ECE 252: Systems Programming and Concurrency Fall 2020

Lecture 26 — Asynchronous I/O with select, poll

Jeff Zarnett *2020-11-25*

Asynchronous (non-blocking) I/O

Consider some of the usual file read code:

```
int fd = open( "example.txt", O_RDONLY );
int bytes_read = read( fd, buffer, num_bytes );
close( fd );
```

As we discussed much earlier, the `read` call is blocking, as expected. So, your program waits for the I/O operation to be complete before continuing on to the next statements (whatever they are). This is sometimes, but not always, sensible. If you need the data in the next statement, you can't go on until the data is present.

If you are waiting for the bus, do you stare off blankly into space while waiting for it to arrive? Probably not. More likely you pull out your phone and start to use it for something. Whether that is productive or not (e.g., answering a project e-mail or liking posts on Facebook) is up to you, but you are doing something and making use of the time.

Our main solution until now is threads: if one thread gets blocked on the I/O the other ones can continue and is fine. But maybe you don't want to use threads, or maybe you can't due to: race conditions, thread stack size overhead, or limitations on the maximum number of threads. The last one might seem ridiculous, but in some embedded system you may not have the option to make new threads (or at least not as many as you want).

Sometimes, also, your programming language (e.g., Javascript) doesn't allow you to make multiple threads and you really have no choice but to use asynchronous I/O. It's a useful tool to have in the toolbox so let's get into it.

The simplest example:

```
int fd = open( "example.txt", O_RDONLY | O_NONBLOCK );
int bytes_read = read( fd, buffer, num_bytes ); /* Returns instantly! */
close( fd );
```

If we opened the file in non-blocking, the `read` call returns instantly. Whether or not results are ready. Unfortunately, this doesn't work here. The `O_NONBLOCK` option is not helpful, because this call says we should not wait for data when there is no data available. But a file *always* has data available. It might take a long time to load it up from disk, but the data is there. Do we know any scenarios where we don't have data always available?

Sure! Sockets. If we haven't received something, we would get blocked waiting for some data to arrive. But we can change that behaviour on a socket if we wish, by setting the socket to be nonblocking [Hal15]:

```
sockfd = socket( PF_INET, SOCK_STREAM, 0 );
fcntl( sockfd, F_SETFL, O_NONBLOCK );
```

This means that calls to `accept()`, `recv()`, or `recvfrom()` would not block. If you call those and there's no data to receive, you get back a return value of `-1` and `errno` is going to be either `EAGAIN` or `EWOULDBLOCK`. Sadly, the specification does not say which it would be, so the fully correct approach is to check for both. Not great, but it's how we are sure.

Suppose that you are writing a server application that's going to listen on several sockets. This is a common enough scenario. You could have different threads listening on their individual sockets but – see the reasoning above as to why we might not have that option. And we no longer have to!

But if we are a server and there aren't any incoming requests, what exactly are we supposed to do with our time? If we just poll each socket using, for example, `accept()`, this amounts to tight polling and is CPU intensive and wastes the CPU's time. But we don't want to get blocked on a particular socket either, because what if it's not the one where the next packet arrives? What we need is a third option.

Third option: `select()`

Our wish is granted. The third option is called `select()` – it allows us to monitor a group of sockets, telling us about the state of each of them. A socket could be ready for a read, ready for a write (of small size – you can't write a huge chunk to a socket without getting blocked at some point), or whether an exception has occurred. So actually, select works on three sets of sockets:

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

If we call this function, we'll get blocked until something happens on one of the sockets so that it becomes "ready" – data is available to read, space is available to write, etc. or until we reach a timeout. While blocked, we could also get interrupted by something (e.g., a signal).

The first parameter, `nfds`, is supposed to be the value of the highest number file descriptor in any of the three sets plus 1. So. It has come to this. One of the problems with `select()` is that it is pretty... arcane. According to the documentation, the reason why it is this, is because `select()` will scan through all the file descriptors from 0 up to `nfds-1` to figure out if we care about them.

