I/O multiplexing

Why we need I/O multiplexing?

I/O multiplexing is typically used in networking applications in the following scenarios:

- When a client is handling multiple descriptors (normally interactive input and a network socket)
- When a client to handle multiple sockets at the same time (this is possible, but rare)
- If a TCP server handles both a listening socket and its connected sockets
- If a server handles both TCP and UDP
- If a server handles multiple services and perhaps multiple protocols

I/O Models

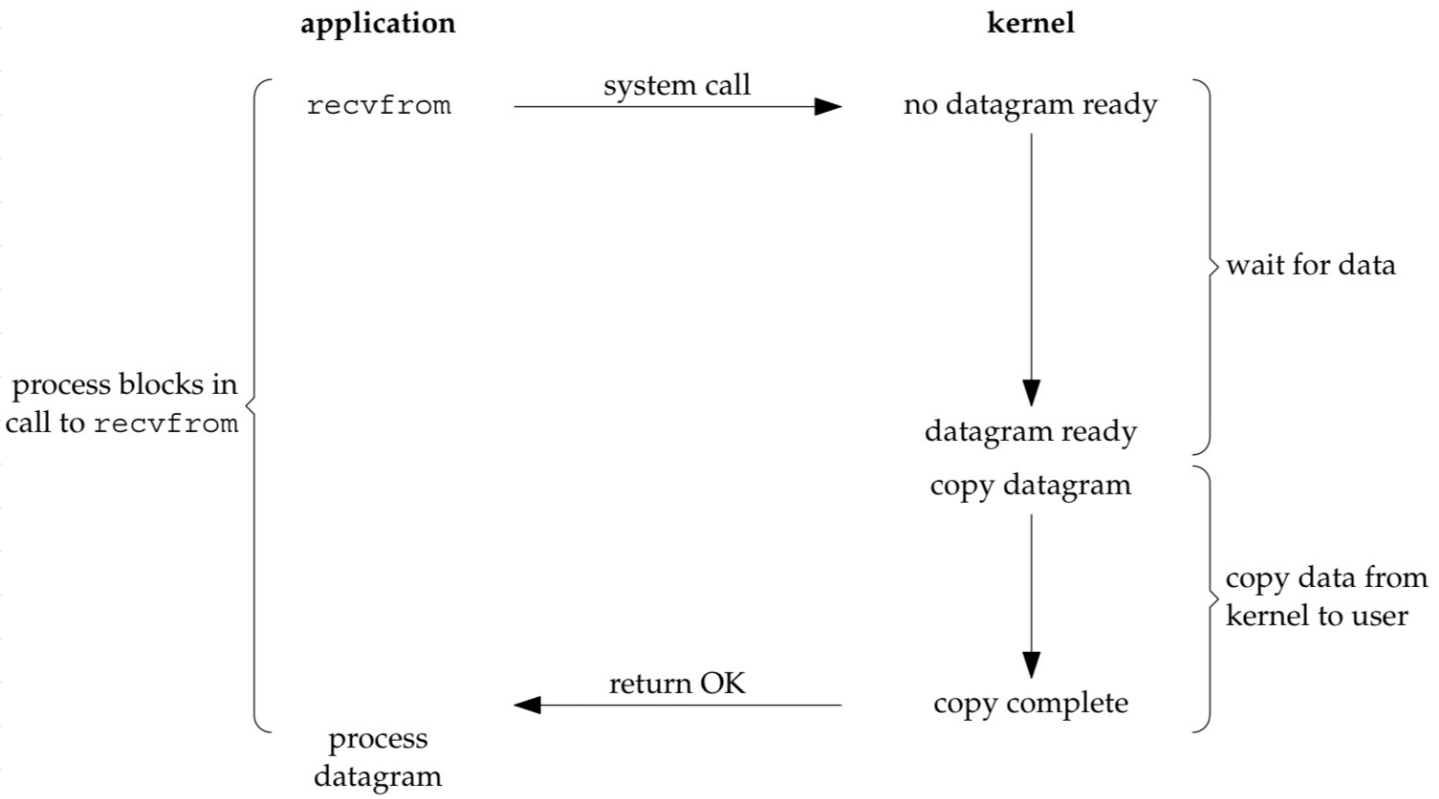
I/O models that are available to us under Unix:

