COSC 2P95 – Lab 9 – Section 01/02 – Signals, Mutexes, and Threads

Sometimes, we can't achieve what we want via a single execution pipeline. Whether it's for responsiveness, or to process more data simultaneously, we rely on techniques that allow for *concurrency*.

The primary goal of this lab is to learn to use *threading* so we can make multiple concurrent attempts to solve the same problem (and thus exploit the additional computing resources afforded by multiprocessing devices).

Additionally, we'll also get some practice with a companion tool to facilitate that: the *mutex*; and additionally take a quick look at how execution can be suspended to deal with *signals*.

In fact, let's start there!

Phase 1 – Signals

To start, we're going to need something that can be interrupted. Let's create a basic dummy program that just wastes CPU cycles:

```
#include <iostream>
#include <unistd.h>

bool continuing;

//At this level, this header signature can be ignored
void* busywork(void* unnecessary) {
     while (continuing) {
          usleep(100000);
     }
}

int main() {
     continuing=true;
     busywork(NULL);
}
```

Before we get any further, let's have a quick chat? (You really can skip this part if you aren't interested)

- The `usleep` function is for sleeping for the provided number of *microseconds*. Depending on the platform, the level of actual precision you get from that may be inconsistent (potentially as bad as being measured in milliseconds). In this case, we're saying to wait 100ms, or 0.1s per cycle
  - If you want to sleep for entire seconds, just use `sleep`
- If you have the CPU usage being displayed (either by adding the applet to the top panel on your Linux screen, or by also running `top`), you might be tempted to think that that's why it's using such little CPU time: because it's sleeping for a tenth of a second at a time. But that's not it.
  - Temporarily, try changing it to `usleep(1);`
  - Didn't change much, did it?
  - On the other hand, try commenting it out entirely, recompiling, and rerunning
    - Geez, takes up a fool core! But… how?

This actually relates to how processes/threads are scheduled in a modern computer:

- Normally, when there's a lot to process, the kernel cycles between processes, allotting a *time slice* to each

- When you sleep, irrespective of how long it's for, you forfeit your current time slice
- If there are other things to schedule, this immediately yields to them
- Because of this, even if it's only for 1 μs, any *sleep* at all can drastically improve the responsiveness of a program (particularly a highly interactive one like a game)

Anyhoo, let's resume…

When you tried running this, presumably you pressed *ctrl+c* to cancel, right?
Since ctrl+c really just sends an INTR character for the SIGINT interrupt signal, we can formalize this:

```
#include <iostream>
#include <unistd.h>
#include <cstdlib>
#include <signal.h>

bool continuing;

//At this level, this header signature can be ignored
void* busywork(void* unnecessary) {
	while (continuing) {
		usleep(100000);
	}
}

//We don't really care what 'sig' is for this example
void interrupted(int sig) {
	std::cout<<"\nComputations complete.\nHalting now..."<<std::endl;
	continuing=false;
}

int main() {
	continuing=true;

	if (signal(SIGINT,interrupted)==SIG_ERR) {
		std::cout<<"Unable to change signal handler."<<std::endl;
		return 1;
	}

	busywork(NULL);
}
```

In this case, when you press ctrl+c, instead of automatically halting, it instead invokes the SIGINT handler, which in turn changes the boolean's value, which is what triggers the halting.

There's just one small tweak to make: The boolean should also be `volatile`, to ensure that we'll always be looking at its current value.
Let's tweak our example slightly:

```
#include <iostream>
#include <unistd.h>
#include <cstdlib>
#include <signal.h>

volatile bool continuing;

//At this level, this header signature can be ignored
void* busywork(void* unnecessary) {
	while (continuing) {
		usleep(100000);
```

```
	}
}

void peek(int sig) {
	std::cout<<"Currently processing: "<<(continuing?"Yes":"No")<<std::endl;
}

//We don't really care what 'sig' is for this example
void interrupted(int sig) {
	std::cout<<"\nComputations complete.\nHalting now..."<<std::endl;
	continuing=false;
}

int main() {
	continuing=true;

	std::cout<<"About to commence; PID: "<<getpid()<<std::endl;

	if (signal(SIGINT,interrupted)==SIG_ERR) {
		std::cout<<"Unable to change signal handler."<<std::endl;
		return 1;
	}

	if (signal(SIGUSR1,peek)==SIG_ERR) {
		std::cout<<"Unable to change signal handler."<<std::endl;
		return 1;
	}

	busywork(NULL);
}
```

Note that, upon starting, it immediately displays the program's own Process Identifier (PID). This can be used, in conjunction with `kill`, to send it signals (like SIGINT or SIGUSR1).
For example my first execution started with:
`About to comment; PID: 5464`

So, I *could* have stopped my program via `kill 5464`, but that wouldn't help us here.
`kill -s SIGUSR1 5464`
would trigger the program to report on its current status.

