Using TCP Through Sockets

Revised February 1999

1 Introduction

This document is a tutorial for using UNIX sockets. All code appears in the 6.033 locker beneath `lab/src/one/`.

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.

Section 2.1 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.

- `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 this case 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 non-negative integer known as a file descriptor. All `read` and `write` operations must be performed on file descriptors. 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.

[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.

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`. If it 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` in some circumstances. Write returns 0 to indicate an end of file, and −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. 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 '^]'.
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 3.3 shows the source code to a simple finger client that does the equivalent of the last `telnet` example of the previous section. 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 cannot immediately read and write data to a file descriptor 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, 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 does not 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 code 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.

These 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. Section 3.5 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 3.2.

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

  This function assigns an address to a socket, as described in Section 3.2. Unlike the finger client, which did not care about its local port number, here we specify a specific port number.

  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` should not 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.

- `int fork (void);`

  fork creates a new process, identical to the current one. 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 does not return. On failure, it returns −1.

3.5 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 input and standard output 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 3.3 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 does not 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 4.5 shows some utility functions for non-blocking I/O. The function `make_async` (left for you to implement) sets the `O_NONBLOCK` bit of a file descriptor non-blocking with the fcntl system call. 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`.

- write. Like read, write will return 0 on an end of file, and −1 with an `errno` of `EAGAIN` if there is no buffer space. 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 is worth noting that file descriptors returned by accept have `O_NONBLOCK` clear, whether or not the listening socket is non-blocking. In an asynchronous server, 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 do not limit the number of descriptors your program uses, you must prepare for system calls such as 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). Do not 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.

Do not poll with select. Students sometimes misunderstand the use of the select call by implementing a polling mechanism. This is precisely what select can avoid. Polling wastes

CPU time by continually checking file descriptors for I/O. Consider this to the analogy of an impatient kid who constantly asks, “Mom, are we there yet? Mom, are we there yet?” This is polling. Now consider a well-behaved kid who just says, “Hey Mom, tell me when we get there.” This is a callback. Which sounds more reasonable to you?

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.

Example

Section 4.5 shows a simple select-based dispatcher. There are three functions, which we leave up to you to implement:

- `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.

4.3 Using shutdown instead of close

In network I/O, a server may shutdown one direction of a socket connection. For instance, a web browser may send a request, shutdown writing on the client side, read a response from the web server, then shutdown writing. In contrast, the close system call will shutdown both directions of a socket connection. You will find the shutdown system call useful in a proxy when passing EOFs.

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

  The shutdown system call closes part or all of a full-duplex connection described by the file descriptor `s`. shutdown informs the kernel that no one is interested in further reading or writing to a file descriptor. Future reads or writes can be independently disallowed. In contrast to the close system call, shutdown enables fine-grained control over a full-duplex connection.

  Setting `how` to 0 disallows future reads. Setting `how` to 1 disallows future writes (particularly useful on TCP sockets). Setting `how` to 2 disallows both reads and writes.

4.4 `async.h`: Interface to `async.c`

```
#ifndef _ASYNC_H_ /* Guard against multiple inclusion */
#define _ASYNC_H_ 1

/* Enable stress-tests. */
/* #define SMALL_LIMITS 1 */

#include <sys/types.h>

#if __GNUC__ != 2
/* The __attribute__ keyword helps make gcc -Wall more useful, but
 * doesn't apply to other C compilers.  You don't need to worry about
 * what __attribute__ does (though if you are curious you can consult
 * the gcc info pages). */
#define __attribute__(x)
#endif /* __GNUC__ != 2 */

/* 1 + highest file descriptor number expected */
#define FD_MAX 64

/* The number of TCP connections we will use.  This can be no higher
 * than FD_MAX, but we reserve a few file descriptors because 0, 1,
 * and 2 are already in use as stdin, stdout, and stderr.  Moreover,
 * libc can make use of a few file descriptors for functions like
 * gethostbyname. */
```

```
#define NCON_MAX FD_MAX - 8

void fatal (const char *msg, ...)
     __attribute__ ((noreturn, format (printf, 1, 2)));

/* Malloc-like functions that don't fail. */
void *xrealloc (void *, size_t);
#define xmalloc(size) xrealloc (0, size)
#define xfree(ptr) xrealloc (ptr, 0)
#define bzero(ptr, size) memset (ptr, 0, size)

void make_async (int);
void cb_add (int, int, void (*fn)(void *), void *arg);
void cb_free (int, int);
void cb_check (void);

#endif /* !_ASYNC_H_ */
```

4.5 async.c: Handy routines for asynchronous 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);
}
```

4.6 Putting it all together

We now present an example that demonstrates the power of non-blocking socket I/O. Section 4.7 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.

4.7 `multifinger.c`: A mostly[3] 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 */

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);
```

[3] `gethostbyname` performs synchronous socket I/O.


```
  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 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.

6 About this document

David Mazières developed the original version of this document in 1998. Jeff Hu and Kevin Fu changed the content for the 6.033 lab in 1999.