63

ALTERNATIVE I/O MODELS

This chapter discusses three alternatives to the conventional file I/O model that we have employed in most programs shown in this book:

- I/O multiplexing (the *select()* and *poll()* system calls);
- signal-driven I/O; and
- the Linux-specific *epoll* API.

63.1 Overview

Most of the programs that we have presented so far in this book employ an I/O model under which a process performs I/O on just one file descriptor at a time, and each I/O system call blocks until the data is transferred. For example, when reading from a pipe, a *read()* call normally blocks if no data is currently present in the pipe, and a *write()* call blocks if there is insufficient space in the pipe to hold the data to be written. Similar behavior occurs when performing I/O on various other types of files, including FIFOs and sockets.

> Disk files are a special case. As described in Chapter 13, the kernel employs the buffer cache to speed disk I/O requests. Thus, a *write()* to a disk returns as soon as the requested data has been transferred to the kernel buffer cache, rather than waiting until the data is written to disk (unless the `O_SYNC` flag was specified when opening the file). Correspondingly, a *read()* transfers data from the buffer cache to a user buffer, and if the required data is not in the buffer cache, then the kernel puts the process to sleep while a disk read is performed.

The traditional blocking I/O model is sufficient for many applications, but not all. In particular, some applications need to able to do one or both of the following:

- Check whether I/O is possible on a file descriptor without blocking if it is not possible.
- Monitor multiple file descriptors to see if I/O is possible on any of them.

We have already encountered two techniques that can be used to partially address these needs: nonblocking I/O and the use of multiple processes or threads.

We described nonblocking I/O in some detail in Sections 5.9 and 44.9. If we place a file descriptor in nonblocking mode by enabling the `O_NONBLOCK` open file status flag, then an I/O system call that can't be immediately completed returns an error instead of blocking. Nonblocking I/O can be employed with pipes, FIFOs, sockets, terminals, pseudoterminals, and some other types of devices.

Nonblocking I/O allows us to periodically check ("poll") whether I/O is possible on a file descriptor. For example, we can make an input file descriptor nonblocking, and then periodically perform nonblocking reads. If we need to monitor multiple file descriptors, then we mark them all nonblocking, and poll each of them in turn. However, polling in this manner is usually undesirable. If polling is done only infrequently, then the latency before an application responds to an I/O event may be unacceptably long; on the other hand, polling in a tight loop wastes CPU time.

> In this chapter, we use the word *poll* in two distinct ways. One of these is as the name of the I/O multiplexing system call, *poll()*. In the other use, we mean "performing a nonblocking check on the status of a file descriptor."

If we don't want a process to block when performing I/O on a file descriptor, we can instead create a new process to perform the I/O. The parent process can then carry on to perform other tasks, while the child process blocks until the I/O is complete. If we need to handle I/O on multiple file descriptors, we can create one child for each descriptor. The problems with this approach are expense and complexity. Creating and maintaining processes places a load on the system, and, typically, the child processes will need to use some form of IPC to inform the parent about the status of I/O operations.

Using multiple threads instead of processes is less demanding of resources, but the threads will probably still need to communicate information to one another about the status of I/O operations, and the programming can be complex, especially if we are using thread pools to minimize the number of threads used to handle large numbers of simultaneous clients. (One place where threads can be particularly useful is if the application needs to call a third-party library that performs blocking I/O. An application can avoid blocking in this case by making the library call in a separate thread.)

Because of the limitations of both nonblocking I/O and the use of multiple threads or processes, one of the following alternatives is often preferable:

- *I/O multiplexing* allows a process to simultaneously monitor multiple file descriptors to find out whether I/O is possible on any of them. The *select()* and *poll()* system calls perform I/O multiplexing.
- *Signal-driven I/O* is a technique whereby a process requests that the kernel send it a signal when input is available or data can be written on a specified file descriptor. The process can then carry on performing other activities, and is notified when I/O becomes possible via receipt of the signal. When monitoring large numbers of file descriptors, signal-driven I/O provides significantly better performance than *select()* and *poll()*.
- The *epoll* API is a Linux-specific feature that first appeared in Linux 2.6. Like the I/O multiplexing APIs, the *epoll* API allows a process to monitor multiple file descriptors to see if I/O is possible on any of them. Like signal-driven I/O, the *epoll* API provides much better performance when monitoring large numbers of file descriptors.

> In the remainder of this chapter, we'll generally frame the discussion of the above techniques in terms of processes. However, these techniques can also be employed in multithreaded applications.

In effect, I/O multiplexing, signal-driven I/O, and *epoll* are all methods of achieving the same result—monitoring one or, commonly, several file descriptors simultaneously to see if they are *ready* to perform I/O (to be precise, to see whether an I/O system call could be performed without blocking). The transition of a file descriptor into a ready state is triggered by some type of I/O *event*, such as the arrival of input, the completion of a socket connection, or the availability of space in a previously full socket send buffer after TCP transmits queued data to the socket peer. Monitoring multiple file descriptors is useful in applications such as network servers that must simultaneously monitor multiple client sockets, or applications that must simultaneously monitor input from a terminal and a pipe or socket.

Note that none of these techniques performs I/O. They merely tell us that a file descriptor is ready. Some other system call must then be used to actually perform the I/O.

> One I/O model that we don't describe in this chapter is POSIX asynchronous I/O (AIO). POSIX AIO allows a process to queue an I/O operation to a file and then later be notified when the operation is complete. The advantage of POSIX AIO is that the initial I/O call returns immediately, so that the process is not tied up waiting for data to be transferred to the kernel or for the operation to complete. This allows the process to perform other tasks in parallel with the I/O (which may include queuing further I/O requests). For certain types of applications, POSIX AIO can provide useful performance benefits. Currently, Linux provides a threads-based implementation of POSIX AIO within *glibc*. At the time of writing, work is ongoing toward providing an in-kernel implementation of POSIX AIO, which should provide better scaling performance. POSIX AIO is described in [Gallmeister, 1995] and [Robbins & Robbins, 2003].

Which technique?

During the course of this chapter, we'll consider the reasons we may choose one of these techniques rather than another. In the meantime, we summarize a few points:

- The *select()* and *poll()* system calls are long-standing interfaces that have been present on UNIX systems for many years. Compared to the other techniques, their primary advantage is portability. Their main disadvantage is that they don't scale well when monitoring large numbers (hundreds or thousands) of file descriptors.
- The key advantage of the *epoll* API is that it allows an application to efficiently monitor large numbers of file descriptors. Its primary disadvantage is that it is a Linux-specific API.

  > Some other UNIX implementations provide (nonstandard) mechanisms similar to *epoll*. For example, Solaris provides the special `/dev/poll` file (described in the Solaris *poll(7d)* manual page), and some of the BSDs provide the *kqueue* API (which provides a more general-purpose monitoring facility than *epoll*). [Stevens et al., 2004] briefly describes these two mechanisms; a longer discussion of *kqueue* can be found in [Lemon, 2001].

- Like *epoll*, signal-driven I/O allows an application to efficiently monitor large numbers of file descriptors. However, *epoll* provides a number of advantages over signal-driven I/O:
  - We avoid the complexities of dealing with signals.
  - We can specify the kind of monitoring that we want to perform (e.g., ready for reading or ready for writing).
  - We can select either level-triggered or edge-triggered notification (described in Section 63.1.1).

  Furthermore, taking full advantage of signal-driven I/O requires the use of nonportable, Linux-specific features, and if we do this, signal-driven I/O is no more portable than *epoll*.

Because, on the one hand, *select()* and *poll()* are more portable, while signal-driven I/O and *epoll* deliver better performance, for some applications, it can be worthwhile writing an abstract software layer for monitoring file descriptor events. With such a layer, portable programs can employ *epoll* (or a similar API) on systems that provide it, and fall back to the use of *select()* or *poll()* on other systems.

> The *libevent* library is a software layer that provides an abstraction for monitoring file descriptor events. It has been ported to a number of UNIX systems. As its underlying mechanism, *libevent* can (transparently) employ any of the techniques described in this chapter: *select()*, *poll()*, signal-driven I/O, or *epoll*, as well as the Solaris specific `/dev/poll` interface or the BSD *kqueue* interface. (Thus, *libevent* also serves as an example of how to use each of these techniques.) Written by Niels Provos, *libevent* is available at *http://monkey.org/~provos/libevent/*.

63.1.1 Level-Triggered and Edge-Triggered Notification

Before discussing the various alternative I/O mechanisms in detail, we need to distinguish two models of readiness notification for a file descriptor:

- *Level-triggered notification*: A file descriptor is considered to be ready if it is possible to perform an I/O system call without blocking.
- *Edge-triggered notification*: Notification is provided if there is I/O activity (e.g., new input) on a file descriptor since it was last monitored.

Table 63-1 summarizes the notification models employed by I/O multiplexing, signal-driven I/O, and *epoll*. The *epoll* API differs from the other two I/O models in that it can employ both level-triggered notification (the default) and edge-triggered notification.

Table 63-1: Use of level-triggered and edge-triggered notification models

I/O model	Level-triggered?	Edge-triggered?
select(), *poll()*	•	
Signal-driven I/O		•
epoll	•	•

Details of the differences between these two notification models will become clearer during the course of the chapter. For now, we describe how the choice of notification model affects the way we design a program.

When we employ level-triggered notification, we can check the readiness of a file descriptor at any time. This means that when we determine that a file descriptor is ready (e.g., it has input available), we can perform some I/O on the descriptor, and then repeat the monitoring operation to check if the descriptor is still ready (e.g., it still has more input available), in which case we can perform more I/O, and so on. In other words, because the level-triggered model allows us to repeat the I/O monitoring operation at any time, it is not necessary to perform as much I/O as possible (e.g., read as many bytes as possible) on the file descriptor (or even perform any I/O at all) each time we are notified that a file descriptor is ready.

By contrast, when we employ edge-triggered notification, we receive notification only when an I/O event occurs. We don't receive any further notification until another I/O event occurs. Furthermore, when an I/O event is notified for a file descriptor, we usually don't know how much I/O is possible (e.g., how many bytes are available for reading). Therefore, programs that employ edge-triggered notification are usually designed according to the following rules:

- After notification of an I/O event, the program should—at some point—perform as much I/O as possible (e.g., read as many bytes as possible) on the corresponding file descriptor. If the program fails to do this, then it might miss the opportunity to perform some I/O, because it would not be aware of the need to operate on the file descriptor until another I/O event occurred. This could lead to spurious data loss or blockages in a program. We said "at some point," because sometimes it may not be desirable to perform all of the I/O immediately after we determine that the file descriptor is ready. The problem is that we may

starve other file descriptors of attention if we perform a large amount of I/O on one file descriptor. We consider this point in more detail when we describe the edge-triggered notification model for *epoll* in Section 63.4.6.

- If the program employs a loop to perform as much I/O as possible on the file descriptor, and the descriptor is marked as blocking, then eventually an I/O system call will block when no more I/O is possible. For this reason, each monitored file descriptor is normally placed in nonblocking mode, and after notification of an I/O event, I/O operations are performed repeatedly until the relevant system call (e.g., *read()* or *write()*) fails with the error `EAGAIN` or `EWOULDBLOCK`.

63.1.2 Employing Nonblocking I/O with Alternative I/O Models

Nonblocking I/O (the `O_NONBLOCK` flag) is often used in conjunction with the I/O models described in this chapter. Some examples of why this can be useful are the following:

- As explained in the previous section, nonblocking I/O is usually employed in conjunction with I/O models that provide edge-triggered notification of I/O events.
- If multiple processes (or threads) are performing I/O on the same open file descriptions, then, from a particular process's point of view, a descriptor's readiness may change between the time the descriptor was notified as being ready and the time of the subsequent I/O call. Consequently, a blocking I/O call could block, thus preventing the process from monitoring other file descriptors. (This can occur for all of the I/O models that we describe in this chapter, regardless of whether they employ level-triggered or edge-triggered notification.)
- Even after a level-triggered API such as *select()* or *poll()* informs us that a file descriptor for a stream socket is ready for writing, if we write a large enough block of data in a single *write()* or *send()*, then the call will nevertheless block.
- In rare cases, level-triggered APIs such as *select()* and *poll()* can return spurious readiness notifications—they can falsely inform us that a file descriptor is ready. This could be caused by a kernel bug or be expected behavior in an uncommon scenario.

  > Section 16.6 of [Stevens et al., 2004] describes one example of spurious readiness notifications on BSD systems for a listening socket. If a client connects to a server's listening socket and then resets the connection, a *select()* performed by the server between these two events will indicate the listening socket as being readable, but a subsequent *accept()* that is performed after the client's reset will block.

63.2 I/O Multiplexing

I/O multiplexing allows us to simultaneously monitor multiple file descriptors to see if I/O is possible on any of them. We can perform I/O multiplexing using either of two system calls with essentially the same functionality. The first of these, *select()*, appeared along with the sockets API in BSD. This was historically the more widespread of the two system calls. The other system call, *poll()*, appeared in System V. Both *select()* and *poll()* are nowadays required by SUSv3.

We can use *select()* and *poll()* to monitor file descriptors for regular files, terminals, pseudoterminals, pipes, FIFOs, sockets, and some types of character devices. Both system calls allow a process either to block indefinitely waiting for file descriptors to become ready or to specify a timeout on the call.

