Using TCP Through Sockets

David Mazières dm@cs.nyu.edu
Revised by Frank Dabek frank@mit.edu
and Eric Peterson ericp@mit.edu

1 Introduction

This document provides an introduction to using sockets on Unix systems with a focus on asynchronous I/O. We first cover the basics of Unix file I/O and then discuss the use of sockets to communicate over a network using TCP. Section three describes the use of non-blocking I/O to improve performance and presents, as an example, an asynchronous finger client. Finally, a library that simplifies the use of asynchronous sockets is presented.

Readers of this document are assumed to be proficient in the C and C++ programming languages including template classes and be comfortable using a Unix operating system to develop programs. Detailed knowledge of network standards or Unix I/O is not required.

Noting the following typesetting conventions may assist the reader: words set in sans serif type (such as open) are Unix system calls; any text set in a mono spaced font (such as `nbytes`) is either a C code fragment or a C object or name.

2 File descriptors

Most I/O on Unix systems takes place through the read and write system calls[1]. Before discussing network I/O, it helps to understand how these functions work even on simple files. If you are already familiar with file descriptors and the read and write system calls, you can skip to the next section.

All read and write operations must be performed on file descriptors, non-negative integers which are created via a call to open (see below). File descriptors remain bound to files even when files are renamed or deleted or undergo permission changes that revoke access[2].

By convention, file descriptors numbers 0, 1, and 2 correspond to standard input, standard output, and standard error respectively. Thus a call to printf will result in a write to file descriptor 1.

Section ?? shows a very simple program that prints the contents of files to the standard output—just like the UNIX cat command. The function `typefile` uses four system calls to copy the contents of a file to the standard output.

[1]High-level I/O functions such as fread and fprintf are implemented in terms of read and write.
[2]Note that not all network file systems properly implement these semantics.

- `int open(char *path, int flags, ...);`

  The open system call requests access to a particular file. `path` specifies the name of the file to access; `flags` determines the type of access being requested—in the case of this example read-only access. open ensures that the named file exists (or can be created, depending on `flags`) and checks that the invoking user has sufficient permission for the mode of access.

  If successful, open returns a file descriptor.

  If unsuccessful, open returns −1 and sets the global variable errno to indicate the nature of the error. The routine perror will print "filename: error message" to the standard error based on errno.

- `int read (int fd, void *buf, int nbytes);`

  read will read up to `nbytes` bytes of data into memory starting at `buf`. It returns the number of bytes actually read, which may very well be less than `nbytes`. The case in which read returns fewer than `nbytes` is often called a "short read" and is a common source of errors. If read returns 0, this indicates an end of file. If it returns −1, this indicates an error.

- `int write (int fd, void *buf, int nbytes);`

  write will write up to `nbytes` bytes of data at `buf` to file descriptor `fd`. It returns the number of bytes actually written, which unfortunately may be less than `nbytes` if the file descriptor is non-blocking (see Section ??). Write returns −1 to indicate an error.

- `int close (int fd);`

  close deallocates a file descriptor. Systems typically limit each process to 64 file descriptors by default (though the limit can sometimes be raised substantially with the setrlimit system call). Thus, it is a good idea to close file descriptors after their last use so as to prevent "too many open files" errors.

2.1 `type.c`: Copy file to standard output

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

void
typefile (char *filename)
{
  int fd, nread;
  char buf[1024];

  fd = open (filename, O_RDONLY);
  if (fd == -1) {
    perror (filename);
    return;
  }
```

```
  while ((nread = read (fd, buf, sizeof (buf))) > 0)
    write (1, buf, nread);

  close (fd);
}

int
main (int argc, char **argv)
{
  int argno;
  for (argno = 1; argno < argc; argno++)
    typefile (argv[argno]);
  exit (0);
}
```

3 TCP/IP Connections

3.1 Introduction

TCP is the reliable protocol many applications use to communicate over the Internet. TCP provides a stream abstraction: Two processes, possibly on different machines, each have a file descriptor. Data written to either descriptor will be returned by a read from the other. Such network file descriptors are called sockets in Unix.

Every machine on the Internet has a unique, 32-bit IP (Internet protocol) address[3]. An IP address is sufficient to route network packets to a machine from anywhere on the Internet. However, since multiple applications can use TCP simultaneously on the same machine, another level of addressing is needed to disambiguate which process and file descriptor incoming TCP packets correspond to. For this reason, each end of a TCP connection is named by 16-bit port number in addition to its 32-bit IP address.

So how does a TCP connection get set up? Typically, a server will listen for connections on an IP address and port number. Clients can then allocate their own ports and connect to that server. Servers usually listen on well-known ports. For instance, finger servers listen on port 79, web servers on port 80 and mail servers on port 25. A list of well-known port numbers can be found in the file `/etc/services` on any Unix machine.

The Unix `telnet` utility will allow to you connect to TCP servers and interact with them. By default, `telnet` connects to port 23 and speaks to a `telnet` daemon that runs login. However, you can specify a different port number. For instance, port 7 on many machines runs a TCP echo server:

```
athena% telnet athena.dialup.mit.edu 7
...including Athena's default telnet options: "-ax"
Trying 18.184.0.39...
Connected to ten-thousand-dollar-bill.dialup.mit.edu.
Escape character is '^]'.
```

[3]This discussion will assume IP version 4. Version 6 of IP significantly expands the address space

```
repeat after me...
repeat after me...
The echo server works!
The echo server works!
quit
quit
^]
telnet> q
Connection closed.
athena%
```

Note that in order to quit telnet, you must type Control-] followed by q and return. The echo server will happily echo anything you type like quit.

As another example, let's look at the finger protocol, one of the simplest widely used TCP protocols. The Unix `finger` command takes a single argument of the form *user@host*. It then connects to port 79 of *host*, writes the *user* string and a carriage-return line-feed over the connection, and dumps whatever the server sends back to the standard output. We can simulate the `finger` command using `telnet`. For instance, using `telnet` to do the equivalent of the command `finger help@mit.edu`, we get:

```
athena% telnet mit.edu 79
...including Athena's default telnet options: "-ax"
Trying 18.72.0.100...
Connected to mit.edu.
Escape character is '^]'.
help
These help topics are available:

about           general         options         restrictions    url
change-info     motd            policy          services        wildcards

To view one of these topics, enter "help name-of-topic-you-want".
...
Connection closed by foreign host.
athena%
```

3.2 TCP client programming

Now let's see how to make use of sockets in C. Section **??** shows the source code to a simple finger client that does the equivalent of the last `telnet` example of the previous section. In general, a client wishing to create a TCP connection to a server first calls socket to create a socket, optionally calls bind to specify a local address, and finally connects to the server using the connect system call.

The function `tcpconnect` shows all the steps necessary to connect to a TCP server. It makes the following system calls:

- `int socket (int domain, int type, int protocol);`

  The socket system call creates a new socket, just as open creates a new file descriptor. socket returns a non-negative file descriptor number on success, or −1 on an error.

  When creating a TCP socket, `domain` should be `AF_INET`, signifying an IP socket, and `type` should be `SOCK_STREAM`, signifying a reliable stream. Since the reliable stream protocol for IP is TCP, the first two arguments already effectively specify TCP. Thus, the third argument can be left 0, letting the Operating System assign a default protocol (which will be `IPPROTO_TCP`).

  Unlike file descriptors returned by open, you can't immediately read and write data to a socket returned by socket. You must first assign the socket a local IP address and port number, and in the case of TCP you need to connect the other end of the socket to a remote machine. The bind and connect system calls accomplish these tasks.

- `int bind (int s, struct sockaddr *addr, int addrlen);`

  bind sets the local address and port number of a socket. `s` is the file descriptor number of a socket. For IP sockets, `addr` must be a structure of type `sockaddr_in`, usually as follows (in `/usr/include/netinet/in.h`). `addrlen` must be the size of `struct sockaddr_in` (or whichever structure one is using).