- blocking I/O
- nonblocking I/O
- I/O multiplexing
- signal driven I/O
- asynchronous I/O

There are two phases for any input operation

1. Waiting for data to be ready
2. Copying data from kernel to process

Blocking I/O Model

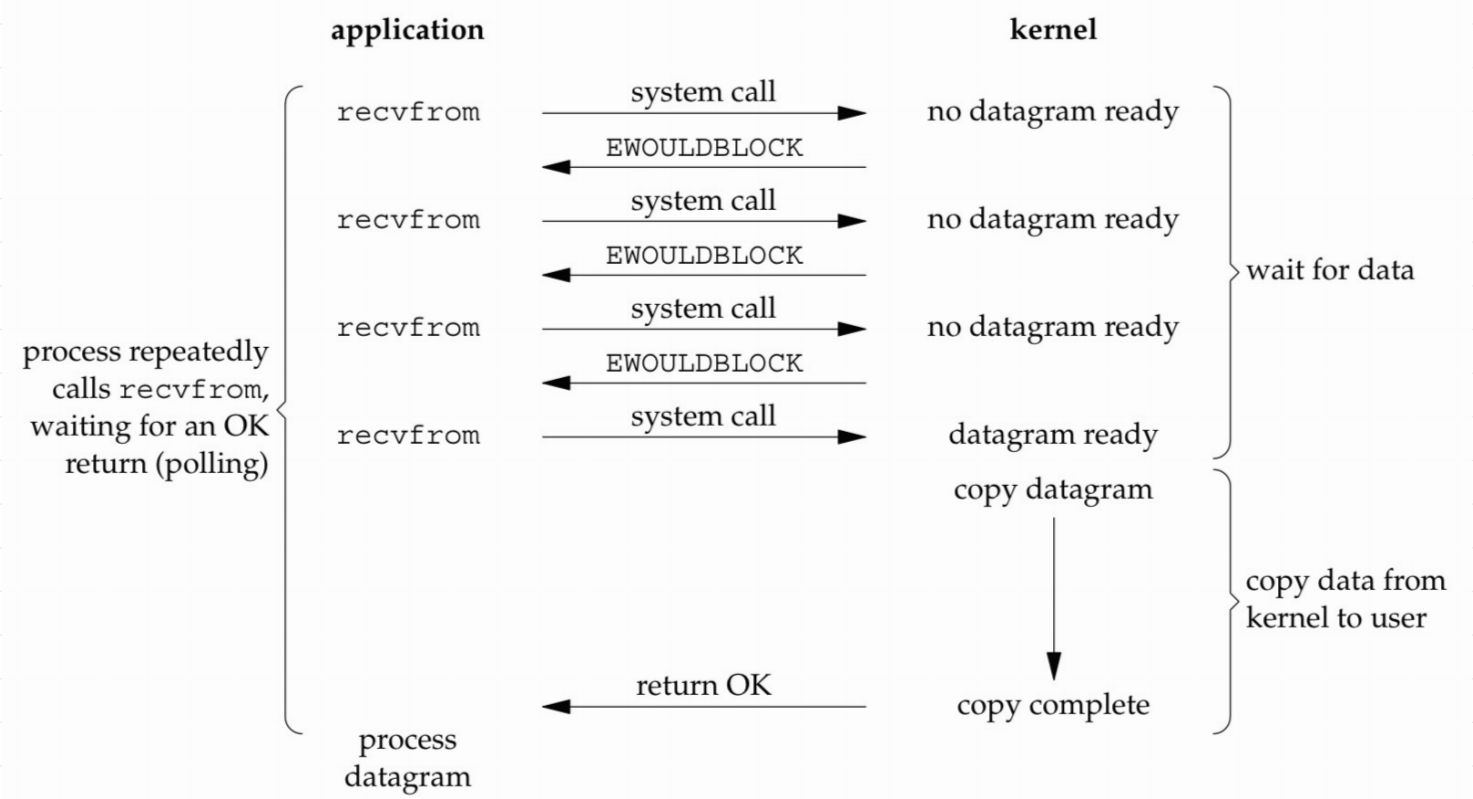


The system call does not return until data has arrived and been copied into our application. Sort of made feasible in a multithreaded environment, where meaningful work can occur even if some threads are blocked.