`kill -s SIGINT 5464`
would tell it to exit gracefully.

Of course, the second time it runs, it will no longer be 5464, so you need to pay attention to that.
Alternatively, since I was compiling my program to `s1`, I could type `pidof s1` into bash.

You could also try sending SIGUSR2, but we haven't defined a handler for that.

And that's it for the first phase!

Phase 2 – Threads and Mutual Exclusion

Next, we want to break the busywork off into a separate thread.
This will entail including another header, and creating a handle for our *pthread* (POSIX Thread).

Important: remember to add `-pthread` to your g++ line!

```
#include <iostream>
#include <unistd.h>
#include <cstdlib>
#include <signal.h>
#include <pthread.h>

volatile bool continuing;

//At this level, this header signature can be ignored
void* busywork(void* unnecessary) {
      while (continuing) {
            usleep(100000);
      }
}

void peek(int sig) {
      std::cout<<"Currently processing: "<<(continuing?"Yes":"No")<<std::endl;
}

//We don't really care what 'sig' is for this example
void interrupted(int sig) {
      std::cout<<"\nComputations complete.\nHalting now..."<<std::endl;
      continuing=false;
}

int main() {
      pthread_t ct;//our child thread

      continuing=true;

      std::cout<<"About to commence; PID: "<<getpid()<<std::endl;

      if (signal(SIGINT,interrupted)==SIG_ERR) {
            std::cout<<"Unable to change signal handler."<<std::endl;
            return 1;
      }

      if (signal(SIGUSR1,peek)==SIG_ERR) {
            std::cout<<"Unable to change signal handler."<<std::endl;
            return 1;
      }

      pthread_create(&ct, NULL, &busywork, NULL);

      while (continuing)
            sleep(1);
}
```

Fun fact: though it's easy to find instructions on how to use system calls online, you can also look the up in `man`! e.g.:

```
man pthread_create
```

So, this may *seem* reasonable, but it really isn't.
There are two problems:

- `continuing` can be changed while being read
  - Not the end of the world, but also not preferable
- The 'main loop' can potentially end before the thread. If the thread was performing calculations, we'd want to clean it up *before* leaving the main loop!
  - We could try addressing this by adding another boolean, or a counter for the number of threads in use, but what happens when multiple threads try to access it simultaneously?

What we really need is some form of ***mutual exclusion***.
A *mutex* is a lock, that can only be reserved by one block at a time. If two threads try to reserve the same lock, then one will need to wait.

Let's rewrite this to use more than one thread, and to also keep track of the number of threads running:

```
#include <iostream>
#include <unistd.h>
#include <cstdlib>
#include <signal.h>
#include <pthread.h>

static const int NUMTHREADS=4;
volatile bool continuing;
volatile int occupied;
pthread_mutex_t lock; //Our mutual exclusion lock

//At this level, this header signature can be ignored
void* busywork(void* unnecessary) {
	while (continuing) {
		usleep(100000);
	}
	pthread_mutex_lock(&lock);
	occupied--;
	std::cout<<"Exiting thread."<<std::endl;
	pthread_mutex_unlock(&lock);
}

void peek(int sig) {
	std::cout<<"Currently processing: "<<(continuing?"Yes":"No")<<std::endl;
}

//We don't really care what 'sig' is for this example
void interrupted(int sig) {
	std::cout<<"\nComputations complete.\nHalting now..."<<std::endl;
	continuing=false;
}

int main() {
	pthread_t ct[NUMTHREADS];//our child threads

	continuing=true;

	std::cout<<"About to commence; PID: "<<getpid()<<std::endl;

	if (signal(SIGINT,interrupted)==SIG_ERR) {
		std::cout<<"Unable to change signal handler."<<std::endl;
		return 1;
	}

	if (signal(SIGUSR1,peek)==SIG_ERR) {
		std::cout<<"Unable to change signal handler."<<std::endl;
		return 1;
	}

	for (int i=0;i<NUMTHREADS;i++) {
		pthread_mutex_lock(&lock);//reserve lock
		pthread_create(&ct[i], NULL, &busywork, NULL);
		occupied++;
		pthread_mutex_unlock(&lock);//release lock
	}

	//we don't need the mutex here, because we aren't changing occupied
	while (occupied>0)
		sleep(1);
	std::cout<<"Execution complete."<<std::endl;
}
```

So, now, we know enough to create *worker threads*: multiple instances of the same code, used in a pool to process multiple copies of data in parallel.

