Portable Multithreading

The Signal Stack Trick For User-Space Thread Creation

Ralf S. Engelschall
rse@engelschall.com
Weblinger Weg 28, 85221 Dachau, Germany.

August 1999

Document Version 0.9.15

Abstract

This paper describes a pragmatic but mostly portable fallback approach for creating and dispatching between multiple threads of execution on Unix systems which lack a dedicated user-space context switching facility. The described approach closely follow the techniques implemented in the author's user-space multithreading library *GNU Portable Threads* (Pth)[1] and usually can be easily adapted by other user-space implementations as their fallback approach. It is entirely based on Unix system facilities and ANSI C language features and especially does not require any assembly code or platform specific tricks at all. The most interesting issue is the technique of creating threads, which is explained in detail.

Keywords: portability, multithreading, Unix, POSIX, ANSI C, user-space, context creation, context switching, signal handler, stack, makecontext, switchcontext, sigaltstack, setjmp, longjmp.

1 Introduction

1.1 Multithreading

The paradigm of programming with multiple threads of execution (aka *multithreading*) is already a very old one and dates back to the decades of programming with *coroutines* [1, 2]. Paradoxically the use of threads on Unix platforms became popular the first time in the early 1990s.

Multithreading Advantages

Multithreading can provide many benefits for applications (good runtime concurrency, parallel programming techniques can be easier implemented, the popular procedural programming style can be combined with multiple threads of execution, etc.) but the most interesting ones are usually performance gains and reduced resource consumption. Because in contrast to multiprocess applications, multithreaded ones usually require less system resources (mainly memory) and their internal communication part can leverage from the shared address space.

Multithreading and Application

Nevertheless there still exist just a few real applications in the free software world which use multithreading for their benefit, although their application areas are predestinated for multithreading. For instance the popular Apache webserver [2] as of version 1.3 still uses a pre-forking process model for serving HTTP requests, although experiments with multithreaded Apache variants already in 1996 showed great performance boosts. The same applies to other applications.

The reason mainly is, that for a long time multithreading facilities under Unix were rare. The situation became better after some vendors like Sun and DEC incorporated threading facilities into their Unix flavors and POSIX

[1] see http://www.gnu.org/software/pth/
[2] see http://www.apache.org/

standardized a threading API (known as Pthreads). [3]. But an API and a few vendor implementation are not enough to fulfill the portability requirements of free software packages. Here portable multithreading libraries are needed first.

The author currently knows about over fourteen (mostly user-space) freely available multithreading solutions for Unix systems [4], but only a few of them are really portable. And even the mostly portable ones suffer from the fact that they partly depend on assembly code or platform specific tricks usually related to the creation and dispatching of the individual threads. This means that the number of platforms they are working on are usually a fixed set and applications which are based on them are only portable to those platforms. This situation is not satisfactory, so application authors still avoid the use multithreading when they want to (or have to) achieve maximum portability for their application.

So a pragmatic but mostly portable fallback technique for implementing user-space threads would be useful, because this way the possibilities to use multithreading in free software applications could be greatly improved.

Ingredients Of A Thread

A Unix process has many ingredients, but the most important ones are its memory mapping table, the signal dispatch table, the signal mask, the set of file descriptors and the machine context, which in turn consists of at least the CPU registers including the program counter and the stack pointer. On the other hand, there can be light-weight processes (LWP) or threads which share all ingredients with the underlying (heavy-weight) process except for the machine context.

Kernel-Space vs. User-Space

Those LWPs or threads, on a Unix platform classically can be implemented either inside kernel-space or user-space. When implemented in kernel-space, one usually calls them LWPs, else (user-space) threads. If threads are implemented by the kernel, the thread context switches are performed by the kernel without notice by the application, similar to the dispatching of processes. If threads are implemented in user-space, the thread context switches are performed usually by an application library without notice by the kernel. Additionally, there exist hybrid threading approaches, where typically a user-space library binds one or more user-space threads to one or more kernel-space LWPs.

Thread Models

For instance the vendor threading facilities under *Solaris*, *AIX*, *Tru64* and *IRIX* use a **M:N** mapping [15, 24], i.e., **M** user-space threads are mapped onto **N** kernel-space LWPs. On the other hand *LinuxThreads* [23] under *GNU/Linux* uses a **1:1** mapping and pure user-space implementations like *GNU Pth*, *FSU pthreads* or *MIT pthreads*, etc. use a **M:1** mapping [19, 16, 17].

