Lecture 26 — Asynchronous I/O with select, poll

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 3, 2020

Read Me A Book

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 it's there.

Waiting for the Bus

If you are waiting for the bus, do you stare off blankly into space while waiting for it to arrive?

Haha, As If

More likely you pull out your phone and start to use it for something.

We Have Threads!

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 for a reason?

Some languages give you no choice, like JavaScript!

Jump To 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 );
```

That was easy, right?

There's a Catch

If we opened the file in non-blocking, the `read` call returns instantly.

Whether or not results are ready.

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.

Do we know any scenarios where we don't have data always available?

If we haven't received data, we'd 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:

```
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.

Server Setup

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!

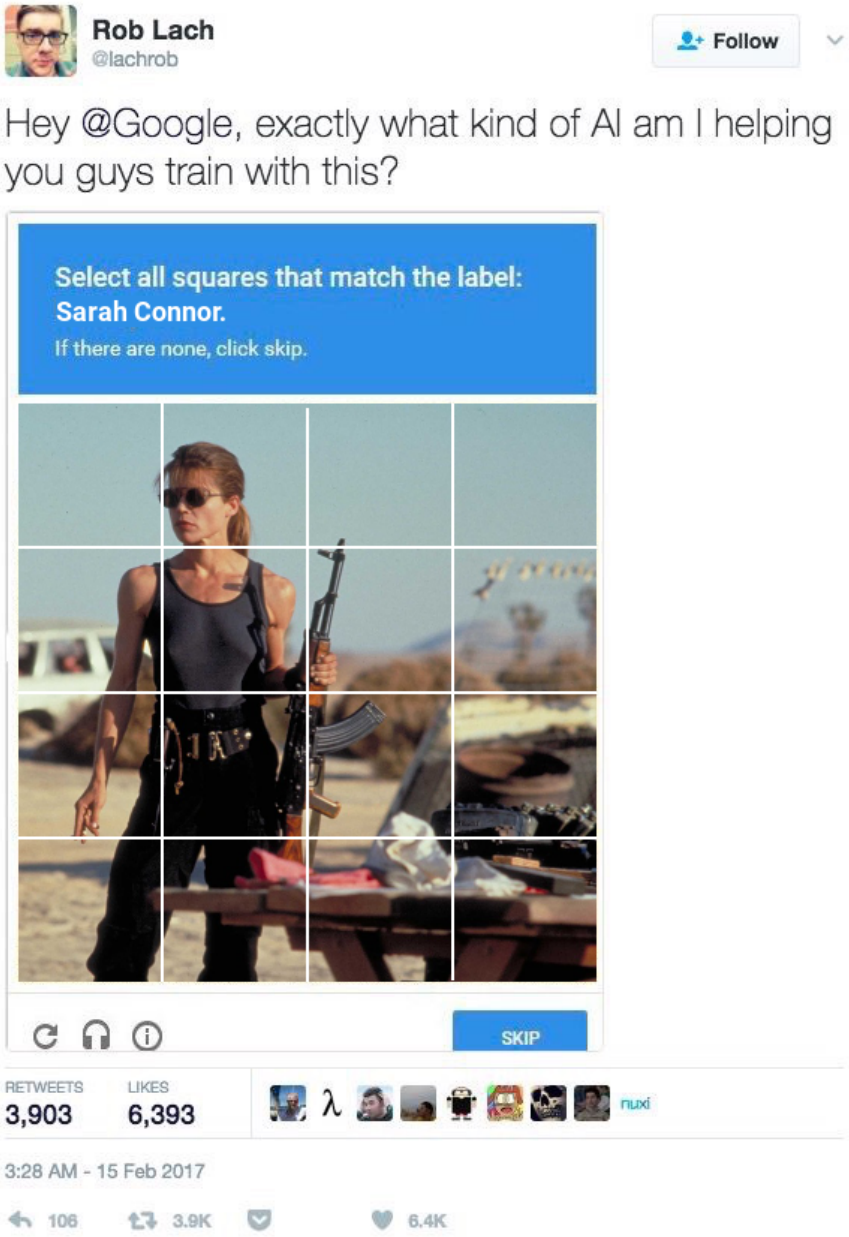
Wasting Our Time?

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.

What we need is a third option.

Select...


Rob Lach
@lachrob
Follow
Hey @Google, exactly what kind of AI am I helping you guys train with this?
Select all squares that match the label:
Sarah Connor.
If there are none, click skip.
SKIP
RETWEETS
3,903
LIKES
6,393
3:28 AM - 15 Feb 2017
106
3.9K
6.4K

Select!

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), or whether an exception has occurred.

