NAME

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

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

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

typedef /* ... */ fd_set;

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

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);

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

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

pselect():
_POSIX_C_SOURCE >= 200112L

DESCRIPTION

WARNING: **select**() can monitor only file descriptors numbers that are less than **FD_SETSIZE** (1024)—an unreasonably low limit for many modern applications—and this limitation will not change. All modern applications should instead use **poll**(2) or **epoll**(7), which do not suffer this limitation.

select() allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). A file descriptor is considered ready if it is possible to perform a corresponding I/O operation (e.g., **read**(2), or a sufficiently small **write**(2)) without blocking.

fd_set

A structure type that can represent a set of file descriptors. According to POSIX, the maximum number of file descriptors in an *fd_set* structure is the value of the macro **FD_SETSIZE**.

File descriptor sets

The principal arguments of **select**() are three "sets" of file descriptors (declared with the type *fd_set*), which allow the caller to wait for three classes of events on the specified set of file descriptors. Each of the *fd_set* arguments may be specified as NULL if no file descriptors are to be watched for the corresponding class of events.

Note well: Upon return, each of the file descriptor sets is modified in place to indicate which file descriptors are currently "ready". Thus, if using **select**() within a loop, the sets *must be reinitialized* before each call.

The contents of a file descriptor set can be manipulated using the following macros:

FD_ZERO()
: This macro clears (removes all file descriptors from) *set*. It should be employed as the first step in initializing a file descriptor set.

FD_SET()
: This macro adds the file descriptor *fd* to *set*. Adding a file descriptor that is already present in the set is a no-op, and does not produce an error.

FD_CLR()
This macro removes the file descriptor *fd* from *set*. Remo ving a file descriptor that is not present in the set is a no-op, and does not produce an error.

FD_ISSET()
select() modifies the contents of the sets according to the rules described below. After calling**select**(), the **FD_ISSET**() macro can be used to test if a file descriptor is still present in a set. **FD_ISSET**() returns nonzero if the file descriptor *fd* is present in *set*, and zero if it is not.

Arguments

The arguments of **select**() are as follows:

readfds The file descriptors in this set are watched to see if they are ready for reading. A file descriptor is ready for reading if a read operation will not block; in particular, a file descriptor is also ready on end-of-file.

After **select**() has returned, *readfds* will be cleared of all file descriptors except for those that are ready for reading.

writefds
The file descriptors in this set are watched to see if they are ready for writing. A file descriptor is ready for writing if a write operation will not block. However, even if a file descriptor indicates as writable, a large write may still block.

After **select**() has returned, *writefds* will be cleared of all file descriptors except for those that are ready for writing.

exceptfds
The file descriptors in this set are watched for "exceptional conditions". For examples of some exceptional conditions, see the discussion of **POLLPRI** in **poll**(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 argument should be set to the highest-numbered file descriptor in any of the three sets, plus 1. The indicated file descriptors in each set are checked, up to this limit (but see BUGS).

timeout The *timeout* argument is a *timeval* structure (shown below) that specifies the interval that **select**() should block waiting for a file descriptor to become ready. The call will block until either:

- a file descriptor becomes ready;
- the call is interrupted by a signal handler; or
- the timeout expires.

Note that the *timeout* interval will be rounded up to the system clock granularity, and kernel scheduling delays mean that the blocking interval may overrun by a small amount.

If both fields of the *timeval* structure are zero, then **select**() returns immediately. (This is useful for polling.)

If *timeout* is specified as NULL, **select**() blocks indefinitely waiting for a file descriptor to become ready.

pselect()

The **pselect**() system call allows an application to safely wait until either a file descriptor becomes ready or until a signal is caught.

The operation of **select**() and **pselect**() is identical, other than these three differences:

- **select**() uses a timeout that is a *struct timeval* (with seconds and microseconds), while **pselect**() uses a *struct timespec* (with seconds and nanoseconds).
- **select**() may update the *timeout* argument to indicate how much time was left. **pselect**() does not change this argument.

- **select**() has no *sigmask* argument, and behaves as **pselect**() called with NULL *sigmask*.

sigmask is a pointer to a signal mask (see **sigprocmask**(2)); if it is not NULL, then **pselect**() first replaces the current signal mask by the one pointed to by *sigmask*, then does the "select" function, and then restores the original signal mask. (If *sigmask* is NULL, the signal mask is not modified during the **pselect**() call.)