By default, all sockets are blocking.

In Figure,the process calls recvfrom and the system call does not return until the datagram arrives and is copied into our application buffer, or an error occurs. The most common error is the system call being interrupted by a signal. We say that our process is blocked the entire time from when it calls recvfrom until it returns. When recvfrom returns successfully, our application processes the datagram.

Nonblocking I/O Model

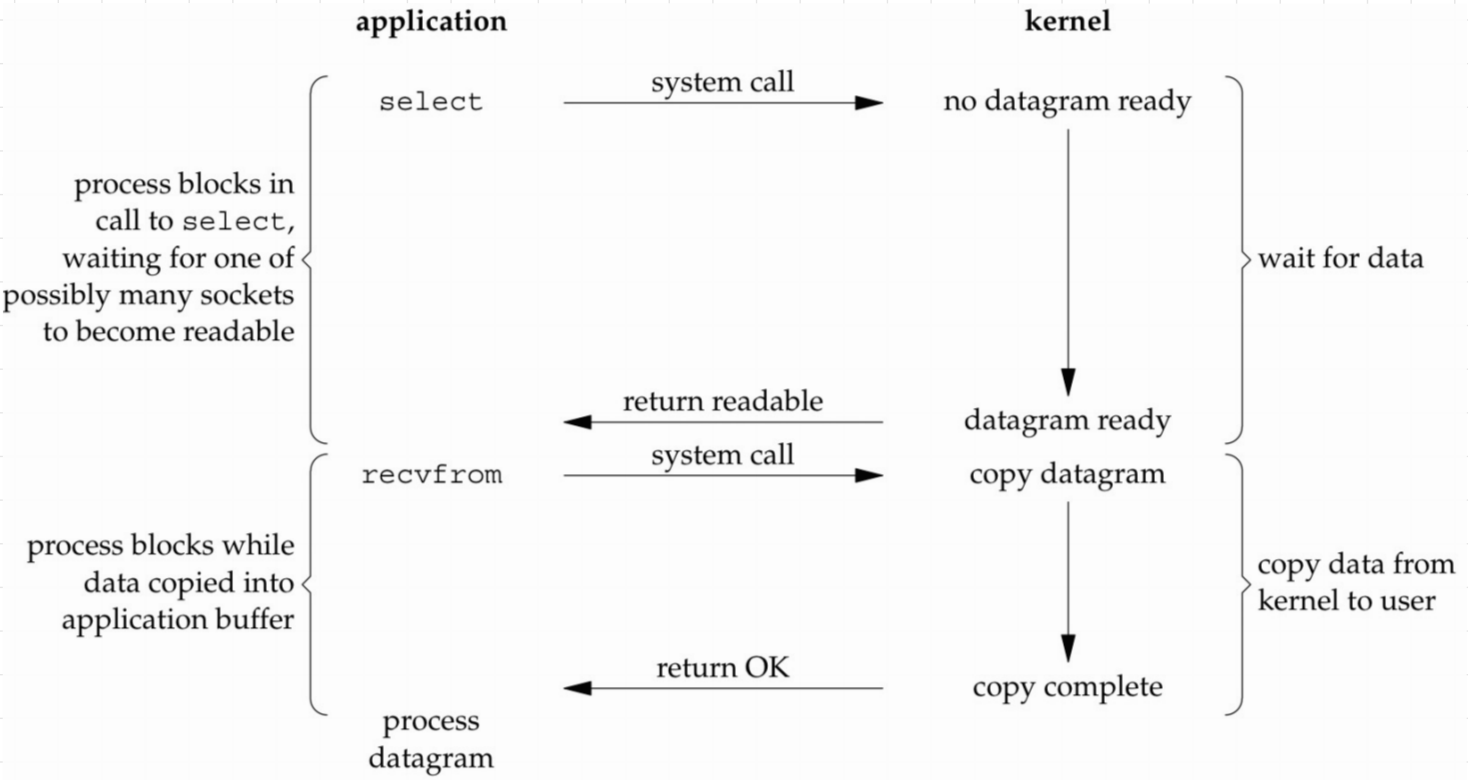


Instead of blocking, the system calls can immediately return EWOULDBLOCK if data is not available. This means we can now loop through all connections and serve them as needed, but if nothing is available we end up polling (busy waiting) and burning through CPU.

In figure, the first three times that we call recvfrom, there is no data to return, so the kernel immediately returns an error of EWOULDBLOCK instead. The fourth time we call recvfrom, a datagram is ready, it is copied into our application buffer, and recvfrom returns successfully. We then process the data.

When an application sits in a loop calling recvfrom on a nonblocking descriptor like this, it is called polling. The application is continually polling the kernel to see if some operation is ready. This is often a waste of CPU time, but this model is occasion- ally encountered, normally on systems dedicated to one function.