What's this about? Well, the `fd_set` structure can have up to 1024 file descriptors, and is actually implemented as a bitfield; by specifying the highest file descriptor that we are interested in, the kernel can stop looking once we reached the last one (and the `fd_set` structure doesn't have to be a linked list or array or anything large!) [SR13].

Well, we were going to have to figure out what the `fd_set` means anyway. It represents a set of file descriptors, just as the name says. There are then four functions for manipulating a set:

```
void FD_ZERO( fd_set *set ); /* Clear the set */
void FD_SET( int fd, fd_set *set ); /* Add fd to the set */
void FD_CLR( int fd, fd_set *set ); /* Remove fd from the set */
int  FD_ISSET( int fd, fd_set *set); /* Tests if fd is a part of the set */
```

When we create a new set, we should first initialize it with a `FD_ZERO` call. That could also be used to reset it, if desired, at some later point. Then we can add file descriptors that we want to have. To add one, use `FD_SET` with the file descriptor to add; to remove one that has been added, use `FD_CLR` with the file descriptor to remove.

But that last one, `FD_ISSET`, is a little different. It's not as if we would forget whether we put a file descriptor in the set. It's really for us to see what happens after `select()` is called – we can find out whether a given file descriptor is in a particular set or not.

The `readfds` are obviously sockets we are interested in reading from and `writefds` are accordingly those we are interested in writing to. But what about the `exceptfds` – this isn't Java, it's not like we're going to get a `SocketDoesNotFeelLikeDoingWorkRightNowException`. No, this is for sockets that are in an exceptional state, which usually means there is Out-Of-Band (OOB) data on a TCP socket. We didn't cover this earlier in network communication and we will also not be going into this subject now. But you can find out if a socket is in that state if you have a reason.

We don't have to use all three of `readfds`, `writefds`, and `exceptfds` in a call to `select()` if we do not need them all. If we have only read sockets, we put them all in the `readfds` set and can just give `NULL` or empty `fd_sets` in for the other parameters [Hal15].

Finally, there is a timeout parameter: we can specify a maximum amount of time we are willing to wait. If nothing happens before the timeout amount of time occurs, then `select()` returns. The format of this is a fairly simple structure `struct timeval`:

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

If both fields of the `struct timeval` are zero, then `select()` returns immediately; if the pointer to it is `NULL` then there is no timeout and we will wait as long as it takes for something – anything – to happen.

When `select()` returns, however that happened, some if not all of the parameters other than `nfds` got updated (argh). The file descriptors passed in are modified in-place to see if they changed status. And the `struct timeval` parameter may (but also may not) be updated to reflect how much time was left before the timeout. Because different systems may or may not change this value, it is not safe to re-use and should be overwritten if you plan to use that structure again.

After select has returned, then we check the file descriptors in our sets. All of them. Yes, really. The function returns when there is something to do – but we aren't told what socket is ready. So we would need to check each socket to see if, for example, it's ready for reading (if that's the plan). To do that we check `FD_ISSET` with the socket and the set to see if it's in the desired state.

Because the sets of file descriptors may have been modified, you probably need to rebuild them after each call, if you plan to use them again. If you were waiting, for example, three sockets, not all of them necessarily became ready at the same time. So if you want to go on waiting for those three, you will need to put them in a set again.

Let's imagine a brief example where we have a server that is going to listen on some different sockets for different services, imaginatively called service1, service2, and service3.

```
void listen_for_connections( int service1_sock, int service2_sock, int service3_sock ) {
  int nfds = 1 + (service1_sock > service2_sock
    ? service1_sock > service3_sock ? service1_sock : service3_sock
    : service2_sock > service3_sock ? service2_sock : service3_sock);

  fd_set s;
  struct timeval tv;
  printf( "Going to start listening for socket events.\n" );

  while( !quit ) {

    FD_ZERO( &s );
    FD_SET( service1_sock, &s );
    FD_SET( service2_sock, &s );
    FD_SET( service3_sock, &s );

    tv.tv_sec = 30;
    tv.tv_usec = 0;

    int res = select( nfds, &s, NULL, NULL, &tv );
    if ( res == -1 ) { /* An error occurred */
      printf( "An error occurred in select(): %s.\n", strerror( errno ) );
      quit = 1;
    } else if ( res == 0 ) { /* 0 sockets had events occur */
      printf( "Still waiting; nothing occurred recently.\n" );
    } else { /* Things happened */
      if ( FD_ISSET( service1_sock, &s ) {
        service1_activate( );
      }
      if ( FD_ISSET( service2_sock, &s ) {
        service2_activate( );
      }
      if ( FD_ISSET( service3_sock, &s ) {
        service3_activate( );
      }
  }
}
```

