Threading and Concurrency Monday!

- **Assigned Reading**
  - You're reading all of Section 12.1 and Sections 12.3 through Sections 12.8 right now, skipping those subsections that refer to networking and servers.
    - Chapter 12 is actually the fourth chapter of the reader.
    - Code examples are in C, but the concepts are largely the same regardless of language.
  - Once we learn networking, we'll come back and hit on some of these excluded sections.
  - Assignment 3 is due a week from tonight, just before midnight.
- **Today**
  - We need to introduce the notion of a thread, discuss scenarios where they're useful, and highlight some of the concurrency issues that can begin to rear their ugly heads when multiple threads try to access the same data.
  - We need to work through all of the working examples from the slide deck I posted last Wednesday and probably dive in to the initial example of last Friday's slide deck.

Redux on Ticket Agent Example

- **The ticker agent example from the last slide deck is the most involved yet.**
  - We rely on global variables to store data that needs to be shared with all the threads. (Threading is often used to split the processor into multiple lightweight processes that collectively work toward a common goal, and the ticket agent example is an example that does just that).
  - Some portions of the code cannot be partially executed and then pulled from the processor, because doing so introduces a synchronization issue called a *race condition* if some other *thread* gets processor time and make partial progress through the same block of code
  - Even individual C and C++ expressions as simple as **`remainingTickets--;`** aren't always atomic, because they may compile to two or more assembly code instructions. Individual assembly code instructions are always atomic, but groups of two or three instructions aren't guaranteed to be executed in sequence within the same time splice.
  - Blocks of code (or even single expressions) that must be executed in full without any competition from competing threads are called *critical region*s.
  - We introduced the notion of a **`mutex`**, which provides atomic methods called **`lock`** and **`unlock`**.
    - If a **`mutex`** is unlocked (as it's constructed to be), then a call to **`lock`** flips the **`mutex`** into a locked state and returns without blocking or yielding.
    - If a **`mutex`** is locked by some other thread, then another thread's attempt to **`lock`** the same **`mutex`** causes that thread to block indefinitely (within the **`lock`** call) until the thread that **owns the lock on the `mutex` releases it by calling `unlock`.**
    - **`lock`** and **`unlock`** are used generally used to mark the beginning and end of a critical region, because doing so guarantees that at most one thread is executing any part of that critical region at one time.
      It's only after that thread exits the critical region (and properly calls **`unlock`**) that some other thread is able to acquire the lock on the same **`mutex`** and enter the same critical region.
    - The overall effect is that at most one thread is in the critical region at one time, so that all code within the critical region is effectively executed as one atomic transaction.
  - The implementations of **`lock`** and **`unlock`** rely on privileged OS access (e.g. the ability to turn off interrupts) and/or a dedicated assembly code instruction (e.g. test-and-set) to ensure that they themselves are implemented to run atomically without threat of any race conditions.
  - In general, it's considered good manners to keep the critical regions as small as is reasonable without introducing any concurrency issues.

The Dining Philosophers Problem

- **Canonical concurrency problem used to illustrate the threat of deadlock.**
  - Five philosophers sit around a circular table, each in front of a big plate of spaghetti.
  - A single fork is placed in between adjacently seated philosophers.
  - Every day, each philosopher comes to the table to think, eat, think, eat, think, and eat.
    That's three square meals of spaghetti after three extended think sessions.
    - Each philosopher keeps to himself as he thinks. Sometime he thinks for a long time, and sometimes he thinks for only a short time.
    - After each philosopher has thought for a while, he proceeds to eat one of his three daily meals. In order to eat, he must grab hold of two forks—the one to his left, and the one to his right. Once that has happened, he chows down on spaghetti to nourish his big, philosophizing brain. When he's full, he puts down the forks in the same order he picked them up and returns to thinking for a while.
  - Here's the core of a C++ program simulating what we just described (click here for full program):

```
static const unsigned int kNumPhilosophers = 5; // must be 2 or greater
static const unsigned int kNumForks = kNumPhilosophers;
static const unsigned int kNumMeals = 3;

static mutex forks[kNumForks]; // forks modeled as mutexes

static void think(unsigned int id) {
  cout << oslock << id << " starts thinking." << endl << osunlock;
  sleep_for(getThinkTime());
  cout << oslock << id << " all done thinking. " << endl << osunlock;
}

static void eat(unsigned int id) {
  unsigned int left = id;
  unsigned int right = (id + 1) % kNumForks;
  forks[left].lock();
  forks[right].lock();
  cout << oslock << id << " starts eating om nom nom nom." << endl << osunlock;
  sleep_for(getEatTime());
  cout << oslock << id << " all done eating." << endl << osunlock;
  forks[left].unlock();
  forks[right].unlock();
}

static void philosopher(unsigned int id) {
  for (unsigned int i = 0; i < kNumMeals; i++) {
    think(id);
    eat(id);
  }
}

int main(int argc, const char *argv[]) {
  thread philosophers[kNumPhilosophers];
  for (unsigned int i = 0; i < kNumPhilosophers; i++)
    philosophers[i] = thread(philosopher, i);
  for (thread& p: philosophers)
    p.join();
  return 0;
}
```

  - The program models each of the forks as a **`mutex`**. Each philosopher either has the fork or doesn't, and we want the act of grabbing the fork to operate as atomic and trasactional.

