

12

Stevens

Advanced Programming in the
UNIX Environment

Addison Wesley 1993

Advanced I/O

12.1 Introduction

This chapter covers numerous topics and functions that we lump under the term "advanced I/O." This includes nonblocking I/O, record locking, System V streams, I/O multiplexing (the `select` and `poll` functions), the `readv` and `writew` functions, and memory mapped I/O (`mmap`). We need to cover these topics before describing interprocess communication in Chapters 14 and 15, and many of the examples in later chapters.

12.2 Nonblocking I/O

In Section 10.5 we said that the system calls are divided into two categories: the "slow" ones, and all the others. The slow system calls are those that can block forever:

- reads from files that can block the caller forever, if data isn't present (pipes, terminal devices, and network devices),
- writes to these same files that can block forever, if the data can't be accepted immediately,
- opens of files block until some condition occurs (such as an open of a terminal device that waits until an attached modem answers the phone, or an open of a FIFO for writing-only when no other process has the FIFO open for reading),
- reads and writes of files that have mandatory record locking enabled,
- certain `ioctl` operations,
- some of the interprocess communication functions (Chapter 14).

We also said that system calls related to disk I/O are not considered slow, even though the read or write of a disk file can block the caller temporarily.

Nonblocking I/O lets us issue an I/O operation, such as an open, read, or write, and not have it block forever. If the operation cannot be completed, return is made immediately with an error noting that the operation would have blocked.

There are two ways to specify nonblocking I/O for a given descriptor.

1. If we call open to get the descriptor, we can specify the `O_NONBLOCK` flag (Section 3.3).
2. For a descriptor that is already open, we call `fcntl` to turn on the `O_NONBLOCK` file status flag (Section 3.13). Program 3.5 shows a function that we can call to turn on any of the file status flags for a descriptor.

Earlier versions of System V used the flag `O_NDELAY` to specify the nonblocking mode. These versions of System V returned a value of 0 from the read function if there wasn't any data to be read. Since this use of a return value of 0 overlapped with the normal Unix convention of 0 meaning the end of file, POSIX.1 chose to provide a nonblocking flag with a different name and different semantics. Indeed, with these older versions of System V we don't know when we get a return of 0 from read whether the call would have blocked, or if the end of file was encountered. We'll see that POSIX.1 requires that read return -1 with `errno` set to `EAGAIN` if there is no data to read from a nonblocking descriptor. SVR4 supports both the older `O_NDELAY` and the POSIX.1 `O_NONBLOCK`, but in this text we'll only use the POSIX.1 feature. The older `O_NDELAY` is for backward compatibility and should not be used in new applications.

4.3BSD provided the `FNDELAY` flag for `fcntl`, and its semantics were slightly different. Instead of just affecting the file status flags for the descriptor, the flags for either the terminal device or the socket were also changed to be nonblocking, affecting all users of the terminal or socket, not just the users sharing the same file table entry (4.3BSD nonblocking I/O only worked on terminals and sockets). Also, 4.3BSD returned `EWOULDBLOCK` if an operation on a nonblocking descriptor could not complete without blocking. 4.3+BSD provides the POSIX.1 `O_NONBLOCK` flag, but the semantics are similar to those for `FNDELAY` under 4.3BSD. A common use for nonblocking I/O is for dealing with a terminal device or a network connection, and these devices are normally used by one process at a time. This means that the change in the BSD semantics normally doesn't affect us. The different error return, `EWOULDBLOCK`, instead of the POSIX.1 `EAGAIN`, continues to be a portability difference that we must deal with. 4.3+BSD also supports FIFOs, and nonblocking I/O works with FIFOs too.

Example

Let's look at an example of nonblocking I/O. Program 12.1 reads up to 100,000 bytes from the standard input and attempts to write it to the standard output. The standard output is first set nonblocking. The output is in a loop, with the results of each write being printed on the standard error. The function `clr_fl` is similar to the function `set_fl` that we showed in Program 3.5. This new function just clears one or more of the flag bits.

```
#in
#in
#in
#in
cha.
int
main
{
```

```
}
```

If the

```
!
-
:
}
r
e
-
```

But i
some

ven though

l, or write,
rn is made

BLOCK flag

_NONBLOCK
e can call to

g mode. These
sn't any data to
convention of 0
different name
n't know when
end of file was
set to EAGAIN if
both the older
OSIX.1 feature.
in new applica-

ightly different.
her the terminal
f the terminal or
cking I/O only
n operation on a
des the POSIX.1
4.3BSD. A com-
work connection,
at the change in
. EWOULDBLOCK,
e must deal with.

