Lecture 19-20: Asynchronous IO

Synchronous vs Asynchronous IO

By now you should be quite familiar with examples of synchronous and asynchronous things. For example, one big challenge with threads is that they are asynchronous relative to one another and thus data exchange between threads is challenging.

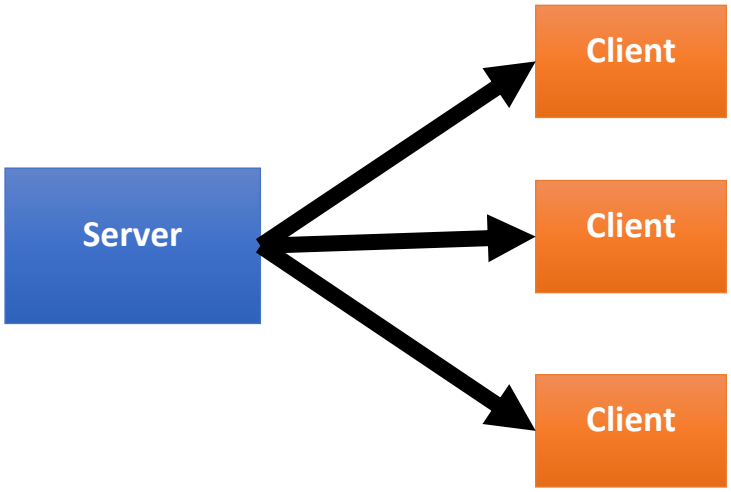
What about Input and Output (IO)? The kind of input and output you are probably most familiar with is Standard Input (stdin) and Standard Output (stdout). These are both examples of synchronous IO. How can one tell? Well, when you call a read from stdin (either through scanf, cin (in c++) or a flat read) the whole program comes to a grinding halt. If you don't give the process an EOF character, the process will hang forever waiting for that EOF character. Why? Well, that exchange is what causes the function to exit which provides the 'synchronization' for the rest of the program. This is a rather boring kind of synchronization because it is essentially… "do the next instruction now". Likewise, if for some reason you were printing to stdout and there was an issue with the terminal where the characters couldn't get out (this is entirely less likely), then the whole program would likewise hang indefinitely until it had printed those characters.

Now, examples of asynchronous IO should be far more obvious to you. If you think outside the programs that you have written and think about programs other people have written (be it Microsoft Word, Firefox, Chrome, or any computer game), it is rare that the program stops working simply because you either want to type something in. Imagine the following example. You are using a web-browser to watch a video (say one of my glorious lectures). Then because you need to do something else you click in the search bar to type in a new address. It is not like the video pauses. If this IO was synchronous you could imagine the video pausing until you clicked on it again. Another obvious example is a computer game. It is not like the 'baddies' in any computer game stop moving when you press a button, nor do they slow down for you to enter what you want to do (please take one step to the left).

The idea here is that the program is happy to accept input at any time without interrupting its other processes (i.e. displaying stuff).

Server Example

Let us consider a more complex example:



Here we have a server, which is running its own process. There are several clients attached to this server. The clients will intermittently send information to the server which then processes that information.

Now let us assume that we are trying to handle this in an asynchronous manner. There are several questions we need to address to get this kind of system working.

What happens if the server is 'mid-analysis' when a request or multiple requests are received?

What if I want to send data to client whose buffer is full? I do not want to wait until it is ready to receive data before sending data to another client that is ready to receive.

At the very least, I need a list or a buffer or some kind of structure which will record each of the events so that I can deal with them when I have the time to do so.

Level Tracking vs Edge Tracking

If you are trying to keep track of things, there are two effective ways of doing so. The first is to keep a list of everything. This is what is known as **Level Tracking**. Each thing in existence is in a list and its state is stored in that list. Now there are some fairly obvious downsides to this. The first is that the list is going to be big, and you need to know in advance what can be in the list. Secondly if you want to use the list, you need to search through it to ask about all the elements in it to check which ones have a useful state.

Imagine for example you were organising a concert. Theoretically everyone on the planet **could** buy tickets to your concert. So, you construct an array with ten billion elements where each element has the name of a person and a **boolean 'ticket'** which is set to 1 if they have bought a ticket and 0 otherwise. Now you need to send an email to all the ticket holders, so you search through the list to find the 10 ones. While this example is patently absurd, it gives a concrete example of how **Level Tracking** can be needlessly expensive. There are good examples of level tracking (signals from the last lecture being a prime example).

