Lecture 11: Multithreading and Condition Variables

Principles of Computer Systems

Spring 2019

Stanford University

Computer Science Department

Lecturer: Chris Gregg

PDF of this presentation

No Labs this week – very short video instead

Because of the midterm exam, we won't have labs this week.

Please watch the following 5-minute video, instead:

CS Storytelling

When you have finished watching, fill out your name and SUNet for lab week credit.

From Lecture 10: confused-ticket-agent compile bug

- During the last class, I had a bug in my code for confused-ticket-agents.cc which I couldn't find during lecture. As promised, I went back and found the error.

```
$ make
/usr/bin/g++-5  -g -Wall -pedantic -O0 -std=c++0x -D_GLIBCXX_USE_NANOSLEEP -D_GLIBCXX_USE_SCHED_YIELD -I/usr/class/cs110/local/include/  -c -o confus
In file included from /usr/include/c++/5/mutex:42:0,
                 from confused-ticket-agents.cc:1:
/usr/include/c++/5/functional: In instantiation of ‘struct std::_Bind_simple<void (*(long unsigned int, std::reference_wrapper<long unsigned int>))(l
/usr/include/c++/5/thread:137:59:   required from ‘std::thread::thread(_Callable&&, _Args&& ...) [with _Callable = void (&)(long unsigned int, unsign
confused-ticket-agents.cc:62:65:   required from here
/usr/include/c++/5/functional:1505:61: error: no type named ‘type’ in ‘class std::result_of<void (*(long unsigned int, std::reference_wrapper<long un
       typedef typename result_of<_Callable(_Args...)>::type result_type;
                                                             ^
/usr/include/c++/5/functional:1526:9: error: no type named ‘type’ in ‘class std::result_of<void (*(long unsigned int, std::reference_wrapper<long uns
         _M_invoke(_Index_tuple<_Indices...>)
         ^
<builtin>: recipe for target 'confused-ticket-agents.o' failed
make: *** [confused-ticket-agents.o] Error 1
```

```
45 static void ticketAgent(size_t id, unsigned int& remainingTickets) {
46     while (remainingTickets > 0) {
47         handleCall(); // sleep for a small amount of time to emulate conversation time.
48         remainingTickets--;
...

58 int main(int argc, const char *argv[]) {
59     thread agents[10];
60     size_t remainingTickets = 250;
61     for (size_t i = 0; i < 10; i++)
62         agents[i] = thread(ticketAgent, 101 + i, ref(remainingTickets));
63     for (thread& agent: agents) agent.join();
64     cout << "End of Business Day!" << endl;
65     return 0;
66 }
```

- The error? (drumroll...)

From Lecture 10: confused-ticket-agent compile bug

- During the last class, I had a bug in my code for confused-ticket-agents.cc which I couldn't find during lecture. As promised, I went back and found the error.

```
static void ticketAgent(size_t id, unsigned int& remainingTickets) {
    while (remainingTickets > 0) {
        handleCall(); // sleep for a small amount of time to emulate conversation time.
        remainingTickets--;
...
```

```
int main(int argc, const char *argv[]) {
    thread agents[10];
    size_t remainingTickets = 250;
    for (size_t i = 0; i < 10; i++)
        agents[i] = thread(ticketAgent, 101 + i, ref(remainingTickets));
    for (thread& agent: agents) agent.join();
    cout << "End of Business Day!" << endl;
    return 0;
}
```

- Type error! I passed in a reference to a **`size_t`** but the function was expecting an **`unsigned int`**. Doh!

Lecture 11: Multithreading and Condition Variables

- The Dining Philosophers Problem
  - This is a canonical multithreading example used to illustrate the potential for deadlock and how to avoid it.
    - Five philosophers sit around a table, each in front of a big plate of spaghetti.
    - A single fork (the utensil, not the system call) is placed 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 11: Multithreading and Condition Variables

- The Dining Philosophers Problem
  - The program models each of the forks as a **mutex**, and each philosopher either holds a fork or 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 release.

```
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;
}
```

Lecture 11: Multithreading and Condition Variables

- The Dining Philosophers Problem
  - 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 he needs to eat.

```
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 11: Multithreading and Condition Variables

- The program appears to work well (we'll run it several times), but it doesn't guard against this: 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, each would be stuck waiting for a second fork to become available. That's a real deadlock threat.
- 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 concurrency issue, then you have a larger problem to be solved.