100,000 bytes
The standard
of each write
the function
ne or more of

```
#include <sys/types.h>
#include <errno.h>
#include <fcntl.h>
#include "ourhdr.h"

char buf[100000];

int
main(void)
{
    int ntwrite, nwrite;
    char *ptr;

    ntwrite = read(STDIN_FILENO, buf, sizeof(buf));
    fprintf(stderr, "read %d bytes\n", ntwrite);

    set_fl(STDOUT_FILENO, O_NONBLOCK); /* set nonblocking */

    for (ptr = buf; ntwrite > 0; ) {
        errno = 0;
        nwrite = write(STDOUT_FILENO, ptr, ntwrite);
        fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);
        if (nwrite > 0) {
            ptr += nwrite;
            ntwrite -= nwrite;
        }
    }

    clr_fl(STDOUT_FILENO, O_NONBLOCK); /* clear nonblocking */
    exit(0);
}
```

Program 12.1 Large nonblocking write.

If the standard output is a regular file, we expect the write to be executed once.

```
$ ls -l /etc/termcap                print file size
-rw-rw-r-- 1 root 133439 Oct 11 1990 /etc/termcap
$ a.out < /etc/termcap > temp.file  try a regular file first
read 100000 bytes
nwrite = 100000, errno = 0          a single write
$ ls -l temp.file                   verify size of output file
-rw-rw-r-- 1 stevens 100000 Nov 21 16:27 temp.file
```

But if the standard output is a terminal, we expect the write to return a partial count sometimes and an error at other times. This is what we see.

```

$ a.out < /etc/termcap 2>stderr.out          output to terminal
                                              lots of output to terminal ...

$ cat stderr.out
read 100000 bytes
nwrite = 8192, errno = 0
nwrite = 8192, errno = 0
nwrite = -1, errno = 11                      211 of these errors
. . .
nwrite = 4096, errno = 0
nwrite = -1, errno = 11                      658 of these errors
. . .
nwrite = 4096, errno = 0
nwrite = -1, errno = 11                      604 of these errors
. . .
nwrite = 4096, errno = 0
nwrite = -1, errno = 11                      1047 of these errors
. . .
nwrite = -1, errno = 11                      1046 of these errors
. . .
nwrite = 4096, errno = 0
                                              and so on ...

```

On this system the `errno` of 11 is `EAGAIN`. The terminal driver on this system always accepted 4096 or 8192 bytes at a time. On another system the first three writes returned 2005, 1822, and 1811, followed by 96 errors, followed by a write of 1846, and so on. How much data is accepted on each write is system dependent.

The behavior of this program under SVR4 is completely different from the preceding—when the output was to the terminal only a single write was needed to output the entire input file. Apparently the nonblocking mode makes no difference! A bigger input file was created and the program's buffer was increased. This behavior of the program (one write for the entire file) continued until the size of the input file was about 700,000 bytes. At that point every write returned the error `EAGAIN`. (The input file was never output to the terminal—the program just generated a continual stream of error messages.)

What's going on here is that the terminal driver in SVR4 is connected to the program through the stream I/O system. (We describe streams in detail in Section 12.4.) The streams system has its own buffers and is capable of accepting more data at a time from the program. The SVR4 behavior also depends on the type of terminal—hard-wired terminal, console device, or a pseudo terminal. □

In this example the program issues thousands of write calls, when only around 20 are required to output the data. The rest just return an error. This type of loop, called *polling*, is a waste of CPU time on a multiuser system. In Section 12.5 we'll see that I/O multiplexing with a nonblocking descriptor is a more efficient way to do this.

We'll encounter nonblocking I/O in Chapter 17 when we output to a terminal device (a PostScript printer) and want to make certain we don't block on a write.