- The problem appears to work well (we'll run it several times in lecture), but it doesn't guard against this possibility: that each philosopher emerges from his deep thinking, successfully grabs the fork to his left, and then gets pulled off the processor because his time slice is over.
    - If this pathological scheduling pattern presents itself, eventually all each philosopher will be blocked from grabbing the fork on his right.
    - That will leave the program in a state where all five threads are entrenched in a state of deadlock, because each philosopher is stuck waiting for the philosopher to his right to let go of his left fork.
    - It's like one, big, intellectual silent contest, and the program is prevented from making progress.

Preventing Deadlock

- **When coding with threads, you need to ensure:**
  - that there are never any race conditions, no matter how remote the chances are, and...
  - that there's no possibility of deadlock, lest a subset of threads are blocked forever and starve for processor time.
- **`mutex`es are generally the solution to race conditions, as we've seen with the ticket agent example.**
- **Deadlock can be *programmatically* prevented by implanting directives to limit the number of threads that try to participate in what could otherwise result in deadlock.**
  - We could, for instance, recognize that it's impossible for three or more philosophers to be eating at the same time, via a simple pigeonhole principle argument (e.g. three philosophers can be eating at the same time if and only if there are six forks, and there are not). We can, therefore, limit the number of philosophers grabbing forks to 2.
  - We can also argue that it's okay to let up to four (but not all five) philosophers to transition into the eat portion of their think-eat cycle, knowing that at least one will succeed in grabbing both forks.
  - Here's the core of a program that takes the second approach (click here for full program):

```
static const unsigned int kNumPhilosophers = 5;
static const unsigned int kNumForks = kNumPhilosophers;
static const unsigned int kNumMeals = 3;
static mutex forks[kNumForks];
static unsigned int numAllowed = kNumPhilosophers - 1; // impose limit to avoid deadlock
static mutex numAllowedLock;

static void think(unsigned int id) {
  cout << oslock << id << " starts thinking." << endl << osunlock;
  sleep_for(getThinkTime());
  cout << oslock << id << " all done thinking. " << endl << osunlock;
}

static void waitForPermission() {
  while (true) {
    numAllowedLock.lock();
    if (numAllowed > 0) break;
    numAllowedLock.unlock();
    sleep_for(10);
  }
  numAllowed--;
  numAllowedLock.unlock();
}

static void grantPermission() {
  numAllowedLock.lock();
  numAllowed++;
  numAllowedLock.unlock();
}

static void eat(unsigned int id) {
  unsigned int left = id;
  unsigned int right = (id + 1) % kNumForks;
  waitForPermission();
  forks[left].lock();
  forks[right].lock();
  cout << oslock << id << " starts eating om nom nom nom." << endl << osunlock;
  sleep_for(getEatTime());
```

```
    cout << oslock << id << " all done eating." << endl << osunlock;
    grantPermission();
    forks[left].unlock();
    forks[right].unlock();
  }
```

- The above program always works, because there's no way all five philosophers can be mutually blocked on one another. We only allow four to go on and grab forks.
- It does, however, have one design flaw, and we'll discuss that design flaw as a means for coming up with an even better solution.

Busy Waiting

- **Reasonable Question: What is the design flaw with the solution just presented?**
  - Answer: the solution uses *busy waiting*, which in the eyes of systems programmers is usually a big no-no unless you have no other options.
  - To see what I mean, focus in on the implementation of **`waitForPermission`**:

```
static void waitForPermission() {
  while (true) {
    numAllowedLock.lock();
    if (numAllowed > 0) break;
    numAllowedLock.unlock();
    sleep_for(10);
  }
  numAllowed--;
  numAllowedLock.unlock();
}
```