The alternative is **Edge Tracking**. This is where instead of tracking the levels of everything, you only ever track the changes. This requires a more dynamic system but makes much more sense when the list of elements in that list is quite small but drawn from a much larger list. Writing a webserver is a great example where **Edge Tracking** makes more sense. You could imagine that **Google** or **Facebook**

have millions of users, but only a small fraction are logged in at any time. It is much cheaper to simply track whenever someone joins or leaves rather than constantly flagging who has joined and left.

Interestingly, the human brain is more akin to **Edge Tracking**. Did you know that most of the time, your brain only tracks changes and completely adapts away any of the underlying signal? You might have noticed this with sound where a continuous tone seems to disappear after while (because it isn't changing). There are also numerous optical illusions that rely on adaptation. "Stare at the dot until such and such disappears". This is a useful insight because it demonstrates that **Edge Tracking**, despite being more complex to implement is going to be the more efficient system, particularly at scale.

Solution #1: Threads

Well, that is the end of this topic. It is pretty obvious that you could have a multi-threaded program, one for doing stuff, and the second for taking input. All the synchronization problems could be dealt with using mutexes, waits and signals. Problem solved.

OK, so why might we not use threads? Threads do create a bit of overhead. They require additional memory and switching between threads has a non-zero cost. Not all processers can do multi-threading. This is especially relevant with embedded applications which might be running on tiny bits of hardware but require very rapid IO capability. Finally, debugging a large multi-threaded program can be… "fun" (intense sarcasm).

This is not an indictment on using threads, just a "they aren't the only solution" argument.

SIGIO (which we will cover shortly) does not create threads.

Solution #2: Signals

The main advantage of using signals is that they are properly asynchronous. Programs can communicate via signals though there are limits on what can be communicated. Signals can be used to detect quite a few things as we have seen (Floating Point Errors, Segfaults). What if they could be used to detect IO?

So how would this paradigm work?

Let us first consider how normal IO works.

Normal IO

It waits for the user to type. It displays whatever the user types (such a lie). Does it really display everything you type? You may have used a program which does type everything. Whenever you hit up or backspace, it will print out something like this: "[[V2@" or some-such. This is because terminals have all these special hack rules like: "arrows moves the cursor", "up means get the last instruction", "tab means auto-complete", "backspace means remove a character in front of the cursor", "ctrl-C ends the program". The typed message will only be sent when a 'return' character is typed.

So here we have a few issues to tidy up. Number one, the program hangs indefinitely until it receives input. The program interferes with the characters as they come in and the program waits for that return signal.

SIGIO

The SIGIO signal is sent when a file-descriptor is ready to perform input or output. It is disabled by default but once active can be used to detect all kinds of things. Unfortunately, just enabling the signal

is not even close to sufficient. We also need to deal with the automated behaviour of the terminal. To get any of this done we first need to understand fcntl.

fcntl – Control

To get fcntl you first need to include it:

```
#include <fcntrl>
```

After which you can use it do stuff via:

```
int fcntl(int fd, int cmd, …);
```

Here 'fd' refers to a file descriptor and 'cmd' refers to a thing you want to do with said file descriptor. Are you lost? Did these notes just take a strange turn into some random function without explaining what the point is? Mysterious. Don't worry, it will make sense in moment.

There is a nice **cmd** available in fcntl, one called **F_SETFL** (which stands for set flag). The flag in question we want to set is one called **O_ASYNC** (aha, this is beginning to make sense I think). The point of this aside is that in order to allow a file descriptor to send SIGIO signals, it must first be set to **O_ASYNC**.

Actually, it is a bit more complex than that syntax-wise (see below):

```
fcntl(fd, F_SETOWN, getpid());
int fd_flags = fcntl(0, F_GETFL);
fcntl(fd, F_SETFL, (fd_flags | O_ASYNC);
```

So, the basic idea here is that first you set the owner, then you get the existing flags then you set the flag you want using the (old_flags | new_flag) syntax. This syntax relies on the idea that all the flags are bits in a long integer and you set each bit as the bit-wise OR of the old flags (i.e. keep all the old flags) plus the new flag.

O_ASYNC will actually be something like: 00000000000000000001000000000000 (this is not literally true, just a 'theoretical example of how flags work').