```
struct in_addr {
   u_int32_t s_addr;
};

struct sockaddr_in {
   short sin_family;
   u_short sin_port;
   struct in_addr sin_addr;
   char sin_zero[8];
};
```

  Different versions of Unix may have slightly different structures. However, all will have the fields `sin_family`, `sin_port`, and `sin_addr`. All other fields should be set to zero. Thus, before using a `struct sockaddr_in`, you must call `bzero` on it, as is done in `tcpconnect`. Once a `struct sockaddr_in` has been zeroed, the `sin_family` field must be set to the value `AF_INET` to indicate that this is indeed a `sockaddr_in`. (Bind cannot take this for granted, as its argument is a more generic `struct sockaddr *`.)

  `sin_port` specifies which 16-bit port number to use. It is given in network (big-endian) byte order[4], and so must be converted from host to network byte order with `htons`. It is often the case when writing a TCP client that one wants a port number but

[4] To avoid ambiguity, the socket standard specifies a 'network' byte order (which happens to be big-endian). The calls htons and ntohs convert shorts back and forth between byte orders. These calls are macros and on big-endian machines have no effect. To ensure portability, however, they should always be used.

doesn't care which one. Specifying a `sin_port` value of 0 tells the OS to choose the port number. The operating system will select an unused port number between 1,024 and 5,000 for the application. Note that only the super-user can bind port numbers under 1,024. Many system services such as mail servers listen for connections on well-known port numbers below 1,024. Allowing ordinary users to bind these ports would potentially also allow them to do things like intercept mail with their own rogue mail servers.

`sin_addr` contains a 32-bit IP address for the local end of a socket. The special value `INADDR_ANY` tells the operating system to choose the IP address. This is usually what one wants when binding a socket, since one typically does not care about the IP address of the machine on which it is running.

- `int connect (int s, struct sockaddr *addr, int addrlen);`

  connect specifies the address of the remote end of a socket. The arguments are the same as for bind, with the exception that one cannot specify a port number of 0 or an IP address of `INADDR_ANY`. Connect returns 0 on success or -1 on failure.

  Note that one can call connect on a TCP socket without first calling bind. In that case, connect will assign the socket a local address as if the socket had been bound to port number 0 with address `INADDR_ANY`. The example finger calls bind for illustrative purposes only.

The above three system calls create a connected TCP socket, over which the finger program writes the name of the user being fingered and reads the response. Most of the rest of the code should be straight-forward, except you might wish to note the use of `gethostbyname` to translate a hostname into a 32-bit IP address.

3.3 `myfinger.c`: A simple network finger client

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define FINGER_PORT 79
#define bzero(ptr, size) memset (ptr, 0, size)

/* Create a TCP connection to host and port.  Returns a file
 * descriptor on success, -1 on error. */
int
tcpconnect (char *host, int port)
{
  struct hostent *h;
  struct sockaddr_in sa;
```

```
  int s;

  /* Get the address of the host at which to finger from the
   * hostname.  */
  h = gethostbyname (host);
  if (!h || h->h_length != sizeof (struct in_addr)) {
    fprintf (stderr, "%s: no such host\n", host);
    return -1;
  }

  /* Create a TCP socket. */
  s = socket (AF_INET, SOCK_STREAM, 0);

  /* Use bind to set an address and port number for our end of the
   * finger TCP connection. */
  bzero (&sa, sizeof (sa));
  sa.sin_family = AF_INET;
  sa.sin_port = htons (0);                  /* tells OS to choose a port */
  sa.sin_addr.s_addr = htonl (INADDR_ANY);  /* tells OS to choose IP addr */
  if (bind (s, (struct sockaddr *) &sa, sizeof (sa)) < 0) {
    perror ("bind");
    close (s);
    return -1;
  }

  /* Now use h to set set the destination address. */
  sa.sin_port = htons (port);
  sa.sin_addr = *(struct in_addr *) h->h_addr;

  /* And connect to the server */
  if (connect (s, (struct sockaddr *) &sa, sizeof (sa)) < 0) {
    perror (host);
    close (s);
    return -1;
  }

  return s;
}

int
main (int argc, char **argv)
{
  char *user;
  char *host;
  int s;
  int nread;
  char buf[1024];

  /* Get the name of the host at which to finger from the end of the
   * command line argument. */
  if (argc == 2) {
    user = malloc (1 + strlen (argv[1]));
    if (!user) {
```

```
      fprintf (stderr, "out of memory\n");
      exit (1);
    }
    strcpy (user, argv[1]);
    host = strrchr (user, '@');
  }
  else
    user = host = NULL;
  if (!host) {
    fprintf (stderr, "usage: %s user@host\n", argv[0]);
    exit (1);
  }
  *host++ = '\0';

  /* Try connecting to the host. */
  s = tcpconnect (host, FINGER_PORT);
  if (s < 0)
    exit (1);

  /* Send the username to finger */
  if (write (s, user, strlen (user)) < 0
      || write (s, "\r\n", 2) < 0) {
    perror (host);
    exit (1);
  }

  /* Now copy the result of the finger command to stdout. */
  while ((nread = read (s, buf, sizeof (buf))) > 0)
    write (1, buf, nread);

  exit (0);
}
```

3.4 TCP server programming

Now let's look at what happens in a TCP server. A TCP server, like a client, begins by calling socket to create a socket and by binding the socket to a well-known port using bind (although optional for clients, servers nearly always call bind to specify the port on which they will operate). Following the bind operation, server and client paths diverge: instead of connecting the socket, a server will instead call listen followed by accept. These functions, described below, alert the operating system to accept new connects and, for each connection, create a new, connected socket which will be returned by accept.

Section **??** shows the complete source code to a simple finger server. It listens for clients on the finger port, 79. Then, for each connection established, it reads a line of data, interprets it as the name of a user to finger, and runs the local finger utility directing its output back over the socket to the client.

The function `tcpserv` takes a port number as an argument, binds a socket to that port, tells the kernel to listen for TCP connections on that socket, and returns the socket file descriptor number, or -1 on an error. This requires three main system calls:

- `int socket (int domain, int type, int protocol);`

  This function creates a socket, as described in Section ??.

- `int bind (int s, struct sockaddr *addr, int addrlen);`

  This function assigns an address to a socket, as described in Section ??. Unlike the finger client, which did not care about its local port number, here we specify a specific port number. `INADDR_ANY` can still be specified as the local IP address: on a multi-homed machine the socket will accept connections on any of the server's addresses.

  Binding a specific port number can cause complications when killing and restarting servers (for instance during debugging). Closed TCP connections can sit for a while in a state called `TIME_WAIT` before disappearing entirely. This can prevent a restarted TCP server from binding the same port number again, even if the old process no longer exists. The `setsockopt` system call shown in `tcpserv` avoids this problem. It tells the operating system to let the socket be bound to a port number already in use.

- `int listen (int s, int backlog);`

  `listen` tells the operating system to accept network connections. It returns 0 on success, and −1 on error. `s` is an unconnected socket bound to the port on which to accept connections. `backlog` formerly specified the number of connections the operating system would accept ahead of the application. That argument is ignored by most modern Unix operating systems, however. People traditionally use the value 5.

  Once you have called `listen` on a socket, you cannot call `connect`, `read`, or `write`, as the socket has no remote end. Instead, a new system call, `accept`, creates a new socket for each client connecting to the port `s` is bound to.

Once `tcpserv` has begun listening on a socket, `main` accepts connections from clients, with the system call `accept`.

- `int accept (int s, struct sockaddr *addr, int *addrlenp);`

  Accept takes a socket `s` on which one is listening and returns a new socket to which a client has just connected. If no clients have connected, `accept` will block until one does. `accept` returns −1 on an error.