63.2.1 The *select()* System Call

The *select()* system call blocks until one or more of a set of file descriptors becomes ready.

```
#include <sys/time.h>         /* For portability */
#include <sys/select.h>

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

Returns number of ready file descriptors, 0 on timeout, or −1 on error

The *nfds*, *readfds*, *writefds*, and *exceptfds* arguments specify the file descriptors that *select()* is to monitor. The *timeout* argument can be used to set an upper limit on the time for which *select()* will block. We describe each of these arguments in detail below.

> In the prototype for *select()* shown above, we include `<sys/time.h>` because that was the header specified in SUSv2, and some UNIX implementations require this header. (The `<sys/time.h>` header is present on Linux, and including it does no harm.)

File descriptor sets

The *readfds*, *writefds*, and *exceptfds* arguments are pointers to *file descriptor sets*, represented using the data type *fd_set*. These arguments are used as follows:

- *readfds* is the set of file descriptors to be tested to see if input is possible;
- *writefds* is the set of file descriptors to be tested to see if output is possible; and
- *exceptfds* is the set of file descriptors to be tested to see if an exceptional condition has occurred.

The term *exceptional condition* is often misunderstood to mean that some sort of error condition has arisen on the file descriptor. This is not the case. An exceptional condition occurs in just two circumstances on Linux (other UNIX implementations are similar):

- A state change occurs on a pseudoterminal slave connected to a master that is in packet mode (Section 64.5).
- Out-of-band data is received on a stream socket (Section 61.13.1).

Typically, the *fd_set* data type is implemented as a bit mask. However, we don't need to know the details, since all manipulation of file descriptor sets is done via four macros: `FD_ZERO()`, `FD_SET()`, `FD_CLR()`, and `FD_ISSET()`.

```
#include <sys/select.h>

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

int FD_ISSET(int fd, fd_set *fdset);
```
Returns true (1) if *fd* is in *fdset*, or false (0) otherwise

These macros operate as follows:

- `FD_ZERO()` initializes the set pointed to by *fdset* to be empty.
- `FD_SET()` adds the file descriptor *fd* to the set pointed to by *fdset*.
- `FD_CLR()` removes the file descriptor *fd* from the set pointed to by *fdset*.
- `FD_ISSET()` returns true if the file descriptor *fd* is a member of the set pointed to by *fdset*.

A file descriptor set has a maximum size, defined by the constant `FD_SETSIZE`. On Linux, this constant has the value 1024. (Other UNIX implementations have similar values for this limit.)

> Even though the `FD_*` macros are operating on user-space data structures, and the kernel implementation of *select()* can handle descriptor sets with larger sizes, *glibc* provides no simple way of modifying the definition of `FD_SETSIZE`. If we want to change this limit, we must modify the definition in the *glibc* header files. However, for reasons that we describe later in this chapter, if we need to monitor large numbers of descriptors, then using *epoll* is probably preferable to the use of *select()*.

The *readfds*, *writefds*, and *exceptfds* arguments are all value-result. Before the call to *select()*, the *fd_set* structures pointed to by these arguments must be initialized (using `FD_ZERO()` and `FD_SET()`) to contain the set of file descriptors of interest. The *select()* call modifies each of these structures so that, on return, they contain the set of file descriptors that are ready. (Since these structures are modified by the call, we must ensure that we reinitialize them if we are repeatedly calling *select()* from within a loop.) The structures can then be examined using `FD_ISSET()`.

If we are not interested in a particular class of events, then the corresponding *fd_set* argument can be specified as `NULL`. We say more about the precise meaning of each of the three event types in Section 63.2.3.

The *nfds* argument must be set one greater than the highest file descriptor number included in any of the three file descriptor sets. This argument allows *select()* to be more efficient, since the kernel then knows not to check whether file descriptor numbers higher than this value are part of each file descriptor set.

The *timeout* argument

The *timeout* argument controls the blocking behavior of *select()*. It can be specified either as NULL, in which case *select()* blocks indefinitely, or as a pointer to a *timeval* structure:

```
struct timeval {
    time_t      tv_sec;         /* Seconds */
    suseconds_t tv_usec;        /* Microseconds (long int) */
};
```

If both fields of *timeout* are 0, then *select()* doesn't block; it simply polls the specified file descriptors to see which ones are ready and returns immediately. Otherwise, *timeout* specifies an upper limit on the time for which *select()* is to wait.

Although the *timeval* structure affords microsecond precision, the accuracy of the call is limited by the granularity of the software clock (Section 10.6). SUSv3 specifies that the timeout is rounded upward if it is not an exact multiple of this granularity.

> SUSv3 requires that the maximum permissible timeout interval be at least 31 days. Most UNIX implementations allow a considerably higher limit. Since Linux/x86-32 uses a 32-bit integer for the *time_t* type, the upper limit is many years.

When *timeout* is NULL, or points to a structure containing nonzero fields, *select()* blocks until one of the following occurs:

- at least one of the file descriptors specified in *readfds*, *writefds*, or *exceptfds* becomes ready;
- the call is interrupted by a signal handler; or
- the amount of time specified by *timeout* has passed.

> In older UNIX implementations that lacked a sleep call with subsecond precision (e.g., *nanosleep()*), *select()* was used to emulate this functionality by specifying *nfds* as 0; *readfds*, *writefds*, and *exceptfds* as NULL; and the desired sleep interval in *timeout*.

On Linux, if *select()* returns because one or more file descriptors became ready, and if *timeout* was non-NULL, then *select()* updates the structure to which *timeout* points to indicate how much time remained until the call would have timed out. However, this behavior is implementation-specific. SUSv3 also allows the possibility that an implementation leaves the structure pointed to by *timeout* unchanged, and most other UNIX implementations *don't* modify this structure. Portable applications that employ *select()* within a loop should always ensure that the structure pointed to by *timeout* is initialized before each *select()* call, and should ignore the information returned in the structure after the call.

SUSv3 states that the structure pointed to by *timeout* may be modified only on a successful return from *select()*. However, on Linux, if *select()* is interrupted by a signal handler (so that it fails with the error EINTR), then the structure is modified to indicate the time remaining until a timeout would have occurred (i.e., like a successful return).

If we use the Linux-specific *personality()* system call to set a personality that includes the STICKY_TIMEOUTS personality bit, then *select()* doesn't modify the structure pointed to by *timeout*.

Return value from *select()*

As its function result, *select()* returns one of the following:

- A return value of −1 indicates that an error occurred. Possible errors include EBADF and EINTR. EBADF indicates that one of the file descriptors in *readfds*, *writefds*, or *exceptfds* is invalid (e.g., not currently open). EINTR, indicates that the call was interrupted by a signal handler. (As noted in Section 21.5, *select()* is never automatically restarted if interrupted by a signal handler.)
- A return value of 0 means that the call timed out before any file descriptor became ready. In this case, each of the returned file descriptor sets will be empty.
- A positive return value indicates that one or more file descriptors is ready. The return value is the number of ready descriptors. In this case, each of the returned file descriptor sets must be examined (using FD_ISSET()) in order to find out which I/O events occurred. If the same file descriptor is specified in more than one of *readfds*, *writefds*, and *exceptfds*, it is counted multiple times if it is ready for more than one event. In other words, *select()* returns the total number of file descriptors marked as ready in all three returned sets.

Example program

The program in Listing 63-1 demonstrates the use of *select()*. Using command-line arguments, we can specify the *timeout* and the file descriptors that we wish to monitor. The first command-line argument specifies the *timeout* for *select()*, in seconds. If a hyphen (-) is specified here, then *select()* is called with a timeout of NULL, meaning block indefinitely. Each of the remaining command-line arguments specifies the number of a file descriptor to be monitored, followed by letters indicating the operations for which the descriptor is to be checked. The letters we can specify here are *r* (ready for read) and *w* (ready for write).

Listing 63-1: Using *select()* to monitor multiple file descriptors

altio/t_select.c

```
#include <sys/time.h>
#include <sys/select.h>
#include "tlpi_hdr.h"

static void
usageError(const char *progName)
{
    fprintf(stderr, "Usage: %s {timeout|-} fd-num[rw]...\n", progName);
    fprintf(stderr, "    - means infinite timeout; \n");
    fprintf(stderr, "    r = monitor for read\n");
    fprintf(stderr, "    w = monitor for write\n\n");
    fprintf(stderr, "    e.g.: %s - 0rw 1w\n", progName);
    exit(EXIT_FAILURE);
}
```

```
int
main(int argc, char *argv[])
{
    fd_set readfds, writefds;
    int ready, nfds, fd, numRead, j;
    struct timeval timeout;
    struct timeval *pto;
    char buf[10];                       /* Large enough to hold "rw\0" */

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageError(argv[0]);

    /* Timeout for select() is specified in argv[1] */

    if (strcmp(argv[1], "-") == 0) {
        pto = NULL;                     /* Infinite timeout */
    } else {
        pto = &timeout;
        timeout.tv_sec = getLong(argv[1], 0, "timeout");
        timeout.tv_usec = 0;            /* No microseconds */
    }

    /* Process remaining arguments to build file descriptor sets */

    nfds = 0;
    FD_ZERO(&readfds);
    FD_ZERO(&writefds);

    for (j = 2; j < argc; j++) {
        numRead = sscanf(argv[j], "%d%2[rw]", &fd, buf);
        if (numRead != 2)
            usageError(argv[0]);
        if (fd >= FD_SETSIZE)
            cmdLineErr("file descriptor exceeds limit (%d)\n", FD_SETSIZE);

        if (fd >= nfds)
            nfds = fd + 1;              /* Record maximum fd + 1 */
        if (strchr(buf, 'r') != NULL)
            FD_SET(fd, &readfds);
        if (strchr(buf, 'w') != NULL)
            FD_SET(fd, &writefds);
    }

    /* We've built all of the arguments; now call select() */

    ready = select(nfds, &readfds, &writefds, NULL, pto);
                                        /* Ignore exceptional events */
    if (ready == -1)
        errExit("select");

    /* Display results of select() */

    printf("ready = %d\n", ready);
```

```
    for (fd = 0; fd < nfds; fd++)
        printf("%d: %s%s\n", fd, FD_ISSET(fd, &readfds) ? "r" : "",
                FD_ISSET(fd, &writefds) ? "w" : "");

    if (pto != NULL)
        printf("timeout after select(): %ld.%03ld\n",
               (long) timeout.tv_sec, (long) timeout.tv_usec / 10000);
    exit(EXIT_SUCCESS);
}
```

altio/t_select.c

In the following shell session log, we demonstrate the use of the program in Listing 63-1. In the first example, we make a request to monitor file descriptor 0 for input with a 10-second *timeout*:

```
$ ./t_select 10 0r
Press Enter, so that a line of input is available on file descriptor 0
ready = 1
0: r
timeout after select(): 8.003
$                                          Next shell prompt is displayed
```

The above output shows us that *select()* determined that one file descriptor was ready. This was file descriptor 0, which was ready for reading. We can also see that the *timeout* was modified. The final line of output, consisting of just the shell `$` prompt, appeared because the *t_select* program didn't read the newline character that made file descriptor 0 ready, and so that character was read by the shell, which responded by printing another prompt.

In the next example, we again monitor file descriptor 0 for input, but this time with a *timeout* of 0 seconds:

```
$ ./t_select 0 0r
ready = 0
timeout after select(): 0.000
```

The *select()* call returned immediately, and found no file descriptor was ready.

In the next example, we monitor two file descriptors: descriptor 0, to see if input is available, and descriptor 1, to see if output is possible. In this case, we specify the *timeout* as `NULL` (the first command-line argument is a hyphen), meaning infinity:

```
$ ./t_select - 0r 1w
ready = 1
0:
1: w
```

The *select()* call returned immediately, informing us that output was possible on file descriptor 1.

63.2.2 The *poll()* System Call

The *poll()* system call performs a similar task to *select()*. The major difference between the two system calls lies in how we specify the file descriptors to be monitored. With *select()*, we provide three sets, each marked to indicate the file descriptors of interest. With *poll()*, we provide a list of file descriptors, each marked with the set of events of interest.

```
#include <poll.h>

int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

Returns number of ready file descriptors, 0 on timeout, or −1 on error

The *fds* argument and the *pollfd* array (*nfds*) specify the file descriptors that *poll()* is to monitor. The *timeout* argument can be used to set an upper limit on the time for which *poll()* will block. We describe each of these arguments in detail below.

The *pollfd* array

The *fds* argument lists the file descriptors to be monitored by *poll()*. This argument is an array of *pollfd* structures, defined as follows:

```
struct pollfd {
    int   fd;               /* File descriptor */
    short events;           /* Requested events bit mask */
    short revents;          /* Returned events bit mask */
};
```

The *nfds* arguments specifies the number of items in the *fds* array. The *nfds_t* data type used to type the *nfds* argument is an unsigned integer type.

The *events* and *revents* fields of the *pollfd* structure are bit masks. The caller initializes *events* to specify the events to be monitored for the file descriptor *fd*. Upon return from *poll()*, *revents* is set to indicate which of those events actually occurred for this file descriptor.

Table 63-2 lists the bits that may appear in the *events* and *revents* fields. The first group of bits in this table (POLLIN, POLLRDNORM, POLLRDBAND, POLLPRI, and POLLRDHUP) are concerned with input events. The next group of bits (POLLOUT, POLLWRNORM, and POLLWRBAND) are concerned with output events. The third group of bits (POLLERR, POLLHUP, and POLLNVAL) are set in the *revents* field to return additional information about the file descriptor. If specified in the *events* field, these three bits are ignored. The final bit (POLLMSG) is unused by *poll()* on Linux.