  - The narrative is pretty straightforward: Continually poll the value of **`numAllowed`** until the value is positive. At that point, decrement it to emulate the consumption of a shared resource—in this case, a permission slip allowing a philosopher to start grabbing forks. Since there are multiple threads potentially examining and decrementing **`numAllowed`**, identify the critical region as one that needs to be guarded by a **`mutex`**. And if the philosopher notices the value of **`numAllowed`** is 0 (e.g. all permissions slips are out), then release the lock and yield the processor to some other philosopher thread who can actually do some useful work.
  - What's the outrage? The above solution uses busy waiting, which is concurrency jargon used when a thread periodically checks to see whether some condition has changed so it can move on to do more meaningful work. The problem with busy waiting, in most situations is that it hoards the processor from time to time, when it would be better to ensure that other threads—those can presumably have meaningful work to do—get the processor instead.
  - A better solution: if a philosopher doesn't have permission to advance (e.g. **`numAllowed`** is atomically confirmed to be zero), then that thread should be put to sleep *indefinitely* until some other thread sees reason to wake it up. In this example, another philosopher thread, after it increments **`numAllowed`** within **`grantPermission`**, could notify the indefinitely blocked thread that some permissions slips are now available.
  - Implementing this idea requires a more sophisticated concurrency directive that supports a different form of thread communication. Fortunately, C++11 provides a standard (albeit difficult-to-understand, imo) directive called the **`condition_variable_any`**.

Preventing Deadlock sans Busy Waiting

- **A better solution—one that doesn't allow threads to busy wait—can be framed in terms of a `condition_variable_any`.**
  - Take 3: Much of the program stays the same, but there are some new global variables, and the implementations of **`waitForPermission`** and **`grantPermission`** ar changed. You can click here to see the full program.
  - The code below summarizes what's different in take 3:

```
static mutex forks[kNumForks];
static int numAllowed = kNumForks - 1;
static mutex m;
static condition_variable_any cv;

static void waitForPermission() {
  lock_guard<mutex> lg(m);
  cv.wait(m, []{ return numAllowed > 0; });
  numAllowed--;
}

static void grantPermission() {
  lock_guard<mutex> lg(m);
  numAllowed++;
  if (numAllowed == 1) cv.notify_all();
}
```

  - For the moment, forget about the **`mutex`** called **`m`** and focus on the **`condition_variable_any`** called **`cv`**.
  - The **`condition_variable_any`** is the core concurrency directive that can preempt and block a thread until some condition is met.
    - In this example, the philosopher seeking permission to eat **`wait`**s indefinitely until some condition holds (unless the condition holds already, at which point it doesn't need to wait).
      - If **`numAllowed`** is positive at the moment **`wait`** is called, then it returns immediately without blocking.
      - If **`numAllowed`** is zero at the moment **`wait`** is called, then the calling thread is pulled off the processor, marked as unrunnable and blocked until the thread manager is informed (by another thread) that the value of **`numAllowed`** has changed.
    - In this example, the philosopher just finishing up a meal increments **`numAllowed`**, and if the value of **`numAllowed`** goes from 0 to 1, the same philosopher signals (via **`notify_all`**) all threads blocked on the **`condition_variable_any`** that a meaningful update has occurred. That prompts the thread manager to reexamine the condition on behalf of all threads blocked by **`wait`**, and potentially allow one or more of them to emerge from their long nap and move on to the work they weren't allowed to move on to before.

Gritty Details

- **The details are subtle but important.**
  - Here's the code again:

```
static mutex forks[kNumForks];
static int numAllowed = kNumForks - 1;
static mutex m;
static condition_variable_any cv;

static void waitForPermission() {
  lock_guard<mutex> lg(m);
  cv.wait(m, []{ return numAllowed > 0; });
  numAllowed--;
}

static void grantPermission() {
  lock_guard<mutex> lg(m);
  numAllowed++;
  if (numAllowed == 1) cv.notify_all();
}
```

  - Because **numAllowed** is being examined and potentially changed concurrently by many threads, and because a condition framed in terms of it (that is, **numAllowed > 0**) influences whether a thread blocks or continues uninterrupted, we need a traditional **mutex** here so that competing threads can lock down exclusive access to all things **numAllowed**.
  - The **lock_guard** class exists to automate the locking and unlocking of a **mutex**.
    - The **lock_guard** constructor binds an internal reference to the supplied **mutex** and calls **lock** on it (blocking—within the constructor—until the lock can be acquired).
    - The **lock_guard** destructor releases the lock on the same **mutex**.
    - The overall effect—at least in the above two functions—is that the combination of both implementations is marked as one big critical region.
      - Java gurus: the **lock_guard** class is the best thing C++ has in place to emulate Java's **synchronized** keyword.
    - The **lock_guard** is a template, because its implementation only requires its template type respond to zero-argument **lock** and **unlock** methods. The **mutex** class certainly does that, but many others (e.g. C++11's **recursive_mutex**, or its **timed_mutex**) do as well, and the **lock_guard** didn't want to hard code in **mutex**.
  - **condition_variable_any::wait** requires a **mutex** be supplied to help manage its synchronization on the supplied condition.
  - If **condition_variable_any::wait** notices that the supplied condition isn't being met, it puts the current thread to sleep indefinitely and releases the supplied **mutex**. When the **condition_variable_any** is notified and a waiting thread is permitted to continue, the thread re-acquires the **mutex** it released just prior to sleeping.