  For TCP, `addr` should be a `struct sockaddr_in *`. `addrlenp` must be a pointer to an integer containing the value `sizeof (struct sockaddr_in)`. `accept` will adjust `*addrlenp` to contain the actual length of the `struct sockaddr` it copies into `*addr`. In the case of TCP, all `struct sockaddr_in`'s are the same size, so `*addrlenp` shouldn't change.

The finger daemon makes use of a few more Unix system calls which, while not network-specific, are often encountered in network servers. With `fork` it creates a new process. This new process calls `dup2` to redirect its standard output and error over the accepted socket. Finally, a call to `execl` replaces the new process with an instance of the finger program. Finger inherits its standard output and error, so these go straight back over the network to the client.

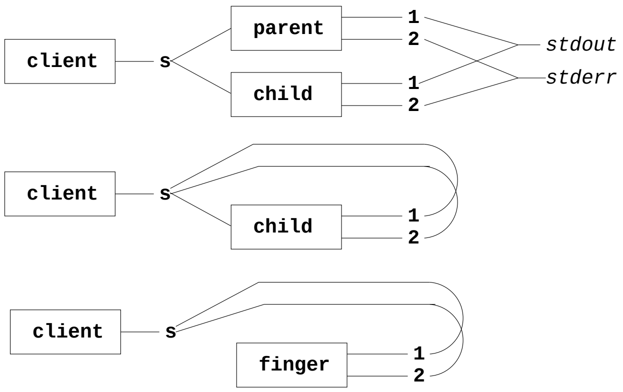

Figure 1: **file descriptors in a forking server** are inherited by the child process (top figure). In the second figure, the child process has called dup2 to connect file descriptors 1 and 2 which normally correspond to standard output and standard error to the network socket. In the final figure, the child has called execl and the local finger process runs, sending its standard output over s.

- `int fork (void);`

  fork creates a new process, identical to the current one. fork returns twice: in the old process, fork returns a process ID of the new process. In the new or "child" process, fork returns 0. fork returns -1 if there is an error.

- `int dup2(int oldfd, int newfd);`

  dup2 closes file descriptor number `newfd`, and replaces it with a copy of `oldfd`. When the second argument is 1, this changes the destination of the standard output. When that argument is 2, it changes the standard error.

- `int execl(char *path, char *arg0, ..., NULL);`

  The execl system call runs a command—as if you had typed it on the command line. The command executed will inherit all file descriptors except those with the close-on-exec flag set. execl replaces the currently executing program with the one specified by `path`. On success, it therefore doesn't return. On failure, it returns -1.

3.5 Closing a socket

If the close system call is passed the only remaining file descriptor reference to a socket, communication in both directions will be closed. If another reference to the socket exists (even in another process), communications are unaffected over the remaining descriptors. It is sometimes convenient to transmit an end-of-file over a socket without closing the socket—either because not all descriptors can be closed, or because one wishes to read from the socket even after writing an end-of-file.

Consider, for example, a protocol in which a client sends a single query and then receives a response from the server. The client might signal the end of the query with an end-of-file—effectively closing the write half of its TCP connection. Once the server receives the end-of-file, it parses and responds to the query. The client must still be able to read from the socket even after sending an end of file. It can do so using the shutdown system call.

- `int shutdown (int fd, int how);`

  shutdown shuts down communications over a socket in one or both directions, without deallocating the file descriptor and regardless of how many other file descriptor references there are to the socket. The argument `how` can be either 0, 1, or 2. 0 shuts down the socket for reading, 1 for writing, and 2 for both. When a TCP socket is shut down for writing, the process at the other end of the socket will see a 0-length read, indicating end-of-file, but data can continue to flow in the other direction.

  The TCP protocol has no way of indicating to the remote end that a socket has been shut down for reading. Thus, it is almost never useful tu call shutdown on a TCP socket with a `how` argument of 0 or 2.

3.6 `myfingerd.c`: A simple network finger server

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>
#include <signal.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define FINGER_PORT 79
#define FINGER_COMMAND "/usr/bin/finger"
#define bzero(ptr, size) memset (ptr, 0, size)

/* Create a TCP socket, bind it to a particular port, and call listen
 * for connections on it.  These are the three steps necessary before
 * clients can connect to a server.  */
int
tcpserv (int port)
{
  int s, n;
  struct sockaddr_in sin;

  /* The address of this server */
  bzero (&sin, sizeof (sin));
  sin.sin_family = AF_INET;
  sin.sin_port = htons (port);
  /* We are interested in listening on any and all IP addresses this
   * machine has, so use the magic IP address INADDR_ANY. */
  sin.sin_addr.s_addr = htonl (INADDR_ANY);
```

```
  s = socket (AF_INET, SOCK_STREAM, 0);
  if (s < 0) {
    perror ("socket");
    return -1;
  }

  /* Allow the program to run again even if there are old connections
   * in TIME_WAIT.  This is the magic you need to do to avoid seeing
   * "Address already in use" errors when you are killing and
   * restarting the daemon frequently.  */
  n = 1;
  if (setsockopt (s, SOL_SOCKET, SO_REUSEADDR, (char *)&n, sizeof (n)) < 0) {
    perror ("SO_REUSEADDR");
    close (s);
    return -1;
  }
  /* This function sets the close-on-exec bit of a file descriptor.
   * That way no programs we execute will inherit the TCP server file
   * descriptor. */
  fcntl (s, F_SETFD, 1);

  if (bind (s, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
    fprintf (stderr, "TCP port %d: %s\n", port, strerror (errno));
    close (s);
    return -1;
  }
  if (listen (s, 5) < 0) {
    perror ("listen");
    close (s);
    return -1;
  }

  return s;
}

/* Read a line of input from a file descriptor and return it.  Returns
 * NULL on EOF/error/out of memory.  May over-read, so don't use this
 * if there is useful data after the first line. */
static char *
readline (int s)
{
  char *buf = NULL, *nbuf;
  int buf_pos = 0, buf_len = 0;
  int i, n;

  for (;;) {
    /* Ensure there is room in the buffer */
    if (buf_pos == buf_len) {
      buf_len = buf_len ? buf_len << 1 : 4;
      nbuf = realloc (buf, buf_len);
      if (!nbuf) {
        free (buf);
        return NULL;
```

```
      }
      buf = nbuf;
    }

    /* Read some data into the buffer */
    n = read (s, buf + buf_pos, buf_len - buf_pos);
    if (n <= 0) {
      if (n < 0)
        perror ("read");
      else
        fprintf (stderr, "read: EOF\n");
      free (buf);
      return NULL;
    }

    /* Look for the end of a line, and return if we got it.  Be
     * generous in what we consider to be the end of a line. */
    for (i = buf_pos; i < buf_pos + n; i++)
      if (buf[i] == '\0' || buf[i] == '\r' || buf[i] == '\n') {
        buf[i] = '\0';
        return buf;
      }

    buf_pos += n;
  }
}

static void
runfinger (int s)
{
  char *user;

  /* Read the username being fingered. */
  user = readline (s);

  /* Now connect standard output and standard error to the socket,
   * instead of the invoking user's terminal. */
  if (dup2 (s, 1) < 0 || dup2 (s, 2) < 0) {
    perror ("dup2");
    exit (1);
  }
  close (s);

  /* Run the finger command.  It will inherit our standard output and
   * error, and therefore send its results back over the network. */
  execl (FINGER_COMMAND, "finger", "--", *user ? user : NULL, NULL);

  /* We should never get here, unless we couldn't run finger. */
  perror (FINGER_COMMAND);
  exit (1);
}

int
```

```
main (int argc, char **argv)
{
  int ss, cs;
  struct sockaddr_in sin;
  int sinlen;
  int pid;

