Lectures 14-16: Threads

ps & kill

We have covered this before. This should be old hat by now, but to really understand threads, one must first get over the confusion about threads and processes. Namely, threads and processes, while similar are subtly different.

If you type:

```
ps -a
```

Into a terminal, you will get a list of all the processes running on a machine. It gives you information about the process ID, who is running it, when did it start and what it is called. Big list. Simple.

When you use 'kill' to end one of those processes, there are a number of things that can happen depending on which flags you use.

The most common is 'kill -kill' which is an uninterruptible form of process ending. In this case the operating system does not ask for permission from the process, it simply knocks it off the process chain. The other forms of 'kill', such as '-TERM', '-STOP', '-CONT' (the last two being used to pause and resume processes) send different signals to a process. Some will ask the process nicely "do you have a way of shutting down, if so, please do it". Many processes are good like this and it is somewhat analogous to pressing the 'X' button on a windowed application or File > Quit or some such.

Jobs

'jobs' is a fun little function and works well with two others 'fg' and 'bg' (standing for foreground/background). To make this work, you will also need to know Ctrl-Z which is a short-hand for kill -STOP and will pause a process. Also it helps if you know what adding '&' to the end of a command does.

Let us say that I have started running two copies of xeyes using 'xeyes &' so I can have both running as a background task. If I type 'jobs' it will tell me what programs are running from my terminal (i.e. two copies of xeyes). I can then use 'fg' to bring one of them ('fg 1' or 'fg 2' if I want to choose which one) to the 'foreground'. This will basically replace my 'xeyes &' call with 'xeyes'. Now I can no longer use the terminal other than signal shortcuts like 'Ctrl-C'. Two processes are running, one in the foreground, one in the background. To send a process to the background, I can use 'Ctrl-Z' to pause it. I can then type bg which will send the current process to the background and have two background processes again. When you call 'jobs' it will list the number necessary to perform this 'to-foreground' task so you can choose which process you want to interact with.

Threads

So, we are quite familiar with processes. What then, is a thread? Let us start by talking about some of the differences. First and foremost, threads share memory. What does this mean? It means that both 'threads' are running in the same memory context and have the same heap, data and code bit of memory to talk to. Two processes will have separate memory allocated to them and having processes interact can be a bit tricky. This also tells us that two threads likely have the same process ID (which they do).

Now, this might be a bit confusing. If we have 'fork' and can spawn processes, what is the point of a second thread? The answer lies in this sharing of memory. The simplest example is of a computer game. Often, games will split the tasks into two main threads. One will be the game engine (i.e. what the game does under the hood such as calculating gravity, A.I.s and other game related computation) and the other will be the display (put pixels on the screen). The reason for the split is because you wouldn't want minor issues with display to affect the speed the game runs at and vice versa. That way if the graphics card is having trouble, the frame-rate drops, but the game continues to run at the same speed. The reason you wouldn't use two processes to do this is because of the cost of communication between the game engine and the display. Where would the display get its data from? A file? Some kind of pipe? It seems like a lot of work when it could just directly access objects, structs and variables from a program.

The other conclusion is that if threads have shared memory, it means they are not managed by the operating system. They do not have separate process IDs, rather they only have one. However, many computers have the option of multi-threading which means they use different computational resources to run different threads. The difference is that even when the work is split, the computer still thinks of the two threads as the same process.

Stack

While threads share memory, each thread has its own stack. This makes intuitive sense. If they did not, how could they function at all? The stack is essentially the 'currently running' program. In multithreaded programs, the program keeps track of two separate stacks, each of which defines the execution of each thread.

Threads, How To

Includes and Compilation

```
#include <pthread.h>
```

Pretty self-explanatory, but we are not done. It turns out that 'c' does not automatically think in terms of threads and needs to told explicitly that threads are happening. Thus:

```
gcc -pthread -o "my_exe" file1.c file2.c
```

This flag tells the compiler that threads are happening. This is sensible given that having threads implies a second stack and the underlying memory configuration for a single-thread vs multi-thread program cannot be the same.

Making the Thread

A call to 'pthread_create':

```
pthread_create(tptr, attrptr, func_ptr, arg_ptr)
```

The parts:

tptr: How to refer to your new thread. Before calling pthread_create, you are first required to have a variable of type 'pthread'. This lets 'c' know which thread you want to talk to.

attrptr: (Worry about this later)

func_ptr: The function where the thread starts. This is a fairly obvious point. If a new thread involves a new stack, where does that stack begin. Instead of always defaulting to main or some-such, a new thread may pick where it begins.

arg_ptr: Parameters of the function mentioned above in 'func_ptr'

Legal Starting Points

So how does 'c' know what kind of function? Well, a 'thread starting function' must return a void pointer (void *). It must also have a single parameter… which is also a void pointer (void *). Which seems like it would work, but basically is just a stand in for 'whatever you want'.

