Lecture 15: Multithreading and Condition Variables

- The Dining Philosophers Problem
  - This is a canonical multithreading and concurrency example illustrating the potential for deadlock (and how to avoid it).
    - Five philosophers sit around a circular table, each in front of a big plate of spaghetti.
    - A single fork (the utensil, not the system call) is placed in between neighboring philosophers.
      - 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 barely thinks at all.
      - 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—one on his left, then one on his right. With two forks in hand, he chows 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.
  - The next two slides present the core of our first stab at the program that codes to this problem description. (The full program is right here.)
Lecture 15: Multithreading and Condition Variables

- The Dining Philosophers Problem
  - The program models each of the forks as a `mutex`. Each philosopher either has a fork or it doesn't. By modeling the fork as a `mutex`, we can rely on `mutex::lock` to model a thread-safe fork grab and `mutex::unlock` to model a thread-safe fork drop.

```cpp
static void philosopher(size_t id, mutex& left, mutex& right) {
    for (size_t i = 0; i < 3; i++) {
        think(id);
        eat(id, left, right);
    }
}

int main(int argc, const char *argv[]) {
    mutex forks[5];
    thread philosophers[5];
    for (size_t i = 0; i < 5; i++) {
        mutex& left = forks[i], & right = forks[(i + 1) % 5];
        philosophers[i] = thread(philosopher, i, ref(left), ref(right));
    }
    for (thread& p: philosophers) p.join();
    return 0;
}
```
The implementation of `think` is straightforward. It's designed to emulate the time a philosopher spends thinking without interacting with forks or other philosophers.

The implementation of `eat` is almost as straightforward, provided you understand the thread subroutine is being fed references to the two forks needed to eat.

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

static void eat(size_t id, mutex& left, mutex& right) {
    left.lock();
    right.lock();
    cout << oslock << id << " starts eating om nom nom nom." << endl << osunlock;
    sleep_for(getEatTime());
    cout << oslock << id << " all done eating." << endl << osunlock;
    left.unlock();
    right.unlock();
}
```
Lecture 15: Multithreading and Condition Variables

- The program appears to work well (we'll run it several times), but it doesn't guard against this possibility: each philosopher emerges from deep thought, successfully grabs the fork to his left, and is then forced off the processor because his time slice is up.
- If all five philosopher threads are subjected to the same scheduling pattern (unlikely, but possible), each philosopher would be stuck waiting for a second fork that will never be available. There's a real potential for deadlock, so the program is broken.
- The deadlock is more or less guaranteed if we insert a `sleep_for` call in between the two calls to `lock`, as we have in the version of `eat` presented below.
  - We should be able to insert a `sleep_for` call anywhere in a thread routine. If it surfaces an otherwise rare concurrency issue, then you have a larger problem to be solved.

```cpp
static void eat(size_t id, mutex& left, mutex& right) {
    left.lock();
    sleep_for(5000); // artificially force off the processor
    right.lock();
    cout << oslock << id << " starts eating om nom nom nom." << endl << osunlock;
    sleep_for(getEatTime());
    cout << oslock << id << " all done eating." << endl << osunlock;
    left.unlock();
    right.unlock();
}
```
Lecture 15: Multithreading and Condition Variables

• When coding with threads, you need to ensure that
  ○ there are no race conditions, even if they rarely cause problems, and
  ○ there's no threat of deadlock, lest a subset of threads are forever blocked and starving for processor time.
• mutexes are generally the solution to race conditions, as we've seen with the ticket agent example. We can use them to mark the boundaries of critical regions and limit the number of threads present within to be at most one.
• Deadlock can be programmatically prevented by implanting directives to limit the number of threads competing for a shared resource, like forks.
  ○ We could, for instance, recognize it's impossible for three philosophers to be eating at the same time. That means we could limit the number of philosophers who have permission to grab forks to a mere 2.
  ○ We could also argue it's okay to let four (though certainly not all five) philosophers grab forks, knowing that at least one will succeed in grabbing both.
  ○ My personal preference? Impose a limit of four. My rationale? Implant the minimal amount of bottlenecking needed to remove the threat of deadlock, and trust the thread manager to otherwise make good choices.
Lecture 15: Multithreading and Condition Variables

- Here's the core of a program that limits the number of philosophers grabbing forks to four. (The full program can be found right here.)
  - I impose this limit by introducing the notion of a permission slip (or permit). Before grabbing forks, a philosopher must first acquire one of four permission slips.
  - These permission slips need to be acquired and released without race condition.
  - For now, I'll model a permit using a counter—I call it `permits`—and a companion `mutex`—I call it `permitsLock`—that must be acquired before examining or altering `permits`.

```c
int main(int argc, const char *argv[]) {
    size_t permits = 4;
    mutex forks[5], permitsLock;
    thread philosophers[5];
    for (size_t i = 0; i < 5; i++) {
        mutex& left = forks[i],
            & right = forks[(i + 1) % 5];
        philosophers[i] =
            thread(philosopher, i, ref(left), ref(right), ref(permits), ref(permitsLock));
    }
    for (thread& p: philosophers) p.join();
    return 0;
}
```
Lecture 15: Multithreading and Condition Variables

- The implementation of `think` stays the same, so I don't present it again.
- The implementation of `eat`, however, changed (and the changes are highlighted in **bold**).
  - It accepts two additional references: one to the number of available `permits`, and a second to the `mutex` used to guard against simultaneous access to `permits`.