  /* This system call allows one to call fork without worrying about
   * calling wait.  Don't worry about what it means unless you start
   * caring about the exit status of forked processes, in which case
   * you should delete this line and read the manual pages for wait
   * and waitpid.  For a description of what this signal call really
   * does, see the manual page for sigaction and look for
   * SA_NOCLDWAIT.  Signal is an older signal interface which when
   * invoked this way is equivalent to setting SA_NOCLDWAIT. */
  signal (SIGCHLD, SIG_IGN);

  ss = tcpserv (FINGER_PORT);
  if (ss < 0)
    exit (1);

  for (;;) {
    sinlen = sizeof (sin);
    cs = accept (ss, (struct sockaddr *) &sin, &sinlen);
    if (cs < 0) {
      perror ("accept");
      exit (1);
    }
    printf ("connection from %s\n", inet_ntoa (sin.sin_addr));

    pid = fork ();
    if (!pid)
      /* Child process */
      runfinger (cs);

    close (cs);
  }
}
```

4 Non-blocking I/O

4.1 The `O_NONBLOCK` flag

The finger client in Section **??** is only as fast as the server to which it talks. When the program calls connect, read, and sometimes even write, it must wait for a response from the server before making any further progress. This doesn't ordinarily pose a problem; if finger blocks, the operating system will schedule another process so the CPU can still perform useful work.

On the other hand, suppose you want to finger some huge number of users. Some servers may take a long time to respond (for instance, connection attempts to unreachable servers will take over a minute to time out). Thus, your program itself may have plenty of useful work to do, and you may not want to schedule another process every time a server is slow to respond.

For this reason, Unix allows file descriptors to be placed in a non-blocking mode. A bit associated with each file descriptor `O_NONBLOCK`, determines whether it is in non-blocking mode or not. Section **??** shows some utility functions for non-blocking I/O. The function `make_async` sets the `O_NONBLOCK` bit of a file descriptor non-blocking with the fcntl system call.

There are a number of ways to achieve this goal other than placing sockets in non-blocking mode. For instance, a threads package could be used to spawn a process for each request; requests would then be made in concurrently in blocking mode. An alternative, signal-driven asynchronous socket interface also exists in a number of Unix operating systems. The method described here, however, avoids the common pitfalls of preemptive concurrency and is more portable than signal driven methods.

Many system calls behave slightly differently on file descriptors which have `O_NONBLOCK` set:

- read. When there is data to read, read behaves as usual. When there is an end of file, read still returns 0. If, however, a process calls read on a non-blocking file descriptor when there is no data to be read yet, instead of waiting for data, read will return -1 and set `errno` to `EAGAIN`. In general, a read call to a non-blocking socket will return without error any time a call to a blocking socket would return immediately. Under this definition, data is available to be read when it has been loaded into a kernel buffer by the OS and is ready to be copied into the user-specified buffer.

- write. Like read, write will return -1 with an `errno` of `EAGAIN` if there is no buffer space available for the operating system to copy data to. If, however, there is some buffer space but not enough to contain the entire write request, write will take as much data as it can and return a value smaller than the length specified as its third argument. Code must handle such "short writes" by calling write again later on the rest of the data.

- connect. A TCP connection request requires a response from the listening server. When called on a non-blocking socket, connect cannot wait for such a response before returning. For this reason, connect on a non-blocking socket usually returns -1 with `errno` set to `EINPROGRESS`. Occasionally, however, connect succeeds or fails immediately even on a non-blocking socket, so you must be prepared to handle this case.

- accept. When there are connections to accept, accept will behave as usual. If there are no pending connections, however, accept will return -1 and set `errno` to `EWOULDBLOCK`. It's worth noting that, on some operating systems, file descriptors returned by accept have `O_NONBLOCK` clear, whether or not the listening socket is non-blocking. In a asynchronous servers, one often sets `O_NONBLOCK` immediately on any file descriptors accept returns.

4.2 select: Finding out when sockets are ready

O_NONBLOCK allows an application to keep the CPU when an I/O system call would ordinarily block. However, programs can use several non-blocking file descriptors and still find none of them ready for I/O. Under such circumstances, programs need a way to avoid wasting CPU time by repeatedly polling individual file descriptors. The select system call solves this problem by letting applications sleep until one or more file descriptors in a set is ready for I/O.

select usage

- `int select (int nfds, fd_set *rfds, fd_set *wfds, fd_set *efds,`
  `            struct timeval *timeout);`

  select takes pointers to sets of file descriptors and a timeout. It returns when one or more of the file descriptors are ready for I/O, or after the specified timeout. Before returning, select modifies the file descriptor sets so as to indicate which file descriptors actually are ready for I/O. select returns the number of ready file descriptors, or -1 on an error.

  select represents sets of file descriptors as bit vectors—one bit per descriptor. The first bit of a vector is 1 if that set contains file descriptor 0, the second bit is 1 if it contains descriptor 1, and so on. The argument `nfds` specifies the number of bits in each of the bit vectors being passed in. Equivalently, `nfds` is one more than highest file descriptor number select must check on.

  These file descriptor sets are of type `fd_set`. Several macros in system header files allow easy manipulation of this type. If `fd` is an integer containing a file descriptor, and `fds` is a variable of type `fd_set`, the following macros can manipulate `fds`:

  - `FD_ZERO (&fds);`
    Clears all bits in a `fds`.
  - `FD_SET (fd, &fds);`
    Sets the bit corresponding to file descriptor `fd` in `fds`.
  - `FD_CLR (fd, &fds);`
    Clears the bit corresponding to file descriptor `fd` in `fds`.
  - `FD_ISSET (fd, &fds);`
    Returns a true if and only if the bit for file descriptor `fd` is set in `fds`.

  select takes three file descriptor sets as input. `rfds` specifies the set of file descriptors on which the process would like to perform a read or accept. `wfds` specifies the set of file descriptors on which the process would like to perform a write. `efds` is a set of file descriptors for which the process is interested in exceptional events such as the arrival of out of band data. In practice, people rarely use `efds`. Any of the `fd_set *` arguments to select can be `NULL` to indicate an empty set.

The argument `timeout` specifies the amount of time to wait for a file descriptor to become ready. It is a pointer to a structure of the following form:

```
struct timeval {
   long tv_sec;          /* seconds */
   long tv_usec;         /* and microseconds */
};
```

`timeout` can also be `NULL`, in which case select will wait indefinitely.

Tips and subtleties

File descriptor limits. Programmers using select may be tempted to write code capable of using arbitrarily many file descriptors. Be aware that the operating system limits the number of file descriptors a process can have. If you don't bound the number of descriptors your program uses, you must be prepared for system calls like socket and accept to fail with errors like `EMFILE`. By default, a modern Unix system typically limits processes to 64 file descriptors (though the setrlimit system call can sometimes raise that limit substantially). Don't count on using all 64 file descriptors, either. All processes inherit at least three file descriptors (standard input, output, and error), and some C library functions need to use file descriptors, too. It should be safe to assume you can use 56 file descriptors, though.

If you do raise the maximum number of file descriptors allowed to your process, there is another problem to be aware of. The `fd_set` type defines a vector with `FD_SETSIZE` bits in it (typically 256). If your program uses more than `FD_SETSIZE` file descriptors, you must allocate more memory for each vector than than an `fd_set` contains, and you can no longer use the `FD_ZERO` macro.

Using select with connect. After connecting a non-blocking socket, you might like to know when the connect has completed and whether it succeeded or failed. TCP servers can accept connections without writing to them (for instance, our finger server waited to read a username before sending anything back over the socket). Thus, selecting for readability will not necessarily notify you of a connect's completion; you must check for writability.

When select does indicate the writability of a non-blocking socket with a pending connect, how can you tell if that connect succeeded? The simplest way is to try writing some data to the file descriptor to see if the write succeeds. This approach has two small complications. First, writing to an unconnected socket does more than simply return an error code; it kills the current process with a `SIGPIPE` signal. Thus, any program that risks writing to an unconnected socket should tell the operating system that it wants to ignore `SIGPIPE`. The signal system call accomplishes this:

```
signal (SIGPIPE, SIG_IGN);
```

The second complication is that you may not have any data to write to a socket, yet still wish to know if a non-blocking connect has succeeded. In that case, you can find out whether a socket is connected with the getpeername system call. getpeername takes the same argument types as accept, but expects a connected socket as its first argument. If getpeername returns

0 (meaning success), then you know the non-blocking connect has succeeded. If it returns -1, then the connect has failed.

4.3 Putting it all together

We now present an example that demonstrates the power of non-blocking socket I/O. Section ?? shows the source code to multifinger—an asynchronous finger client. When fingering many hosts, multifinger performs an order of magnitude better than a traditional Unix finger client. It connects to `NCON_MAX` hosts in parallel using non-blocking I/O. Some simple testing showed this client could finger 1,000 hosts in under 2 minutes.

The program presented here uses helper routines (async.c) to link readability and writability conditions to callbacks, a common technique. In the next section, we will present a robust implementation of an asynchronous socket library based on callbacks.

The simple callback-based socket library presented here implements three functions:

- `void cb_add (int fd, int write, void (*fn)(void *), void *arg);`

  Tells the dispatcher to call function `fn` with argument `arg` when file descriptor `fd` is ready for reading, if `write` is 0, or writing, otherwise.

- `void cb_free (int fd, int write);`

  Tells the dispatcher it should no longer call any function when file descriptor `fd` is ready for reading or writing (depending on the value of `write`).

- `void cb_check (void);`

  Wait until one or more of the registered file descriptors is ready, and make any appropriate callbacks.

The function `cb_add` maintains two `fd_set` variables, `rfds` for descriptors with read callbacks, and `wfds` for ones with write callbacks. It also records the function calls it needs to make in two arrays of `cb` ("callback") structures.

`cb_check` calls select on the file descriptors in `rfds` and `wfds`. Since select overwrites the `fd_set` structures it gets, `cb_check` must first copy the sets it is checking. `cb_check` then loops through the sets of ready descriptors making any appropriate function calls.

4.4 `async.c`: Helper routines for callback-based socket I/O

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdarg.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <assert.h>
#include <sys/socket.h>
```

```
#include "async.h"

/* Callback to make when a file descriptor is ready */
struct cb {
  void (*cb_fn) (void *);     /* Function to call */
  void *cb_arg;               /* Argument to pass function */
};
static struct cb rcb[FD_MAX], wcb[FD_MAX];  /* Per fd callbacks */
static fd_set rfds, wfds;                   /* Bitmap of cb's in use */

void
cb_add (int fd, int write, void (*fn)(void *), void *arg)
{
  struct cb *c;

  assert (fd >= 0 && fd < FD_MAX);
  c = &(write ? wcb : rcb)[fd];
  c->cb_fn = fn;
  c->cb_arg = arg;
  FD_SET (fd, write ? &wfds : &rfds);
}

void
cb_free (int fd, int write)
{
  assert (fd >= 0 && fd < FD_MAX);
  FD_CLR (fd, write ? &wfds : &rfds);
}

void
cb_check (void)
{
  fd_set trfds, twfds;
  int i, n;

  /* Call select.  Since the fd_sets are both input and output
   * arguments, we must copy rfds and wfds. */
  trfds = rfds;
  twfds = wfds;
  n = select (FD_MAX, &trfds, &twfds, NULL, NULL);
  if (n < 0)
    fatal ("select: %s\n", strerror (errno));

  /* Loop through and make callbacks for all ready file descriptors */
  for (i = 0; n && i < FD_MAX; i++) {
    if (FD_ISSET (i, &trfds)) {
      n--;
      /* Because any one of the callbacks we make might in turn call
       * cb_free on a higher numbered file descriptor, we want to make
       * sure each callback is wanted before we make it.  Hence check
       * rfds. */
      if (FD_ISSET (i, &rfds))
        rcb[i].cb_fn (rcb[i].cb_arg);
```

```
    }
    if (FD_ISSET (i, &twfds)) {
      n--;
      if (FD_ISSET (i, &wfds))
        wcb[i].cb_fn (wcb[i].cb_arg);
    }
  }
}

void
make_async (int s)
{
  int n;

  /* Make file file descriptor nonblocking. */
  if ((n = fcntl (s, F_GETFL)) < 0
      || fcntl (s, F_SETFL, n | O_NONBLOCK) < 0)
    fatal ("O_NONBLOCK: %s\n", strerror (errno));

  /* You can pretty much ignore the rest of this function... */

  /* Many asynchronous programming errors occur only when slow peers
   * trigger short writes.  To simulate this during testing, we set
   * the buffer size on the socket to 4 bytes.  This will ensure that
   * each read and write operation works on at most 4 bytes--a good
   * stress test. */
#if SMALL_LIMITS
#if defined (SO_RCVBUF) && defined (SO_SNDBUF)
  /* Make sure this really is a stream socket (like TCP).  Code using
   * datagram sockets will simply fail miserably if it can never
   * transmit a packet larger than 4 bytes. */
  {
    int sn = sizeof (n);
    if (getsockopt (s, SOL_SOCKET, SO_TYPE, (char *)&n, &sn) < 0
        || n != SOCK_STREAM)
      return;
  }

  n = 4;
  if (setsockopt (s, SOL_SOCKET, SO_RCVBUF, (void *)&n, sizeof (n)) < 0)
    return;
  if (setsockopt (s, SOL_SOCKET, SO_SNDBUF, (void *)&n, sizeof (n)) < 0)
    fatal ("SO_SNDBUF: %s\n", strerror (errno));
#else /* !SO_RCVBUF || !SO_SNDBUF */
#error "Need SO_RCVBUF/SO_SNDBUF for SMALL_LIMITS"
#endif /* SO_RCVBUF && SO_SNDBUF */
#endif /* SMALL_LIMITS */

  /* Enable keepalives to make sockets time out if servers go away. */
  n = 1;
  if (setsockopt (s, SOL_SOCKET, SO_KEEPALIVE, (void *) &n, sizeof (n)) < 0)
    fatal ("SO_KEEPALIVE: %s\n", strerror (errno));
}
```

```
void *
xrealloc (void *p, size_t size)
{
  p = realloc (p, size);
  if (size && !p)
    fatal ("out of memory\n");
  return p;
}

void
fatal (const char *msg, ...)
{
  va_list ap;

  fprintf (stderr, "fatal: ");
  va_start (ap, msg);
  vfprintf (stderr, msg, ap);
  va_end (ap);
  exit (1);
}
```

4.5 multifinger.c: A mostly[5] asynchronous finger client

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <netdb.h>
#include <signal.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>

#include "async.h"

#define FINGER_PORT 79
#define MAX_RESP_SIZE 16384

struct fcon {
  int fd;
  char *host;          /* Host to which we are connecting */
  char *user;          /* User to finger on that host */
  int user_len;        /* Lenght of the user string */
  int user_pos;        /* Number bytes of user already written to network */
  void *resp;          /* Finger response read from network */
  int resp_len;        /* Number of allocated bytes resp points to */
  int resp_pos;        /* Number of resp bytes used so far */
};

int ncon;              /* Number of open TCP connections */
```

[5] gethostbyname performs synchronous socket I/O.