> On UNIX implementations providing STREAMS devices, POLLMSG indicates that a message containing a SIGPOLL signal has reached the head of the stream. POLLMSG is unused on Linux, because Linux doesn't implement STREAMS.

Table 63-2: Bit-mask values for *events* and *revents* fields of the *pollfd* structure

Bit	Input in *events*?	Returned in *revents*?	Description
POLLIN	•	•	Data other than high-priority data can be read
POLLRDNORM	•	•	Equivalent to POLLIN
POLLRDBAND	•	•	Priority data can be read (unused on Linux)
POLLPRI	•	•	High-priority data can be read
POLLRDHUP	•	•	Shutdown on peer socket
POLLOUT	•	•	Normal data can be written
POLLWRNORM	•	•	Equivalent to POLLOUT
POLLWRBAND	•	•	Priority data can be written
POLLERR		•	An error has occurred
POLLHUP		•	A hangup has occurred
POLLNVAL		•	File descriptor is not open
POLLMSG			Unused on Linux (and unspecified in SUSv3)

It is permissible to specify *events* as 0 if we are not interested in events on a particular file descriptor. Furthermore, specifying a negative value for the *fd* field (e.g., negating its value if nonzero) causes the corresponding *events* field to be ignored and the *revents* field always to be returned as 0. Either of these techniques can be used to (perhaps temporarily) disable monitoring of a single file descriptor, without needing to rebuild the entire *fds* list.

Note the following further points regarding the Linux implementation of *poll()*:

- Although defined as separate bits, POLLIN and POLLRDNORM are synonymous.
- Although defined as separate bits, POLLOUT and POLLWRNORM are synonymous.
- POLLRDBAND is generally unused; that is, it is ignored in the *events* field and not set in *revents*.

  The only place where POLLRDBAND is set is in code implementing the (obsolete) DECnet networking protocol.

- Although set for sockets in certain circumstances, POLLWRBAND conveys no useful information. (There are no circumstances in which POLLWRBAND is set when POLLOUT and POLLWRNORM are not also set.)

  POLLRDBAND and POLLWRBAND are meaningful on implementations that provide System V STREAMS (which Linux does not). Under STREAMS, a message can be assigned a nonzero priority, and such messages are queued to the receiver in decreasing order of priority, in a band ahead of normal (priority 0) messages.

- The _XOPEN_SOURCE feature test macro must be defined in order to obtain the definitions of the constants POLLRDNORM, POLLRDBAND, POLLWRNORM, and POLLWRBAND from <poll.h>.

- POLLRDHUP is a Linux-specific flag available since kernel 2.6.17. In order to obtain this definition from <poll.h>, the _GNU_SOURCE feature test macro must be defined.
- POLLNVAL is returned if the specified file descriptor was closed at the time of the *poll()* call.

Summarizing the above points, the *poll()* flags of real interest are POLLIN, POLLOUT, POLLPRI, POLLRDHUP, POLLHUP, and POLLERR. We consider the meanings of these flags in greater detail in Section 63.2.3.

The *timeout* argument

The *timeout* argument determines the blocking behavior of *poll()* as follows:

- If *timeout* equals −1, block until one of the file descriptors listed in the *fds* array is ready (as defined by the corresponding *events* field) or a signal is caught.
- If *timeout* equals 0, do not block—just perform a check to see which file descriptors are ready.
- If *timeout* is greater than 0, block for up to *timeout* milliseconds, until one of the file descriptors in *fds* is ready, or until a signal is caught.

As with *select()*, the accuracy of *timeout* is limited by the granularity of the software clock (Section 10.6), and SUSv3 specifies that *timeout* is always rounded upward if it is not an exact multiple of the clock granularity.

Return value from *poll()*

As its function result, *poll()* returns one of the following:

- A return value of −1 indicates that an error occurred. One possible error is EINTR, indicating that the call was interrupted by a signal handler. (As noted in Section 21.5, *poll()* is never automatically restarted if interrupted by a signal handler.)
- A return of 0 means that the call timed out before any file descriptor became ready.
- A positive return value indicates that one or more file descriptors are ready. The returned value is the number of *pollfd* structures in the *fds* array that have a nonzero *revents* field.

> Note the slightly different meaning of a positive return value from *select()* and *poll()*. The *select()* system call counts a file descriptor multiple times if it occurs in more than one returned file descriptor set. The *poll()* system call returns a count of ready file descriptors, and a file descriptor is counted only once, even if multiple bits are set in the corresponding *revents* field.

Example program

Listing 63-2 provides a simple demonstration of the use of *poll()*. This program creates a number of pipes (each pipe uses a consecutive pair of file descriptors), writes bytes to the write ends of randomly selected pipes, and then performs a *poll()* to see which pipes have data available for reading.

The following shell session shows an example of what we see when running this program. The command-line arguments to the program specify that ten pipes should be created, and writes should be made to three randomly selected pipes.

```
$ ./poll_pipes 10 3
Writing to fd:   4 (read fd:   3)
Writing to fd:  14 (read fd:  13)
Writing to fd:  14 (read fd:  13)
poll() returned: 2
Readable:   3
Readable:  13
```

From the above output, we can see that *poll()* found two pipes had data available for reading.

Listing 63-2: Using *poll()* to monitor multiple file descriptors

altio/poll_pipes.c

```
#include <time.h>
#include <poll.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int numPipes, j, ready, randPipe, numWrites;
    int (*pfds)[2];                     /* File descriptors for all pipes */
    struct pollfd *pollFd;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s num-pipes [num-writes]\n", argv[0]);

    /* Allocate the arrays that we use. The arrays are sized according
       to the number of pipes specified on command line */

    numPipes = getInt(argv[1], GN_GT_0, "num-pipes");

    pfds = calloc(numPipes, sizeof(int [2]));
    if (pfds == NULL)
        errExit("malloc");
    pollFd = calloc(numPipes, sizeof(struct pollfd));
    if (pollFd == NULL)
        errExit("malloc");

    /* Create the number of pipes specified on command line */

    for (j = 0; j < numPipes; j++)
        if (pipe(pfds[j]) == -1)
            errExit("pipe %d", j);

    /* Perform specified number of writes to random pipes */

    numWrites = (argc > 2) ? getInt(argv[2], GN_GT_0, "num-writes") : 1;
```

```
    srandom((int) time(NULL));
    for (j = 0; j < numWrites; j++) {
        randPipe = random() % numPipes;
        printf("Writing to fd: %3d (read fd: %3d)\n",
                pfds[randPipe][1], pfds[randPipe][0]);
        if (write(pfds[randPipe][1], "a", 1) == -1)
            errExit("write %d", pfds[randPipe][1]);
    }

    /* Build the file descriptor list to be supplied to poll(). This list
       is set to contain the file descriptors for the read ends of all of
       the pipes. */

    for (j = 0; j < numPipes; j++) {
        pollFd[j].fd = pfds[j][0];
        pollFd[j].events = POLLIN;
    }

    ready = poll(pollFd, numPipes, -1);         /* Nonblocking */
    if (ready == -1)
        errExit("poll");

    printf("poll() returned: %d\n", ready);

    /* Check which pipes have data available for reading */

    for (j = 0; j < numPipes; j++)
        if (pollFd[j].revents & POLLIN)
            printf("Readable: %d %3d\n", j, pollFd[j].fd);

    exit(EXIT_SUCCESS);
}
```

altio/poll_pipes.c

63.2.3 When Is a File Descriptor Ready?

Correctly using *select()* and *poll()* requires an understanding of the conditions under which a file descriptor indicates as being ready. SUSv3 says that a file descriptor (with `O_NONBLOCK` clear) is considered to be ready if a call to an I/O function would not block, *regardless of whether the function would actually transfer data*. The key point is italicized: *select()* and *poll()* tell us whether an I/O operation would not block, rather than whether it would successfully transfer data. In this light, let us consider how these system calls operate for different types of file descriptors. We show this information in tables containing two columns:

- The *select()* column indicates whether a file descriptor is marked as readable (`r`), writable (`w`), or having an exceptional condition (`x`).
- The *poll()* column indicates the bit(s) returned in the *revents* field. In these tables, we omit mention of `POLLRDNORM`, `POLLWRNORM`, `POLLRDBAND`, and `POLLWRBAND`. Although some of these flags may be returned in *revents* in various circumstances (if they are specified in *events*), they convey no useful information beyond that provided by `POLLIN`, `POLLOUT`, `POLLHUP`, and `POLLERR`.

Regular files

File descriptors that refer to regular files are always marked as readable and writable by *select()*, and returned with POLLIN and POLLOUT set in *revents* for *poll()*, for the following reasons:

- A *read()* will always immediately return data, end-of-file, or an error (e.g., the file was not opened for reading).
- A *write()* will always immediately transfer data or fail with some error.

> SUSv3 says that *select()* should also mark a descriptor for a regular file as having an exceptional condition (though this has no obvious meaning for regular files). Only some implementations do this; Linux is one of those that do not.

Terminals and pseudoterminals

Table 63-3 summarizes the behavior of *select()* and *poll()* for terminals and pseudoterminals (Chapter 64).

When one half of a pseudoterminal pair is closed, the *revents* setting returned by *poll()* for the other half of the pair depends on the implementation. On Linux, at least the POLLHUP flag is set. However, other implementations return various flags to indicate this event—for example, POLLHUP, POLLERR, or POLLIN. Furthermore, on some implementations, the flags that are set depend on whether it is the master or the slave device that is being monitored.

Table 63-3: *select()* and *poll()* indications for terminals and pseudoterminals

Condition or event	*select()*	*poll()*
Input available	r	POLLIN
Output possible	w	POLLOUT
After *close()* by pseudoterminal peer	rw	See text
Pseudoterminal master in packet mode detects slave state change	x	POLLPRI

Pipes and FIFOs

Table 63-4 summarizes the details for the read end of a pipe or FIFO. The *Data in pipe?* column indicates whether the pipe has at least 1 byte of data available for reading. In this table, we assume that POLLIN was specified in the *events* field for *poll()*.

On some other UNIX implementations, if the write end of a pipe is closed, then, instead of returning with POLLHUP set, *poll()* returns with the POLLIN bit set (since a *read()* will return immediately with end-of-file). Portable applications should check to see if either bit is set in order to know if a *read()* will block.

Table 63-5 summarizes the details for the write end of a pipe. In this table, we assume that POLLOUT was specified in the *events* field for *poll()*. The *Space for PIPE_BUF bytes?* column indicates whether the pipe has room to atomically write PIPE_BUF bytes without blocking. This is the criterion on which Linux considers a pipe ready for writing. Some other UNIX implementations use the same criterion; others consider a pipe writable if even a single byte can be written. (In Linux 2.6.10 and earlier, the capacity of a pipe is the same as PIPE_BUF. This means that a pipe is considered unwritable if it contains even a single byte of data.)

On some other UNIX implementations, if the read end of a pipe is closed, then, instead of returning with POLLERR set, *poll()* returns with either the POLLOUT bit or the POLLHUP bit set. Portable applications need to check to see if any of these bits is set to determine if a *write()* will block.

Table 63-4: *select()* and *poll()* indications for the read end of a pipe or FIFO

Condition or event		*select()*	*poll()*
Data in pipe?	**Write end open?**		
no	no	r	POLLHUP
yes	yes	r	POLLIN
yes	no	r	POLLIN | POLLHUP

Table 63-5: *select()* and *poll()* indications for the write end of a pipe or FIFO

Condition or event		*select()*	*poll()*
Space for PIPE_BUF bytes?	**Read end open?**		
no	no	w	POLLERR
yes	yes	w	POLLOUT
yes	no	w	POLLOUT | POLLERR

Sockets

Table 63-6 summarizes the behavior of *select()* and *poll()* for sockets. For the *poll()* column, we assume that *events* was specified as (POLLIN | POLLOUT | POLLPRI). For the *select()* column, we assume that the file descriptor is being tested to see if input is possible, output is possible, or an exceptional condition occurred (i.e., the file descriptor is specified in all three sets passed to *select()*). This table covers just the common cases, not all possible scenarios.

> The Linux *poll()* behavior for UNIX domain sockets after a peer *close()* differs from that shown in Table 63-6. As well as the other flags, *poll()* additionally returns POLLHUP in *revents*.

Table 63-6: *select()* and *poll()* indications for sockets

Condition or event	*select()*	*poll()*
Input available	r	POLLIN
Output possible	w	POLLOUT
Incoming connection established on listening socket	r	POLLIN
Out-of-band data received (TCP only)	x	POLLPRI
Stream socket peer closed connection or executed *shutdown(SHUT_WR)*	rw	POLLIN | POLLOUT | POLLRDHUP

The Linux-specific POLLRDHUP flag (available since Linux 2.6.17) needs a little further explanation. This flag—actually in the form of EPOLLRDHUP—is designed primarily for use with the edge-triggered mode of the *epoll* API (Section 63.4). It is returned when the remote end of a stream socket connection has shut down the writing half

of the connection. The use of this flag allows an application that uses the *epoll* edge-triggered interface to employ simpler code to recognize a remote shutdown. (The alternative is for the application to note that the POLLIN flag is set and then perform a *read()*, which indicates the remote shutdown with a return of 0.)