Other than the difference in the precision of the *timeout* argument, the following **pselect**() call:

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

is equivalent to *atomically* executing the following calls:

```
sigset_t origmask;

pthread_sigmask(SIG_SETMASK, &sigmask, &origmask);
ready = select(nfds, &readfds, &writefds, &exceptfds, timeout);
pthread_sigmask(SIG_SETMASK, &origmask, NULL);
```

The reason that **pselect**() is needed is that if one wants to wait for either a signal or for a file descriptor to become ready, then an atomic test is needed to prevent race conditions. (Suppose the signal handler sets a global flag and returns. Then a test of this global flag followed by a call of **select**() could hang indefinitely if the signal arrived just after the test but just before the call. By contrast, **pselect**() allows one to first block signals, handle the signals that have come in, then call **pselect**() with the desired *sigmask*, avoiding the race.)

The timeout

The *timeout* argument for **select**() is a structure of the following type:

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

The corresponding argument for **pselect**() is a **timespec**(3) structure.

On Linux, **select**() modifies *timeout* to reflect the amount of time not slept; most other implementations do not do this. (POSIX.1 permits either behavior.) This causes problems both when Linux code which reads *timeout* is ported to other operating systems, and when code is ported to Linux that reuses a *struct timeval* for multiple **select**()s in a loop without reinitializing it. Consider *timeout* to be undefined after **select**() returns.

RETURN VALUE

On success, **select**() and **pselect**() return the number of file descriptors contained in the three returned descriptor sets (that is, the total number of bits that are set in *readfds*, *writefds*, *exceptfds*). The return value may be zero if the timeout expired before any file descriptors became ready.

On error, −1 is returned, and *errno* is set to indicate the error; the file descriptor sets are unmodified, and *timeout* becomes undefined.

ERRORS

EBADF

An invalid file descriptor was given in one of the sets. (Perhaps a file descriptor that was already closed, or one on which an error has occurred.) However, see BUGS.

EINTR

A signal was caught; see **signal**(7).

EINVAL

nfds is negative or exceeds the **RLIMIT_NOFILE** resource limit (see **getrlimit**(2)).

EINVAL

The value contained within *timeout* is invalid.

ENOMEM
Unable to allocate memory for internal tables.

VERSIONS

pselect() was added in Linux 2.6.16. Prior to this, **pselect**() was emulated in glibc (but see BUGS).

STANDARDS

select() conforms to POSIX.1-2001, POSIX.1-2008, and 4.4BSD (**select**() first appeared in 4.2BSD). Generally portable to/from non-BSD systems supporting clones of the BSD socket layer (including System V variants). However, note that the System V variant typically sets the timeout variable before returning, but the BSD variant does not.

pselect() is defined in POSIX.1g, and in POSIX.1-2001 and POSIX.1-2008.

fd_set is defined in POSIX.1-2001 and later.

NOTES

The following header also provides the *fd_set* type: <*sys/time .h*>.

An *fd_set* is a fixed size buffer. Executing **FD_CLR**() or **FD_SET**() with a value of *fd* that is ne gative or is equal to or larger than **FD_SETSIZE** will result in undefined behavior. Moreover, POSIX requires *fd* to be a valid file descriptor.

The operation of **select**() and **pselect**() is not affected by the **O_NONBLOCK** flag.

On some other UNIX systems, **select**() can fail with the error **EAGAIN** if the system fails to allocate kernel-internal resources, rather than **ENOMEM** as Linux does. POSIX specifies this error for **poll**(2), but not for **select**(). Portable programs may wish to check for**EA GAIN** and loop, just as with **EINTR**.

The self-pipe trick

On systems that lack **pselect**(), reliable (and more portable) signal trapping can be achieved using the self-pipe trick. In this technique, a signal handler writes a byte to a pipe whose other end is monitored by **select**() in the main program. (To avoid possibly blocking when writing to a pipe that may be full or reading from a pipe that may be empty, nonblocking I/O is used when reading from and writing to the pipe.)

Emulating usleep(3)

Before the advent of **usleep**(3), some code employed a call to **select**() with all three sets empty, *nfds* zero, and a non-NULL *timeout* as a fairly portable way to sleep with subsecond precision.

Correspondence between select() and poll() notifications

Within the Linux kernel source, we find the following definitions which show the correspondence between the readable, writable, and exceptional condition notifications of **select**() and the event notifications provided by **poll**(2) and **epoll**(7):

```
#define POLLIN_SET  (EPOLLRDNORM | EPOLLRDBAND | EPOLLIN |
                     EPOLLHUP | EPOLLERR)
                    /* Ready for reading */
#define POLLOUT_SET (EPOLLWRBAND | EPOLLWRNORM | EPOLLOUT |
                     EPOLLERR)
                    /* Ready for writing */
#define POLLEX_SET  (EPOLLPRI)
                    /* Exceptional condition */
```

Multithreaded applications

If a file descriptor being monitored by **select**() is closed in another thread, the result is unspecified. On some UNIX systems, **select**() unblocks and returns, with an indication that the file descriptor is ready (a subsequent I/O operation will likely fail with an error, unless another process reopens the file descriptor between the time **select**() returned and the I/O operation is performed). On Linux (and some other systems), closing the file descriptor in another thread has no effect on **select**(). In summary, any application that relies on a particular behavior in this scenario must be considered buggy.