All we're missing is a task to try solving.

Phase 3 – Baby's First Cryptocurrency

Bitcoin is a semi-popular *cryptocurrency* (nerd-speak for *"I think there's a stable financial future in using a form of currency that most people have never heard of, let alone understand or be willing to accept for goods/services"*). What's interesting for this lab is how new Bitcoins are created: they're "mined" by solving difficult cryptographic hash functions.

The extremely simplified version is thus:

- *Hash functions* are one-way mathematical functions mapping an input to a fixed-length value
  - e.g. `md5(wokkawokka!)→124B4B72CA3b319313d2DE21C82FCB8C`
  - This makes md5 a 128-bit hash
- Any generated hash value is really just a very long number, which means it may be represented as a binary number
  - This makes the hash above equivalent to `00010010010010110100101101110010110010100011101100110001100100110001001111010010110111100010000111001000001011111100101110001100`
- Hash functions can operate on any data that can be reduced to binary data, but we *can* choose to use a fixed-length, arbitrary input number called a *nonce*
- The goal of mining is to find randomly generated nonces that correspond to hash values, such that:
  - A valid hash value has X leading zeroes (starting at the most significant/left end)
  - Only contiguous zeroes starting at the left end count
  - The longer the string of zeroes, obviously the harder to guess at a corresponding nonce
- If you're the first person to claim a hash (and demonstrate a corresponding nonce), then you can claim a bitcoin (and do some bookkeeping steps that don't interest us here)

As such, if MD5 were the hash (it isn't, but pretend it is), *wokkawokka!* would correspond to a relatively easy hash, and so that hash would have been discovered early on. The more time passes, the more ambitious you need to be to find an unclaimed hash value.

For this task, we'll be writing a *very* simple hash function, with pretty simple nonces.
First, let's be sure we know how to report our values. We can display a 32-bit binary pattern as such:

```
void printThirtyTwo(unsigned int word) {
	for (int i=0;i<32;i++)
		std::cout<<(((0x80000000>>i)&word)?1:0);
	std::cout<<std::endl;
}
```

and a 64-bit pattern thusly:

```
void printSixtyFour(unsigned long word) {
	for (int i=0;i<64;i++)
		std::cout<<(((0x8000000000000000>>i)&word)?1:0);
	std::cout<<std::endl;
}
```

Our hash is trivial:

```
//Breaks nonce up into 16 4-bit tokens, to generate hash value
unsigned int calchash(unsigned long nonce) {
	unsigned int hash=0;
	for (int i=15;i>=0;i--) {
		hash=hash*17+((nonce>>(4*i))&0x0F);
	}
	return hash;
}
```

We'll generate a nonce as such:

```
void genULong(unsigned long &nonce) {
     nonce=0;
     for (int i=63;i>=0;i--) {
          nonce<<=1;
          nonce|=random()%2;
     }

}
```

Important: This assumes we've randomized the seed!
e.g. at the beginning of the main function:

```
srand(time(NULL));
```

Lastly, we'll need to know how many *leading zeroes* a 32-bit unsigned int has:

```
int leadingZeroes(unsigned int value) {
     for (int i=0;i<32;i++)
          if ((value>>(31-i))&1) return i;
     return 32;
}
```

Between these individual functions, and what we learned up to phase 2, we can now implement our actual task:
The main program consists of a simple menu with two options:

- Try to find nonces that can create hashes with *at least* a user-selected number of leading zeroes
- Quit

When the user chooses to start generating hashes, those calculations will continue until the user presses ctrl+c.

- Once the user *does* press ctrl+c, it simply returns to the menu, for the next user selection

Hashes ***must*** be generated by ***four*** threads (all copies of the same function).

- Whenever any of the threads finds a nonce/hash pair, it should display them both on the screen
- It would be unacceptable to have two threads printing simultaneously, so you must use a mutex/lock to ensure that only one thread may generate output at a time

Submission Task
Simply create the threaded mining program described above.

Submission
When you're finished, simply demonstrate your sample program to get checked off.