63.2.4 Comparison of *select()* and *poll()*

In this section, we consider some similarities and differences between *select()* and *poll()*.

Implementation details

Within the Linux kernel, *select()* and *poll()* both employ the same set of kernel-internal *poll* routines. These *poll* routines are distinct from the *poll()* system call itself. Each routine returns information about the readiness of a single file descriptor. This readiness information takes the form of a bit mask whose values correspond to the bits returned in the *revents* field by the *poll()* system call (Table 63-2). The implementation of the *poll()* system call involves calling the kernel *poll* routine for each file descriptor and placing the resulting information in the corresponding *revents* field.

To implement *select()*, a set of macros is used to convert the information returned by the kernel *poll* routines into the corresponding event types returned by *select()*:

```
#define POLLIN_SET  (POLLRDNORM | POLLRDBAND | POLLIN | POLLHUP | POLLERR)
                                    /* Ready for reading */
#define POLLOUT_SET (POLLWRBAND | POLLWRNORM | POLLOUT | POLLERR)
                                    /* Ready for writing */
#define POLLEX_SET  (POLLPRI)       /* Exceptional condition */
```

These macro definitions reveal the semantic correspondence between the information returned by *select()* and *poll()*. (If we look at the *select()* and *poll()* columns in the tables in Section 63.2.3, we see that the indications provided by each system call are consistent with the above macros.) The only additional information we need to complete the picture is that *poll()* returns POLLNVAL in the *revents* field if one of the monitored file descriptors was closed at the time of the call, while *select()* returns −1 with *errno* set to EBADF.

API differences

The following are some differences between the *select()* and *poll()* APIs:

- The use of the *fd_set* data type places an upper limit (FD_SETSIZE) on the range of file descriptors that can be monitored by *select()*. By default, this limit is 1024 on Linux, and changing it requires recompiling the application. By contrast, *poll()* places no intrinsic limit on the range of file descriptors that can be monitored.
- Because the *fd_set* arguments of *select()* are value-result, we must reinitialize them if making repeated *select()* calls from within a loop. By using separate *events* (input) and *revents* (output) fields, *poll()* avoids this requirement.

- The *timeout* precision afforded by *select()* (microseconds) is greater than that afforded by *poll()* (milliseconds). (The accuracy of the timeouts of both of these system calls is nevertheless limited by the software clock granularity.)
- If one of the file descriptors being monitored was closed, then *poll()* informs us exactly which one, via the `POLLNVAL` bit in the corresponding *revents* field. By contrast, *select()* merely returns −1 with *errno* set to `EBADF`, leaving us to determine which file descriptor is closed by checking for an error when performing an I/O system call on the descriptor. However, this is typically not an important difference, since an application can usually keep track of which file descriptors it has closed.

Portability

Historically, *select()* was more widely available than *poll()*. Nowadays, both interfaces are standardized by SUSv3 and widely available on contemporary implementations. However, there is some variation in the behavior of *poll()* across implementations, as noted in Section 63.2.3.

Performance

The performance of *poll()* and *select()* is similar if either of the following is true:

- The range of file descriptors to be monitored is small (i.e., the maximum file descriptor number is low).
- A large number of file descriptors are being monitored, but they are densely packed (i.e., most or all of the file descriptors from 0 up to some limit are being monitored).

However, the performance of *select()* and *poll()* can differ noticeably if the set of file descriptors to be monitored is sparse; that is, the maximum file descriptor number, *N*, is large, but only one or a few descriptors in the range 0 to *N* are being monitored. In this case, *poll()* can perform better than *select()*. We can understand the reasons for this by considering the arguments passed to the two system calls. With *select()*, we pass one or more file descriptor sets and an integer, *nfds*, which is one greater than the maximum file descriptor to be examined in each set. The *nfds* argument has the same value, regardless of whether we are monitoring all file descriptors in the range 0 to *(nfds − 1)* or only the descriptor *(nfds − 1)*. In both cases, the kernel must examine *nfds* elements in each set in order to check exactly which file descriptors are to be monitored. By contrast, when using *poll()*, we specify only the file descriptors of interest to us, and the kernel checks only those descriptors.

> The difference in performance for *poll()* and *select()* with sparse descriptor sets was quite significant in Linux 2.4. Some optimizations in Linux 2.6 have narrowed the performance gap considerably.

We consider the performance of *select()* and *poll()* further in Section 63.4.5, where we compare the performance of these system calls against *epoll*.

63.2.5 Problems with *select()* and *poll()*

The *select()* and *poll()* system calls are the portable, long-standing, and widely used methods of monitoring multiple file descriptors for readiness. However, these APIs suffer some problems when monitoring a large number of file descriptors:

- On each call to *select()* or *poll()*, the kernel must check all of the specified file descriptors to see if they are ready. When monitoring a large number of file descriptors that are in a densely packed range, the time required for this operation greatly outweighs the time required for the next two operations.
- In each call to *select()* or *poll()*, the program must pass a data structure to the kernel describing all of the file descriptors to be monitored, and, after checking the descriptors, the kernel returns a modified version of this data structure to the program. (Furthermore, for *select()*, we must initialize the data structure before each call.) For *poll()*, the size of the data structure increases with the number of file descriptors being monitored, and the task of copying it from user to kernel space and back again consumes a noticeable amount of CPU time when monitoring many file descriptors. For *select()*, the size of the data structure is fixed by `FD_SETSIZE`, regardless of the number of file descriptors being monitored.
- After the call to *select()* or *poll()*, the program must inspect every element of the returned data structure to see which file descriptors are ready.

The consequence of the above points is that the CPU time required by *select()* and *poll()* increases with the number of file descriptors being monitored (see Section 63.4.5 for more details). This creates problems for programs that monitor large numbers of file descriptors.

The poor scaling performance of *select()* and *poll()* stems from a simple limitation of these APIs: typically, a program makes repeated calls to monitor the same set of file descriptors; however, the kernel doesn't remember the list of file descriptors to be monitored between successive calls.

Signal-driven I/O and *epoll*, which we examine in the following sections, are both mechanisms that allow the kernel to record a persistent list of file descriptors in which a process is interested. Doing this eliminates the performance scaling problems of *select()* and *poll()*, yielding solutions that scale according to the number of I/O events that occur, rather than according to the number of file descriptors being monitored. Consequently, signal-driven I/O and *epoll* provide superior performance when monitoring large numbers of file descriptors.

63.3 Signal-Driven I/O

With I/O multiplexing, a process makes a system call (*select()* or *poll()*) in order to check whether I/O is possible on a file descriptor. With signal-driven I/O, a process requests that the kernel send it a signal when I/O is possible on a file descriptor. The process can then perform any other activity until I/O is possible, at which time

the signal is delivered to the process. To use signal-driven I/O, a program performs the following steps:

1. Establish a handler for the signal delivered by the signal-driven I/O mechanism. By default, this notification signal is SIGIO.
2. Set the *owner* of the file descriptor—that is, the process or process group that is to receive signals when I/O is possible on the file descriptor. Typically, we make the calling process the owner. The owner is set using an *fcntl()* F_SETOWN operation of the following form:

```
fcntl(fd, F_SETOWN, pid);
```

3. Enable nonblocking I/O by setting the O_NONBLOCK open file status flag.
4. Enable signal-driven I/O by turning on the O_ASYNC open file status flag. This can be combined with the previous step, since they both require the use of the *fcntl()* F_SETFL operation (Section 5.3), as in the following example:

```
flags = fcntl(fd, F_GETFL);                 /* Get current flags */
fcntl(fd, F_SETFL, flags | O_ASYNC | O_NONBLOCK);
```

5. The calling process can now perform other tasks. When I/O becomes possible, the kernel generates a signal for the process and invokes the signal handler established in step 1.
6. Signal-driven I/O provides edge-triggered notification (Section 63.1.1). This means that once the process has been notified that I/O is possible, it should perform as much I/O (e.g., read as many bytes) as possible. Assuming a nonblocking file descriptor, this means executing a loop that performs I/O system calls until a call fails with the error EAGAIN or EWOULDBLOCK.

On Linux 2.4 and earlier, signal-driven I/O can be employed with file descriptors for sockets, terminals, pseudoterminals, and certain other types of devices. Linux 2.6 additionally allows signal-driven I/O to be employed with pipes and FIFOs. Since Linux 2.6.25, signal-driven I/O can also be used with *inotify* file descriptors.

In the following pages, we first present an example of the use of signal-driven I/O, and then explain some of the above steps in greater detail.

> Historically, signal-driven I/O was sometimes referred to as *asynchronous I/O*, and this is reflected in the name (O_ASYNC) of the associated open file status flag. However, nowadays, the term *asynchronous I/O* is used to refer to the type of functionality provided by the POSIX AIO specification. Using POSIX AIO, a process requests the kernel to perform an I/O operation, and the kernel *initiates* the operation, but immediately passes control back to the calling process; the process is then later notified when the I/O operation completes or an error occurs.
>
> O_ASYNC was specified in POSIX.1g, but was not included in SUSv3 because the specification of the required behavior for this flag was deemed insufficient.
>
> Several UNIX implementations, especially older ones, don't define the O_ASYNC constant for use with *fcntl()*. Instead, the constant is named FASYNC, and *glibc* defines this latter name as a synonym for O_ASYNC.

Example program

Listing 63-3 provides a simple example of the use of signal-driven I/O. This program performs the steps described above for enabling signal-driven I/O on standard input, and then places the terminal in cbreak mode (Section 62.6.3), so that input is available a character at a time. The program then enters an infinite loop, performing the "work" of incrementing a variable, *cnt*, while waiting for input to become available. Whenever input becomes available, the SIGIO handler sets a flag, *gotSigio,* that is monitored by the main program. When the main program sees that this flag is set, it reads all available input characters and prints them along with the current value of *cnt*. If a hash character (#) is read in the input, the program terminates.

Here is an example of what we see when we run this program and type the *x* character a number of times, followed by a hash (#) character:

```
$ ./demo_sigio
cnt=37; read x
cnt=100; read x
cnt=159; read x
cnt=223; read x
cnt=288; read x
cnt=333; read #
```

Listing 63-3: Using signal-driven I/O on a terminal

altio/demo_sigio.c

```
#include <signal.h>
#include <ctype.h>
#include <fcntl.h>
#include <termios.h>
#include "tty_functions.h"      /* Declaration of ttySetCbreak() */
#include "tlpi_hdr.h"

static volatile sig_atomic_t gotSigio = 0;
                                /* Set nonzero on receipt of SIGIO */

static void
sigioHandler(int sig)
{
    gotSigio = 1;
}

int
main(int argc, char *argv[])
{
    int flags, j, cnt;
    struct termios origTermios;
    char ch;
    struct sigaction sa;
    Boolean done;
```

```
    /* Establish handler for "I/O possible" signal */

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sa.sa_handler = sigioHandler;
    if (sigaction(SIGIO, &sa, NULL) == -1)
        errExit("sigaction");

    /* Set owner process that is to receive "I/O possible" signal */

    if (fcntl(STDIN_FILENO, F_SETOWN, getpid()) == -1)
        errExit("fcntl(F_SETOWN)");

    /* Enable "I/O possible" signaling and make I/O nonblocking
       for file descriptor */

    flags = fcntl(STDIN_FILENO, F_GETFL);
    if (fcntl(STDIN_FILENO, F_SETFL, flags | O_ASYNC | O_NONBLOCK) == -1)
        errExit("fcntl(F_SETFL)");

    /* Place terminal in cbreak mode */

    if (ttySetCbreak(STDIN_FILENO, &origTermios) == -1)
        errExit("ttySetCbreak");

    for (done = FALSE, cnt = 0; !done ; cnt++) {
        for (j = 0; j < 100000000; j++)
            continue;                   /* Slow main loop down a little */

        if (gotSigio) {                 /* Is input available? */

            /* Read all available input until error (probably EAGAIN)
               or EOF (not actually possible in cbreak mode) or a
               hash (#) character is read */

            while (read(STDIN_FILENO, &ch, 1) > 0 && !done) {
                printf("cnt=%d; read %c\n", cnt, ch);
                done = ch == '#';
            }

            gotSigio = 0;
        }
    }

    /* Restore original terminal settings */

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &origTermios) == -1)
        errExit("tcsetattr");
    exit(EXIT_SUCCESS);
}
```

altio/demo_sigio.c

Establish the signal handler before enabling signal-driven I/O

Because the default action of SIGIO is to terminate the process, we should enable the handler for SIGIO before enabling signal-driven I/O on a file descriptor. If we enable signal-driven I/O before establishing the SIGIO handler, then there is a time window during which, if I/O becomes possible, delivery of SIGIO will terminate the process.

> On some UNIX implementations, SIGIO is ignored by default.

Setting the file descriptor owner

We set the file descriptor owner using an *fcntl()* operation of the following form:

```
fcntl(fd, F_SETOWN, pid);
```

We may specify that either a single process or all of the processes in a process group are to be signaled when I/O is possible on the file descriptor. If *pid* is positive, it is interpreted as a process ID. If *pid* is negative, its absolute value specifies a process group ID.

> On older UNIX implementations, an *ioctl()* operation—either FIOSETOWN or SIOCSPGRP—was used to achieve the same effect as F_SETOWN. For compatibility, these *ioctl()* operations are also provided on Linux.