C library/kernel differences

The Linux kernel allows file descriptor sets of arbitrary size, determining the length of the sets to be checked from the value of *nfds*. However, in the glibc implementation, the *fd_set* type is fixed in size. See also BUGS.

The **pselect**() interface described in this page is implemented by glibc. The underlying Linux system call is named **pselect6**(). This system call has somewhat different behavior from the glibc wrapper function.

The Linux **pselect6**() system call modifies its *timeout* argument. However, the glibc wrapper function hides this behavior by using a local variable for the timeout argument that is passed to the system call. Thus, the glibc **pselect**() function does not modify its *timeout* argument; this is the behavior required by POSIX.1-2001.

The final argument of the **pselect6**() system call is not a *sigset_t* * pointer, but is instead a structure of the form:

```
struct {
    const kernel_sigset_t *ss;   /* Pointer to signal set */
    size_t ss_len;               /* Size (in bytes) of object
                                    pointed to by 'ss' */
};
```

This allows the system call to obtain both a pointer to the signal set and its size, while allowing for the fact that most architectures support a maximum of 6 arguments to a system call. See **sigprocmask**(2) for a discussion of the difference between the kernel and libc notion of the signal set.

Historical glibc details

glibc 2.0 provided an incorrect version of **pselect**() that did not take a *sigmask* argument.

From glibc 2.1 to glibc 2.2.1, one must define **_GNU_SOURCE** in order to obtain the declaration of **pselect**() from <*sys/select.h*>.

BUGS

POSIX allows an implementation to define an upper limit, advertised via the constant **FD_SETSIZE**, on the range of file descriptors that can be specified in a file descriptor set. The Linux kernel imposes no fixed limit, but the glibc implementation makes *fd_set* a fixed-size type, with **FD_SETSIZE** defined as 1024, and the **FD_***() macros operating according to that limit. To monitor file descriptors greater than 1023, use **poll**(2) or **epoll**(7) instead.

The implementation of the *fd_set* arguments as value-result arguments is a design error that is avoided in **poll**(2) and **epoll**(7).

According to POSIX, **select**() should check all specified file descriptors in the three file descriptor sets, up to the limit *nfds−1*. However, the current implementation ignores any file descriptor in these sets that is greater than the maximum file descriptor number that the process currently has open. According to POSIX, any such file descriptor that is specified in one of the sets should result in the error **EBADF**.

Starting with glibc 2.1, glibc provided an emulation of **pselect**() that was implemented using **sigprocmask**(2) and **select**(). This implementation remained vulnerable to the very race condition that **pselect**() was designed to prevent. Modern versions of glibc use the (race-free) **pselect**() system call on kernels where it is provided.

On Linux, **select**() may report a socket file descriptor as "ready for reading", while nevertheless a subsequent read blocks. This could for example happen when data has arrived but upon examination has the wrong checksum and is discarded. There may be other circumstances in which a file descriptor is spuriously reported as ready. Thus it may be safer to use**O_NONBLOCK** on sockets that should not block.

On Linux, **select**() also modifies *timeout* if the call is interrupted by a signal handler (i.e., the **EINTR** error return). This is not permitted by POSIX.1. The Linux**pselect**() system call has the same behavior, but the glibc wrapper hides this behavior by internally copying the *timeout* to a local variable and passing that variable to the system call.

EXAMPLES

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

int
main(void)
{
    int             retval;
    fd_set          rfds;
    struct timeval  tv;

    /* Watch stdin (fd 0) to see when it has input. */

    FD_ZERO(&rfds);
    FD_SET(0, &rfds);

    /* Wait up to five seconds. */

    tv.tv_sec = 5;
    tv.tv_usec = 0;

    retval = select(1, &rfds, NULL, NULL, &tv);
    /* Don't rely on the value of tv now! */

    if (retval == -1)
        perror("select()");
    else if (retval)
        printf("Data is available now.\n");
        /* FD_ISSET(0, &rfds) will be true. */
    else
        printf("No data within five seconds.\n");

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

accept(2), **connect**(2), **poll**(2), **read**(2), **recv**(2), **restart_syscall**(2), **send**(2), **sigprocmask**(2), **write**(2), **timespec**(3), **epoll**(7), **time**(7)

For a tutorial with discussion and examples, see **select_tut**(2).