The Problem with Threads

Hopefully, you have been diligent and gone off and programmed your first threaded program (or looked at an example). The best place to start is with simple toy programs which either print endlessly to the command line or increment variables. In most programs like this there will be very little problem with threads. It is only when threads start getting used for real applications that the inherent problem of threads comes to bear.

Shared Memory

As stated previously, threads share memory. This means one thread can, via a pointer have direct access to a variable that another thread also has access to. This leads to the problem of handover. Think of a relay race. There is a baton, there is one thread (the first runner) who carries the baton to the second runner. The second runner takes the baton from the first, and everything is wonderful. Except… that does not always happen. What if the first runner drops the baton too early or does not let it go properly.

The same problem exists in multi-threaded programs. In the game example, the display thread probably does not really care what it is displaying. It may appear as a glitch but if I read an empty buffer, the worst that can happen is a black pixel, or may just a black screen (that is pretty bad actually). But what if instead of just displaying a value, I wanted to use that value to perform some calculation. For example, what if that value was the acceleration reading from a gyroscope and I was using that value to update the position commands on an aircraft.

What could go wrong? First, perhaps the thread which writes the gyro reading is slightly delayed, which means the reading thread reads an out-of-date value. This could cause an aircraft to make bad calculations and potentially crash. Yet this isn't as bad as it gets when it comes to threads. Consider the following function:

```
void incrementX()
{
      x = x + 1;
}
```

Now, it might seem that little can go wrong with this function. However, if I had two threads repetitively calling this function… lets say Thread A calls it 100 times and Thread B calls it 50 times. Let us assume that 'x' is a global integer that is initialised to zero. So… after Thread A & B have finished… what is the value of 'x'? Now you might be thinking 'this is obviously a trick question, 150 seems too obvious'. You would be right. It can actually be less than 150. How might this happen.

To understand this we need to break down the line. The program accesses 'x', then increments it, creating a temporary value, which it then writes back into 'x'. While this may look like a single line of

code, it is actually quite a few different operations. This means, that when two threads are calling this function, the order of these 'sub-operations' is very important. Consider:

Scenario 1

A) Accesses 'x' (x is zero)

A) Increments 'x' (temporary value is x + 1)

A) Writes new 'x' (x + 1)

B) Accesses 'x' (x is still zero)

B) Increments 'x' (temporary value is x + 1)

B) Writes new 'x' (x + 1)

Scenario 2

A) Accesses 'x' (x is zero)

B) Accesses 'x' (x is still zero)

A) Increments 'x' (temporary value is x + 1)

B) Increments 'x' (temporary value is x + 1)

A) Writes new 'x' (x + 1)

B) Writes new 'x' (x + 1)

In scenario 2, x is now just x + 1, not x + 2 like we might expect. What this means is that the order in which the operation A writes and B reads can change the outcome. Because this depends on the whims of the operating system, we regard this as a 'Race Condition'.

Race Condition

Simply put, a race condition is when two independent 'tasks' which are interdependent produce different results depending on the timing of how those two tasks are implemented.

Another simple example is me putting down a cup and you picking it up. Now, if the order in which the tasks are performed affects the outcome (such as you trying to pick up a cup I haven't put down) then a race condition has occurred. This isn't a concept unique to threads or processes, but threads are a good place to explore it.

Mutexes as a Solution to Race Conditions

Let's begin with a description of what a Mutex is. Mutex, which stands for *Mutual Exclusive* is a 'resource' which is designed to be fundamentally singular in its existence. That is to say, there is only one and as such, only one person can have it at a time. The example I like to use is the conch shell form the novel 'Lord of the Flies'. In the novel, there is a group of boys who have a council, and the rule is, the person holding the 'conch shell' must be listened to. As only one such shell exists it is a way of preventing the boys interrupting each other mid-speech.

Mutexes function the same way. There is only one mutex. Only one thread at a time can have access to the mutex. Thus, a very simple design pattern becomes apparent. If there is a bit of shared memory,

only the thread which 'has the mutex' can mess with the memory. Other threads must wait until they have the mutex before they can do anything.

Now one obvious disadvantage of this technique immediately jumps to mind. Does this mean that threads need to wait for each other? Short answer… yes. This makes a lot of sense. If the other thread is updating some variables, you would want to wait until the changes have been made, just like you would wait for me to put something down before trying to pick it up.