Typically, *pid* is specified as the process ID of the calling process (so that the signal is sent to the process that has the file descriptor open). However, it is possible to specify another process or a process group (e.g., the caller's process group), and signals will be sent to that target, subject to the permission checks described in Section 20.5, where the sending process is considered to be the process that does the F_SETOWN.

The *fcntl()* F_GETOWN operation returns the ID of the process or process group that is to receive signals when I/O is possible on a specified file descriptor:

```
id = fcntl(fd, F_GETOWN);
if (id == -1)
    errExit("fcntl");
```

A process group ID is returned as a negative number by this call.

> The *ioctl()* operation that corresponds to F_GETOWN on older UNIX implementations was FIOGETOWN or SIOCGPGRP. Both of these *ioctl()* operations are also provided on Linux.

A limitation in the system call convention employed on some Linux architectures (notably, x86) means that if a file descriptor is owned by a process group ID less than 4096, then, instead of returning that ID as a negative function result from the *fcntl()* F_GETOWN operation, *glibc* misinterprets it as a system call error. Consequently, the *fcntl()* wrapper function returns −1, and *errno* contains the (positive) process group ID. This is a consequence of the fact that the kernel system call interface indicates errors by returning a negative *errno* value as a function result, and there are a few cases where it is necessary to distinguish such results from a successful call that returns a valid negative value. To make this distinction, *glibc* interprets negative

system call returns in the range −1 to −4095 as indicating an error, copies this (absolute) value into *errno*, and returns −1 as the function result for the application program. This technique is generally sufficient for dealing with the few system call service routines that can return a valid negative result; the *fcntl()* `F_GETOWN` operation is the only practical case where it fails. This limitation means that an application that uses process groups to receive "I/O possible" signals (which is unusual) can't reliably use `F_GETOWN` to discover which process group owns a file descriptor.

> Since *glibc* version 2.11, the *fcntl()* wrapper function fixes the problem of `F_GETOWN` with process group IDs less than 4096. It does this by implementing `F_GETOWN` in user space using the `F_GETOWN_EX` operation (Section 63.3.2), which is provided by Linux 2.6.32 and later.

63.3.1 When Is "I/O Possible" Signaled?

We now consider the details of when "I/O possible" is signaled for various file types.

Terminals and pseudoterminals

For terminals and pseudoterminals, a signal is generated whenever new input becomes available, even if previous input has not yet been read. "Input possible" is also signaled if an end-of-file condition occurs on a terminal (but not on a pseudoterminal).

There is no "output possible" signaling for terminals. A terminal disconnect is also not signaled.

Starting with kernel 2.4.19, Linux provides "output possible" signaling for the slave side of a pseudoterminal. This signal is generated whenever input is consumed on the master side of the pseudoterminal.

Pipes and FIFOs

For the read end of a pipe or FIFO, a signal is generated in these circumstances:

- Data is written to the pipe (even if there was already unread input available).
- The write end of the pipe is closed.

For the write end of a pipe or FIFO, a signal is generated in these circumstances:

- A read from the pipe increases the amount of free space in the pipe so that it is now possible to write `PIPE_BUF` bytes without blocking.
- The read end of the pipe is closed.

Sockets

Signal-driven I/O works for datagram sockets in both the UNIX and the Internet domains. A signal is generated in the following circumstances:

- An input datagram arrives on the socket (even if there were already unread datagrams waiting to be read).
- An asynchronous error occurs on the socket.

Signal-driven I/O works for stream sockets in both the UNIX and the Internet domains. A signal is generated in the following circumstances:

- A new connection is received on a listening socket.
- A TCP *connect()* request completes; that is, the active end of a TCP connection entered the ESTABLISHED state, as shown in Figure 61-5 (page 1272). The analogous condition is not signaled for UNIX domain sockets.
- New input is received on the socket (even if there was already unread input available).
- The peer closes its writing half of the connection using *shutdown()*, or closes its socket altogether using *close()*.
- Output is possible on the socket (e.g., space has become available in the socket send buffer).
- An asynchronous error occurs on the socket.

inotify file descriptors

A signal is generated when the *inotify* file descriptor becomes readable—that is, when an event occurs for one of the files monitored by the *inotify* file descriptor.

63.3.2 Refining the Use of Signal-Driven I/O

In applications that need to simultaneously monitor very large numbers (i.e., thousands) of file descriptors—for example, certain types of network servers—signal-driven I/O can provide significant performance advantages by comparison with *select()* and *poll()*. Signal-driven I/O offers superior performance because the kernel "remembers" the list of file descriptors to be monitored, and signals the program only when I/O events actually occur on those descriptors. As a result, the performance of a program employing signal-driven I/O scales according to the number of I/O events that occur, rather than the number of file descriptors being monitored.

To take full advantage of signal-driven I/O, we must perform two steps:

- Employ a Linux-specific *fcntl()* operation, `F_SETSIG`, to specify a realtime signal that should be delivered instead of `SIGIO` when I/O is possible on a file descriptor.
- Specify the `SA_SIGINFO` flag when using *sigaction()* to establish the handler for the realtime signal employed in the previous step (see Section 21.4).

The *fcntl()* `F_SETSIG` operation specifies an alternative signal that should be delivered instead of `SIGIO` when I/O is possible on a file descriptor:

```
if (fcntl(fd, F_SETSIG, sig) == -1)
    errExit("fcntl");
```

The `F_GETSIG` operation performs the converse of `F_SETSIG`, retrieving the signal currently set for a file descriptor:

```
sig = fcntl(fd, F_GETSIG);
if (sig == -1)
    errExit("fcntl");
```

(In order to obtain the definitions of the F_SETSIG and F_GETSIG constants from <fcntl.h>, we must define the _GNU_SOURCE feature test macro.)

Using F_SETSIG to change the signal used for "I/O possible" notification serves two purposes, both of which are needed if we are monitoring large numbers of I/O events on multiple file descriptors:

- The default "I/O possible" signal, SIGIO, is one of the standard, nonqueuing signals. If multiple I/O events are signaled while SIGIO is blocked—perhaps because the SIGIO handler is already invoked—all notifications except the first will be lost. If we use F_SETSIG to specify a realtime signal as the "I/O possible" signal, multiple notifications can be queued.
- If the handler for the signal is established using a *sigaction()* call in which the SA_SIGINFO flag is specified in the *sa.sa_flags* field, then a *siginfo_t* structure is passed as the second argument to the signal handler (Section 21.4). This structure contains fields identifying the file descriptor on which the event occurred, as well as the type of event.

Note that the use of *both* F_SETSIG and SA_SIGINFO is required in order for a valid *siginfo_t* structure to be passed to the signal handler.

If we perform an F_SETSIG operation specifying *sig* as 0, then we return to the default behavior: SIGIO is delivered, and a *siginfo_t* argument is not supplied to the handler.

For an "I/O possible" event, the fields of interest in the *siginfo_t* structure passed to the signal handler are as follows:

- *si_signo*: the number of the signal that caused the invocation of the handler. This value is the same as the first argument to the signal handler.
- *si_fd*: the file descriptor for which the I/O event occurred.
- *si_code*: a code indicating the type of event that occurred. The values that can appear in this field, along with their general descriptions, are shown in Table 63-7.
- *si_band*: a bit mask containing the same bits as are returned in the *revents* field by the *poll()* system call. The value set in *si_code* has a one-to-one correspondence with the bit-mask setting in *si_band*, as shown in Table 63-7.

Table 63-7: *si_code* and *si_band* values in the *siginfo_t* structure for "I/O possible" events

si_code	*si_band* mask value	Description
POLL_IN	POLLIN | POLLRDNORM	Input available; end-of-file condition
POLL_OUT	POLLOUT | POLLWRNORM | POLLWRBAND	Output possible
POLL_MSG	POLLIN | POLLRDNORM | POLLMSG	Input message available (unused)
POLL_ERR	POLLERR	I/O error
POLL_PRI	POLLPRI | POLLRDNORM	High-priority input available
POLL_HUP	POLLHUP | POLLERR	Hangup occurred

In an application that is purely input-driven, we can further refine the use of F_SETSIG. Instead of monitoring I/O events via a signal handler, we can block the nominated "I/O possible" signal, and then accept the queued signals via calls to *sigwaitinfo()* or

sigtimedwait() (Section 22.10). These system calls return a *siginfo_t* structure that contains the same information as is passed to a signal handler established with SA_SIGINFO. Accepting signals in this manner returns us to a synchronous model of event processing, but with the advantage that we are much more efficiently notified about the file descriptors on which I/O events have occurred than if we use *select()* or *poll()*.

Handling signal-queue overflow

We saw in Section 22.8 that there is a limit on the number of realtime signals that may be queued. If this limit is reached, the kernel reverts to delivering the default SIGIO signal for "I/O possible" notifications. This informs the process that a signal-queue overflow occurred. When this happens, we lose information about which file descriptors have I/O events, because SIGIO is not queued. (Furthermore, the SIGIO handler doesn't receive a *siginfo_t* argument, which means that the signal handler can't determine the file descriptor that generated the signal.)

We can reduce the likelihood of signal-queue overflows by increasing the limit on the number of realtime signals that can be queued, as described in Section 22.8. However, this doesn't eliminate the need to handle the possibility of an overflow. A properly designed application using F_SETSIG to establish a realtime signal as the "I/O possible" notification mechanism must also establish a handler for SIGIO. If SIGIO is delivered, then the application can drain the queue of realtime signals using *sigwaitinfo()* and temporarily revert to the use of *select()* or *poll()* to obtain a complete list of file descriptors with outstanding I/O events.

Using signal-driven I/O with multithreaded applications

Starting with kernel 2.6.32, Linux provides two new, nonstandard *fcntl()* operations that can be used to set the target for "I/O possible" signals: F_SETOWN_EX and F_GETOWN_EX.

The F_SETOWN_EX operation is like F_SETOWN, but as well as allowing the target to be specified as a process or process group, it also permits a thread to be specified as the target for "I/O possible" signals. For this operation, the third argument of *fcntl()* is a pointer to a structure of the following form:

```
struct f_owner_ex {
    int   type;
    pid_t pid;
};
```

The *type* field defines the meaning of the *pid* field, and has one of the following values:

F_OWNER_PGRP
: The *pid* field specifies the ID of a process group that is to be the target of "I/O possible" signals. Unlike with F_SETOWN, a process group ID is specified as a positive value.

F_OWNER_PID
: The *pid* field specifies the ID of a process that is to be the target of "I/O possible" signals.

F_OWNER_TID
: The *pid* field specifies the ID of a thread that is to be the target of "I/O possible" signals. The ID specified in *pid* is a value returned by *clone()* or *gettid()*.

The F_GETOWN_EX operation is the converse of the F_GETOWN_EX operation. It uses the *f_owner_ex* structure pointed to by the third argument of *fcntl()* to return the settings defined by a previous F_SETOWN_EX operation.

> Because the F_SETOWN_EX and F_GETOWN_EX operations represent process group IDs as positive values, F_GETOWN_EX doesn't suffer the problem described earlier for F_GETOWN when using process group IDs less than 4096.

63.4 The *epoll* API

Like the I/O multiplexing system calls and signal-driven I/O, the Linux *epoll* (event poll) API is used to monitor multiple file descriptors to see if they are ready for I/O. The primary advantages of the *epoll* API are the following:

- The performance of *epoll* scales much better than *select()* and *poll()* when monitoring large numbers of file descriptors.
- The *epoll* API permits either level-triggered or edge-triggered notification. By contrast, *select()* and *poll()* provide only level-triggered notification, and signal-driven I/O provides only edge-triggered notification.

The performance of *epoll* and signal-driven I/O is similar. However, *epoll* has some advantages over signal-driven I/O:

- We avoid the complexities of signal handling (e.g., signal-queue overflow).
- We have greater flexibility in specifying what kind of monitoring we want to perform (e.g., checking to see if a file descriptor for a socket is ready for reading, writing, or both).

The *epoll* API is Linux-specific, and is new in Linux 2.6.

The central data structure of the *epoll* API is an *epoll instance*, which is referred to via an open file descriptor. This file descriptor is not used for I/O. Instead, it is a handle for kernel data structures that serve two purposes:

- recording a list of file descriptors that this process has declared an interest in monitoring—the *interest list*; and
- maintaining a list of file descriptors that are ready for I/O—the *ready list*.

The membership of the ready list is a subset of the interest list.

For each file descriptor monitored by *epoll*, we can specify a bit mask indicating events that we are interested in knowing about. These bit masks correspond closely to the bit masks used with *poll()*.

The *epoll* API consists of three system calls:

- The *epoll_create()* system call creates an *epoll* instance and returns a file descriptor referring to the instance.

- The *epoll_ctl()* system call manipulates the interest list associated with an *epoll* instance. Using *epoll_ctl()*, we can add a new file descriptor to the list, remove an existing descriptor from the list, and modify the mask that determines which events are to be monitored for a descriptor.
- The *epoll_wait()* system call returns items from the ready list associated with an *epoll* instance.

63.4.1 Creating an *epoll* Instance: *epoll_create()*

The *epoll_create()* system call creates a new *epoll* instance whose interest list is initially empty.

```
#include <sys/epoll.h>

int epoll_create(int size);
```

Returns file descriptor on success, or −1 on error

The *size* argument specifies the number of file descriptors that we expect to monitor via the *epoll* instance. This argument is not an upper limit, but rather a hint to the kernel about how to initially dimension internal data structures. (Since Linux 2.6.8, the *size* argument is ignored, because changes in the implementation meant that the information it provided is no longer required.)