```
static void
fcon_free (struct fcon *fc)
{
  if (fc->fd >= 0) {
    cb_free (fc->fd, 0);
    cb_free (fc->fd, 1);
    close (fc->fd);
    ncon--;
  }
  xfree (fc->host);
  xfree (fc->user);
  xfree (fc->resp);
  xfree (fc);
}

void
finger_done (struct fcon *fc)
{
  printf ("[%s]\n", fc->host);
  fwrite (fc->resp, 1, fc->resp_pos, stdout);
  fcon_free (fc);
}

static void
finger_getresp (void *_fc)
{
  struct fcon *fc = _fc;
  int n;

  if (fc->resp_pos == fc->resp_len) {
    fc->resp_len = fc->resp_len ? fc->resp_len << 1 : 512;
    if (fc->resp_len > MAX_RESP_SIZE) {
      fprintf (stderr, "%s: response too large\n", fc->host);
      fcon_free (fc);
      return;
    }
    fc->resp = xrealloc (fc->resp, fc->resp_len);
  }

  n = read (fc->fd, fc->resp + fc->resp_pos, fc->resp_len - fc->resp_pos);
  if (n == 0)
    finger_done (fc);
  else if (n < 0) {
    if (errno == EAGAIN)
      return;
    else
      perror (fc->host);
    fcon_free (fc);
    return;
  }

  fc->resp_pos += n;
```

```
}

static void
finger_senduser (void *_fc)
{
  struct fcon *fc = _fc;
  int n;

  n = write (fc->fd, fc->user + fc->user_pos, fc->user_len - fc->user_pos);
  if (n <= 0) {
    if (n == 0)
      fprintf (stderr, "%s: EOF\n", fc->host);
    else if (errno == EAGAIN)
      return;
    else
      perror (fc->host);
    fcon_free (fc);
    return;
  }

  fc->user_pos += n;
  if (fc->user_pos == fc->user_len) {
    cb_free (fc->fd, 1);
    cb_add (fc->fd, 0, finger_getresp, fc);
  }
}

static void
finger (char *arg)
{
  struct fcon *fc;
  char *p;
  struct hostent *h;
  struct sockaddr_in sin;

  p = strrchr (arg, '@');
  if (!p) {
    fprintf (stderr, "%s: ignored -- not of form 'user@host'\n", arg);
    return;
  }

  fc = xmalloc (sizeof (*fc));
  bzero (fc, sizeof (*fc));

  fc->fd = -1;
  fc->host = xmalloc (strlen (p));
  strcpy (fc->host, p + 1);
  fc->user_len = p - arg + 2;
  fc->user = xmalloc (fc->user_len + 1);
  memcpy (fc->user, arg, fc->user_len - 2);
  memcpy (fc->user + fc->user_len - 2, "\r\n", 3);

  h = gethostbyname (fc->host);
```

```
  if (!h) {
    fprintf (stderr, "%s: hostname lookup failed\n", fc->host);
    fcon_free (fc);
    return;
  }

  fc->fd = socket (AF_INET, SOCK_STREAM, 0);
  if (fc->fd < 0)
    fatal ("socket: %s\n", strerror (errno));
  ncon++;
  make_async (fc->fd);

  bzero (&sin, sizeof (sin));
  sin.sin_family = AF_INET;
  sin.sin_port = htons (FINGER_PORT);
  sin.sin_addr = *(struct in_addr *) h->h_addr;
  if (connect (fc->fd, (struct sockaddr *) &sin, sizeof (sin)) < 0
      && errno != EINPROGRESS) {
    perror (fc->host);
    fcon_free (fc);
    return;
  }

  cb_add (fc->fd, 1, finger_senduser, fc);
}

int
main (int argc, char **argv)
{
  int argno;

  /* Writing to an unconnected socket will cause a process to receive
   * a SIGPIPE signal.  We don't want to die if this happens, so we
   * ignore SIGPIPE.  */
  signal (SIGPIPE, SIG_IGN);

  /* Fire off a finger request for every argument, but don't let the
   * number of outstanding connections exceed NCON_MAX. */
  for (argno = 1; argno < argc; argno++) {
    while (ncon >= NCON_MAX)
      cb_check ();
    finger (argv[argno]);
  }

  while (ncon > 0)
    cb_check ();
  exit (0);
}
```

5 Scatter-gather I/O

The read and write system calls move data to and from contiguous memory in a program. However, sometimes one wants to write non-contiguous data or read data into a non-contiguous buffer. Rather than incur the overhead of multiple read and write system calls or use gratuitous copies through temporary buffers, one can tell the kernel about non-contiguous memory with the readv and writev system calls. These system calls make use of the iovec structure defined in sys/uio.h:

```
struct iovec {
  void *iov_base;        /* Base address. */
  size_t iov_len;        /* Length. */
};
```

- `int readv (int fd, const struct iovec *iov, int iovcnt);`

  readv reads from a file descriptor into non-contiguous memory regions. fd specifies the file descriptor from which to read data. iov points to an array of iovcnt iovec structures describing regions of memory into which the data should be placed. readv returns the number of bytes actually read, 0 on an end-of-file, or −1 in the case of error.

- `int writev (int fd, const struct iovec *iov, int iovcnt);`

  writev writes the contents of non-contiguous memory regions to a file descriptor. fd specifies the file descriptor to write to. iov points to an array of iovcnt iovec structures describing the memory regions to be written. writev returns the number of bytes actually written, or −1 in case of error. When a file descriptor is non-blocking, writev may write less than the full amount requested.

Note that both readv and writev have a maximum permissible value for iovcnt, defined in system header files as UIO_MAXIOV.

6 libasync – an asynchronous socket library

In the above example, the routines in async.c allowed the programmer to associate function callbacks with readability and writability conditions on sockets. libasync provides similar functionality using C++ templates. It also provides a number of helpful utility functions for creating sockets and connections and buffering data under conditions where the write system call may not write the full amount requested.

6.1 Memory allocation & debugging

libasync provides C++ support for a debugging malloc. To make the most of this support, you should emply two macros New and vNew instead of the built-in C++ operator new. Use

New where you would ordinarily use new. It is a wrapper around new that also records line-number information, making it easy to debug memory leaks and other problems. vNew is simply a call to New cast to void. Use it to avoid compiler warnings when you ignore the pointer returned by New. You should still use new for per-class allocators and placement new (don't worry if you don't know what these are).

libasync also provides three classes for reporting errors to the user, warn, fatal, and panic. warn simply prints a message to the termial, much like the C++ standard library's cerr. For example, one might say:

```
int n = write (fd, buf, nbytes);
if (n < 0 && errno != EAGAIN) {
  warn << "could not write to socket: "
       << strerror (errno) << "\n";
  // ...
}
```

Unlike cerr, however, warn is asynchronous. warn also makes efforts to accumulate and write data to the terminal in chunks with reasonable boundaries. Thus, when several processes write to the same terminal using warn, the combined output is considerably easier to read than if they had both used cerr. fatal is like warn, except it prepends the word "fatal: " and exits after printing. panic causes a core dump after printing the message. It should be used for assertion failures, when the occurrence of an event indicates a bug in the program.

6.2 The suio class

The suio class maintains arrays of iovec structures and helps deal with the annoyance of short writes. The suio class has the following core methods:

- `void print (const void *base, size_t len);`

  suio::print adds len bytes of data stored at base to the end of the array of iovec structures maintained by suio. suio may or may not copy the contents of the memory at base. Thus, **you must not under any circumstances modify the data** until it has been removed from the suio. Modifying data will cause a crash when the library is compiled for memory debugging.

- `void copy (const void *base, size_t len);`

  This is the same as suio::print, except it makes a copy of the data into a temporary buffer managed by suio. Thus, one can modify the memory at base immediately after calling suio::copy.

- `void rembytes (size_t n);`

  Removes n bytes from the beginning of the iovec array.

- `const iovec *iov () const;`

  Returns an array of iovec structures corresponding to all the data that has been accumulated via suio::print calls.

- `size_t iovcnt () const;`

  Returns the length of the array returned by suio::iov.

- `size_t resid () const;`

  Returns the total number of bytes in the `iovec` array.

As an example, the following shows a rather convoluted function `greet ()`, which prints the words "hello world" to standard output. It behaves correctly even when standard output is in non-blocking mode.