12.3 Rec

Wha
tems
are a
tain
new
Cha
l
vent
or m
since
locki

History

Figu
tems

We c
this
ing,
versi
I
locke
any
I
func

```

#include <stropts.h>
#include "ourhdr.h"

#define BUFFSIZE 8192

int
main(void)
{
    int n, flag;
    char ctlbuf[BUFFSIZE], datbuf[BUFFSIZE];
    struct strbuf ctl, dat;

    ctl.buf = ctlbuf;
    ctl.maxlen = BUFFSIZE;
    dat.buf = datbuf;
    dat.maxlen = BUFFSIZE;
    for ( ; ; ) {
        flag = 0; /* return any message */
        if ( (n = getmsg(STDIN_FILENO, &ctl, &dat, &flag)) < 0)
            err_sys("getmsg error");
        fprintf(stderr, "flag = %d, ctl.len = %d, dat.len = %d\n",
                flag, ctl.len, dat.len);
        if (dat.len == 0)
            exit(0);
        else if (dat.len > 0)
            if (write(STDOUT_FILENO, dat.buf, dat.len) != dat.len)
                err_sys("write error");
    }
}

```

Program 12.11 Copy standard input to standard output using getmsg.

12.5 I/O Multiplexing

When we read from one descriptor and write to another, we can use blocking I/O in a loop such as

```

while ( (n = read(STDIN_FILENO, buf, BUFSIZ)) > 0)
    if (write(STDOUT_FILENO, buf, n) != n)
        err_sys("write error");

```

We see this form of blocking I/O over and over again. What if we have to read from two descriptors? In this case we can't do a blocking read on either descriptor, as data may appear on one descriptor while we're blocked in a read on the other. A different technique is required to handle this case.

Let's skip ahead and look at the modem dialer in Chapter 18. In this program we read from the terminal (standard input) and write to the modem, and we read from the modem and write to the terminal (standard output). Figure 12.11 shows a picture of this.

The p
the in
C
(using
Figur

If we
proble
mode:
paren
an enc
signal
W
tors n
and p:
same
few se
is call
won't
have t
any sy
tem.

As
us wit
First, r
future,
only if
but it l
or net

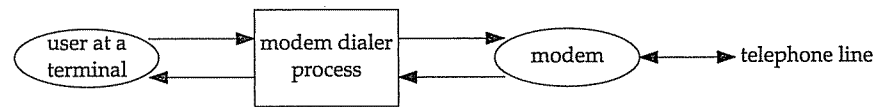


Figure 12.11 Overview of modem dialer program.

The process has two inputs and two outputs. We can't do a blocking read on either of the inputs, as we never know which input will have data for us.

One way to handle this particular problem is to divide the process in two pieces (using `fork`) with each half handling one direction of data. We show this in Figure 12.12.

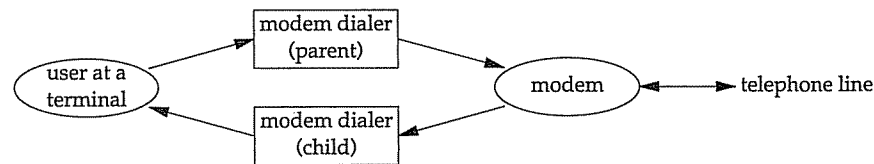


Figure 12.12 Modem dialer using two processes.

If we use two processes we can let each process do a blocking read. But this leads to a problem when the operation terminates. If an end of file is received by the child (the modem is hung up by the other end of the phone line) then the child terminates and the parent is notified by the `SIGCHLD` signal. But if the parent terminates (the user enters an end of file at the terminal) then the parent has to tell the child to stop. We can use a signal for this (`SIGUSR1`, for example) but it does complicate the program somewhat.

We could use nonblocking I/O in a single process. To do this we set both descriptors nonblocking, and issue a `read` on the first descriptor. If data is present, we read it and process it. If there is no data to read, the call returns immediately. We then do the same thing with the second descriptor. After this we wait for some amount of time (a few seconds perhaps), then try to read from the first descriptor again. This type of loop is called *polling*. The problem is that it is a waste of CPU time. Most of the time there won't be data to read, so we waste the time performing the read system calls. We also have to guess how long to wait each time around the loop. Although polling works on any system that supports nonblocking I/O, it should be avoided on a multitasking system.