As its function result, *epoll_create()* returns a file descriptor referring to the new *epoll* instance. This file descriptor is used to refer to the *epoll* instance in other *epoll* system calls. When the file descriptor is no longer required, it should be closed in the usual way, using *close()*. When all file descriptors referring to an *epoll* instance are closed, the instance is destroyed and its associated resources are released back to the system. (Multiple file descriptors may refer to the same *epoll* instance as a consequence of calls to *fork()* or descriptor duplication using *dup()* or similar.)

> Starting with kernel 2.6.27, Linux supports a new system call, *epoll_create1()*. This system call performs the same task as *epoll_create()*, but drops the obsolete *size* argument and adds a *flags* argument that can be used to modify the behavior of the system call. One flag is currently supported: `EPOLL_CLOEXEC`, which causes the kernel to enable the close-on-exec flag (`FD_CLOEXEC`) for the new file descriptor. This flag is useful for the same reasons as the *open()* `O_CLOEXEC` flag described in Section 4.3.1.

63.4.2 Modifying the *epoll* Interest List: *epoll_ctl()*

The *epoll_ctl()* system call modifies the interest list of the *epoll* instance referred to by the file descriptor *epfd*.

```
#include <sys/epoll.h>

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *ev);
```

Returns 0 on success, or −1 on error

The *fd* argument identifies which of the file descriptors in the interest list is to have its settings modified. This argument can be a file descriptor for a pipe, FIFO, socket, POSIX message queue, *inotify* instance, terminal, device, or even another *epoll* descriptor (i.e., we can build a kind of hierarchy of monitored descriptors). However, *fd* can't be a file descriptor for a regular file or a directory (the error `EPERM` results).

The *op* argument specifies the operation to be performed, and has one of the following values:

`EPOLL_CTL_ADD`
: Add the file descriptor *fd* to the interest list for *epfd*. The set of events that we are interested in monitoring for *fd* is specified in the buffer pointed to by *ev*, as described below. If we attempt to add a file descriptor that is already in the interest list, *epoll_ctl()* fails with the error `EEXIST`.

`EPOLL_CTL_MOD`
: Modify the events setting for the file descriptor *fd*, using the information specified in the buffer pointed to by *ev*. If we attempt to modify the settings of a file descriptor that is not in the interest list for *epfd*, *epoll_ctl()* fails with the error `ENOENT`.

`EPOLL_CTL_DEL`
: Remove the file descriptor *fd* from the interest list for *epfd*. The *ev* argument is ignored for this operation. If we attempt to remove a file descriptor that is not in the interest list for *epfd*, *epoll_ctl()* fails with the error `ENOENT`. Closing a file descriptor automatically removes it from all of the *epoll* interest lists of which it is a member.

The *ev* argument is a pointer to a structure of type *epoll_event*, defined as follows:

```
struct epoll_event {
    uint32_t     events;        /* epoll events (bit mask) */
    epoll_data_t data;          /* User data */
};
```

The *data* field of the *epoll_event* structure is typed as follows:

```
typedef union epoll_data {
    void        *ptr;           /* Pointer to user-defined data */
    int          fd;            /* File descriptor */
    uint32_t     u32;           /* 32-bit integer */
    uint64_t     u64;           /* 64-bit integer */
} epoll_data_t;
```

The *ev* argument specifies settings for the file descriptor *fd*, as follows:

- The *events* subfield is a bit mask specifying the set of events that we are interested in monitoring for *fd*. We say more about the bit values that can be used in this field in the next section.
- The *data* subfield is a union, one of whose members can be used to specify information that is passed back to the calling process (via *epoll_wait()*) if *fd* later becomes ready.

Listing 63-4 shows an example of the use of *epoll_create()* and *epoll_ctl()*.

Listing 63-4: Using *epoll_create()* and *epoll_ctl()*

```
int epfd;
struct epoll_event ev;

epfd = epoll_create(5);
if (epfd == -1)
    errExit("epoll_create");

ev.data.fd = fd;
ev.events = EPOLLIN;
if (epoll_ctl(epfd, EPOLL_CTL_ADD, fd, ev) == -1)
    errExit("epoll_ctl");
```

The `max_user_watches` limit

Because each file descriptor registered in an *epoll* interest list requires a small amount of nonswappable kernel memory, the kernel provides an interface that defines a limit on the total number of file descriptors that each user can register in all *epoll* interest lists. The value of this limit can be viewed and modified via `max_user_watches`, a Linux-specific file in the `/proc/sys/fs/epoll` directory. The default value of this limit is calculated based on available system memory (see the *epoll(7)* manual page).

63.4.3 Waiting for Events: *epoll_wait()*

The *epoll_wait()* system call returns information about ready file descriptors from the *epoll* instance referred to by the file descriptor *epfd*. A single *epoll_wait()* call can return information about multiple ready file descriptors.

```
#include <sys/epoll.h>

int epoll_wait(int epfd, struct epoll_event *evlist, int maxevents, int timeout);
```

Returns number of ready file descriptors, 0 on timeout, or –1 on error

The information about ready file descriptors is returned in the array of *epoll_event* structures pointed to by *evlist*. (The *epoll_event* structure was described in the previous section.) The *evlist* array is allocated by the caller, and the number of elements it contains is specified in *maxevents*.

Each item in the array *evlist* returns information about a single ready file descriptor. The *events* subfield returns a mask of the events that have occurred on this descriptor. The *data* subfield returns whatever value was specified in *ev.data* when we registered interest in this file descriptor using *epoll_ctl()*. Note that the *data* field provides the only mechanism for finding out the number of the file

descriptor associated with this event. Thus, when we make the *epoll_ctl()* call that places a file descriptor in the interest list, we should either set *ev.data.fd* to the file descriptor number (as shown in Listing 63-4) or set *ev.data.ptr* to point to a structure that contains the file descriptor number.

The *timeout* argument determines the blocking behavior of *epoll_wait()*, as follows:

- If *timeout* equals −1, block until an event occurs for one of the file descriptors in the interest list for *epfd* or until a signal is caught.
- If *timeout* equals 0, perform a nonblocking check to see which events are currently available on the file descriptors in the interest list for *epfd*.
- If *timeout* is greater than 0, block for up to *timeout* milliseconds, until an event occurs on one of the file descriptors in the interest list for *epfd*, or until a signal is caught.

On success, *epoll_wait()* returns the number of items that have been placed in the array *evlist*, or 0 if no file descriptors were ready within the interval specified by *timeout*. On error, *epoll_wait()* returns −1, with *errno* set to indicate the error.

In a multithreaded program, it is possible for one thread to use *epoll_ctl()* to add file descriptors to the interest list of an *epoll* instance that is already being monitored by *epoll_wait()* in another thread. These changes to the interest list will be taken into account immediately, and the *epoll_wait()* call will return readiness information about the newly added file descriptors.

epoll events

The bit values that can be specified in *ev.events* when we call *epoll_ctl()* and that are placed in the *evlist[].events* fields returned by *epoll_wait()* are shown in Table 63-8. With the addition of an `E` prefix, most of these bits have names that are the same as the corresponding event bits used with *poll()*. (The exceptions are `EPOLLET` and `EPOLLONESHOT`, which we describe in more detail below.) The reason for this correspondence is that, when specified as input to *epoll_ctl()* or returned as output via *epoll_wait()*, these bits convey exactly the same meaning as the corresponding *poll()* event bits.

Table 63-8: Bit-mask values for the *epoll events* field

Bit	Input to *epoll_ctl()*?	Returned by *epoll_wait()*?	Description
`EPOLLIN`	•	•	Data other than high-priority data can be read
`EPOLLPRI`	•	•	High-priority data can be read
`EPOLLRDHUP`	•	•	Shutdown on peer socket (since Linux 2.6.17)
`EPOLLOUT`	•	•	Normal data can be written
`EPOLLET`	•		Employ edge-triggered event notification
`EPOLLONESHOT`	•		Disable monitoring after event notification
`EPOLLERR`		•	An error has occurred
`EPOLLHUP`		•	A hangup has occurred

The EPOLLONESHOT flag

By default, once a file descriptor is added to an *epoll* interest list using the *epoll_ctl()* EPOLL_CTL_ADD operation, it remains active (i.e., subsequent calls to *epoll_wait()* will inform us whenever the file descriptor is ready) until we explicitly remove it from the list using the *epoll_ctl()* EPOLL_CTL_DEL operation. If we want to be notified only once about a particular file descriptor, then we can specify the EPOLLONESHOT flag (available since Linux 2.6.2) in the *ev.events* value passed in *epoll_ctl()*. If this flag is specified, then, after the next *epoll_wait()* call that informs us that the corresponding file descriptor is ready, the file descriptor is marked inactive in the interest list, and we won't be informed about its state by future *epoll_wait()* calls. If desired, we can subsequently reenable monitoring of this file descriptor using the *epoll_ctl()* EPOLL_CTL_MOD operation. (We can't use the EPOLL_CTL_ADD operation for this purpose, because the inactive file descriptor is still part of the *epoll* interest list.)

Example program

Listing 63-5 demonstrates the use of the *epoll* API. As command-line arguments, this program expects the pathnames of one or more terminals or FIFOs. The program performs the following steps:

- Create an *epoll* instance ①.
- Open each of the files named on the command line for input ② and add the resulting file descriptor to the interest list of the *epoll* instance ③, specifying the set of events to be monitored as EPOLLIN.
- Execute a loop ④ that calls *epoll_wait()* ⑤ to monitor the interest list of the *epoll* instance and handles the returned events from each call. Note the following points about this loop:
  - After the *epoll_wait()* call, the program checks for an EINTR return ⑥, which may occur if the program was stopped by a signal in the middle of the *epoll_wait()* call and then resumed by SIGCONT. (Refer to Section 21.5.) If this occurs, the program restarts the *epoll_wait()* call.
  - It the *epoll_wait()* call was successful, the program uses a further loop to check each of the ready items in *evlist* ⑦. For each item in *evlist*, the program checks the *events* field for the presence of not just EPOLLIN ⑧, but also EPOLLHUP and EPOLLERR ⑨. These latter events can occur if the other end of a FIFO was closed or a terminal hangup occurred. If EPOLLIN was returned, then the program reads some input from the corresponding file descriptor and displays it on standard output. Otherwise, if either EPOLLHUP or EPOLLERR occurred, the program closes the corresponding file descriptor ⑩ and decrements the counter of open files (*numOpenFds*).
  - The loop terminates when all open file descriptors have been closed (i.e., when *numOpenFds* equals 0).

The following shell session logs demonstrate the use of the program in Listing 63-5. We use two terminal windows. In one window, we use the program in Listing 63-5 to monitor two FIFOs for input. (Each open of a FIFO for reading by this program will complete only after another process has opened the FIFO for writing, as

described in Section 44.7.) In the other window, we run instances of *cat(1)* that write data to these FIFOs.

```
Terminal window 1                    Terminal window 2
$ mkfifo p q
$ ./epoll_input p q
                                     $ cat > p
Opened "p" on fd 4
                                     Type Control-Z to suspend cat
                                     [1]+  Stopped      cat >p
                                     $ cat > q
Opened "q" on fd 5
About to epoll_wait()
Type Control-Z to suspend the epoll_input program
[1]+  Stopped     ./epoll_input p q
```

Above, we suspended our monitoring program so that we can now generate input on both FIFOs, and close the write end of one of them:

```
                                     qqq
                                     Type Control-D to terminate "cat > q"
                                     $ fg %1
                                     cat >p
                                     ppp
```

Now we resume our monitoring program by bringing it into the foreground, at which point *epoll_wait()* returns two events:

```
$ fg
./epoll_input p q
About to epoll_wait()
Ready: 2
  fd=4; events: EPOLLIN
    read 4 bytes: ppp

  fd=5; events: EPOLLIN EPOLLHUP
    read 4 bytes: qqq

    closing fd 5
About to epoll_wait()
```

The two blank lines in the above output are the newlines that were read by the instances of *cat*, written to the FIFOs, and then read and echoed by our monitoring program.

