NAME

select, pselect, FD_CLR, FD_ISSET, FD_SET, FD_ZERO − synchronous I/O multiplexing

SYNOPSIS

```
/* According to POSIX.1-2001, POSIX.1-2008 */
#include <sys/select.h>

/* According to earlier standards */
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

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

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

#include <sys/select.h>

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

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

pselect(): _POSIX_C_SOURCE >= 200112L

DESCRIPTION

select() (or **pselect**()) is used to efficiently monitor multiple file descriptors, to see if any of them is, or becomes, "ready"; that is, to see whether I/O becomes possible, or an "exceptional condition" has occurred on any of the file descriptors.

Its principal arguments are three "sets" of file descriptors: *readfds*, *writefds*, and *exceptfds*. Each set is declared as type *fd_set*, and its contents can be manipulated with the macros **FD_CLR**(), **FD_ISSET**(), **FD_SET**(), and **FD_ZERO**(). A newly declared set should first be cleared using **FD_ZERO**(). **select**() modifies the contents of the sets according to the rules described below; after calling **select**() you can test if a file descriptor is still present in a set with the **FD_ISSET**() macro. **FD_ISSET**() returns nonzero if a specified file descriptor is present in a set and zero if it is not. **FD_CLR**() removes a file descriptor from a set.

Arguments

readfds This set is watched to see if data is available for reading from any of its file descriptors. After **select**() has returned, *readfds* will be cleared of all file descriptors except for those that are immediately available for reading.

writefds

This set is watched to see if there is space to write data to any of its file descriptors. After **select**() has returned, *writefds* will be cleared of all file descriptors except for those that are immediately available for writing.

exceptfds

This set is watched for "exceptional conditions". In practice, only one such exceptional condition is common: the availability of *out-of-band* (OOB) data for reading from a TCP socket. See recv(2), send(2), and tcp(7) for more details about OOB data. (One other less common case where select(2) indicates an exceptional condition occurs with pseudoterminals in packet mode; see ioctl_tty(2).) After **select**() has returned, *exceptfds* will be cleared of all file descriptors except for those for which an exceptional condition has occurred.

nfds This is an integer one more than the maximum of any file descriptor in any of the sets. In other words, while adding file descriptors to each of the sets, you must calculate the maximum integer value of all of them, then increment this value by one, and then pass this as *nfds*.

utimeout
This is the longest time **select**() may wait before returning, even if nothing interesting happened. If this value is passed as NULL, then **select**() blocks indefinitely waiting for a file descriptor to become ready. *utimeout* can be set to zero seconds, which causes **select**() to return immediately, with information about the readiness of file descriptors at the time of the call. The structure *struct timeval* is defined as:

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

ntimeout
This argument for **pselect**() has the same meaning as *utimeout*, but *struct timespec* has nanosecond precision as follows:

```
struct timespec {
long tv_sec;    /* seconds */
long tv_nsec;   /* nanoseconds */
};
```

sigmask
This argument holds a set of signals that the kernel should unblock (i.e., remove from the signal mask of the calling thread), while the caller is blocked inside the **pselect**() call (see sigaddset(3) and sigprocmask(2)). It may be NULL, in which case the call does not modify the signal mask on entry and exit to the function. In this case, **pselect**() will then behave just like **select**().

Combining signal and data events

pselect() is useful if you are waiting for a signal as well as for file descriptor(s) to become ready for I/O. Programs that receive signals normally use the signal handler only to raise a global flag. The global flag will indicate that the event must be processed in the main loop of the program. A signal will cause the **select**() (or **pselect**()) call to return with *errno* set to **EINTR**. This behavior is essential so that signals can be processed in the main loop of the program, otherwise **select**() would block indefinitely. Now, somewhere in the main loop will be a conditional to check the global flag. So we must ask: what if a signal arrives after the conditional, but before the **select**() call? The answer is that **select**() would block indefinitely, even though an event is actually pending. This race condition is solved by the **pselect**() call. This call can be used to set the signal mask to a set of signals that are to be received only within the **pselect**() call. For instance, let us say that the event in question was the exit of a child process. Before the start of the main loop, we would block **SIGCHLD** using sigprocmask(2). Our **pselect**() call would enable **SIGCHLD** by using an empty signal mask. Our program would look like:

```
static volatile sig_atomic_t got_SIGCHLD = 0;

static void
child_sig_handler(int sig)
{
got_SIGCHLD = 1;
}

int
main(int argc, char *argv[])
{
sigset_t sigmask, empty_mask;
struct sigaction sa;
fd_set readfds, writefds, exceptfds;
int r;
```