From now on we talk about such **M:1** user space threading approaches, where one or more user space threads are implemented inside a single kernel space process. The exercise is to implement this by using standardized Unix system and ANSI C language facilities *only*.

1.2 The Exercise

As we have mentioned, a thread shares all things with the underlying process except for the machine context. So the major task for a user-space threading library is to create and dispatch those machine contexts for the application.

In practice, the second major task it has to do is to make sure that no thread by accident blocks the whole process (and thereby all other threads). Instead when an operation would block, the task of the threading library is to suspend only the execution of the current thread and in the meantime dispatch the remaining threads. But this task is outside the scope of this paper. We focus only on the aspect of machine context handling.

1.3 The Curse of Portability

Our goal of real portability for a threading library causes some non-trivial problems which have to be solved. The most obvious one is that dealing with machine contexts usually suffers greatly from portability, because it is a highly CPU dependent task, although in principle it is just a matter of switching a few CPU registers (mainly the program counter and the stack pointer).

Assembly Code Considered Harmful

Additionally we disallow the use of any assembly solutions or platform specific tricks, because then the threading library again would be only semi-portable, i.e., it can be ported to **N** platforms but on the **(N+1)**th platform one has to manually adjust or even extend the threading library to work there, too.

That is not acceptable for us, even when it also makes solving the problems harder, of course. At least all

[3] IEEE POSIX 1003.1c-1995 aka ISO/IEC 9945-1:1996

[4] see the GNU Pth webpage for references to them

other known free software user-space threading libraries [16, 17, 18] do not restrict themself to this and therefore are just semi-portable. But real portability is our major goal.

2 Problem Analysis

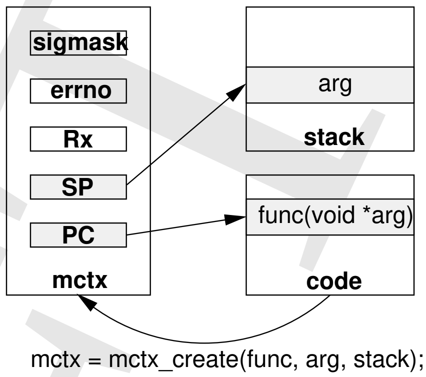
2.1 The Task in Detail

Our task sounds simple: Create and initialize a machine context (mctx) for a thread, which when switched to the first time, lets the thread start executing at a startup routine (func) with a startup argument (arg) on a particular run-time stack (stack). When the thread then again yields execution its individual thread ingredients are stored in mctx and on the next dispatching they are again restored from there.

The Proposed API

In detail we propose the following *Application Programmers Interface* (API) for the machine context handling:

- A data structure **mctx_t** which holds the machine context.
- A function "**mctx_t** `*mctx_create(` **void** `(*`*func*`)(`**void**`),` **void** `*`*arg*, **void** `*`*skaddr*, **size_t** *sksize*`)`" which creates a machine context structure out of a start routine *func*, a start argument *arg* and a stack starting at *skaddr*, which is *sksize* bytes in size. It returns the created **mctx_t** data structure.
- A function "**int** `mctx_save(`**mctx_t** `*`*mctx*`)`" which saves the current machine context into the machine context structure *mctx*. It returns `FALSE` when the machine context was saved and it returns the first time, and `TRUE` when the machine context was restored by `mctx_restore` and it returns the second time (see below).
- A function "**void** `mctx_restore(` **mctx_t** `*`*mctx*`)`" which restores the machine context from the machine context structure *mctx*. This function does not return at all. Instead it returns at the location of the `mctx_save` function but with a return of `TRUE`.