Another technique is called *asynchronous I/O*. To do this we tell the kernel to notify us with a signal when a descriptor is ready for I/O. There are two problems with this. First, not all systems support this feature (it is not yet part of POSIX, but may be in the future). SVR4 provides the `SIGPOLL` signal for this technique, but this signal works only if the descriptor refers to a streams device. 4.3+BSD has a similar signal, `SIGIO`, but it has similar limitations—it works only on descriptors that refer to terminal devices or networks. The second problem with this technique is that there is only one of these

signals per process (SIGPOLL or SIGIO). If we enable this signal for two descriptors (in the example we've been talking about, reading from two descriptors) the occurrence of the signal doesn't tell us which descriptor is ready. To determine which descriptor is ready, we still need to set each nonblocking and try them in sequence. We describe asynchronous I/O briefly in Section 12.6.

A better technique is to use *I/O multiplexing*. To do this we build a list of the descriptors that we are interested in (usually more than one descriptor) and call a function that doesn't return until one of the descriptors is ready for I/O. On return from the function we are told which descriptors are ready for I/O.

I/O multiplexing is not yet part of POSIX. The `select` function is provided by both SVR4 and 4.3+BSD to do I/O multiplexing. The `poll` function is provided only by SVR4. SVR4 actually implements `select` using `poll`.

I/O multiplexing was provided with the `select` function in 4.2BSD. This function has always worked with any descriptor, although its main use has been for terminal I/O and network I/O. SVR3 added the `poll` function when streams were added. Until SVR4, however, `poll` only worked with streams devices. SVR4 supports `poll` on any descriptor.

Interruptibility of `select` and `poll`

When the automatic restarting of interrupted system calls was introduced with 4.2BSD (Section 10.5), the `select` function was never restarted. This characteristic continues with 4.3+BSD (and most systems derived from earlier BSD systems) even if the `SA_RESTART` option is specified. But under SVR4, if `SA_RESTART` is specified, even `select` and `poll` are automatically restarted. To prevent this from catching us when we port software to SVR4, we'll always use the `signal_intr` function (Program 10.13) if the signal could interrupt a call to `select` or `poll`.

12.5.1 `select` Function

The `select` function lets us do I/O multiplexing under both SVR4 and 4.3+BSD. The arguments we pass to `select` tell the kernel

1. Which descriptors we're interested in.
2. What conditions we're interested in for each descriptor. (Do we want to read from a given descriptor? Do we want to write to a given descriptor? Are we interested in an exception condition for a given descriptor?)
3. How long we want to wait. (We can wait forever, wait a fixed amount of time, or not wait at all.)

On the return from `select` the kernel tells us

1. The total count of the number of descriptors that are ready.
2. Which descriptors are ready for each of the three conditions (read, write, or exception condition).

With
wri

#

i

Let's

Then

t

t

t

]

sets.

tions

fd_s

one b

in Fig

1

type,

one c

With this return information we can call the appropriate I/O function (usually read or write) and know that the function won't block.

```
#include <sys/types.h> /* fd_set data type */
#include <sys/time.h> /* struct timeval */
#include <unistd.h> /* function prototype might be here */

int select(int maxfdp1, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *tvptr);
```

Returns: count of ready descriptors, 0 on timeout, -1 on error

Let's look at the last argument first. This specifies how long we want to wait.

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

There are three conditions.

`tvptr == NULL`

Wait forever. This infinite wait can be interrupted if we catch a signal. Return is made when one of the specified descriptors is ready or when a signal is caught. If a signal is caught, `select` returns -1 with `errno` set to `EINTR`.

`tvptr->tv_sec == 0 && tvptr->tv_usec == 0`

Don't wait at all. All the specified descriptors are tested and return is made immediately. This is a way to poll the system to find out the status of multiple descriptors, without blocking in the `select` function.