```
sigemptyset(&sigmask);
sigaddset(&sigmask, SIGCHLD);
if (sigprocmask(SIG_BLOCK, &sigmask, NULL) == -1) {
perror("sigprocmask");
exit(EXIT_FAILURE);
}

sa.sa_flags = 0;
sa.sa_handler = child_sig_handler;
sigemptyset(&sa.sa_mask);
if (sigaction(SIGCHLD, &sa, NULL) == -1) {
perror("sigaction");
exit(EXIT_FAILURE);
}

sigemptyset(&empty_mask);

for (;;) {          /* main loop */
/* Initialize readfds, writefds, and exceptfds
before the pselect() call. (Code omitted.) */

r = pselect(nfds, &readfds, &writefds, &exceptfds,
NULL, &empty_mask);
if (r == -1 && errno != EINTR) {
/* Handle error */
}

if (got_SIGCHLD) {
got_SIGCHLD = 0;

/* Handle signalled event here; e.g., wait() for all
terminated children. (Code omitted.) */
}

/* main body of program */
}
}
```

Practical

So what is the point of **select**()? Can't I just read and write to my file descriptors whenever I want? The point of **select**() is that it watches multiple descriptors at the same time and properly puts the process to sleep if there is no activity. UNIX programmers often find themselves in a position where they have to handle I/O from more than one file descriptor where the data flow may be intermittent. If you were to merely create a sequence of read(2) and write(2) calls, you would find that one of your calls may block waiting for data from/to a file descriptor, while another file descriptor is unused though ready for I/O. **select**() efficiently copes with this situation.

Select law

Many people who try to use **select**() come across behavior that is difficult to understand and produces nonportable or borderline results. For instance, the above program is carefully written not to block at any point, even though it does not set its file descriptors to nonblocking mode. It is easy to introduce subtle errors that will remove the advantage of using **select**(), so here is a list of essentials to watch for when using **select**().

1. You should always try to use **select**() without a timeout. Your program should have nothing to do if there is no data available. Code that depends on timeouts is not usually portable and is difficult to debug.

2. The value *nfds* must be properly calculated for efficiency as explained above.
3. No file descriptor must be added to any set if you do not intend to check its result after the **select**() call, and respond appropriately. See next rule.
4. After **select**() returns, all file descriptors in all sets should be checked to see if they are ready.
5. The functions read(2), recv(2), write(2), and send(2) do *not* necessarily read/write the full amount of data that you have requested. If they do read/write the full amount, it's because you have a low traffic load and a fast stream. This is not always going to be the case. You should cope with the case of your functions managing to send or receive only a single byte.
6. Never read/write only in single bytes at a time unless you are really sure that you have a small amount of data to process. It is extremely inefficient not to read/write as much data as you can buffer each time. The buffers in the example below are 1024 bytes although they could easily be made larger.
7. Calls to read(2), recv(2), write(2), send(2), and **select**() can fail with the error **EINTR**, and calls to read(2), recv(2) write(2), and send(2) can fail with *errno* set to **EAGAIN** (**EWOULDBLOCK**). These results must be properly managed (not done properly above). If your program is not going to receive any signals, then it is unlikely you will get **EINTR**. If your program does not set nonblocking I/O, you will not get **EAGAIN**.
8. Never call read(2), recv(2), write(2), or send(2) with a buffer length of zero.
9. If the functions read(2), recv(2), write(2), and send(2) fail with errors other than those listed in **7.**, or one of the input functions returns 0, indicating end of file, then you should *not* pass that file descriptor to **select**() again. In the example below, I close the file descriptor immediately, and then set it to −1 to prevent it being included in a set.
10. The timeout value must be initialized with each new call to **select**(), since some operating systems modify the structure. **pselect**() however does not modify its timeout structure.
11. Since **select**() modifies its file descriptor sets, if the call is being used in a loop, then the sets must be reinitialized before each call.

Usleep emulation

On systems that do not have a usleep(3) function, you can call **select**() with a finite timeout and no file descriptors as follows:

```
struct timeval tv;
tv.tv_sec = 0;
tv.tv_usec = 200000;  /* 0.2 seconds */
select(0, NULL, NULL, NULL, &tv);
```

This is guaranteed to work only on UNIX systems, however.

RETURN VALUE

On success, **select**() returns the total number of file descriptors still present in the file descriptor sets.

If **select**() timed out, then the return value will be zero. The file descriptors set should be all empty (but may not be on some systems).

A return value of −1 indicates an error, with *errno* being set appropriately. In the case of an error, the contents of the returned sets and the *struct timeout* contents are undefined and should not be used. **pselect**() however never modifies *ntimeout*.

NOTES

Generally speaking, all operating systems that support sockets also support **select**(). **select**() can be used to solve many problems in a portable and efficient way that naive programmers try to solve in a more complicated manner using threads, forking, IPCs, signals, memory sharing, and so on.

The poll(2) system call has the same functionality as **select**(), and is somewhat more efficient when monitoring sparse file descriptor sets. It is nowadays widely available, but historically was less portable than **select**().

The Linux-specific epoll(7) API provides an interface that is more efficient than select(2) and poll(2) when monitoring large numbers of file descriptors.

EXAMPLE

Here is an example that better demonstrates the true utility of **select**(). The listing below is a TCP forwarding program that forwards from one TCP port to another.

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

static int forward_port;

#undef max
#define max(x,y) ((x) > (y) ? (x) : (y))

static int
listen_socket(int listen_port)
{
struct sockaddr_in addr;
int lfd;
int yes;

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

yes = 1;
if (setsockopt(lfd, SOL_SOCKET, SO_REUSEADDR,
&yes, sizeof(yes)) == -1) {
perror("setsockopt");
close(lfd);
return -1;
}

memset(&addr, 0, sizeof(addr));
addr.sin_port = htons(listen_port);
addr.sin_family = AF_INET;
if (bind(lfd, (struct sockaddr *) &addr, sizeof(addr)) == -1) {
perror("bind");
close(lfd);
return -1;
}

printf("accepting connections on port %d\n", listen_port);
listen(lfd, 10);
return lfd;
}

static int
```

```
connect_socket(int connect_port, char *address)
{
struct sockaddr_in addr;
int cfd;

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

memset(&addr, 0, sizeof(addr));
addr.sin_port = htons(connect_port);
addr.sin_family = AF_INET;

if (!inet_aton(address, (struct in_addr *) &addr.sin_addr.s_addr)) {
perror("bad IP address format");
close(cfd);
return -1;
}

if (connect(cfd, (struct sockaddr *) &addr, sizeof(addr)) == -1) {
perror("connect()");
shutdown(cfd, SHUT_RDWR);
close(cfd);
return -1;
}
return cfd;
}

#define SHUT_FD1 do {                                \
if (fd1 >= 0) {                 \
shutdown(fd1, SHUT_RDWR);   \
close(fd1);                 \
fd1 = -1;                   \
}                               \
} while (0)

#define SHUT_FD2 do {                                \
if (fd2 >= 0) {                 \
shutdown(fd2, SHUT_RDWR);   \
close(fd2);                 \
fd2 = -1;                   \
}                               \
} while (0)

#define BUF_SIZE 1024

int
main(int argc, char *argv[])
{
int h;
int fd1 = -1, fd2 = -1;
char buf1[BUF_SIZE], buf2[BUF_SIZE];
int buf1_avail = 0, buf1_written = 0;
int buf2_avail = 0, buf2_written = 0;

if (argc != 4) {
fprintf(stderr, "Usage\n\tfwd <listen-port> "
```

```
"<forward-to-port> <forward-to-ip-address>\n");
exit(EXIT_FAILURE);
}

signal(SIGPIPE, SIG_IGN);

forward_port = atoi(argv[2]);

h = listen_socket(atoi(argv[1]));
if (h == -1)
exit(EXIT_FAILURE);

for (;;) {
int ready, nfds = 0;
ssize_t nbytes;
fd_set readfds, writefds, exceptfds;