```cpp
static void eat(size_t id, mutex& left, mutex& right, size_t& permits, mutex& permitsLock) {
    waitForPermission(permits, permitsLock); // on next slide
    left.lock(); 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(permits, permitsLock); // on next slide
    left.unlock(); right.unlock();
}

static void philosopher(size_t id, mutex& left, mutex& right, size_t& permits, mutex& permitsLock) {
    for (size_t i = 0; i < kNumMeals; i++) {
        think(id);
        eat(id, left, right, permits, permitsLock);
    }
}
```
The implementation of `eat` on the prior slide deck introduces calls to `waitForPermission` and `grantPermission`.

- The implementation of `grantPermission` is certainly the easier of the two to understand: transactionally increment the number of `permits` by one.
- The implementation of `waitForPermission` is less obvious. Because we don't know what else to do (yet!), we busy wait with short naps until the number of `permits` is positive. Once that happens, we consume a permit and then return.

```cpp
static void waitForPermission(size_t& permits, mutex& permitsLock) {
    while (true) {
        permitsLock.lock();
        if (permits > 0) break;
        permitsLock.unlock();
        sleep_for(10);
    }
    permits--;
    permitsLock.unlock();
}

static void grantPermission(size_t& permits, mutex& permitsLock) {
    permitsLock.lock();
    permits++;
    permitsLock.unlock();
}
```
Lecture 15: Multithreading and Condition Variables

- The second version of the program works, in the sense that it never deadlocks.
- It does, however, suffer from busy waiting, which the systems programmer gospel says is a big no-no unless there are absolutely no other options.
  - A better solution? if a philosopher doesn't have permission to advance, then that thread should sleep until another thread sees reason to wake it up. In this example, another philosopher thread, after it increments \texttt{permits} within \texttt{grantPermission}, could notify the sleeping thread that a permit just became available.
  - Implementing this idea requires a more sophisticated concurrency directive that supports a different form of thread communication—one akin to the use of signals and \texttt{sigsuspend} to support communication between processes. Fortunately, C++ provides a standard directive called the \texttt{condition_variable_any} to do exactly this.

```cpp
class condition_variable_any {
  public:
    void wait(mutex& m);
    template <typename Pred> void wait(mutex& m, Pred pred);
    void notify_one();
    void notify_all();
};
```
Lecture 15: Multithreading and Condition Variables

- Here's the main thread routine that introduces a condition_variable_any to support the notification model we'll use in place of busy waiting. (Full program: here)
  - All of the variables needed to foster inter-thread communication are highlighted in bold.
  - The philosopher thread routine and the eat thread subroutine accept references to permits, cv, and m, because references to all three need to be passed on to waitForPermission and grantPermission.
  - I go with the simpler name m instead of permitsLock for reasons I'll get to soon.

```c
int main(int argc, const char *argv[]) {
    size_t permits = 4;
    mutex forks[5], m;
    condition_variable_any cv;
    thread philosophers[5];
    for (size_t i = 0; i < 5; i++) {
        mutex& left = forks[i], & right = forks[(i + 1) % 5];
        philosophers[i] =
            thread(philosopher, i, ref(left), ref(right), ref(permits), ref(cv), ref(m));
    }
    for (thread& p: philosophers) p.join();
    return 0;
}
```
Lecture 15: Multithreading and Condition Variables

- The new implementations of `waitForPermission` and `grantPermission` are below:

```cpp
static void waitForPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
    lock_guard<mutex> lg(m);
    while (permits == 0) cv.wait(m);
    permits--;
}

static void grantPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
    lock_guard<mutex> lg(m);
    permits++;
    if (permits == 1) cv.notify_all();
}
```

- The `lock_guard` is a convenience class whose constructor calls `lock` on the supplied `mutex` and whose destructor calls `unlock` on the same `mutex`. It's a convenience class used to ensure the lock on a `mutex` is released no matter how the function exits (early return, standard return at end, thrown exception, etc.)

- `grantPermission` is a straightforward thread-safe increment, save for the fact that if `permits` just went from 0 to 1, it's possible other threads are waiting for a permit to become available. That's why the conditional call to `cv.notify_all()` is there.
The new implementations of `waitForPermission` and `grantPermission` are below:

```cpp
static void waitForPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
    lock_guard<mutex> lg(m);
    while (permits == 0) cv.wait(m);
    permits--;
}

static void grantPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
    lock_guard<mutex> lg(m);
    permits++;
    if (permits == 1) cv.notify_all();
}
```

The implementation of `waitForPermission` will eventually grant a permit to the calling thread, though it may need to wait a while for one to become available.
- If there aren't any permits, the thread is forced to sleep via `cv.wait(m)`. The thread manager releases the lock on `m` just as it's putting the thread to sleep.
- When `cv` is notified within `grantPermission`, the thread manager marks the thread as runnable, but mandates it reacquire the lock on `m` (very much needed to properly reevaluate `permits == 0`) before returning from `cv.wait(m)` and proceeding.