Context Switching

There are two good reasons for the particular semantics of `mctx_save`: First, the underlying implementation we will choose later, will support exactly these semantics. Second, we want to use an easy dispatching function which looks like this:

```
void
mctx_switch(mctx_t *from,
            mctx_t *to)
{
  if (mctx_save(from) == FALSE)
      mctx_restore(to);
  return;
}
```

2.2 Possibilities

When we poke around in the references of the ANSI C language and the Unix standards we find the following functions on which we can base our implementation:

- There are the functions `getcontext`(3), `makecontext`(3), `swapcontext`(3) and `setcontext`(3) which conform to the Single Unix Specification, Version 2 (SUSv2 [14], aka Unix95/98). Unfortunately these are available on modern Unix platforms only.
- There are the functions `setjmp`(3) and `longjmp`(3) which conform to ISO 9899:1990 (ISO C) and `sigsetjmp`(3) and `siglongjmp`(3) which conform to IEEE Std1003.1-1988 (POSIX), and Single Unix Specification, Version 2 (SUSv2 [14], aka Unix95/98). The first two functions are available really on all Unix platforms, the last two are available only on some of them.

- There is the function `sigaltstack`(2) which conforms to the Single Unix Specification, Version 2 (SUSv2 [14]) and its ancestor function `sigstack`(2) from 4.2BSD. The last one exists only on BSD-derived platforms, but the first function already exists on all current Unix platforms.

2.3 Maximum Portability Solution

The maximum portable solution obviously is to use the standardized `makecontext`(3) function to create threads and `switchcontext`(2) or `getcontext`(2)/`setcontext`(3) to dispatch them. And actually these are the preferred functions modern user-space multithreading libraries are using. Unfortunately there are still a lot of platforms where this approach cannot be used. So our fallback approach has to use the remaining possibilities.

2.4 Remaining Possibilities

Our problem can be divided into two parts, an easy one and a difficult one.

The Easy Part

That `setjmp`(3) and `longjmp`(3) can be used to implement user-space threads is commonly known, of course. Mostly all older portable user-space threading libraries are based on them, although some problems are known with these facilities (see below). So it becomes clear, that we have to use the these functions and base our machine context (**mctx_t**) on their `jmp_buf` data structure.

We immediately recognize that this way we have at least solved the dispatching problem, because our `mctx_save` and `mctx_restore` functions were modeled wisely after `setjmp`(3) and `longjmp`(3).

The Difficult Part

Nevertheless the difficult problem of how to create the machine context remains. Even knowing that our machine context is `jmp_buf` based is no advantage to us. A `jmp_buf` has to be treated by us as an opaque data structure — for portability reasons. The only operations we can perform on it, are `setjmp`(3) and `longjmp`(3) calls, of course. Additionally, we are forced to use `sigaltstack`(3) for our stack manipulations, because it is the only portable function which actually deals with stacks.

It should be noted, that other thread packages have tried similar approaches. In particular, ROBERT S. THAU's *Really Simple Threads* (rsthreads, [21]) package uses `sigaltstack`(2) in a very similar way for thread creation. And KOTA ABE's *Portable Thread Library* (PTL, [18]) uses a similar `sigstack`(2) approach as its fallback approach for thread creation. But both variants do not work on some System-V-derived platforms, for reasons we will discuss later.

It is clear our implementation for `mctx_create` has to play a few tricks to use a `jmp_buf` for passing execution control to an arbitrary startup routine. And it has to be carefully to not suffer from unexpected side-effects. It should be also obvious that we cannot again expect to find an easy solution (as for `mctx_save` and `mctx_restore`), because `setjmp`(3) and `sigaltstack`(3) cannot be trivially combined to achieve our expected semantics for `mctx_create`.

3 Implementation

As we have already discovered, our implementation contains an easy part (`mctx_save` and `mctx_restore`) and a difficult part (`mctx_create`). Let us start with the easy part, whose implementation is obvious: [5]

```
typedef struct mctx_st {
    jmp_buf jb;
} mctx_t;

#define mctx_save(mctx)
    (setjmp(mctx->jb))

#define mctx_restore(mctx)
    (longjmp(mctx->jb, TRUE))

mctx_t *
mctx_create(void(*func)(void),
            void *arg,
            void *skaddr,
            size_t sksize      )
{
    mctx_t *mctx;

    ...to be filled in...
    return mctx;
}
```

There is only one subtle but important point we should mention: The use of the C pre-processor `#define` directive to implement `mctx_save` and `mctx_restore` is not indiscriminately. For technical reasons related to

[5] All error checks are omitted for better readability.

setjmp(3) and return (which we will explain later in detail) we *cannot* implement these two functions as C functions if we want to achieve maximum portability.