So, why do I have to 'set the owner'? Not only do you have to setup O_ASYNC, you also need to link the file descriptor to the process destined to receive SIGIO signals from changes in the file-descriptor.

But we are not done. One thing remains… Vader… you must confront Vader. I mean… terminal behaviour… you must confront terminal behaviour, then, only then SIGIO, will you have.

STTY

So, you can use 'stty' via the system command to play games with the terminal. The most important part being to remove the default 'echo' behaviour (i.e. it displaying what you type) and the second to remove any preprocessing it does. In linux, this is referred to as 'raw' (unprocessed) and 'cooked' (processed) user input so you need to set the following command:

```
system('stty raw -echo');
```

Followed by

```
system('stty cooked +echo');
```

to re-enable the default behaviour (otherwise your terminal becomes somewhat useless?). The choice of when and how to change raw and cooked input is interesting. It could be done:

- On an adhoc basis
- Manually prior to the program being called (and then correspondingly afterwards)
- Using 'system' as above
- Using C functions

POSIX – AKA Let's do it all again using AIO rather than SIGIO

So, the example above works. However, just like signals, there is a more 'robust' (though actually incomplete) version of a SIGIO-like entity which does the same thing with more options.

Compiler Flags

Let's begin by adding -lrt (to link the rt library).

The Process

Is basically the same. We will redirect SIGIO and set STTY => RAW.

Giant Structs

Like before, these 'special' methods for doing things usually include giant structs:

```
struct aiocb
{
      int             aio_fildes      # file descriptor
      off_t           aio_offset      # file offset
      volatile void   *aio_buf        # pointer to a buffer : )
      size_t          aio_nbytes      # length of a transfer
      int             aio_reqprio     # request priority offset
      struct sigevent aio_sigevent    # a sigevent…
      int             aio_lio_opcode  # operation to be performed
}
```

There are a lot of options here, few of which I will describe but just know they are there.

Coding Time

So, we need to have our aiocb struct (here called my_buffer). We also need to have an input (we are reading one at a time. Because we are reading from stdin we choose .aio_fildes = 0. The aio_sigevent.sigev_signo is set to SIGIO (what we are leveraging here). The rest are essentially about as default values as we can use.

```
struct aiocb my_buffer;

setup ()
{
static char input[1];
my_buffer.aio_fildes        = 0;        // Stdin (i.e. fd = 0)
my_buffer.aio_buf           = input;
my_buffer.aio_nbytes        = 1;        // See input
my_buffer.aio_offset        = 0;        // Offset in file (header?)
my_buffer.aio_sigevent.sigev_notify    = SIGEV_SIGNAL;
my_buffer.aio_sigevent.sigev_signo     = SIGIO;
}
```

Once we have our aiocb buffer setup, we need to get it connected somehow. We do this via the aio_read function:

```
aio_read(&my_buffer);
```

Now all we need to do is to setup our signal for redirection:

```
signal(SIGIO, signal_handling_function);
```

Now we should be ready to roll… except, how do we actually deal with the buffer inside of the signal handler? Well firstly we have the option of handling errors via:

```
aio_error(&my_buffer) != 0
```

This can be used in a standard if statement like making pipes/threads etc.

Then we need to get the actual characters via:

```
aio_return(&my_buffer);
```

However, this is just reading from the buffer… not actually a reference to it. To do that you need to be able to get at my_buffer.aio_buf. This can be done using a simple ‘convenience pointer’:

```
char * character_pointer = (char *) my_buffer.aio_buf;
```

When aio_return is called, the character which was typed in stdin will now be available in the memory pointed to by **character_pointer**. The last tidbit is to make a new request at the end of the handler via:

```
aio_read(&my_buffer);
```

Now we can actually glue it all together into a cohesive program. Check the examples code to get your head around it.

STRACE

Also know as Signal-Trace. It is a wonderful linux tool for worrying about signals. It can be easily invoked as such:

```
strace ./my_exe
```

What it does, is intercepts and records the system calls which are called by a process and the signals which are received by a process (acha… so this is why this is in this lecture). To be honest, the best way to learn about what strace does is to actually use it. Go use it. Have a look at what it prints out. Some things might be familiar, others not so much. Remember you can type:

```
man <system_command>
```

To find out what each <system_command> does. There are a lot of them when you use strace on one of your programs.

Strace and Forks

To do forks, use a -f flag:

```
strace ./my_exe -f
```