```
void
writewait (int fd)
{
  fd_set fds;
  assert (fd < FD_SETSIZE);
  FD_ZERO (&fds);
  FD_SET (fd, &fds);
  select (fd + 1, NULL, &fds, NULL, NULL);
}

void
greet ()
{
  char hello[] = "hello";
  char space[] = " ";
  char world[] = "world";
  char nl[] = "\n";

  suio uio;
  uio.print (hello, sizeof (hello) - 1);
  uio.print (space, sizeof (space) - 1);
  uio.print (world, sizeof (world) - 1);
  uio.print (nl, sizeof (nl) - 1);

  while (uio.resid ()) {
    writewait (1);
    int n = writev (1, uio.iov (),
                    min<int> (uio.iovcnt (), UIO_MAXIOV));
    if (n < 0 && errno != EAGAIN)
      fatal << "stdout: " << strerror (errno) << "\n";
    if (n > 0)
      uio.rembytes (n);
  }
}
```

Writing the contents of a suio structure in this way is so common that there is a method output for doing it. For symmetry, there is also an input function which reads data from a file descriptor into memory managed by the suio.

- `int output (int fd);`

  suio::output writes as much of the data in a suio structure to `fd` as possible. It returns 1 after successfully writing some data to the file descriptor, 0 if it was unable to write because of `EAGAIN`, and -1 if there was some other write error.

- `int input (int fd);`

  Reads data from file descriptor `fd` and appends it to the contents of the suio. Returns the number of bytes read, 0 on end of file, or -1 on error (including `EAGAIN`).

6.3 The str and strbuf classes

One complication of programming with callbacks is dealing with freeing dynamically allocated memory. This is particularly true of strings. If a function takes an argument of type `char *` and does something asynchronously using the string, the caller may not be allowed to free the string until later. Keeping track of who is responsible for what strings and for how long is tricky and error-prone.

For that reason, libasync has a special class str for reference-counted strings.[6] Strings are references to immutable character streams. Thus, for instance, the following code prints "`one is 'one'`":

```
  str two = "one";
  str one = two;
  two = "two";
  warn << "one is '" << one << "'\n";
```

The strbuf structure allows one to build up a string by appending to it. The following code illustrates the use of strbuf:

```
void
func (str arg)
{
  strbuf sb;
  sb << "arg is '" << arg << "'.";
  str result = sb;
  warn << result << "\n";
}
```

[6]Note that C++ has a string class, but the standard does not specify that it has to be reference counted. Thus, a C++ string implementation can incur the overhead copying a string each time it is passed as a function argument.

strbuf is a built around the suio structure. One can access the underlying suio with the strbuf::tosuio method:

- suio *tosuio () const;

  Returns a pointer to the suio in which the strbuf is storing accumulated data.

6.4 Reference counting

Strings are not the only data structure for which deallocation becomes tricky in asynchronous programming. libasync therefore provides a generic mechanism for reference-counted deallocation of arbitrary dynamically allocated data structures. Refcounting is a simple form of automatic garbage collection: each time a refcounted object is referenced or goes out of scope its reference count is incremented or decremented, respectively. If the reference count of an object ever reaches zero, the object's destructor is called and it is deleted. Note that unlike real garbage collection in languages like Java, you cannot use reference-counted deallocation on data structures with pointer cycles.

Reference counted objects are created by allocating an instance of the refcounted template:

```
class foo : public bar { ... };

  ...
  ref<foo> f = New refcounted<foo> ( ... );
  ptr<bar> b = f;
  f = New refcounted<foo> ( ... );
  b = NULL;
```

Given a class named foo, a refcounted<foo> takes the same constructor arguments as foo, except that constructors with more than 7 arguments cannot be called due to the absence of a varargs template. A ptr<foo> p behaves like a foo *p, except that it is reference counted: *p and p->field are valid operations on p whichever its type. However, array subscripts will not work on a ptr<foo>. You can only allocate one reference counted object at a time.

A ref<foo> is like a ptr<foo>, except that a ref<foo> can never be NULL. If you try to assign a NULL ptr<foo> to a ref<foo> you will get an immediate core dump. The statement ref<foo> = NULL will generate a compile time error.

A const ref<foo> cannot change what it is pointing to, but the foo pointed to can be modified. A ref<const foo> points to a foo you cannot change. A ref<foo> can be converted to a ref<const foo>. In general, you can implicitly convert a ref<A> to a ref<B> if you can implicitly convert an A to a B. You can also implicitly convert a ref<foo> or ptr<foo> to a foo *. Many functions can get away with taking a foo instead of a ptr<foo> if they don't eliminate any existing references.

On both the Pentium and Pentium Pro, a function taking a ref<foo> argument usually seems to take 10-15 more cycles the same function with a foo argument. With some versions of g++, though, this number can go as high as 50 cycles unless you compile with '-fno-exceptions'.

Sometimes you want to do something other than simply free an object when its reference count goes to 0. This can usually be accomplished by the reference counted object's destructor. However, after a destructor is run, the memory associated with an object is freed. If you don't want the object to be deleted, you can define a finalize method that gets invoked once the reference count goes to 0. Any class with a finalize method must declare a virtual base class of `refcount`. For example:

```
class foo : public virtual refcount {
  ...
  void finalize () { recycle (this); }
};
```

Occasionally you may want to generate a reference counted `ref` or `ptr` from an ordinary pointer. This might, for instance, be used by the recycle function above. You can do this with the function mkref, but again only if the underlying type has a virtual base class of `refcount`. Given the above definition, recycle might do this:

```
void
recycle (foo *fp)
{
  ref<foo> = mkref (fp);
  ...
}
```

Note that unlike in Java, an object's finalize method will be called every time the reference count reaches 0, not just the first time. Thus, there is nothing morally wrong with "resurrecting" objects as they are being garbage collected.

Use of mkref is potentially dangerous, however. You can disallow its use on a per-class basis by simply not giving your object a public virtual base class of `refcount`.

```
class foo {
  // fine, no mkref or finalize allowed
};

class foo : private virtual refcount {
  void finalize () { ... }
  // finalize will work, but not mkref
};
```

If you like to live dangerously, there are a few more things you can do (but probably shouldn't). If foo has a virtual base class of `refcount`, it will also inherit the methods `refcount_inc()` and `refcount_dec()`. You can use these to create memory leaks and crash your program, respectively.

6.5 Callbacks

Using libasync to perform socket operations also entails the use of the template class `callback`—a type that approximates function currying. An object of type `callback<R, B1, B2>` contains a member `R operator() (B1, B2)`. Thus callbacks are function objects with the first template type specifying the return of the function and the remaining arguments specifying the types of the arguments to pass the function.

Callbacks are limited to 3 arguments by a compile-time default. Template arguments that aren't specified default to void and don't need to be passed in. Thus, a `callback<int>` acts like a function with signature `int fn ()`, a `callback<int, char *>` acts like a function with signature `int fn (char *)`, and so forth.

Each callback class has two type members, `ptr` and `ref` (accessed as `::ptr` and `::ref`), specifying refcounted pointers and references to the callback object respectively (see above for a description of the refcount class)

The function `wrap` is used to create references to callbacks. Given a function with signature `R fn (A1, A2, A3)`", `wrap` can generate the following references:

```
wrap (fn) → callback<R, A1, A2, A3>::ref
wrap (fn, a1) → callback<R, A2, A3>::ref
wrap (fn, a1, a2) → callback<R, A3>::ref
wrap (fn, a1, a2, a3) → callback<R>::ref
```

When the resulting callback is actually called, it invokes `fn`. The argument list it passes fn starts with whatever arguments were initially passed to wrap and then contains whatever arguments are given to the callback object. For example, given fn above, this code ends up calling `fn (a1, a2, a3)` and assigning the return value to r:

```
R r;
A1 a1;
A2 a2;
A3 a3;
callback<R, A1>::ptr cb;

cb = wrap (fn, a1);
r = (*cb) (a2, a3);
```

One can create callbacks from class methods as well as from global functions. To do this, simply pass the object as the first parameter to wrap, and the method as the second. For example:

```
struct foo {
  void bar (int, char *);
  callback<void, char *>::ref baz () {
    return wrap (this, &foo::bar, 7);
  }
};
```

Note the only way to generate pointers to class members in ANSI C++ is with fully qualified member names. `&foo::bar` cannot be abbreviated to `bar` in the above example, though some C++ compilers still accept that syntax.

If wrap is called with a refcounted `ref` or `ptr` to an object, instead of a simple pointer, the resulting callback will maintain a reference to the object, ensuring it is not deleted. For example, in the following code, `baz` returns a callback with a reference to the current object. This ensures that a `foo` will not be deleted until after the callback has been deleted. Without the call to mkref, if a callback happened after the reference count on a foo object went to zero, the foo object would previously have been deleted and its vtable pointer likely clobbered, resulting in a core dump.