Now we type *Control-D* in the second terminal window in order to terminate the remaining instance of *cat*, which causes *epoll_wait()* to once more return, this time with a single event:

```
                                     Type Control-D to terminate "cat >p"
Ready: 1
  fd=4; events: EPOLLHUP
    closing fd 4
All file descriptors closed; bye
```

Listing 63-5: Using the *epoll* API

altio/epoll_input.c

```
#include <sys/epoll.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

#define MAX_BUF     1000        /* Maximum bytes fetched by a single read() */
#define MAX_EVENTS     5        /* Maximum number of events to be returned from
                                   a single epoll_wait() call */

int
main(int argc, char *argv[])
{
    int epfd, ready, fd, s, j, numOpenFds;
    struct epoll_event ev;
    struct epoll_event evlist[MAX_EVENTS];
    char buf[MAX_BUF];

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s file...\n", argv[0]);

①   epfd = epoll_create(argc - 1);
    if (epfd == -1)
        errExit("epoll_create");

    /* Open each file on command line, and add it to the "interest
       list" for the epoll instance */

②   for (j = 1; j < argc; j++) {
        fd = open(argv[j], O_RDONLY);
        if (fd == -1)
            errExit("open");
        printf("Opened \"%s\" on fd %d\n", argv[j], fd);

        ev.events = EPOLLIN;            /* Only interested in input events */
        ev.data.fd = fd;
③       if (epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev) == -1)
            errExit("epoll_ctl");
    }

    numOpenFds = argc - 1;

④   while (numOpenFds > 0) {

        /* Fetch up to MAX_EVENTS items from the ready list */

        printf("About to epoll_wait()\n");
⑤       ready = epoll_wait(epfd, evlist, MAX_EVENTS, -1);
        if (ready == -1) {
⑥           if (errno == EINTR)
                continue;               /* Restart if interrupted by signal */
            else
                errExit("epoll_wait");
        }
```

```
            printf("Ready: %d\n", ready);

            /* Deal with returned list of events */

⑦           for (j = 0; j < ready; j++) {
                printf("  fd=%d; events: %s%s%s\n", evlist[j].data.fd,
                        (evlist[j].events & EPOLLIN)  ? "EPOLLIN "  : "",
                        (evlist[j].events & EPOLLHUP) ? "EPOLLHUP " : "",
                        (evlist[j].events & EPOLLERR) ? "EPOLLERR " : "");

⑧               if (evlist[j].events & EPOLLIN) {
                    s = read(evlist[j].data.fd, buf, MAX_BUF);
                    if (s == -1)
                        errExit("read");
                    printf("    read %d bytes: %.*s\n", s, s, buf);

⑨               } else if (evlist[j].events & (EPOLLHUP | EPOLLERR)) {

                    /* If EPOLLIN and EPOLLHUP were both set, then there might
                       be more than MAX_BUF bytes to read. Therefore, we close
                       the file descriptor only if EPOLLIN was not set.
                       We'll read further bytes after the next epoll_wait(). */

                    printf("    closing fd %d\n", evlist[j].data.fd);
⑩                   if (close(evlist[j].data.fd) == -1)
                        errExit("close");
                    numOpenFds--;
                }
            }
        }

        printf("All file descriptors closed; bye\n");
        exit(EXIT_SUCCESS);
    }
```

altio/epoll_input.c

63.4.4 A Closer Look at *epoll* Semantics

We now look at some subtleties of the interaction of open files, file descriptors, and *epoll*. For the purposes of this discussion, it is worth reviewing Figure 5-2 (page 95), which shows the relationship between file descriptors, open file descriptions, and the system-wide file i-node table.

When we create an *epoll* instance using *epoll_create()*, the kernel creates a new in-memory i-node and open file description, and allocates a new file descriptor in the calling process that refers to the open file description. The interest list for an *epoll* instance is associated with the open file description, not with the *epoll* file descriptor. This has the following consequences:

- If we duplicate an *epoll* file descriptor using *dup()* (or similar), then the duplicated descriptor refers to the same *epoll* interest and ready lists as the original descriptor. We may modify the interest list by specifying either file descriptor

as the *epfd* argument in a call to *epoll_ctl()*. Similarly, we can retrieve items from the ready list by specifying either file descriptor as the *epfd* argument in a call to *epoll_wait()*.

- The preceding point also applies after a call to *fork()*. The child inherits a duplicate of the parent's *epoll* file descriptor, and this duplicate descriptor refers to the same *epoll* data structures.

When we perform an *epoll_ctl()* `EPOLL_CTL_ADD` operation, the kernel adds an item to the *epoll* interest list that records both the number of the monitored file descriptor and a reference to the corresponding open file description. For the purpose of *epoll_wait()* calls, the kernel monitors the open file description. This means that we must refine our earlier statement that when a file descriptor is closed, it is automatically removed from any *epoll* interest lists of which it is a member. The refinement is this: an open file description is removed from the *epoll* interest list once all file descriptors that refer to it have been closed. This means that if we create duplicate descriptors referring to an open file—using *dup()* (or similar) or *fork()*—then the open file will be removed only after the original descriptor and all of the duplicates have been closed.

These semantics can lead to some behavior that at first appears surprising. Suppose that we execute the code shown in Listing 63-6. The *epoll_wait()* call in this code will tell us that the file descriptor *fd1* is ready (in other words, *evlist[0].data.fd* will be equal to *fd1*), even though *fd1* has been closed. This is because there is still one open file descriptor, *fd2*, referring to the open file description contained in the *epoll* interest list. A similar scenario occurs when two processes hold duplicate descriptors for the same open file description (typically, as a result of a *fork()*), and the process performing the *epoll_wait()* has closed its file descriptor, but the other process still holds the duplicate descriptor open.

Listing 63-6: Semantics of *epoll* with duplicate file descriptors

```
int epfd, fd1, fd2;
struct epoll_event ev;
struct epoll_event evlist[MAX_EVENTS];

/* Omitted: code to open 'fd1' and create epoll file descriptor 'epfd' ... */

ev.data.fd = fd1
ev.events = EPOLLIN;
if (epoll_ctl(epfd, EPOLL_CTL_ADD, fd1, ev) == -1)
    errExit("epoll_ctl");

/* Suppose that 'fd1' now happens to become ready for input */

fd2 = dup(fd1);
close(fd1);
ready = epoll_wait(epfd, evlist, MAX_EVENTS, -1);
if (ready == -1)
    errExit("epoll_wait");
```

63.4.5 Performance of *epoll* Versus I/O Multiplexing

Table 63-9 shows the results (on Linux 2.6.25) when we monitor *N* contiguous file descriptors in the range *0* to *N − 1* using *poll()*, *select()*, and *epoll*. (The test was arranged such that during each monitoring operation, exactly one randomly selected file descriptor is ready.) From this table, we see that as the number of file descriptors to be monitored grows large, *poll()* and *select()* perform poorly. By contrast, the performance of *epoll* hardly declines as *N* grows large. (The small decline in performance as *N* increases is possibly a result of reaching CPU caching limits on the test system.)

> For the purposes of this test, `FD_SETSIZE` was changed to 16,384 in the *glibc* header files to allow the test program to monitor large numbers of file descriptors using *select()*.

Table 63-9: Times taken by *poll()*, *select()*, and *epoll* for 100,000 monitoring operations

Number of descriptors monitored (*N*)	*poll()* CPU time (seconds)	*select()* CPU time (seconds)	*epoll* CPU time (seconds)
10	0.61	0.73	0.41
100	2.9	3.0	0.42
1000	35	35	0.53
10000	990	930	0.66

In Section 63.2.5, we saw why *select()* and *poll()* perform poorly when monitoring large numbers of file descriptors. We now look at the reasons why *epoll* performs better:

- On each call to *select()* or *poll()*, the kernel must check all of the file descriptors specified in the call. By contrast, when we mark a descriptor to be monitored with *epoll_ctl()*, the kernel records this fact in a list associated with the underlying open file description, and whenever an I/O operation that makes the file descriptor ready is performed, the kernel adds an item to the ready list for the *epoll* descriptor. (An I/O event on a single open file description may cause multiple file descriptors associated with that description to become ready.) Subsequent *epoll_wait()* calls simply fetch items from the ready list.
- Each time we call *select()* or *poll()*, we pass a data structure to the kernel that identifies all of the file descriptors that are to be monitored, and, on return, the kernel passes back a data structure describing the readiness of all of these descriptors. By contrast, with *epoll*, we use *epoll_ctl()* to build up a data structure *in kernel space* that lists the set of file descriptors to be monitored. Once this data structure has been built, each later call to *epoll_wait()* doesn't need to pass any information about file descriptors to the kernel, and the call returns information about only those descriptors that are ready.

  > In addition to the above points, for *select()*, we must initialize the input data structure prior to each call, and for both *select()* and *poll()*, we must inspect the returned data structure to find out which of the *N* file descriptors are ready. However, some testing showed that the time required for these other steps was

insignificant compared to the time required for the system call to monitor *N* descriptors. Table 63-9 doesn't include the times for the inspection step.

Very roughly, we can say that for large values of *N* (the number of file descriptors being monitored), the performance of *select()* and *poll()* scales linearly with *N*. We start to see this behavior for the *N = 100* and *N = 1000* cases in Table 63-9. By the time we reach *N = 10000*, the scaling has actually become worse than linear.

By contrast, *epoll* scales (linearly) according to the number of I/O events that occur. The *epoll* API is thus particularly efficient in a scenario that is common in servers that handle many simultaneous clients: of the many file descriptors being monitored, most are idle; only a few descriptors are ready.

63.4.6 Edge-Triggered Notification

By default, the *epoll* mechanism provides *level-triggered* notification. By this, we mean that *epoll* tells us whether an I/O operation can be performed on a file descriptor without blocking. This is the same type of notification as is provided by *poll()* and *select()*.

The *epoll* API also allows for *edge-triggered* notification—that is, a call to *epoll_wait()* tells us if there has been I/O activity on a file descriptor since the previous call to *epoll_wait()* (or since the descriptor was opened, if there was no previous call). Using *epoll* with edge-triggered notification is semantically similar to signal-driven I/O, except that if multiple I/O events occur, *epoll* coalesces them into a single notification returned via *epoll_wait()*; with signal-driven I/O, multiple signals may be generated.

To employ edge-triggered notification, we specify the `EPOLLET` flag in *ev.events* when calling *epoll_ctl()*:

```
struct epoll_event ev;

ev.data.fd = fd
ev.events = EPOLLIN | EPOLLET;
if (epoll_ctl(epfd, EPOLL_CTL_ADD, fd, ev) == -1)
    errExit("epoll_ctl");
```

We illustrate the difference between level-triggered and edge-triggered *epoll* notification using an example. Suppose that we are using *epoll* to monitor a socket for input (`EPOLLIN`), and the following steps occur:

1. Input arrives on the socket.
2. We perform an *epoll_wait()*. This call will tell us that the socket is ready, regardless of whether we are employing level-triggered or edge-triggered notification.
3. We perform a second call to *epoll_wait()*.

If we are employing level-triggered notification, then the second *epoll_wait()* call will inform us that the socket is ready. If we are employing edge-triggered notification, then the second call to *epoll_wait()* will block, because no new input has arrived since the previous call to *epoll_wait()*.

As we noted in Section 63.1.1, edge-triggered notification is usually employed in conjunction with nonblocking file descriptors. Thus, the general framework for using edge-triggered *epoll* notification is as follows:

1. Make all file descriptors that are to be monitored nonblocking.
2. Build the *epoll* interest list using *epoll_ctl()*.
3. Handle I/O events using the following loop:
   a) Retrieve a list of ready descriptors using *epoll_wait()*.
   b) For each file descriptor that is ready, process I/O until the relevant system call (e.g., *read()*, *write()*, *recv()*, *send()*, or *accept()*) returns with the error `EAGAIN` or `EWOULDBLOCK`.

Preventing file-descriptor starvation when using edge-triggered notification

Suppose that we are monitoring multiple file descriptors using edge-triggered notification, and that a ready file descriptor has a large amount (perhaps an endless stream) of input available. If, after detecting that this file descriptor is ready, we attempt to consume all of the input using nonblocking reads, then we risk starving the other file descriptors of attention (i.e., it may be a long time before we again check them for readiness and perform I/O on them). One solution to this problem is for the application to maintain a list of file descriptors that have been notified as being ready, and execute a loop that continuously performs the following actions:

1. Monitor the file descriptors using *epoll_wait()* and add ready descriptors to the application list. If any file descriptors are already registered as being ready in the application list, then the timeout for this monitoring step should be small or 0, so that if no new file descriptors are ready, the application can quickly proceed to the next step and service any file descriptors that are already known to be ready.
2. Perform a limited amount of I/O on those file descriptors registered as being ready in the application list (perhaps cycling through them in round-robin fashion, rather than always starting from the beginning of the list after each call to *epoll_wait()*). A file descriptor can be removed from the application list when the relevant nonblocking I/O system call fails with the `EAGAIN` or `EWOULDBLOCK` error.

Although it requires extra programming work, this approach offers other benefits in addition to preventing file-descriptor starvation. For example, we can include other steps in the above loop, such as handling timers and accepting signals with *sigwaitinfo()* (or similar).

Starvation considerations can also apply when using signal-driven I/O, since it also presents an edge-triggered notification mechanism. By contrast, starvation considerations don't necessarily apply in applications employing a level-triggered notification mechanism. This is because we can employ blocking file descriptors with level-triggered notification and use a loop that continuously checks descriptors for readiness, and then performs *some* I/O on the ready descriptors before once more checking for ready file descriptors.

63.5 Waiting on Signals and File Descriptors

Sometimes, a process needs to simultaneously wait for I/O to become possible on one of a set of file descriptors or for the delivery of a signal. We might attempt to perform such an operation using *select()*, as shown in Listing 63-7.

Listing 63-7: Incorrect method of unblocking signals and calling *select()*

```
sig_atomic_t gotSig = 0;

void
handler(int sig)
{
    gotSig = 1;
}

int
main(int argc, char *argv[])
{
    struct sigaction sa;
    ...

    sa.sa_sigaction = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGUSR1, &sa, NULL) == -1)
        errExit("sigaction");

    /* What if the signal is delivered now? */

    ready = select(nfds, &readfds, NULL, NULL, NULL);
    if (ready > 0) {
        printf("%d file descriptors ready\n", ready);
    } else if (ready == -1 && errno == EINTR) {
        if (gotSig)
            printf("Got signal\n");
    } else {
        /* Some other error */
    }

    ...
}
```

The problem with this code is that if the signal (`SIGUSR1` in this example) arrives after establishing the handler but before *select()* is called, then the *select()* call will nevertheless block. (This is a form of race condition.) We now look at some solutions to this problem.

