Lecture 7: Intro to Multithreading

Note: Reading these lecture notes is not a substitute for watching the lecture. I frequently go off script, and you are responsible for understanding everything I talk about in lecture unless I specify otherwise.

What are threads?

So far, we have been talking about processes. Each process has its own virtual address space, its own file descriptor table, its own signal table and signal mask, etc. Each process can use registers on the CPU without other processes changing its values.

Threads are created within processes. A thread is very much like a process (in fact, threads are often called "lightweight processes,") but instead of being isolated, threads share most resources with other threads of the same process. Threads have their own stacks, but they share the same heap, the same globals, the same file descriptor table, and more.

Launching threads

In C++, to spawn a thread, create a `thread` object. The first argument is the function that the thread should run. This example launches 1 extra thread that prints "hello world":

```
static void printHello() {
    cout << "Hello world!" << endl;
}

int main(int argc, char *argv[]) {
    thread helloThread = thread(printHello);
    helloThread.join();
    return 0;
}
```

`thread.join()` is similar to waitpid; it blocks until the thread exits. **Unlike waitpid, your program will crash if you forget to call join()!**

Our first multithreading concurrency issues

printf is *thread safe*: if two threads/processes call it at the same time, one will have to wait for the other to finish printing before it's allowed to print. This ensures that any output on the terminal is coherent, and doesn't have characters interleaved from two running processes or threads.

We don't have the same guarantees from `cout`. If you run twenty "hello world" threads, you'll likely see some interleaving of output:

```
const size_t kNumThreads = 20;

static void printHello(size_t threadNum) {
    cout << "Hello world from thread " << threadNum << "!" << endl;
}

int main(int argc, char *argv[]) {
    thread threads[kNumThreads];
    for (size_t i = 0; i < kNumThreads; i++) {
        threads[i] = thread(printHello, i);
    }
    for (size_t i = 0; i < kNumThreads; i++) {
        threads[i].join();
    }
    return 0;
}
```

We have written two stream manipulators called `ostreamlock` and `ostreamunlock` that bring thread safety to `cout`. For now, you can just use them without understanding how they work (though I will explain how they work next week). The following `printHello` does not have the same interleaving problems:

```
static void printHello(size_t threadNum) {
    cout << oslock << "Hello world from thread " << threadNum << "!" << endl << osu
}
```

Note: `endl` should come before `osunlock`, since `endl` is an actual character (`\n`) that is being printed to the screen.

Note: If you forget `osunlock`, your program will probably hang, because no other thread (yourself included) will be able to get past `oslock`!

Race conditions

Consider the following code, where 10 ticket agents collaboratively sell off 100 tickets:

```
const size_t kNumAgents = 10;
// NOTE: use of globals is still discouraged, but I'm going to do it here for
// simplicity. (The better move is to create a class with private instance
// variables, and implement the below functions in that class.)
size_t remainingTickets = 100;

static void runTicketAgent(size_t id) {
    while (true) {
        talkToCustomer();   // Sleep for random duration

        // Sell ticket:
        if (remainingTickets == 0) break;
        remainingTickets--;
        cout << oslock << "Agent #" << id << " sold a ticket! (" << remainingTicke
            << " more to be sold)." << endl << osunlock;
    }
    cout << oslock << "Agent #" << id << " sees all tickets have been sold. Goodby
        << endl << osunlock;
}

int main(int argc, const char *argv[]) {
    thread agents[kNumAgents];
    for (size_t i = 0; i < kNumAgents; i++) {
        agents[i] = thread(runTicketAgent, 100 + i);
    }
    for (thread& agent: agents) {
        agent.join();
    }
    cout << "End of Business Day!" << endl;
    return 0;
}
```

If one of the threads gets pulled off the processor after the `if (remainingTickets == 0)` test but before `remainingTickets--`, then a *race condition* can occur: the agent thinks there are still tickets available and goes to sell it, but just before it sells the last ticket, a *different* agent sells it. Then this agent gets put back on the processor, executes `remainingTickets--` despite that now being 0, and **remainingTickets underflows.** If you run this code with `sleep_for(100)` in between those two lines, you'll see a very large number of tickets being sold.

This is a race condition in between two lines of C++, but race conditions can happen *within a single line of code* as well. Consider this program:

```
int main(int argc, const char *argv[]) {
    int counter = 0;

    thread thread1 = thread([&] () {
        counter++;
    });
    thread thread2 = thread([&] () {
        counter++;
    });

    thread1.join();
    thread2.join();

    cout << "counter = " << counter << endl;
    return 0;
}
```

(*Note: this code uses lambda functions, which are functions declared inline. We'll be using them at several points in the next few weeks. They are described here.*)

Almost always, this program prints `counter = 2`. However, even though `counter++` looks like a single uninterruptible line, it expands to three assembly instructions:

```
mov 0x12345600, %rax
inc %rax
mov %rax, 0x12345600
```

It's possible that Thread 1 loads `0` into `rax` before getting pulled off the processor. Thread 2 loads `0` into `rax`, increments it to `1`, and writes `1` back to memory. Then, Thread 1 wakes back up, increments its `rax` to `1`, and writes that back to memory. The final value that is printed is `counter = 1`, which is not what we'd expect.

Mutexes to the rescue

A mutex (short for "mutual exclusion") is a synchronization primitive that can prevent two threads from running critical code at the same time. By protecting a critical region with a mutex, we can mutually exclude threads from executing code inside of that region at the same time, avoiding the race conditions described above.

Mutexes can either be locked or unlocked; they are initialized to be in the unlocked state. A thread can call `lock()` on the mutex, and if it's unlocked, `lock()` will lock the lock and return immediately. Otherwise, it will wait for the lock to become unlocked.