`tvptr->tv_sec != 0 || tvptr->tv_usec != 0`

Wait the specified number of seconds and microseconds. Return is made when one of the specified descriptors is ready or when the time-out value expires. If the timeout expires before any of the descriptors is ready, the return value is 0. (If the system doesn't provide microsecond resolution, the `tvptr->tv_usec` value is rounded up to the nearest supported value.) As with the first condition, this wait can also be interrupted by a caught signal.

The middle three arguments, `readfds`, `writefds`, and `exceptfds`, are pointers to *descriptor sets*. These three sets specify which descriptors we're interested in and for which conditions (readable, writable, or an exception condition). A descriptor set is stored in an `fd_set` data type. This data type is chosen by the implementation so that it can hold one bit for each possible descriptor. We can consider it just a big array of bits, as shown in Figure 12.13.

The only thing we can do with the `fd_set` data type is (a) allocate a variable of this type, (b) assign a variable of this type to another variable of the same type, or (c) use one of the following four macros on a variable of this type:

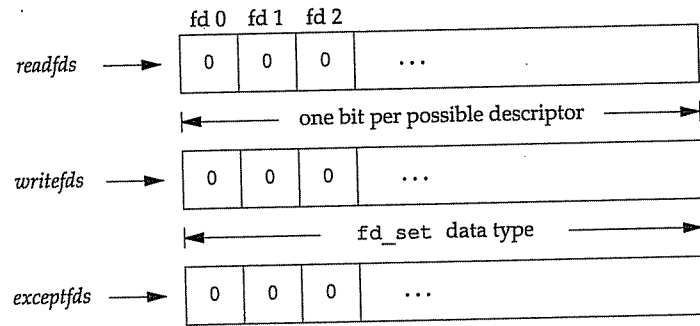


Figure 12.13 Specifying the read, write, and exception descriptors for select.

```

FD_ZERO(fd_set *fdset);          /* clear all bits in fdset */
FD_SET(int fd, fd_set *fdset);   /* turn on bit for fd in fdset */
FD_CLR(int fd, fd_set *fdset);   /* turn off bit for fd in fdset */
FD_ISSET(int fd, fd_set *fdset); /* test bit for fd in fdset */

```

After declaring a descriptor set, as in

```

fd_set  rset;
int     fd;

```

we must zero the set using FD_ZERO.

```

FD_ZERO(&rset);

```

We then set bits in the set for each descriptor that we're interested in:

```

FD_SET(fd, &rset);
FD_SET(STDIN_FILENO, &rset);

```

On return from select we can test whether a given bit in the set is still on using FD_ISSET:

```

if (FD_ISSET(fd, &rset)) {
    ...
}

```

Any (or all) of the middle three arguments to select (the pointers to the descriptor sets) can be null pointers, if we're not interested in that condition. If all three pointers are NULL, then we have a higher precision timer than provided by sleep. (Recall from Section 10.19 that sleep waits for an integral number of seconds. With select we can wait for intervals less than 1 second; the actual resolution depending on the system's clock.) Exercise 12.6 shows such a function.

The first argument to select, maxfdp1, stands for "max fd plus 1." We calculate the highest descriptor that we're interested in, in any of the three descriptor sets, add 1, and that's the first argument. We could just set the first argument to FD_SETSIZE, a constant in <sys/types.h> that specifies the maximum number of descriptors (often

256 c
prob
more
desc
unus
/
1
E
E
E
E
E
s

then

The r
start
(start
T
1
2
3

256 or 1024), but this value is too large for most applications. Indeed, most applications probably use between 3 and 10 descriptors. (There are applications that need many more descriptors, but these aren't the typical Unix program.) By specifying the highest descriptor that we're interested in, the kernel can avoid going through hundreds of unused bits in the three descriptor sets, looking for bits that are turned on.

As an example, if we write

```
fd_set readset, writeset;

FD_ZERO(&readset);
FD_ZERO(&writeset);

FD_SET(0, &readset);
FD_SET(3, &readset);
FD_SET(1, &writeset);
FD_SET(2, &writeset);

select(4, &readset, &writeset, NULL, NULL);
```

then Figure 12.14 shows what the two descriptor sets look like.