FD_ZERO(&readfds);
FD_ZERO(&writefds);
FD_ZERO(&exceptfds);
FD_SET(h, &readfds);
nfds = max(nfds, h);

if (fd1 > 0 && buf1_avail < BUF_SIZE)
FD_SET(fd1, &readfds);
/* Note: nfds is updated below, when fd1 is added to
exceptfds. */
if (fd2 > 0 && buf2_avail < BUF_SIZE)
FD_SET(fd2, &readfds);

if (fd1 > 0 && buf2_avail - buf2_written > 0)
FD_SET(fd1, &writefds);
if (fd2 > 0 && buf1_avail - buf1_written > 0)
FD_SET(fd2, &writefds);

if (fd1 > 0) {
FD_SET(fd1, &exceptfds);
nfds = max(nfds, fd1);
}
if (fd2 > 0) {
FD_SET(fd2, &exceptfds);
nfds = max(nfds, fd2);
}

ready = select(nfds + 1, &readfds, &writefds, &exceptfds, NULL);

if (ready == -1 && errno == EINTR)
continue;

if (ready == -1) {
perror("select()");
exit(EXIT_FAILURE);
}

if (FD_ISSET(h, &readfds)) {
socklen_t addrlen;
struct sockaddr_in client_addr;
int fd;

addrlen = sizeof(client_addr);
memset(&client_addr, 0, addrlen);
```

```
fd = accept(h, (struct sockaddr *) &client_addr, &addrlen);
if (fd == -1) {
perror("accept()");
} else {
SHUT_FD1;
SHUT_FD2;
buf1_avail = buf1_written = 0;
buf2_avail = buf2_written = 0;
fd1 = fd;
fd2 = connect_socket(forward_port, argv[3]);
if (fd2 == -1)
SHUT_FD1;
else
printf("connect from %s\n",
inet_ntoa(client_addr.sin_addr));

/* Skip any events on the old, closed file descriptors. */
continue;
}
}

/* NB: read OOB data before normal reads */

if (fd1 > 0 && FD_ISSET(fd1, &exceptfds)) {
char c;

nbytes = recv(fd1, &c, 1, MSG_OOB);
if (nbytes < 1)
SHUT_FD1;
else
send(fd2, &c, 1, MSG_OOB);
}
if (fd2 > 0 && FD_ISSET(fd2, &exceptfds)) {
char c;

nbytes = recv(fd2, &c, 1, MSG_OOB);
if (nbytes < 1)
SHUT_FD2;
else
send(fd1, &c, 1, MSG_OOB);
}
if (fd1 > 0 && FD_ISSET(fd1, &readfds)) {
nbytes = read(fd1, buf1 + buf1_avail,
BUF_SIZE - buf1_avail);
if (nbytes < 1)
SHUT_FD1;
else
buf1_avail += nbytes;
}
if (fd2 > 0 && FD_ISSET(fd2, &readfds)) {
nbytes = read(fd2, buf2 + buf2_avail,
BUF_SIZE - buf2_avail);
if (nbytes < 1)
SHUT_FD2;
else
buf2_avail += nbytes;
}
```

```
if (fd1 > 0 && FD_ISSET(fd1, &writefds) && buf2_avail > 0) {
nbytes = write(fd1, buf2 + buf2_written,
buf2_avail - buf2_written);
if (nbytes < 1)
SHUT_FD1;
else
buf2_written += nbytes;
}
if (fd2 > 0 && FD_ISSET(fd2, &writefds) && buf1_avail > 0) {
nbytes = write(fd2, buf1 + buf1_written,
buf1_avail - buf1_written);
if (nbytes < 1)
SHUT_FD2;
else
buf1_written += nbytes;
}

/* Check if write data has caught read data */

if (buf1_written == buf1_avail)
buf1_written = buf1_avail = 0;
if (buf2_written == buf2_avail)
buf2_written = buf2_avail = 0;

/* One side has closed the connection, keep
writing to the other side until empty */

if (fd1 < 0 && buf1_avail - buf1_written == 0)
SHUT_FD2;
if (fd2 < 0 && buf2_avail - buf2_written == 0)
SHUT_FD1;
}
exit(EXIT_SUCCESS);
}
```

The above program properly forwards most kinds of TCP connections including OOB signal data transmitted by **telnet** servers. It handles the tricky problem of having data flow in both directions simultaneously. You might think it more efficient to use a fork(2) call and devote a thread to each stream. This becomes more tricky than you might suspect. Another idea is to set nonblocking I/O using fcntl(2). This also has its problems because you end up using inefficient timeouts.

The program does not handle more than one simultaneous connection at a time, although it could easily be extended to do this with a linked list of buffers—one for each connection. At the moment, new connections cause the current connection to be dropped.

SEE ALSO

accept(2), connect(2), ioctl(2), poll(2), read(2), recv(2), select(2), send(2), sigprocmask(2), write(2), sigaddset(3), sigdelset(3), sigemptyset(3), sigfillset(3), sigismember(3), epoll(7)

COLOPHON

This page is part of release 4.16 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man–pages/.