> Since version 2.6.27, Linux provides a further technique that can be used to simultaneously wait on signals and file descriptors: the *signalfd* mechanism described in Section 22.11. Using this mechanism, we can receive signals via a

file descriptor that is monitored (along with other file descriptors) using *select()*, *poll()*, or *epoll_wait()*.

63.5.1 The *pselect()* System Call

The *pselect()* system call performs a similar task to *select()*. The main semantic difference is an additional argument, *sigmask*, that specifies a set of signals to be unmasked while the call is blocked.

```
#define _XOPEN_SOURCE 600
#include <sys/select.h>

int pselect(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
            struct timespec *timeout, const sigset_t *sigmask);
```

Returns number of ready file descriptors, 0 on timeout, or −1 on error

More precisely, suppose we have the following *pselect()* call:

```
ready = pselect(nfds, &readfds, &writefds, &exceptfds, timeout, &sigmask);
```

This call is equivalent to *atomically* performing the following steps:

```
sigset_t origmask;

sigprocmask(SIG_SETMASK, &sigmask, &origmask);
ready = select(nfds, &readfds, &writefds, &exceptfds, timeout);
sigprocmask(SIG_SETMASK, &origmask, NULL);        /* Restore signal mask */
```

Using *pselect()*, we can recode the first part of the body of our main program in Listing 63-7 as shown in Listing 63-8.

Aside from the *sigmask* argument, *select()* and *pselect()* differ in the following ways:

- The *timeout* argument to *pselect()* is a *timespec* structure (Section 23.4.2), which allows the timeout to be specified with nanosecond (instead of microsecond) precision.
- SUSv3 explicitly states that *pselect()* doesn't modify the *timeout* argument on return.

If we specify the *sigmask* argument of *pselect()* as `NULL`, then *pselect()* is equivalent to *select()* (i.e., it performs no manipulation of the process signal mask), except for the differences just noted.

The *pselect()* interface is an invention of POSIX.1g, and is nowadays incorporated in SUSv3. It is not available on all UNIX implementations, and was added to Linux only in kernel 2.6.16.

> Previously, a *pselect()* library function was provided by *glibc*, but this implementation didn't provide the atomicity guarantees that are required for the correct operation of the call. Such guarantees can be provided only by a kernel implementation of *pselect()*.

Listing 63-8: Using *pselect()*

```
sigset_t emptyset, blockset;
struct sigaction sa;

sigemptyset(&blockset);
sigaddset(&blockset, SIGUSR1);

if (sigprocmask(SIG_BLOCK, &blockset, NULL) == -1)
    errExit("sigprocmask");

sa.sa_sigaction = handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;
if (sigaction(SIGUSR1, &sa, NULL) == -1)
    errExit("sigaction");

sigemptyset(&emptyset);
ready = pselect(nfds, &readfds, NULL, NULL, NULL, &emptyset);
if (ready == -1)
    errExit("pselect");
```

The *ppoll()* and *epoll_pwait()* system calls

Linux 2.6.16 also added a new, nonstandard system call, *ppoll()*, whose relationship to *poll()* is analogous to the relationship of *pselect()* to *select()*. Similarly, starting with kernel 2.6.19, Linux also includes *epoll_pwait()*, providing an analogous extension to *epoll_wait()*. See the *ppoll(2)* and *epoll_pwait(2)* manual pages for details.

63.5.2 The Self-Pipe Trick

Since *pselect()* is not widely implemented, portable applications must employ other strategies to avoid race conditions when simultaneously waiting for signals and calling *select()* on a set of file descriptors. One common solution is the following:

1. Create a pipe, and mark its read and write ends as nonblocking.
2. As well as monitoring all of the other file descriptors that are of interest, include the read end of the pipe in the *readfds* set given to *select()*.
3. Install a handler for the signal that is of interest. When this signal handler is called, it writes a byte of data to the pipe. Note the following points about the signal handler:
   - The write end of the pipe was marked as nonblocking in the first step to prevent the possibility that signals arrive so rapidly that repeated invocations of the signal handler fill the pipe, with the result that the signal handler's *write()* (and thus the process itself) is blocked. (It doesn't matter if a write to a full pipe fails, since the previous writes will already have indicated the delivery of the signal.)

- The signal handler is installed after creating the pipe, in order to prevent the race condition that would occur if a signal was delivered before the pipe was created.
- It is safe to use *write()* inside the signal handler, because it is one of the async-signal-safe functions listed in Table 21-1, on page 426.

4. Place the *select()* call in a loop, so that it is restarted if interrupted by a signal handler. (Restarting in this fashion is not strictly necessary; it merely means that we can check for the arrival of a signal by inspecting *readfds*, rather than checking for an `EINTR` error return.)
5. On successful completion of the *select()* call, we can determine whether a signal arrived by checking if the file descriptor for the read end of the pipe is set in *readfds*.
6. Whenever a signal has arrived, read all bytes that are in the pipe. Since multiple signals may arrive, employ a loop that reads bytes until the (nonblocking) *read()* fails with the error `EAGAIN`. After draining the pipe, perform whatever actions must be taken in response to delivery of the signal.

This technique is commonly known as the *self-pipe trick*, and code demonstrating this technique is shown in Listing 63-9.

Variations on this technique can equally be employed with *poll()* and *epoll_wait()*.

Listing 63-9: Using the self-pipe trick

from **altio/self_pipe.c**

```
static int pfd[2];                      /* File descriptors for pipe */

static void
handler(int sig)
{
    int savedErrno;                     /* In case we change 'errno' */

    savedErrno = errno;
    if (write(pfd[1], "x", 1) == -1 && errno != EAGAIN)
        errExit("write");
    errno = savedErrno;
}

int
main(int argc, char *argv[])
{
    fd_set readfds;
    int ready, nfds, flags;
    struct timeval timeout;
    struct timeval *pto;
    struct sigaction sa;
    char ch;
```

```
    /* ... Initialize 'timeout', 'readfds', and 'nfds' for select() */

    if (pipe(pfd) == -1)
        errExit("pipe");

    FD_SET(pfd[0], &readfds);           /* Add read end of pipe to 'readfds' */
    nfds = max(nfds, pfd[0] + 1);       /* And adjust 'nfds' if required */

    flags = fcntl(pfd[0], F_GETFL);
    if (flags == -1)
        errExit("fcntl-F_GETFL");
    flags |= O_NONBLOCK;                /* Make read end nonblocking */
    if (fcntl(pfd[0], F_SETFL, flags) == -1)
        errExit("fcntl-F_SETFL");

    flags = fcntl(pfd[1], F_GETFL);
    if (flags == -1)
        errExit("fcntl-F_GETFL");
    flags |= O_NONBLOCK;                /* Make write end nonblocking */
    if (fcntl(pfd[1], F_SETFL, flags) == -1)
        errExit("fcntl-F_SETFL");

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;           /* Restart interrupted read()s */
    sa.sa_handler = handler;
    if (sigaction(SIGINT, &sa, NULL) == -1)
        errExit("sigaction");

    while ((ready = select(nfds, &readfds, NULL, NULL, pto)) == -1 &&
            errno == EINTR)
        continue;                       /* Restart if interrupted by signal */
    if (ready == -1)                    /* Unexpected error */
        errExit("select");

    if (FD_ISSET(pfd[0], &readfds)) {   /* Handler was called */
        printf("A signal was caught\n");

        for (;;) {                      /* Consume bytes from pipe */
            if (read(pfd[0], &ch, 1) == -1) {
                if (errno == EAGAIN)
                    break;              /* No more bytes */
                else
                    errExit("read");    /* Some other error */
            }

            /* Perform any actions that should be taken in response to signal */
        }
    }

    /* Examine file descriptor sets returned by select() to see
       which other file descriptors are ready */

}
```

from **altio/self_pipe.c**

63.6 Summary

In this chapter, we explored various alternatives to the standard model for performing I/O: I/O multiplexing (*select()* and *poll()*), signal-driven I/O, and the Linux-specific *epoll* API. All of these mechanisms allow us to monitor multiple file descriptors to see if I/O is possible on any of them. None of these mechanisms actually performs I/O. Instead, once we have determined that a file descriptor is ready, we use the traditional I/O system calls to perform the I/O.

The *select()* and *poll()* I/O multiplexing calls simultaneously monitor multiple file descriptors to see if I/O is possible on any of the descriptors. With both system calls, we pass a complete list of to-be-checked file descriptors to the kernel on each system call, and the kernel returns a modified list indicating which descriptors are ready. The fact that complete file descriptor lists are passed and checked on each call means that *select()* and *poll()* perform poorly when monitoring large numbers of file descriptors.

Signal-driven I/O allows a process to receive a signal when I/O is possible on a file descriptor. To enable signal-driven I/O, we must establish a handler for the `SIGIO` signal, set the owner process that is to receive the signal, and enable signal generation by setting the `O_ASYNC` open file status flag. This mechanism offers significant performance benefits over I/O multiplexing when monitoring large numbers of file descriptors. Linux allows us to change the signal used for notification, and if we instead employ a realtime signal, then multiple notifications can be queued, and the signal handler can use its *siginfo_t* argument to determine the file descriptor and event type that generated the signal.

Like signal-driven I/O, *epoll* offers superior performance when monitoring large numbers of file descriptors. The performance advantage of *epoll* (and signal-driven I/O) derives from the fact that the kernel "remembers" the list of file descriptors that a process is monitoring (by contrast with *select()* and *poll()*, where each system call must again tell the kernel which file descriptors to check). The *epoll* API has some notable advantages over the use of signal-driven I/O: we avoid the complexities of dealing with signals and can specify which types of I/O events (e.g., input or output) are to be monitored.

In the course of this chapter, we drew a distinction between level-triggered and edge-triggered readiness notification. With a level-triggered notification model, we are informed whether I/O is currently possible on a file descriptor. By contrast, edge-triggered notification informs us whether I/O activity has occurred on a file descriptor since it was last monitored. The I/O multiplexing system calls offer a level-triggered notification model; signal-driven I/O approximates to an edge-triggered model; and *epoll* is capable of operating under either model (level-triggered is the default). Edge-triggered notification is usually employed in conjunction with nonblocking I/O.

We concluded the chapter by looking at a problem that sometimes faces programs that monitor multiple file descriptors: how to simultaneously also wait for the delivery of a signal. The usual solution to this problem is the so-called self-pipe trick, whereby a handler for the signal writes a byte to a pipe whose read end is included among the set of monitored file descriptors. SUSv3 specifies *pselect()*, a variation of *select()* that provides another solution to this problem. However, *pselect()* is not available on all UNIX implementations. Linux also provides the analogous (but nonstandard) *ppoll()* and *epoll_pwait()*.

Further information

[Stevens et al., 2004] describes I/O multiplexing and signal-driven I/O, with particular emphasis on the use of these mechanisms with sockets. [Gammo et al, 2004] is a paper comparing the performance of *select()*, *poll()*, and *epoll*.

A particularly interesting online resource is at *http://www.kegel.com/c10k.html*. Written by Dan Kegel, and entitled "The C10K problem," this web page explores the issues facing developers of web servers designed to simultaneously serve tens of thousands of clients. The web page includes a host of links to related information.

63.7 Exercises

63-1. Modify the program in Listing 63-2 (`poll_pipes.c`) to use *select()* instead of *poll()*.

63-2. Write an *echo* server (see Sections 60.2 and 60.3) that handles both TCP and UDP clients. To do this, the server must create both a listening TCP socket and a UDP socket, and then monitor both sockets using one of the techniques described in this chapter.

63-3. Section 63.5 noted that *select()* can't be used to wait on both signals and file descriptors, and described a solution using a signal handler and a pipe. A related problem exists when a program needs to wait for input on both a file descriptor and a System V message queue (since System V message queues don't use file descriptors). One solution is to fork a separate child process that copies each message from the queue to a pipe included among the file descriptors monitored by the parent. Write a program that uses this scheme with *select()* to monitor input from both the terminal and a message queue.

63-4. The last step of the description of the self-pipe technique in Section 63.5.2 stated that the program should first drain the pipe, and then perform any actions that should be taken in response to the signal. What might happen if these substeps were reversed?

63-5. Modify the program in Listing 63-9 (`self_pipe.c`) to use *poll()* instead of *select()*.

63-6. Write a program that uses *epoll_create()* to create an *epoll* instance and then immediately waits on the returned file descriptor using *epoll_wait()*. When, as in this case, *epoll_wait()* is given an *epoll* file descriptor with an empty interest list, what happens? Why might this be useful?

63-7. Suppose we have an *epoll* file descriptor that is monitoring multiple file descriptors, all of which are always ready. If we perform a series of *epoll_wait()* calls in which *maxevents* is much smaller than the number of ready file descriptors (e.g., *maxevents* is 1), without performing all possible I/O on the ready descriptors between calls, what descriptor(s) does *epoll_wait()* return in each call? Write a program to determine the answer. (For the purposes of this experiment, it suffices to perform no I/O between the *epoll_wait()* system calls.) Why might this behavior be useful?

63-8. Modify the program in Listing 63-3 (`demo_sigio.c`) to use a realtime signal instead of `SIGIO`. Modify the signal handler to accept a *siginfo_t* argument and display the values of the *si_fd* and *si_code* fields of this structure.