

I/O Multiplexing and Non-blocking I/O

COMP 321

Dave Johnson



COMP 321

Copyright © 2026 David B. Johnson

Page 1

1

read() and write() Normally Block the Process

Consider, for example, the following typical code

```
while ((nch = read(STDIN_FILENO, buffer, BUFFER_SIZE)) > 0)  
    write(STDOUT_FILENO, buffer, nch);
```

- Each read() call “blocks” and doesn’t return until it completes
- Each write() call then “blocks” and doesn’t return until it completes
- This may be fine in simple cases like the above

But what if you want or need to read/write multiple files “at once” ?

COMP 321

Copyright © 2026 David B. Johnson

Page 2

2

Why Might a read() on an fd Block?

Just a few reasons a read() from an fd might block

- Reading from a regular file never blocks
 - Even at the end of the file
- Reading from a network connection can block if no new data has yet arrived
- Reading from a pipe (or fifo) can block if no data is currently available
 - If all writing fds get closed, then read will not block since end of file
- Reading from a terminal might block
 - If you haven't yet typed ENTER then you might backspace away any input
 - Only characters before most recent ENTER are available to be read
 - A read from terminal may block since available input may be backspaced

Why Might a write() on an fd Block?

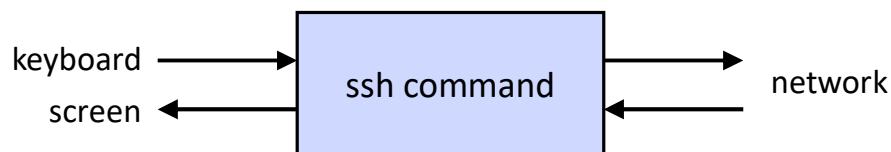
Just a few reasons a write() on an fd might block

- Writing to a regular file never blocks
 - Even if the file system (or disk) is full
- Writing to a network connection can block if output buffering is full
 - Must be able to transmit (and with TCP, get acknowledgement) first
- Writing to a pipe (or fifo) can block if the available buffering is full
- Writing to a terminal can block if the available buffering is full
 - A terminal is a slow device, and it might take a while for previous output to complete and free space in output buffering

A Simple Example of Many read/write at Once

Consider something like the “ssh” command

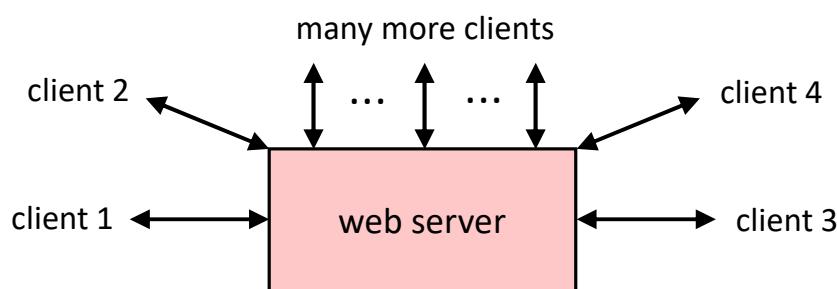
- Read from the keyboard and write to the network connection to the computer you are ssh’d into
- Read from the network connection from the computer you are ssh’d into and write to the screen
- How can the ssh program do all of this (both directions) at once?
- And what about also reading from the mouse and writing to the network?



Another Example

Consider a large, very busy web server

- The web server reads/writes with one web browser client
- While it also reads/writes with another web browser client
- And with another web browser client, etc., for many clients, all at once



A Solution: I/O Multiplexing

The basic idea

- Build a list of all the file descriptors we want to do this kind of I/O on “at once”
- Repeat in a loop
 - Give this list of file descriptors to the kernel
 - The kernel blocks this process until **at least any one** of those descriptors is “**ready**” for next I/O operation (e.g., a read() or a write())
 - Example: you typed something on the keyboard
 - Example: the network is ready for the next data you want to send
 - The process performs the read or write for each descriptors indicated ready
 - For each of those reads or writes, the process shouldn’t need to be blocked since that descriptor is ready for this next I/O