I/O Multiplexing Model

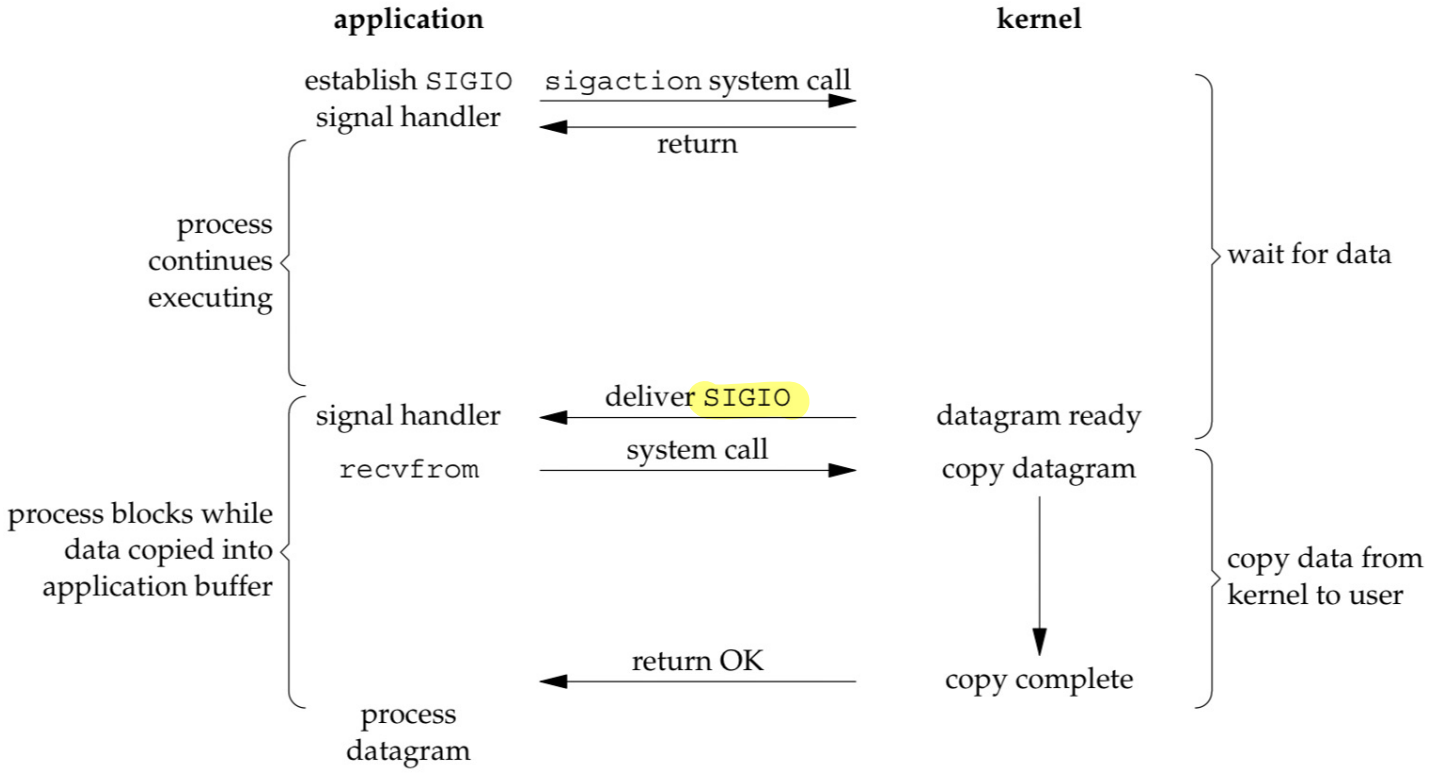


Rather than work on just one file descriptor at a time, we might want a method to monitor changes on lots of file descriptors. These are all just different polling methods/iterations for doing that.

With I/O multiplexing, we call select or poll and block in