You'll notice that we check each of the sockets individually – more than one of them could become ready at once. Now, the `activate()` calls there could take some nontrivial time, and if sockets receive data in the meantime then on the next iteration of the loop then `select()` will return immediately because data is waiting.

This represents a fairly simple scenario, though, where we are just waiting for `accept()` on the three services where we send back as simple response and that's the end of it. But if the connections we were looking at are supposed to stay open for some period of time, such as when there is occasional communication, then we would also have to add and remove sockets from the sets as connections were opened and closed. Managing the set of sockets in such a scenario is non-trivial, because each call to `select` can (and usually does) change the sets.

Chat Server. The more complicated example in [Hal15] is about a chat program (if you ever used IRC you have a pretty good idea of how this would work). When people want to join the chat room they send a message to a server. The server accepts and opens the connection and adds the client to the chat room. But people don't always talk. So if nothing is happening right now there's nothing to send. So the server can use `select()` to keep an eye open for when someone has said something. Either way, we would wait for something to happen, either on one of the sockets from a current client, or the socket for accepting incoming connections.

If it's the socket for accepting incoming connections that activated then accept the incoming connection, and add the new socket to the list that we are going to listen to. A chat server might choose to send a notification to other clients to let them know that a new person has joined the chat. That would be a write to the sockets that represent connected clients.

If another socket activated, most likely someone had something to say. So we can read from the socket that is ready to read and pass on the message that we received to all the other clients. Except, sometimes there isn't one! A socket will show up as being ready for reading if it has closed. But the call to read will return 0 (or a negative number) if the socket has closed (i.e., the client has disconnected). If the connection has been closed, then we should remove that socket from the list that we are interested in. And also we sometimes send a message telling other clients that a person has left the chat.

And when a message is received from a client, then of course the thing for the server to do is to send it on to the other clients, by writing to their sockets so the message is transmitted to them. The clients are waiting to receive messages and will then show them to the user when someone says something. And then look at that: you're talking to other people on the internet! Just remember not to give out your personal information to strangers.

Obviously there are some other considerations that come with managing a chat server. There might be periodic connection-keep-alive kind of messages so that clients remain connected even if nobody is talking right now. And you might want the ability to kick out people who are spammers and perhaps even ban them, amongst other things. But the goal of this explanation wasn't really to learn how to write a chat server in great detail – just to give an example of how you could use `select` to write something useful.

Slightly different, mostly the same: `pselect`. There is also `pselect()` which has the signature:

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

The signature is different in two ways. The first is that instead of `struct timeval` it is a `struct timespec`, which is slightly different in its meaning, and also will never be modified by the call to `pselect`:

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

Yes, the only real difference is that it is seconds and nanoseconds, rather than seconds and microseconds. The other parameter is about setting up a signal mask. We discussed the signal mask much earlier on when we talked about signals for interprocess communication. This is the same, but it allows us to change the signal mask atomically in

the same step as the call to select, so we can choose what signals are allowed to wake us up while we are waiting for something to happen. Given appropriate allocation and initialization of the values, the call:

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

is equivalent to an atomic operation that does all of:

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

If we had tried to do the multi-step sequence then there is always the possibility that something (e.g., a signal!) could happen in between changing the signal mask and the call to `select`. The syntax here for changing the signal mask is slightly different than what we have seen before, but this is the POSIX specification implementation of the `sigprocmask()` call. So, it works the same as we have seen earlier.

I only wanted a nap... It is possible to "misuse" a call to `select` or `pselect` as a way to make a very portable call to put the current thread to sleep. Some of the calls to sleep functions don't translate well to other UNIX-like systems, especially if you want a very short sleep – but `select` is in the standard! So you can do this by calling `select` with all three sets empty and `nfds` at zero and whatever timeout you wish. Clever!