The select() Kernel Call

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

Allows the calling process to ask to watch multiple file descriptors at once

- The file descriptors to watch are specified by three different fd “sets”
- nfds (awkwardly) specifies the limit on which file descriptors to check
 - nfds = highest file descriptor **number** across all 3 sets, **plus 1**
- timeval can specify a timeout after which to return early, even if no I/O is possible yet for any the indicated file descriptors (across all 3 sets)

The Three File Descriptor Sets

Each defines a set of fds to watch for a different type of I/O

- **readfds**
 - The set of fds for the kernel to watch if they are ready for reading from
 - Basically, if it is now possible to do a read from it without blocking
- **writefd**
 - The set of fds for the kernel to watch if they are ready to write to
 - Basically, if it is now possible to do a write to it without blocking
- **exceptfds**
 - The set of fds for the kernel to watch for “exceptional” conditions
 - Largely unused, so we’ll generally ignore it here

Any of the three can be NULL, meaning that set should be treated as empty set

Manipulating an fd_set

```
fd_set my_fd_set;
```

Declares my_fd_set
variable as a set of
file descriptors

- FD_ZERO(&my_fd_set);
 - Initializes the set my_fd_set equal to the empty set
- FD_CLR(int fd, &my_fd_set);
 - Removes fd from the set my_fd_set
- FD_SET(int fd, &my_fd_set);
 - Adds fd to the set my_fd_set
- FD_ISSET(int fd, &my_fd_set);
 - Returns true/false if fd is in the set my_fd_set

The Internal Format of an `fd_set`

A bitmask, with one bit per possible file descriptor number

- Stored as a constant-sized array of integers, using `typedef "fd_set"`
 - If a bit is set in the bitmask, that corresponding fd is “in” the “set”

	fd 0	fd 1	fd 2	fd 3	fd 4	fd 5	fd 6	...
readfds	1	1	0	0	0	1		ignored ...
wrtefds	0	1	1	1	1	0		ignored ...
exceptfds	1	0	1	0	0	0		ignored ...

– ***nfds = 6*** – only 6 bits are used; bits after bit 5 are ignored by `select()`

You should use only the FD_ macros – do not directly access the bits

An Example `select()` Call

```
fd_set read_set;
fd_set write_set;

FD_ZERO(&read_set);
FD_ZERO(&write_set);

FD_SET(0, &read_set);
FD_SET(7, &read_set);
FD_SET(1, &read_set);

FD_SET(4, &write_set);
FD_SET(2, &write_set);

nfds = 8;
status = select(nfds, &read_set, &write_set, NULL, NULL);
```

nfds should be the maximum fd number set across all of the sets on this `select()` call
plus 1, since file descriptor numbers begin at 0, not at 1

The Return Values from select()

For each of the readfds, writefds, and exceptfds fd_set inputs

- Modifies each fd_set **in place** to remove bits for any fds that are not ready
- Leaves **only** the bits set for any fds that are ready for that type of I/O
- **(Overwrites the three original fd_set values!)**

And returns the total count of the specified fds that are ready

- That is, the total count of bits still set, **total across the three sets**
- Will return 0 if it returns due to a timeval timeout instead of any fds becoming ready

Simple Handling of fds on Return from select()

```
status = select(nfds, &read_set, &write_set, NULL, NULL);

if (status > 0 {
    for (i = 0; i < nfds; i++) {
        if (FD_ISSET(i, &read_set)) {
            /* do read() on file descriptor number i */
            ...
        }
        if (FD_ISSET(i, &write_set)) {
            /* do write() on file descriptor number i */
            ...
    }
}
```

The Use of timeval on select()

Can ask the kernel to return after this limit even if none of the fds are ready

- timeval struct has fields for seconds (tv_secs) and microseconds (tv_usecs)
- If timeval pointer == NULL
 - select() waits forever, or until at last some specified fds are ready
- If timeval->tv_secs == 0 && timeval->tv_usecs == 0
 - The kernel checks all of the fds for ready, then always returns **immediately**
- If timeval->tv_secs != 0 || timeval->tv_usecs != 0
 - The kernel returns when any of fds are ready **or** after the specified timeout
 - **Whichever occurs first**
- If **all** of readfds, writefds, and exceptfds are NULL, timeval is still used

An Alternative to select(): poll()

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

A newer, cleaned up interface but roughly still the same

- Input is an array of “struct pollfd”
 - Each specifies fd, events to watch for, and (on return) events that occurred
 - No more bitmaps or overwriting the input sets as in select()

