ECE 252: Systems Programming and Concurrency Spring 2019

Lecture 29 — Asynchronous I/O

Jeff Zarnett *2018-11-01*

Asynchronous (non-blocking) I/O

Cnsider 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.

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. 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.

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);
```

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.