However, having one mutex per bit of data is a bit restrictive. Thus, it is up to the programmer to define how the mutexes behave which firstly means it can be error-prone but also allows some flexibility. Perhaps you can change multiple pieces of data with a mutex. Perhaps you can simply use it as a flag to demonstrate that a piece of code has finished executing.

Mutex Example

A mutex is another special type in C defined as:

```
pthread_mutex_t my_mutex;
```

With two accompanying functions:

```
pthread_mutex_lock(&my_mutex);
pthread_mutex_unlock(&my_mutex);
```

The best way to think of these functions is locking means taking the mutex for yourself, preventing others from having it (i.e. locked). Unlock is what you do when you are finished, leaving it free for others to lock. There is no real cost to unlocking a mutex, it is locking where the burden is incurred. A thread that attempts to lock a mutex which is already locked will find itself waiting for the mutex to be unlocked. In a well-written program, this shouldn't be an issue, but having threads waiting for extended periods of time upon the execution of other threads seems to be a bit of a waste of time.

Producers and Consumers

While there are some small benefits to making a program multithreaded, when there is only really one logical flow within the program, it can sometimes be a bit of a waste of time. The game example only works because there is a tacit assumption that missing a few display frames is not a big deal and letting the graphics card work at its own pace is good. However, most systems are not quite as flexible as game framerates.

An exception to this is the idea of producers and consumers. Sometimes, making something is a lot more effort than using it, or vice versa. Consider how long it takes to make a boiled egg (5 minutes) and how long it takes to eat one (much less than 5 minutes). What if your program consisted of making things and then using them but the rates at which the production and consumption occurred was very different?

You might be inclined to use threads. Perhaps you could have ten threads which processed data but only one that used the processed data because that final step was very quick. There is no point having the consumer wait around for the producer when it could be rapidly doing its thing over and over again. While this sounds simple in theory, getting the timing right can be quite a challenge.

The Mailbox

A mailbox is an example of something where production is slowly and haphazard, but consumption can be quick and immediate. Obviously, you do not want to stand by the letter box all day, but perhaps you would like to read your mail as soon as you get it. If you used a polling approach (checking the

mailbox every so often) you'd be stuck between checking it every second, or only checking it over a longer period and perhaps delaying your response unnecessarily. Ideally, you would want a signal to tell you that mailbox is ready such as a bell or a flag.

In this scenario there are multiple producers (postmen, ad-people) but only one consumer (you).

The Dining Hall

What about the converse. Perhaps there is only one source (the cook) and many consumers (the … consumers). Also in this analogy, the consumers probably don't care which meal they eat. Again, the consumers don't want to be constantly checking whether their meal is ready. Instead, if the cook has a bell they can ring, that can inform the next consumer that there is a meal ready to be consumed.

Signals and Mutexes

How might these examples work? Well, if you want to access the memory you need the mutex. But how do you know if the memory has been written to? What if after someone reads from the shared memory they reset it to zero to signal that it has been read from and is ready for further writing. Do I lock the mutex, check to see if the data is there, realise it is not and then unlock the mutex again? Then what, do I lock it again, check the data again?

pthread_cond_wait

In C, the way to implement this 'signal-based' method of notifying threads is:

```
pthread_cond_wait(&cond, &lock)
```

The idea here is that the thread which is to read from shared memory needs to wait for the data to be populated. That data is associated with a mutex (&lock) and a 'pthread_cond_t' (&cond) which is a kind of condition. In order to call **pthread_cond_wait** the waiting thread (consumer) must first have the mutex. But wait? If the consumer locks the mutex, how will the producer deposit the data into shared memory? Doesn't it need the mutex first. Fortunately (i.e. by design), calling **pthread_cond_wait** unlocks the mutex. Now the producer is free to do its thing, lock the mutex, change some data. But what now?

```
pthread_cond_signal(&cond)
```

The producer can now use this command to 'wake up' the waiting thread. The waiting thread will then try to reacquire the mutex before doing its thing. The order of operations here is very important of course and you should spend a bit of time working through an example by hand. Let's look at the order of operation in the table below.