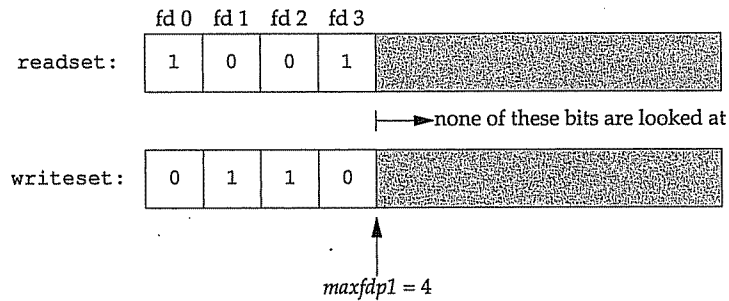


Figure 12.14 Example descriptor sets for select.

The reason we have to add 1 to the maximum descriptor number is because descriptors start at 0, and the first argument is really a count of the number of descriptors to check (starting with descriptor 0).

There are three possible return values from select.

1. A return value of -1 means an error occurred. This can happen, for example, if a signal is caught before any of the specified descriptors are ready.
2. A return value of 0 means no descriptors are ready. This happens if the time limit expires before any of the descriptors are ready.
3. A positive return value specifies the number of descriptors that are ready. In this case the only bits left on in the three descriptor sets are the bits corresponding to the descriptors that are ready.

Be careful not to check the descriptor sets on return unless the return value is greater than 0. The return state of the descriptor sets is implementation dependent if either a signal is caught or the timer expires. Indeed, if the timer expires 4.3+BSD doesn't change the descriptor sets while SVR4 clears the descriptor sets.

anced I/O

t */
set */

still on using

he descriptor
three pointers
(Recall from
select we can
the system's

We calculate
or sets, add 1,
_SETSIZE, a
criptors (often

There is another discrepancy between the SVR4 and BSD implementations of `select`. BSD systems have always returned the sum of the number of ready descriptors in each set. If the same descriptor is ready in two sets (say the read set and the write set), that descriptor is counted twice. SVR4 unfortunately changes this and if the same descriptor is ready in multiple sets, that descriptor is counted only once. This again shows the problems we'll encounter until functions such as `select` are standardized by POSIX.

We now need to be more specific about what "ready" means.

1. A descriptor in the read set (*readfds*) is considered ready if a read from that descriptor won't block.
2. A descriptor in the write set (*writefds*) is considered ready if a write to that descriptor won't block.
3. A descriptor in the exception set (*exceptfds*) is considered ready if there is an exception condition pending on that descriptor. Currently an exception condition corresponds to (a) the arrival of out-of-band data on a network connection, or (b) certain conditions occurring on a pseudo terminal that has been placed into packet mode. (Section 15.10 of Stevens [1990] describes this latter condition.)

It is important to realize that whether a descriptor is blocking or not doesn't affect whether `select` blocks or not. That is, if we have a nonblocking descriptor that we want to read from and we call `select` with a time-out value of 5 seconds, `select` will block for up to 5 seconds. Similarly, if we specify an infinite timeout, `select` blocks until data is ready for the descriptor, or until a signal is caught.

If we encounter the end of file on a descriptor, that descriptor is considered readable by `select`. We then call `read` and it returns 0, the normal Unix way to signify end of file. (Many people incorrectly assume `select` indicates an exception condition on a descriptor when the end of file is reached.)

12.5.2 poll Function

The SVR4 `poll` function is similar to `select`, but the programmer interface is different. As we'll see, `poll` is tied to the streams system, although in SVR4 we are able to use it with any descriptor.

```
#include <stropts.h>
#include <poll.h>

int poll(struct pollfd fdarray[], unsigned long nfd, int timeout);
```

Returns: count of ready descriptors, 0 on timeout, -1 on error

Instead of building a set of descriptors for each condition (readability, writability, and exception condition), as we did with `select`, with `poll` we build an array of `pollfd` structures, with each array element specifying a descriptor number and the conditions that we're interested in for that descriptor.

The

ues i
that
even
mem
read

POL
POL
POL
POL
POL
POL
POL
POL
POL

The f
and t
I
ues a
in the
v
There