Alternative: `poll()`

There's a slightly different function that is very much like `select()` and it is called `poll()`. Like its cousin, it is used to watch file descriptors and see if any of them are ready for an I/O operation. The signature is:

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

The signature is a lot simpler, that's for sure. The first argument is an array of `struct pollfd`, which is our structure for passing in the sockets we wish to monitor. The second parameter is `ndfs` which is just the number of items in the array. Finally, `timeout` is the number of milliseconds to wait before returning with zero meaning don't wait at all, and a negative number meaning wait infinitely.

Alright, let's look at the structure:

```
struct pollfd {
  int fd;        /* file descriptor */
  short events;  /* requested events */
  short revents; /* returned events */
};
```

Someone found a use for a `short` after all! Just kidding, hardware people.

The first parameter is obviously the file descriptor to watch. The second is where we specify what events we want to wait for on this file descriptor, and the third parameter is set by the kernel so we can find out what happened.

What can we add? You can look for data that's ready to read with `POLLIN`, a socket where you can write without blocking with `POLLOUT`, and an exception (as above) with `POLLPRI`. These can be combined with the bitwise OR operator if you want more than one.

The return value will be constructed as a bitwise OR of the fields as well, with the possibility that you could have one of these other return values: `POLLERR` if an error occurred, `POLLHUP` if the other side ended the connection, or `POLLNVAL` if there's a problem with the socket (e.g., uninitialized) [Hal15].

As with `select`, when `poll` returns we have to check which file descriptors have had an event occur and then decide what to do with them. Let's rewrite the `select` example with three services to see its `poll` equivalent:

```
void listen_for_connections( int service1_sock, int service2_sock, int service3_sock ) {
  struct pollfd pollfds[3];
  pollfds[0].fd = service1_sock;
  pollfds[0].events = POLLIN;
  pollfds[1].fd = service2_sock;
  pollfds[1].events = POLLIN;
  pollfds[2].fd = service3_sock;
  pollfds[2].events = POLLIN;

  int timeout = 30 * 1000; /* 30 seconds in ms */
  printf( "Going to start listening for socket events.\n" );

  while( !quit ) {

    int res = poll( &pollfds, 3, timeout );
    if ( res == -1 ) { /* An error occurred */
      printf( "An error occurred in select(): %s.\n", strerror( errno ) );
      quit = 1;
    } else if ( res == 0 ) { /* 0 sockets had events occur */
      printf( "Still waiting; nothing occurred recently.\n" );
    } else { /* Things happened */
      if ( pollfds[0].revents & POLLIN ) {
        service1_activate( );
      }
      if ( pollfds[1].revents & POLLIN ) {
        service2_activate( );
      }
      if ( pollfds[2].revents & POLLIN ) {
        service3_activate( );
      }
  }
}
```

Comparing the two. Ultimately both functions do the same job; it is only the specifics of the API call that make the difference. You might find one or the other to be better, and in particular it is nice that `poll` doesn't ask you to find the maximum file descriptor or manage three sets or have to rebuild the sets on each iteration of the loop.

Both `select` and `poll` are slow, unfortunately. They have linear characteristics: the more file descriptors you give them the slower they get; if you get up to a hundred file descriptors, roughly, then you end up actually spending significant CPU time trying to figure out which of the sockets changed [Ste17].

If we want to do things more efficiently we will want to use a different tool, specifically `libevent`, but we aren't quite done with network communication. After sockets, we learned about cURL, and it turns out we can use nonblocking I/O there as well!

References

[Hal15] Brian "Beej Jorgensen" Hall. Beej's guide to network programming: Using internet sockets, version 3.0.21, 2015. Online: accessed 5-July-2018. URL: `https://beej.us/guide/bgnet/html/single/bgnet.html`.

[SR13] W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment, Third Edition*. Addison-Wesley, 2013.

[Ste17] Daniel Stenberg. poll vs select vs event-based, 2017. Online; accessed 13-November-2018. URL: `https://daniel.haxx.se/docs/poll-vs-select.html`.