Producer	Consumer
Lock the mutex	-
Modify shared data	-
pthread_cond_signal !!	-
Unlock Mutex	-
Do some other things	Lock the mutex
	Check the shared data to see if present, Hooray it is there!!
	Process Data
	Unlock the mutex
Still doing other things	Loop
	Lock the mutex
	Check the shared data to see if present, BOO!! It is NOT there!!
	pthread_cond_wait !! unlocks the mutex
Still doing other things	WAITING!!
Finally, I am done	
Lock the mutex	
Modify shared data	
pthread_cond_signal !!	AWAKE! Try to get mutex… denied… I will wait for it to be unlocked
Unlock Mutex	
Do some other things	Lock the mutex
	Check the shared data to see if present, Hooray it is there!!
	Process Data
	Unlock the mutex

As you can see, the above looks at two circumstances, one where the data is there, one where it is not. Firstly, the data is produced and consumed easily. The second time the consumer has jumped the gun and tried to consume while nothing is there. To avoid calamity, it goes to sleep using wait. Then, the producer finally gets around to repopulating the data. You can see how **pthread_cond_wait** involving the unlocking of the mutex is mandatory for this to work. Otherwise, the producer would never be able to 'produce'. Once the producer calls **pthread_cond_signal** the consumer starts up again but this time is denied by the lack of the mutex (which is fine). When the mutex is unlocked it goes to work again.

Terminating Threads

One more consideration for multi-threaded programs is what happens when a thread ends? Traditionally, when the main program completes its final line and returns zero (or at least it should) the program has reached its conclusion. For a multi-threaded program with several execution stacks, how does the program know it has finished?

This is a non-trivial problem. If you wanted to know if you were the last survivor on earth, you kind of need to know that everyone else is dead? This does not just mean checking the alive status of all the people you have met… they could have had children, children you have never met. You would have to check if, firstly they had had children and secondly that their children were also dead.

So how do threads die? The first and simplest case is when they run out of code to execute. When you call **pthread_create** it links to a function where the thread starts. When that function finds its end, then the thread has nowhere left to go, and the final block of the stack is removed.

There is also the option to explicitly end a thread via **pthread_exit**. This is done implicitly in the above example but there are several advantages to an explicity "this thread is dead" call. Firstly, it can send an exit status which can be picked up and processed to tell another thread something about how well the thread fared (did it do what it was supposed to do or was there a problem).

There is also **pthread_cancel(<thread_id>)**. This is the opposite of **pthread_exit** (which is analogous to self-ending) where one thread (usually a main-thread) decides to abruptly end another thread in process. This makes sense in a producer/consumer paradigm. When the production finishes, the consumers ought be shutdown too, but perhaps the main program still has more to do. **pthread_cancel** gives the user the ability to stop threads that may have no reason to stop themselves.

The last and most problematic way a thread can die, is if the main thread dies. Once **exit()** is called (or an exec-family function) the process dies and thus the threads within it die too.

Joining Threads

One other thing you might want to do with a thread that has reached its end is tell the main program that it has happened. Hence the existence of **pthread_join(<thread_id>, void ** <retval>)**. This function will cause the main thread to wait until the specified thread has **pthread_exit**'ed. The return value of the other thread is given as a pointer which allows the main thread to determine an exit status (if one is given).

In generable, if a thread is joinable (i.e. it can be joined to main without affecting functionality) it is a good idea to explicitly call join. This is also true of **pthread_exit()**. Like using **return** 0 as the last line in a C/C++ program these little attentions to detail will force you to think more carefully about how your programs work (especially with other programs).

Thread Attributes

So, do you remember how **pthread_create** had a second argument called attributes? Well here we are, talking about attributes. Without going into to too much detail, there are several thread attributes that can be altered using this argument. For example it is possible to make a thread 'detached' (i.e. not joinable). What does this mean? Simply, that once the detached thread ends, there is nothing waiting on it and so it releases all its resources immediately instead of waiting for another thread.

Another attribute that can be set is how threads are managed such as the order in which they are executed (FIFO, round-robin etc). You can also specify other things like stack-size and now we are getting into… just go google it territory of detail.

At any rate, to use the attributes you need to specify a **pthread_attr_t** thing and call **pthread_attr_init()**. Then use some extra functions such as **pthread_attr_setdetachstate()** before finally calling **pthread_attr_destroy()**. All in all quite a bit of work, but handy in the right circumstance.

A Final Word on Threads

Well… that was a bit long. The main takeaways I suppose (other than the syntax) are that using threads can be useful, but it should not be your go-to to solve problems (like anything in programming). Any program which over-uses threads immediately becomes VERY complex and often almost impossible to debug. Race conditions are by far the worst kinds of bugs to fix so make sure you go 'softly, softly' when diving into multi-threaded programs. It is the one time in programming where you must spend time thinking through how the program works on paper prior to writing the code because trying to write a multi-threaded program 'on-the-fly' by looking at outputs won't actually work. Even adding in debugging can hide race-conditions because the debugger or debug comments might alter the program's execution time and thus 'hide the bug'. Beware.

As for mutexes. I would recommend spending some time on paper practicing how you would get the handover to work yourself. The reason is that while many 'purport to understand', reproducing an explanation of mutexes and signals is something I find most students trip up on when pressed.