Before we now can proceed with filling in the gaps of the implementation skeleton for mctx_create, let us first discuss the proposed algorithm.

3.1 Overview

The general idea for mctx_create, is to configure the given stack as a signal stack via sigaltstack(2), send the current process a signal to transfer execution control onto this stack, save the machine context there via setjmp(3), get rid of the signal handler scope and bootstrap into the startup routine.

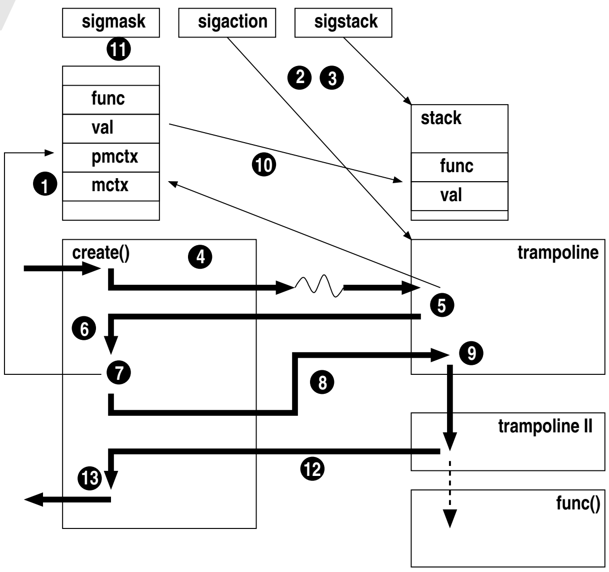
As mentioned, the real problem in this approach comes from the signal handler scope which implies various restrictions on Unix platforms. As we will see, we have to perform a few tricks to get rid of it. The second main problem is: how do we prepare the calling of the start routine without actually entering it.

3.2 Algorithm

The input to the mctx_create function is the thread startup routine *func*, the thread startup argument *arg* and a chunk of memory starting at *skaddr* and of size *sksize*, which should become the threads stack. The algorithm of mctx_create is directly modeled after the implemented algorithm one can find in *GNU Pth*:

1. Allocate a new machine context structure *mctx*.
2. Preserve the current signal mask and block an arbitrary worker signal.
3. Preserve a possibly existing signal action for the worker signal configure a trampoline function as the new temporary signal action.
4. Preserve a possibly active alternate signal stack and configure *skaddr* as the new temporary alternate signal stack of length *sksize*.
5. Send the current process the worker signal, temporarily unblock it and this way allow it to be delivered on the signal stack in order to transfer execution control to the trampoline function.
6. After the trampoline function asynchronously entered, save its machine context in the *mctx* structure and immediately return from it to terminate the signal handler scope.
7. Restore the preserved alternate signal stack and the preserved signal action and mask for worker signal.
8. Save the current machine context of mctx_create.
9. Restore the previously saved machine context of the trampoline function (*mctx*) to again transfer execution control onto the alternate stack, but this time without(!) signal handler scope.
10. After reaching the trampoline function (*mctx*) again, immediately bootstrap into a clean stack frame situation by calling a second trampoline function.
11. Set the new signal mask to be the same as the original signal mask which was active when mctx_create was called.
12. Load the startup information (*func*, *arg*) from mctx_create into local (stack-based) variables.
13. Restore the machine context of mctx_create to transfer execution control back to it.
14. Return the generated machine context *mctx* to the caller.

When we now again switch into the established machine context *mctx*, the thread starts running at routine *func* with argument *arg*. The following figure tries to illustrate the algorithm:



3.3 Source Code

The corresponding ANSI C code, which implements `mctx_create`, is a little bit more complicated, of course. But with the presented algorithm in mind, it is now straight-forward.

```
static mctx_t                mctx_caller;
static volatile sig_atomic_t mctx_called;

static mctx_t  *mctx_creat;
static void   (*mctx_creat_func)(void *);
static void    *mctx_creat_arg;
static sigset_t mctx_creat_sigs;

mctx_t *mctx_create(
    void (*func)(void *), void *arg,
    void *skaddr, size_t sksize)
{
    mctx_t *mctx;
    struct sigaction sa;
    struct sigaction osa;
    struct sigaltstack ss;
    struct sigaltstack oss;
    sigset_t osigs;
    sigset_t sigs;

    /* Step 1: */
    mctx = malloc(sizeof(mctx_t));

    /* Step 2: */
    sigemptyset(&sigs);
    sigaddset(&sigs, SIGUSR1);
    sigprocmask(SIG_BLOCK,
                &sigs, &osigs);

    /* Step 3: */
    memset((void *)&sa, 0,
           sizeof(struct
                  sigaction));
    sa.sa_handler =
        mctx_create_trampoline;
    sa.sa_flags = SA_ONSTACK;
    sigemptyset(&sa.sa_mask);
    sigaction(SIGUSR1, &sa, &osa);

    /* Step 4: */
    ss.ss_sp    = skaddr;
    ss.ss_size  = sksize;
    ss.ss_flags = 0;
    sigaltstack(&ss, &oss);

    /* Step 5: */
    mctx_creat      = mctx;
    mctx_creat_func = func;
    mctx_creat_arg  = arg;
    mctx_creat_sigs = osigs;
    mctx_called     = FALSE;
    kill(getpid(), SIGUSR1);
    sigfillset(&sigs);
    sigdelset(&sigs, SIGUSR1);
    while (!mctx_called)
        sigsuspend(&sigs);

    /* Step 7: */
    sigaltstack(NULL, &ss);
    ss.ss_flags = SS_DISABLE;
    sigaltstack(&ss, NULL);
    if (!(oss.ss_flags & SS_DISABLE))
        sigaltstack(&oss, NULL);
    sigaction(SIGUSR1, &osa, NULL);
    sigprocmask(SIG_SETMASK,
                &osigs, NULL);

    /* Step 8,9: */
    mctx_switch(&mctx_caller, mctx);

    /* Step 14: */
    return mctx;
}

void mctx_create_trampoline(int sig)
{
    /* Step 6: */
    if (mctx_save(mctx_creat) == 0) {
        mctx_called = TRUE;
        return;
    }

    /* Step 10: */
    mctx_create_boot();
}

void mctx_create_boot(void)
{
    mctx_t *mctx_start;
    void (*mctx_start_func)(void *);
    void *mctx_start_arg;

    /* Step 11: */
    sigprocmask(SIG_SETMASK,
                &mctx_creat_sigs,
                NULL);

    /* Step 12: */
    mctx_start      = mctx_creat;
    mctx_start_func = mctx_creat_func;
    mctx_start_arg  = mctx_creat_arg;

    /* Step 13: */
    mctx_switch(mctx_start,
                &mctx_caller);

    /* The thread ``magically'' starts... */
    mctx_start_func(mctx_start_arg);
    abort();
}
```

3.4 Remaining Issues

The presented algorithm and source code can be directly used in practice for implementing a minimal threading system or the concept of co-routines. Its big advantage is, that if the operating system provides the required POSIX primitives, we do not need to know anything at all about the machine we are running on — everything just works. Nevertheless, there remain a few special issues we still have to discuss.

The Waggly longjmp(3) after Return

On some platforms, `longjmp`(3) may not be called after the function which called the `setjmp`(3) returned. When this is done, the stack frame situation is not guaranteed to be in a clean and consistent state. But this is exactly on what our trick is based to get rid of the signal handler scope, when you remember.

The only alternative would be to leave the signal handler via `longjmp`(3), but then we would have another problem as experience showed. For instance, ROBERT S. THAU's *Really Simple Threads* (rsthreads) [21] was ported to several platforms and was used to run an experimental multithreaded version of the Apache web server. THAU's approach differed mainly in the way the signal handler is left. In particular, in an attempt to avoid the unsafe stack frame situation, it uses a `longjmp`(3) call to leave the signal handler, rather than returning from it. But this approach does not work on some System-V-derived kernels, as we already mentioned.

The problem is that these kernels do not "believe" that the code is out of the signal-handling context until the signal handler has returned — and accordingly, refuse to allow readjustment of the signal stack until it has. But with the rsthreads approach, the signal handler for the creation of the first thread never returns, and when rsthreads wants to create the second thread, these kernels refuse to readjust the signal stack, and we are stuck. So with portability in mind, we decided, that it is better to get rid of the signal handler scope with the straight-forward "`return`" and instead fight the mentioned (simpler) stack frame problem.

