously by installing a signal handler for the **SIGIO** signal. Asynchronous notification is accomplished by sending the process a **SIGIO** signal. If a signal handler is not installed for the **SIGIO** signal, asynchronous notification is disabled. The delivery of this instance of the **SIGIO** signal is reliable in that a signal delivered while the handler is executing is not lost. If the client ensures that ***aiowait*** returns nothing (using a polling timeout) before returning from the signal handler, no asynchronous I/O notifications are lost. The ***aiowait*** function is the only way to dequeue an asynchronous notification. Note: **SIGIO** may have several meanings simultaneously: for example, that a descriptor generated **SIGIO** and an asynchronous operation completed. Further, issuing an asynchronous request successfully guarantees that space exists to queue the completion notification.

close, ***exit*** and ***execve*** (see ***exec***) will block until all pending asynchronous I/O operations can be canceled by the system.

It is an error to use the same result buffer in more than one outstanding request. These structures may only be reused after the system has completed the operation.

RETURN VALUES

aioread64 and ***aiowrite64*** return:

0 on success.

-1 on failure and set errno to indicate the error.

ERRORS

EAGAIN	The number of asynchronous requests that the system can handle at any one time has been exceeded
EBADF	fildes is not a valid file descriptor open for reading.
EFAULT	At least one of bufp points to an address outside the address space of the requesting process. See NOTES below.
EINVAL	The parameter resultp is currently being used by an outstanding asynchronous request.
EINVAL	offset is not a valid offset for this file system type.
ENOMEM	Memory resources are unavailable to initiate request.

SEE ALSO

close, exec, exit, llseek, lseek, open64, read, write, aiocancel, aiowait, sigvec

NOTES

Passing an illegal address to bufp will result in setting errno to **EFAULT** only if it is detected by the application process.

SPARC COMPLIANCE DEFINITION 2.4.1 IS

Execution Environment

/dev/zero

NAME

/dev/zero

SYNOPSIS

/dev/zero

DESCRIPTION

The device ***/dev/zero*** is defined to be a special file which is a source of zeroed, unnamed memory. Reads from this device always return a buffer full of zeroes. The file is infinite in length. Writes to this file are always successful, but the data written is ignored. Mapping a zero special file creates a zero-initialized, unnamed memory object of a length equal to the length of the mapping rounded up to the nearest page size as returned by sysconf. Multiple processes can share such a zero special file object provided a common ancestor mapped the object **MAP_SHARED**

SPARC COMPLIANCE DEFINITION 2.4.1 IS

Commands

mkmsgs

NAME

mkmsgs - create message files for use by gettxt

SYNOPSIS

mkmsgs [-o] [-i locale] inputstrings msgfile

DESCRIPTION

The mkmsgs utility is used to create a file of text strings that can be accessed using the text retrieval tools (see ***gettxt***, ***srchtxt***, ***exstr***, and ***gettxt***. It will take as input a file of text strings for a particular geographic locale (see ***setlocale***) and create a file of text strings in a format that can be retrieved by ***gettxt***. By using the -i option, you can install the created file under the ***/usr/lib/locale/locale/LC_MESSAGES*** directory (locale corresponds to the language in which the text strings are written).

inputstrings is the name of the file that contains the original text strings. msgfile is the name of the output file where mkmsgs writes the strings in a format that is readable by ***gettxt***. The name of msgfile can be up to 14 characters in length, but may not contain either \0 (null) or the ASCII code for / (slash) or: (colon).

The input file contains a set of text strings for the particular geographic locale. Text strings are separated by a newline character. Nongraphic characters must be represented as alphabetic escape sequences. Messages are transformed and copied sequentially from inputstrings to msgfile. To generate an empty message in msgfile, leave an empty line at the correct place in inputstrings.

Strings can be changed simply by editing the file input strings. New strings must be added only at the end of the file; then a new msgfile file must be created and installed in the correct place. If this procedure is not followed, the retrieval function will retrieve the wrong string and software compatibility will be broken.

OPTIONS

-o Overwrite msgfile, if it exists.

-i locale Install msgfile in the **/usr/lib/locale/locale/LC_MESSAGES** directory. Only someone who is super-user or a member of group bin can create or overwrite files in this directory. Directories under /usr/lib/locale will be created if they do not exist.

SEE ALSO

exstr, ***gettxt***, ***srchtxt***

SPARC COMPLIANCE DEFINITION 2.4.1 IS

INDEX

Symbols

Numerics

A

E

F

G

H

N

O

U

X

Y

Z