one of these two system calls, instead of blocking in the actual I/O system call.

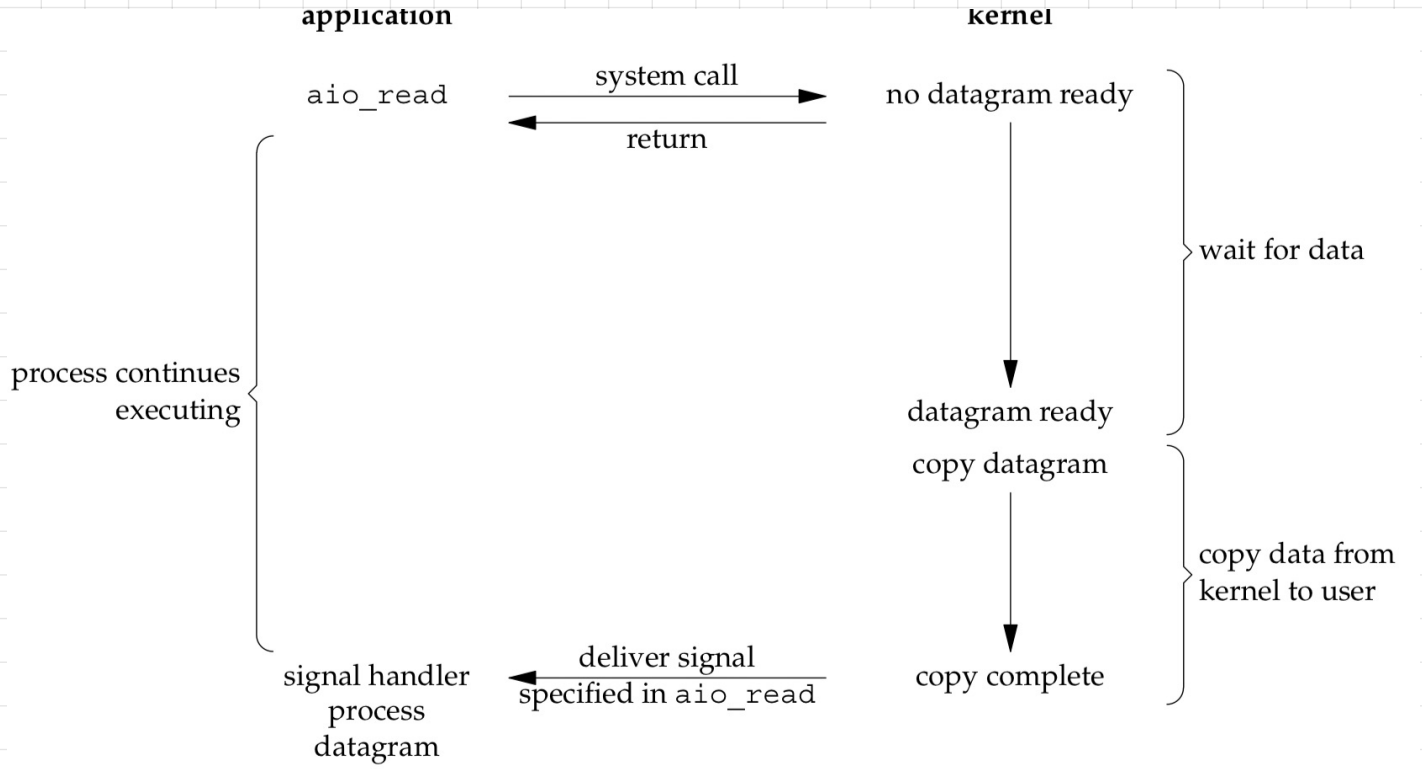
We block in a call to select, waiting for the datagram socket to be readable. When select returns that the socket is readable, we then call recvfrom to copy the datagram into our application buffer.