So actually, select works on three sets of sockets…

Select Definition

```
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"... or until we reach a timeout.

While blocked, we could also get interrupted by something (e.g., a signal).

`nfds`: the value of the highest number file descriptor plus 1.

`timeout`: pretty self-explanatory.

What about the `fd_set` parameters?

File Descriptor Sets

The `fd_set` structure can have up to 1024 file descriptors, and is actually implemented as a bitfield.

The kernel can stop looking once we reached the last one.

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.

File Descriptor Sets

```
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.

To add one, use `FD_SET` with the file descriptor to add.

To remove one, use `FD_CLR` with the file descriptor to remove.

`FD_ISSET`: 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.

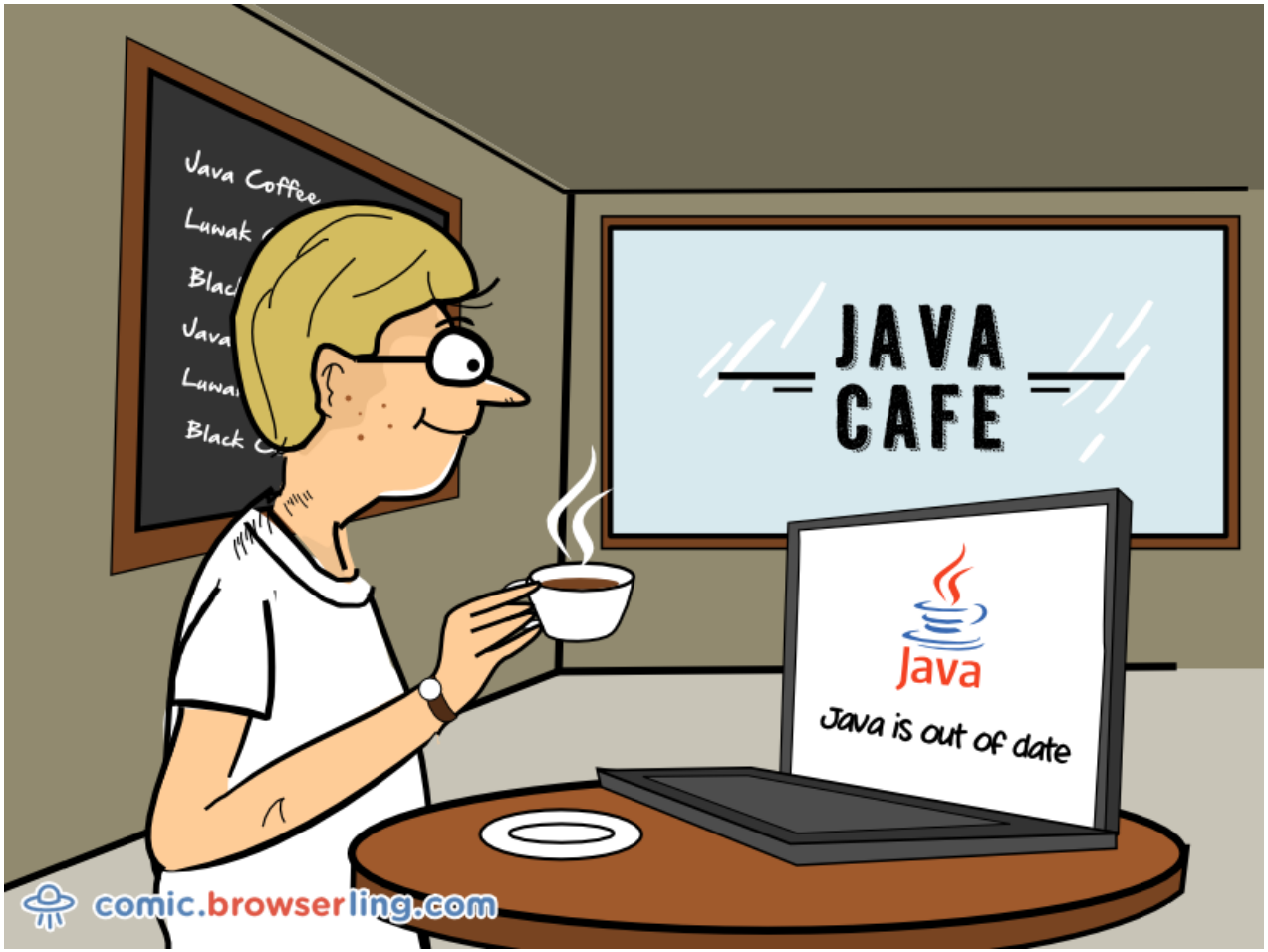
The Trio of Sets

The `readfds` are obviously sockets we are interested in reading from.