Fortunately in practice this is not as problematic as it seems, because evaluations on mostly all current Unix platforms for *GNU Pth* showed, that one can reach a safe stack frame situation again by just calling a function. That is the reason why our algorithm enters the second trampoline function.

The Uncooperative longjmp(3)

Even on operating systems which have working POSIX functions, our approach may theoretically still not work, because `longjmp`(3) does not cooperate. For instance, on some platforms the standard *libc* `longjmp`(3) branches to error-handling code if it detects that the caller tries to jump *up* the stack, i.e., into a stack frame that has already returned.

This is usually implemented by comparing the current stack pointer to the one in the `jmp_buf`. That is why it is important for our algorithm to return from the signal handler and this way enter the (different) stack of the parent. In practice, the implementation in *GNU Pth* showed, that then one no longer suffers from those uncooperative `longjmp`(3) implementations, but one should keep this point in mind when reaching even more uncooperative variants on esoteric Unix platforms. If it still occurs, one can only try to resume the operation by using a possibly existing platform-specific error handling hook.

Garbage at Bottom of Stacks

When you carefully followed the algorithms control flow, you certainly have recognized a subtle side-effect: there remains some garbage at the bottom of each thread stack. The reason is, that when a signal is delivered, the operating system pushes some state onto the stack, which is restored later, when the signal handler returns. But although we return from the signal handler, we jump in again, and this time we enter not directly at the bottom of the stack, because of the `setjmp`(3) in the trampoline function.

Since the operating system has to capture all CPU registers (including those that are ordinarily scratch registers or caller-save registers), there can be a fair amount of memory at the bottom of the established thread stack. For some systems this can be even up to 1 KB of garbage [21]. But except for the additional memory consumption it does not hurt, of course.

We just have to remember this additional stack consumption when deciding the stack size (*sksize*). A reasonable stack size usually is between 16 and 32 KB. Less is neither reasonable nor always allowed (current Unix platforms usually require a stack to be at least 16 KB in size).

Stack Overflows

There is a noticeable difference between the initial "thread" and the explicitly spawned threads: the initial thread runs on the standard process stack. This stack automatically can grow under Unix, while the stacks of the spawned threads are fixed in size. So stack overflows can occur for the spawned threads, of course. This implies that the parent has to make a reasonable guess of the threads stack space requirement already at spawning time.

And there is no really portable solution to this problem, because even when the thread library's scheduler can detect the stack overflow, it cannot easily resize the stack. The reason is simply that the stack initialization goes hand in hand with the initialization of the start routine, when you remember. And this start routine has to be a real C function in order to *call*. But once the thread is running, there no longer exists such an entry point. So, even when the scheduler would be able to give the thread a new enlarged stack, there is no chance to restart the thread on this new stack.

Or more correct, there is no *portable* chance. As with the problems before, when one accept platform specifics, there is a chance, of course. That's why our implementation did not deal with this issue. Instead in practice one usually lets the scheduler just detect the stack overflow and terminate the thread. This is done by using a red zone at the top of the stack which is marked with a magic value the scheduler checks between thread dispatching operations.

More advanced solutions are only possible in semi-portable ways. One approach is to place the thread stacks into shared memory segments and let the scheduler catch `SIGSEGV` signals. When such a signal occurs, because of a stack overflow, the scheduler manually resizes the mapped memory region. Or even more elegant, as *Solaris*, *FreeBSD* and *Tru64* do it: the scheduler allocates the threads inside shared memory segments which are even a few MB in (virtual) size and just relies on the virtual memory system's feature of actually mapping only the actually consumed stack space.

Startup Routine Termination

When you have read the proposed implementation carefully, you should have also recognized a cruel `abort`(3) call at the end of `mctx_trampoline_jumpin`. This means, when the startup routine returns, the process is aborted. That is obviously not reasonable, so why have we written it this way?

When the thread returns from the startup routine, it should be cleanly terminated, of course. But it cannot terminate itself (for instance, because it cannot free its own stack while running on it, etc). The termination handling actually is the task of the thread libraries scheduler. Usually the thread spawning function of the thread library is not directly `mctx_create`.