Signal-Driven I/O Model

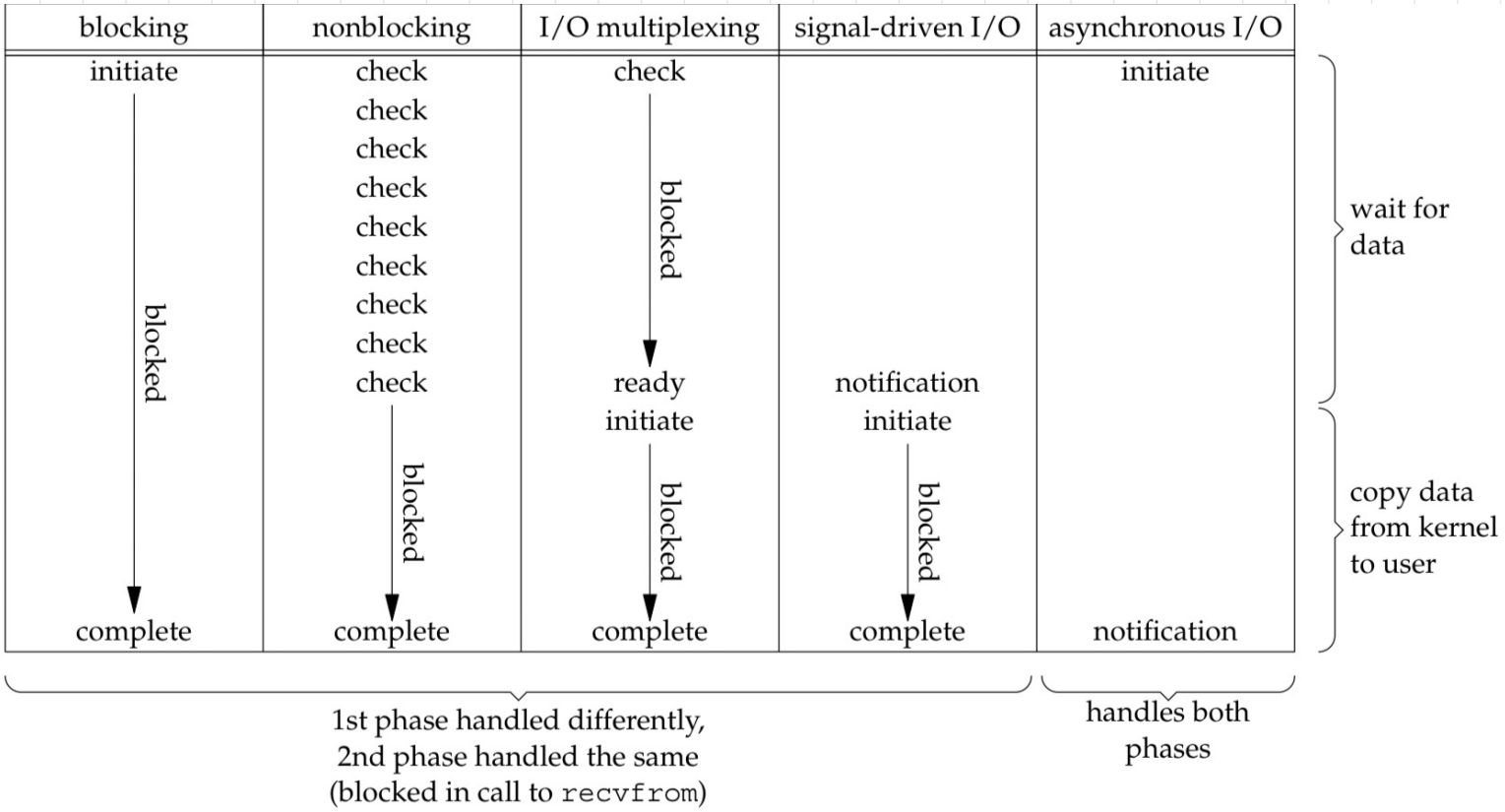


We can also use signals, telling the kernel to notify us with the SIGIO signal when the descriptor is ready.

Asynchronous I/O Model



Comparison of the I/O Models



Synchronous I/O versus Asynchronous I/O

- A synchronous I/O operation causes the requesting process to be blocked until that I/O operation completes.
- An asynchronous I/O operation does not cause the requesting process to be blocked.

select Function

This function allows the process to instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one or more of these events occurs or when a specified amount of time has passed.

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

int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
           const struct timeval *timeout);
```

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

The timeout argument

The timeout argument tells the kernel how long to wait for one of the specified descriptors to become ready. A timeval structure specifies the number of seconds and microseconds.

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

There are three possibilities:

1. Wait forever—Return only when one of the specified descriptors is ready for I/O. For this, we specify the timeout argument as a null pointer.
2. Wait up to a fixed amount of time—Return when one of the specified descriptors is ready for I/O, but do not wait beyond the number of seconds and microseconds specified in the timeval structure pointed to by the timeout argument.
3. Do not wait at all—Return immediately after checking the descriptors. This is called polling. To specify this, the timeout argument must point to a timeval structure and the timer value (the number of seconds and microseconds specified by the structure) must be 0.

The descriptor sets arguments

The three middle arguments, readset, writeset, and exceptset, specify the descriptors that we want the kernel to test for reading, writing, and exception conditions. There are only two exception conditions currently supported:

- The arrival of out-of-band data for a socket.
- The presence of control status information to be read from the master side of a pseudo-terminal that has been put into packet mode. (Not covered in UNP)

select uses descriptor sets, typically an array of integers, with each bit in each integer corresponding to a descriptor. For example, using 32-bit integers, the first element of the array corresponds to descriptors 0 through 31, the second element of the array corresponds to descriptors 32 through 63, and so on. All the implementation details are irrelevant to the application and are hidden in the fd_set datatype and the following four macros:

```
void FD_ZERO(fd_set *fdset);            /* clear all bits in fdset */
void FD_SET(int fd, fd_set *fdset);     /* turn on the bit for fd in fdset */
void FD_CLR(int fd, fd_set *fdset);     /* turn off the bit for fd in fdset */
int  FD_ISSET(int fd, fd_set *fdset);   /* is the bit for fd on in fdset ? */
```

The maxfdp1 argument

The maxfdp1 argument specifies the number of descriptors to be tested. Its value is the maximum descriptor to be tested plus one. The descriptors 0, 1, 2, up through and including maxfdp1–1 are tested.

Return value of select

The return value from this function indicates the total number of bits that are ready across all the descriptor sets. If the timer value expires before any of the descriptors are ready, a value of 0 is returned. A return value of –1 indicates an error

Three conditions are handled with the socket:

1 If the peer TCP sends data, the socket becomes readable and read returns greater than 0 (the number of bytes of data).

2 If the peer TCP sends a FIN (the peer process terminates), the socket becomes readable and read returns 0 (EOF).

3 If the peer TCP sends an RST (the peer host has crashed and rebooted), the socket becomes readable, read returns −1, and errno contains the specific error code.