Comparison between select() and poll()

- select() has existed for a long time and is widely available in Unix-like systems
- poll() is more recent, but it’s still been around a long time
- select() seems to be much more commonly used

A Performance Problem with `select()` and `poll()`

They work well enough for a small number of fds, but not for (very) large

- A typical `select()` or `poll()` application makes this call over and over again, looping, checking for new status on any of the same fds it is interested in
- But the kernel state to block the process (and know when to unblock the process) for each of those calls must be set up “from scratch” on each call
 - Must add the process to a separate kernel list for each of those fds
- And that kernel state is torn down on the completion of each of those calls
 - Must remove the process from each of those separate lists in the kernel
- No state related to a series of `select()` or `poll()` calls can be retained in the kernel (and thus reused) for each of these calls

epoll: More Efficient I/O Multiplexing (Linux Only)

epoll operates through 3 separate kernel calls

- `epoll_create()`
 - Creates a new epoll instance and returns a file descriptor for it
- `epoll_ctl()` (“control”)
 - Used to register interest in some fd for a given epoll instance
 - Can add, remove, or modify settings for an fd within that instance
- `epoll_wait()`
 - Waits for I/O events, blocking the calling process until then
 - The epoll instance state is retained and reused between `epoll_wait()` calls
 - And changes in fd status are tracked in the instance, ready for the next call

FreeBSD and Solaris have similar (also non-standard) facilities

Non-blocking I/O for a File Descriptor

read() and write() normally block the calling process

- I/O multiplexing can tell you which fds are “ready” for reading or writing
 - Meaning it is now possible to do a read (or write) on it without blocking
- But is that enough to ensure your process never blocks on I/O, even if multiplexing told you the fd is “ready” for that I/O?
- Usually yes, but not always, such as
 - After an indication of ready for a write, a *large* write may still block
 - After an indication of ready for a read, if you use, e.g., `rio_readlineb()`, it may do an *additional* read(), if it hasn’t reached the newline yet
 - An indication of ready for read only means ready *then*, but those available characters might get consumed by *another* process before your read
- ***For a solid, robust server, you want to be sure blocking never occurs***

Turning Non-blocking Mode On/Off on an fd

```
int fcntl(int fd, int cmd, ... /* arg */ );
```

A generic interface: do operation indicated by “cmd” on the file descriptor “fd”

- Two fcntl commands are of interest here
 - `F_GETFL` – gets (and returns) the current flags associated with fd
 - `F_SETFL` – sets the flags for the file descriptor fd to the value arg
- To enable non-blocking mode on file descriptor fd

```
fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_NONBLOCK );
```
- Or, if you happen to want to turn it off for file descriptor fd

```
fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) & ~O_NONBLOCK );
```
- Can also set `O_NONBLOCK` in the “flags” argument on `open()`

The Effect of Non-blocking Mode on I/O

Any I/O call on a non-blocking file descriptor will never block

- Any I/O call will still complete normally, if it can
- But if it would instead have to block
 - read()/write() will return -1, with errno = EAGAIN or EWOULDBLOCK (different systems use one of these two errno codes)

Saves your application (e.g., server) from blocking, but then what?

- If the I/O on that fd would have blocked, the return from read()/write() is now instead immediate, with errno set
- But how do you know when to try reading/writing that fd again next
 - You shouldn't just keep retrying immediately in a loop
 - Periodic retries with delays is still wasteful and may wait longer than needed

Using Multiplexing and Non-Blocking I/O Together

Can use together to handle I/O more efficiently

After multiplexing tells you the fd is ready for reading (or writing)

- Then **loop** doing reads for **all** available data (or to write **everything** you want)
- And the loop stops harmlessly on -1 and EAGAIN/EWOULDBLOCK if needed
- Then you wait for multiplexing to tell you again to read (or write)

Handles I/O with fewer kernel calls, since you might get to do multiple reads or writes without doing the I/O multiplexing call before each individual I/O

- Instead of multiplex, read; multiplex, read; multiplex, read; multiplex, read; ...
- Do multiplex, read, read, read, ...; multiplex, read, read, ..., multiplex, ...
(getting **all** pending data after each multiplex instead of just next **single** chunk)