Instead the thread spawning function uses an additional trampoline function as the startup routine. And this trampoline function performs a context switch back into the thread library scheduler when the real thread startup routine returned. The scheduler then can safely remove the thread and its machine context. That is why the `abort`(3) call is never reached in practice.

The sigstack(2) Fallback Situation

Not all platforms provide POSIX `sigaltstack`(2). Instead they at least provide the 4.2BSD ancestor function `sigstack`(2). But one cannot trivially replace `sigaltstack`(2) by `sigstack`(2) in this situation, because in contrast to `sigaltstack`(2) the old `sigstack`(2) does not automatically handle the machine dependent direction of stack growth.

Instead, the caller has to know the direction and always call `sigstack`(2) with the address of the bottom of the stack. So, in a real-world implementation (like *GNU Pth*) one first has to determine the direction of stack growth in order to use `sigstack`(2) as a replacement for `sigaltstack`(2). Fortunately this is easier than it seems on the first look. [6]

The Blind Alley of Brain-Dead Platforms

Finally the world would not be as funny as it is, if really all Unix platforms would be fair to us. Instead, currently at least one platform exists which plays unfair: unfortunately the popular *GNU/Linux*. Although we will discover that it both provides `sigaltstack`(2) and `sigstack`(2), our approach won't work on *Linux* kernels prior to version 2.2 and *glibc* prior to version 2.1.

Why? Because its *libc* provides only dummies of these functions which always return just `-1` with `errno` set to `ENOSYS`. So, this definitely means that our nifty algorithm is useless there, because its central point *is* `sigaltstack`(2)/`sigstack`(2). Nevertheless we do not need to give up, of course. At least not, if we, for a single brain-dead platform, accept to break our general goal of not using any platform dependent things.

So, what can we actually do here? All we have to do, is to fiddle around a little bit with the machine-dependent `jmp_buf` ingredients (by poking around in `setjmp.h` or by disassembling `longjmp`(3) in the debugger). Usually one just has do a `setjmp`(3) to get an initial state in the `jmp_buf` structure and then manually adjust two of its ingredients: the program counter (usually a structure member with "`pc`" in the name) and the stack pointer (usually a structure member with "`sp`" in the name).

That's all and can be acceptable for a real-world implementation which really wants to cover all platforms – at least as long as the special treatment is needed just for one or two platforms. But one has to keep in mind that it at least breaks one of the initial goals and has to be treated as a last chance solution, of course.

More Machine Context Ingredients

Finally for a real-world threading implementation one usually want to put more ingredients into the machine state **mctx_t** structure, of course. For instance one usually want per-thread signal masks in order to fulfill the POSIX.1c signal handling requirements for threads. When `sigsetjmp`(3) and `siglongjmp`(3) exists one can just use them instead of `setjmp`(3) and `longjmp`(2)

[6] See the file `aclocal.m4` in the *GNU Pth* distribution for details

to achieve this. There is only one snare: on some platforms `sigsetjmp/siglongjmp`(3) save also information about the alternate signals stack. So here one has to make sure that even the thread dispatching later can use `sigsetjmp/siglongjmp`(3), the thread creation step has to use plain `setjmp/longjmp`(3) for the trampoline trick.

And if `sigsetjmp/siglongjmp`(3) do not exist, one can easily emulate them by saving and restoring the thread signal masks manually via `sigprocmask`(2) in `mctx_switch`. One just has to be a little bit carefully to prevent race conditions at dispatching.

Additionally it is reasonable to also save and restore the global `errno` variable. For a sophisticated implementation which does also all those additional things, have a look at the source file `pth_mctx.c` in the *GNU Pth* distribution [19].

3.5 Summary

We have presented a pragmatic and mostly portable fallback approach for implementing user-space multithreading, based entirely on Unix system and ANSI C language facilities. The approach was successfully tested in practice on a wide range of Unix flavors by *GNU Pth* and should also adapt to the remaining Unix platforms as long as they behave mostly identical to the current major Unix platforms.