`writefds` are accordingly those we are interested in writing from.

But what about the `exceptfds` – this isn't Java, it's not like we're going to get a `SocketDoesNotFeelLikeDoingWorkRightNowException`.

Is My Joke Out of Date?



Exceptional States

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.

Optionally

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.

We can give in NULL for any we don't care about.

The Timeout

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 0, `select()` returns immediately.

If the pointer is NULL then we'll wait indefinitely.

Change Is Painful

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.

Check the Descriptors

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.

Rebuild It Again

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.

Let's imagine a brief example where we have a server that is going to listen on some different sockets for different services.

Socket Server

```
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( );
      }
  }
}
```

Silly Example

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.

If sockets receive data in the meantime then on the next iteration of the loop then `select()` will return immediately because data is waiting.

What if connections are supposed to stay open for a period of time?

Chat Server

Let's Chat

When people want to join the chat room they send a message to a server.

The server accepts, opens the connection, & 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.

New Person Here!

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.

Someone Said Something

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!

Well, Goodbye Then...

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.

Someone Really Did Say Something

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.

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.

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 );
```

`timespec` is like a timeval, but `pselect` will never alter it.

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

So, seconds and nanoseconds instead of seconds and milliseconds.

And a signal mask!

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.

A Signal Mask

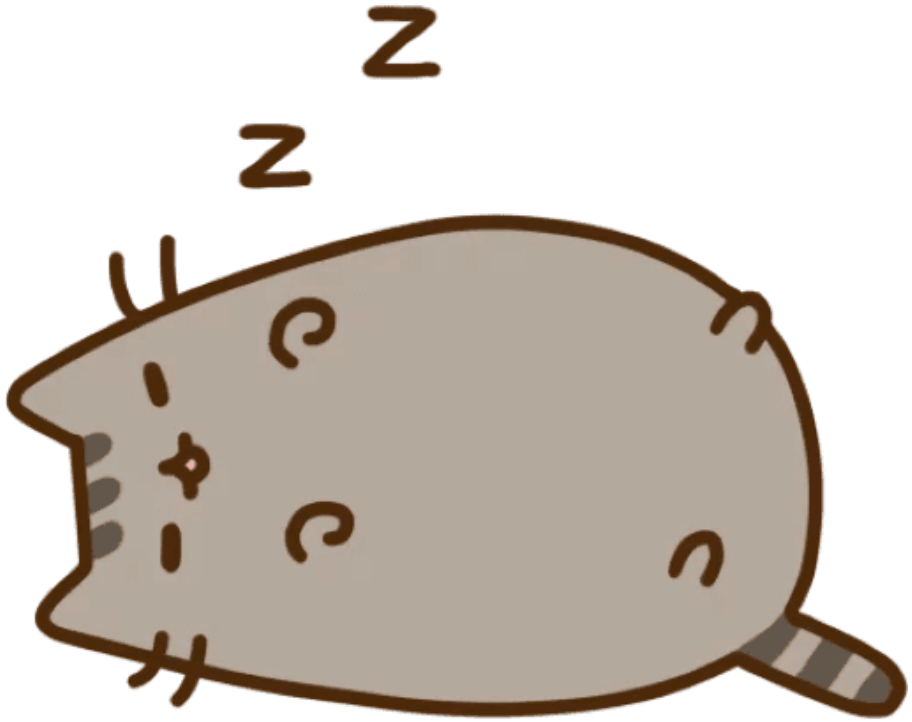
```
ready = pselect( nfds, &readfds, &writefds, &exceptfds, timeout, &sigmask );
```

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.

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.

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.

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

`fds`: pointer to array of `struct pollfd`.

`ndfs`: number of items in the array.

`timeout`: milliseconds to wait.

Poll File Descriptor Type

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

`fd`: file descriptor to watch.

`events`: the events we want to wait for.

`revents`: set by the kernel so we can find out what happened.

What Can We Watch For?

You can look for data that's ready to read with `POLLIN`.

A socket where you can write without blocking with `POLLOUT`.

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.

Return values can also include: `POLLERR`, `POLLHUP`, `POLLNVAL`.

As before, we are told something happened, but we don't know what.

3-Service Example with poll

```
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" );
```

3-Service Example with poll

```
  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( );
      }
  }
}
```

Which is Which?

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.

Both `select` and `poll` are slow, unfortunately. They have linear characteristics.

If we want to do things more efficiently we will want to use a different tool, specifically `libevent`.

After sockets, we learned about cURL, and it turns out we can use nonblocking I/O there as well!