We can fix the ticket agent code by declaring a mutex at the top of the program:

```
static mutex remainingTicketsLock;
```

Then, we use the mutex to ensure that no other thread reads or updates `remainingTickets` while we are working with that value:

```
remainingTicketsLock.lock();
if (remainingTickets == 0) {
    remainingTicketsLock.unlock();
    break;
}
remainingTickets--;
remainingTicketsLock.unlock();
```

Practice

How would we update this program to be safe from race conditions?

```
int x = 0;
int y = 0;
int z = 0;
```

```
static void thread1() {
    if (x > 0) {
        while (y < 10) {
            cout << oslock << "Thread 1 incrementing y to "
                << y + 1 << endl << osunlock;
            y++;
            sleep_for(30);
        }
        cout << oslock << "Thread 1 exiting, y = " << y << endl << osunlock;
    }
}

/* This is identical to thread1, with the exception of z++ */
static void thread2() {
    if (x > 0) {
        while (y < 10) {
            cout << oslock << "Thread 2 incrementing y to "
                << y + 1 << endl << osunlock;
            y++;
            z++;    // <-- This is the only difference vs thread1
            sleep_for(30);
        }
        cout << oslock << "Thread 2 exiting, y = " << y << endl << osunlock;
    }
}

int main(int argc, const char *argv[]) {
    x = 1;
    thread t1(thread1);
    thread t2(thread2);
    t1.join();
    t2.join();
    cout << oslock << "All threads finished! y = " << y
        << ", z = " << z << endl << osunlock;
    return 0;
}
```

Answer:

```
mutex yLock;

static void thread1() {
    if (x > 0) {
        yLock.lock()
        while (y < 10) {
            cout << "Thread 1 incrementing y to "
                << y + 1 << endl;
            y++;
            yLock.unlock();
            sleep_for(30);
            yLock.lock();
        }
        yLock.unlock();
        cout << oslock << "Thread 1 exiting, y = " << y << endl << osunlock;
    }
}

/* This is identical to thread1, with the exception of z++ */
static void thread2() {
    if (x > 0) {
        yLock.lock();
        while (y < 10) {
            cout << oslock << "Thread 2 incrementing y to "
                << y + 1 << endl << osunlock;
            y++;
            yLock.unlock();
            z++;    // This is the only difference vs thread1
            sleep_for(30);
            yLock.lock();
        }
        yLock.unlock();
        cout << oslock << "Thread 2 exiting, y = " << y << endl << osunlock;
    }
}
```

Notes:

- We need to protect `y` with a mutex, because it is written to by two threads, possibly simultaneously.
- We don't need to protect `x` with a mutex. It's *read* by two threads simultaneously, but that doesn't constitute a race condition.
- We don't need to protect `z` with a mutex. It's written to by two threads, but not simultaneously; the write in `main` is guaranteed to come after any writes from `thread2`.

When do I need a mutex?

- When there are multiple threads *writing* to a variable
- When there is a thread *writing* and one or more threads *reading*

Lock guards

A lock guard is a data type that locks a lock in its constructor and unlocks it in its destructor. This is helpful for automatically unlocking a lock when the lock guard goes out of scope, instead of needing to worry about unlocking the lock before every `break`, `return`, or `throw`. In the ticket agents example, we could remove every `remainingTicketsLock.unlock()` call, and replace every `lock()` call with the following:

```
lock_guard<mutex> lg(remainingTicketsLock);
```

Condition variables

We want to create a system where one thread adds work to a queue, and several threads work collaboratively to process the work in that queue. (This is called a *thread pool*, and you'll implement a more robust one in Assignment 4.)

As a super basic example, we won't even have a real queue data structure. We'll just have a counter storing how many items there are to process.

The main thread simply launches a scheduler thread and two worker threads, then waits for them to exit:

```
int main(int argc, const char *argv[]) {
    thread scheduler(runScheduler);
    thread worker1(runWorker, 1);
    thread worker2(runWorker, 2);
```

```
    scheduler.join();
    worker1.join();
    worker2.join();
    return 0;
}
```

The scheduler thread adds work to the queue every 300ms:

```
size_t numQueued = 0;
mutex numQueuedLock;

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
           << numQueued << ")" << endl << osunlock;
    }
}
```

The worker loops indefinitely, processing items to the queue as they are added. (In a real thread pool, the worker should exit when the scheduler is done adding work and all already-added work has been processed. We're going to let it run forever, though! This does mean we'll need to kill our example program with ctrl+c when we want it to exit.)

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        // Somehow, wait for numQueued to become positive if it isn't already...
        // ???

        // Pop from the queue, and do some expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
```

```
            numQueuedLock.unlock();
            sleep_for(1500);
        }
    }
```

Synchronizing the scheduler and worker threads

It would be nice if we could write this code to implement synchronization:

```
// In scheduler, after the cout:
signalWorkers();

// In worker, at the line with ???:
while (numQueued == 0) {
    numQueuedLock.unlock();
    waitForSignal();
    numQueuedLock.lock();
}
```

Condition variables provide this functionality. We can write something *almost* just like this:

```
condition_variable_any queueCv;

// In scheduler:
queueCv.notify_all();

// In workers:
while (numQueued == 0) {
    numQueuedLock.unlock();
    queueCv.wait();
    numQueuedLock.lock();
}
```

As it turns out, `queueCv.wait()` can't be called quite like this, because this introduces a race condition: what if numQueued becomes 0 just before the wait()? The worker will have missed the memo. (This is almost identical to the sigsuspend race condition I explained on Monday.) The real `wait()` function implements the above `while` loop in the operating system in a way that cannot be interrupted. The final synchronization code looks like this:

```
// In scheduler:
queueCv.notify_all();

// In workers:
queueCv.wait(numQueuedLock, [&](){return numQueued > 0;});
```