3.6 Acknowledgements

I would like to thank Robert S. Thau, David Butenhof, Martin Kraemer and Eric Newton for their comments which helped to improve this paper. Additional thanks go to the users of *GNU Pth* for their feedback on the implementation which helped a lot in fine-tuning the presented approach. [rse]

References

[1] M.E. Conway: *Design of a separable transition-diagram compiler.*, Comm. ACM 6:7, 1963, p.396-408

[2] E.W. Dijkstra: *Co-operating sequential processes*, in F. Genuys (Ed.), *Programming Languages*, NATO Advanced Study Institute, Academic Press, London, 1965, p.42-112.

[3] B. Nichols, D. Buttlar, J.P. Farrel: *Pthreads Programming - A POSIX Standard for Better Multiprocessing*, O'Reilly 1996; ISBN 1-56592-115-1

[4] B. Lewis, D. J. Berg: *Threads Primer - A Guide To Multithreaded Programming*, Prentice Hall 1996; ISBN 0-13-443698-9

[5] S. J. Norton, M. D. Dipasquale: *Thread Time - The Multithreaded Programming Guide*, Prentice Hall 1997; ISBN 0-13-190067-6

[6] D. R. Butenhof: *Programming with POSIX Threads*, Addison Wesley 1997; ISBN 0-201-63392-2

[7] W.R.Stevens: *Advanced Programming in the Unix Environment*, Addison-Wesley, 1992; ISBN 0-201-56317-7

[8] D. Lewine: *POSIX Programmer's Guide: Writing Portable Unix Programs*, O'Reilly & Associates,Inc., 1994; ISBN 0-937175-73-0

[9] Bryan O'Sullivan: *Frequently asked questions for comp.os.research*, 1995; http://www.serpentine.com/~bos/os-faq/, ftp://rtfm.mit.edu/pub/usenet/comp.os.research/

[10] Sun Microsystems, Inc: *Threads Frequently Asked Questions*, 1995, http://www.sun.com/workshop/-threads/faq.html

[11] Bryan O'Sullivan: *Frequently asked questions for comp.programming.threads*, 1997; http://www.serpentine.com/~bos/threads-faq/.

[12] Bil Lewis: *Frequently asked questions for comp.programming.threads*, 1999; http://www.lambdacs.com/newsgroup/FAQ.html

[13] Numeric Quest Inc: *Multithreading - Definitions and Guidelines*; 1998; http://www.numeric-quest.com/lang/multi-frame.html

[14] The Open Group: *The Single Unix Specification, Version 2 - Threads*; 1997; http://www.opengroup-.org/onlinepubs/007908799/xsh/threads.html

[15] Sun Microsystems Inc: *SMI Thread Resources*; http://www.sun.com/workshop/threads

[16] Frank Mueller: *FSU pthreads*; 1997; http://www-.cs.fsu.edu/~mueller/pthreads/

[17] Chris Provenzano: *MIT pthreads*; 1993; http://www.mit.edu/people/proven/pthreads.html (old), http://www.humanfactor.com/pthreads/mit-pthreads.html (updated)

[18] Kota Abe: *Portable Threading Library* (PTL); 1999; http://www.media.osaka-cu.ac.jp/~k-abe/PTL/

[19] Ralf S. Engelschall: *GNU Portable Threads* (Pth); 1999; http://www.gnu.org/software/pth/, ftp://ftp.gnu-.org/gnu/pth/

[20] Michael T. Peterson: *POSIX and DCE Threads For Linux* (PCThreads); 1995; http://members.aa-.net/~mtp/PCthreads.html

[21] Robert S. Thau: *Really Simple Threads* (rsthreads); 1996; ftp://ftp.ai.mit.edu/pub/rst/

[22] JOHN BIRRELL: *FreeBSD uthreads*; 1998; ftp://ftp.freebsd.org/pub/FreeBSD/FreeBSD-current/src/lib/libc_r/uthread/

[23] XAVIER LEROY: *The LinuxThreads library*; 1999; http://pauillac.inria.fr/˜xleroy/linuxthreads/

[24] IBM: *AIX Version 4.3 General Programming Concepts: Writing and Debugging Programs; Understanding Threads*; 1998; http://www.rs6000.ibm.com/doc_link/-en_US/a_doc_lib/aixprggd/genprogc/understanding-_threads.htm