```
struct foo : public virtual refcount {
  virtual void bar (int, char *);
  callback<void, char *>::ref baz () {
    return wrap (mkref (this), &foo::bar, 7);
  }
};
```

6.5.1 An example

```
void
printstrings (char *a, char *b, char *c)
{
  printf ("%s %s %s\n", a, b, c);
}

int
main ()
{
  callback<void, char *>::ref cb1 = wrap (printstrings, "cb1a", "cb1b");
  callback<void, char *, char *>::ref cb2 = wrap (printstrings, "cb2a");
  callback<void, char *, char *, char *>::ref cb3 = wrap (printstrings);

  (*cb1) ("cb1c");                  // prints: cb1a cb1b cb1c
  (*cb2) ("cb2b", "cb2c");          // prints: cb2a cb2b cb2c
  (*cb3) ("cb3a", "cb3b", "cb3c");  // prints: cb3a cb3b cb3c

  return 0;
}
```

6.6 libasync routines

Libasync provides a number of useful functions for registering callbacks (fdcb, amain) as well as performing common tasks relating to sockets (tcpconnect, inetsocket). Each of these functions is prototyped in either async.h or amisc.h.

- `void fdcb (int socket, char operation, callback<void>::ptr cb)`

  associates a callback with the specified condition (readability or writability) on socket. The conditions may be specified by the constants selread or selwrite. Exception conditions are not currently supported. To create the refcounted callback to pass to fdcb use wrap. For instance: `wrap(&read_cb, fd)` produces a refcounted callback which matches a function of the signature `void read_cb(int fd);`

  To unregister a callback, call fdcb with the cb argument set to NULL. Note that callbacks do not unregister once they are called and that no more than one callback can be associated with the same condition on any one socket.

- `timecb_t *delaycb (time_t sec, u_int32_t nsec, callback<void>::ref cb)`

  Arranges for a callback to be called a certain number of seconds and nanoseconds in the future. Returns a pointer suitable for passing to `timecb_remove`.

- `void timecb_remove (timecb_t *)`

  Removes a scheduled timer callback.

- `void amain ()`

  Repeatedly checks to see if any registered callbacks should be triggered. amain() does not return and must not be called recursively from a callback.

- `void tcpconnect (str hostname, int port, callback<void, int>::ref cb)`

  creates an asynchronous connection to the specified host and port. The connected socket will be returned as the argument to the callback specified by cb, or that argument will be -1 to indicate an error.

- `int inetsocket (int type, int_16 port, u_int32_t addr)`

  creates a socket, but unlike tcpconnect does not connect it. inetsocket does, however, bind the socket to the specified local port and address. `type` should be `SOCK_STREAM` for a TCP socket, and `SOCK_DGRAM` for a UDP socket. inetsocket does not put the socket into non-blocking mode; you must call make_async yourself.

- `void make_async (int fd)`

  sets the `O_NONBLOCK` flag for the socket s and places the socket in non-blocking mode.

- `void close_on_exec (int fd)`

  sets the close-on-exec flag of the socket to true. When this flag is true any process created by calling exec will not inherit s.

- `void tcp_nodelay (int fd)`

  sets the TCP_NODELAY socket option to true. This command allows TCP to keep more than one small packet of data outstanding at a time. See the tcp(4) manual page or RFC0896.

6.7 multifinger.C

Using libasync, the multifinger example can be implemented in approximately 100 lines of C++ code.

Multifinger begins by registering as many callbacks for finger requests as possible (that number is limited by a conservative estimate of how many file descriptors are available). As each request completes, the `done()` function registers additional callbacks. A global variable, `ncon`, tracks the number of pending transactions: when `ncon` reaches 0 the program terminates by calling `exit`.

For each transaction, the multifinger client moves through a series of states whose boundaries are defined by system calls that potentially return `EAGAIN`. The code progresses through these states by "chaining" callbacks: for instance, when the `write` of the username is complete, the `write_cb` unregisters the write conditioned callback and registers a callback for readability on the socket.

```
#include "async.h"

#define FD_MAX 64
#define NCON_MAX FD_MAX - 8
#define FINGER_PORT 79

char **nexttarget;
char **lasttarget;
void launchmore ();

struct finger {
  static int ncon;

  const str user;
  const str host;
  int fd;
  strbuf buf;

  static void launch (const char *target);
  finger (str u, str h);
  ~finger ();
  void connected (int f);
  void senduser ();
  void recvreply ();
};

int finger::ncon;

void
finger::launch (const char *target)
{
  if (const char *at = strrchr (target, '@')) {
    str user (target, at - target);
    str host (at + 1);
    vNew finger (user, host);
  }
```

```
  else
    warn << target << ": could not parse finger target\n";
}

finger::finger (str u, str h)
  : user (u), host (h), fd (-1)
{
  ncon++;
  buf << user << "\r\n";
  tcpconnect (host, FINGER_PORT, wrap (this, &finger::connected));
}

void
finger::connected (int f)
{
  fd = f;
  if (fd < 0) {
    warn << host << ": " << strerror (errno) << "\n";
    delete this;
    return;
  }
  fdcb (fd, selwrite, wrap (this, &finger::senduser));
}

void
finger::senduser ()
{
  if (buf.tosuio ()->output (fd) < 0) {
    warn << host << ": " << strerror (errno) << "\n";
    delete this;
    return;
  }
  if (!buf.tosuio ()->resid ()) {
    buf << "[" << user << "@" << host << "]\n";
    fdcb (fd, selwrite, NULL);
    fdcb (fd, selread, wrap (this, &finger::recvreply));
  }
}

void
finger::recvreply ()
{
  switch (buf.tosuio ()->input (fd)) {
  case -1:
    if (errno != EAGAIN) {
      warn << host << ": " << strerror (errno) << "\n";
      delete this;
    }
    break;
  case 0:
    buf.tosuio ()->output (1);
    delete this;
    break;
```

```
  }
}

finger::~finger ()
{
  if (fd >= 0) {
    fdcb (fd, selread, NULL);
    fdcb (fd, selwrite, NULL);
    close (fd);
  }
  ncon--;
  launchmore ();
}

void
launchmore ()
{
  while (nexttarget < lasttarget && finger::ncon < NCON_MAX)
    finger::launch (*nexttarget++);
  if (nexttarget == lasttarget && !finger::ncon)
    exit (0);
}

int
main (int argc, char **argv)
{
  make_sync (1);
  nexttarget = argv + 1;
  lasttarget = argv + argc;
  launchmore ();
  amain ();
}
```

7 Finding out more

This document outlines the system calls needed to perform network I/O on UNIX systems. You may find that you wish to know more about the workings of these calls when you are programming. Fortunately these system calls are documented in detail in the Unix manual pages, which you can access via the man command. Section 2 of the manual corresponds to system calls. To look up the manual page for a system call such as socket, you can simply execute the command "`man socket`." Unfortunately, some system calls such as write have names that conflict with Unix commands. To see the manual page for write, you must explicitly specify section two of the manual page, which you can do with "`man 2 write`" on BSD machines or "`man -s 2 write`" on System V. If you are unsure in which section of the manual to look for a command, you can run "`whatis write`" to see a list of sections in which it appears.