```
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 11: 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 zero threat of deadlock, lest a subset of threads are forever starving for processor time.
- **mutex**es 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 them 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 successfully grab 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 11: 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 changing **permits**.

```
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 11: Multithreading and Condition Variables

- The implementation of **think** is the same, so I don't present it again.
- The implementation of **eat**, however, changes.
    - 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**.

```
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);
  }
}
```

Lecture 11: Multithreading and Condition Variables

- 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.

```
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 11: 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 verboten unless there are 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 **permits** within **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 **sigsuspend** to support communication between processes. Fortunately, C++ provides a standard directive called the **condition_variable_any** to do exactly this.

```
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 11: 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)
  - 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 shorter name **m** instead of **permitsLock** for reasons I'll get to soon.

```
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 11: Multithreading and Condition Variables

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

```
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, exception thrown, 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.

Lecture 11: Multithreading and Condition Variables

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

```
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 wakes the sleeping thread, but mandates it reacquire the lock on **m** (very much needed to properly reevaluate **permits == 0**) before returning from **cv.wait(m)**.
    - Yes, **waitForPermission** requires a **while** loop instead an **if** test. Why? It's possible the permit that just became available is immediately consumed by the thread that just returned it. Unlikely, but technically possible.

Lecture 11: Multithreading and Condition Variables

- The Dining Philosophers Problem, continued
  - **while** loops around **cv.wait(m)** calls are so common that the **condition_variable_any** class exports a second, two-argument version of **wait** whose implementation is a **while** loop around the first. That second version looks like this:

```
template <Predicate pred>
void condition_variable_any::wait(mutex& m, Pred pred) {
  while (!pred()) wait(m);
}
```

  - It's a template method, because the second argument supplied via **pred** can be anything capable of standing in for a zero-argument, **bool**-returning function.
  - The first **waitForPermissions** can be rewritten to rely on this new version, as with:

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

Lecture 11: Multithreading and Condition Variables

- Fundamentally, the **size_t**, **condition_variable_any**, and **mutex** are collectively working together to track a resource count—in this case, four permission slips.
  - They provide thread-safe increment in **grantPermission** and thread-safe decrement in **waitForPermission**.
  - They work to ensure that a thread blocked on zero permission slips goes to sleep indefinitely, and that it remains asleep until another thread returns one.
- In our latest **dining-philosopher** example, we relied on these three variables to collectively manage a thread-safe accounting of four permission slips. However!
  - There is little about the implementation that requires the original number be four. Had we gone with 20 philosophers and and 19 permission slips, **waitForPermission** and **grantPermission** would still work as is.
  - The idea of maintaining a thread-safe, generalized counter is so useful that most programming languages include more generic support for it. That support normally comes under the name of a **semaphore**.
  - For reason that aren't entirely clear to me, standard C++ omits the **semaphore** from its standard libraries. My guess as to why? It's easily built in terms of other supported constructs, so it was deemed unnecessary to provide official support for it.

Lecture 11: Multithreading and Condition Variables

- The **semaphore** constructor is so short that it's inlined right in the declaration of the **semaphore** class.
- **semaphore::wait** is our generalization of **waitForPermission**.

```
void semaphore::wait() {
  lock_guard<mutex> lg(m);
  cv.wait(m, [this] { return value > 0; })
  value--;
}
```

- Why does the capture clause include the **this** keyword?
  - Because the anonymous predicate function passed to **cv.wait** is just that—a regular function. Since functions aren't normally entitled to examine the **private** state of an object, the capture clause includes **this** to effectively convert the **bool**-returning function into a **bool**-returning **semaphore** method.
- **semaphore::signal** is our generalization of **grantPermission**.

```
void semaphore::signal() {
  lock_guard<mutex> lg(m);
  value++;
  if (value == 1) cv.notify_all();
}
```

Lecture 11: Multithreading and Condition Variables

- Here's our final version of the **dining-philosophers**.
  - It strips out the exposed **size_t**, **mutex**, and **condition_variable_any** and replaces them with a single **semaphore**.
  - It updates the thread constructors to accept a single reference to that **semaphore**.

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

int main(int argc, const char *argv[]) {
  semaphore permits(4);
  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), ref(permits));
  }
  for (thread& p: philosophers) p.join();
  return 0;
}
```

Lecture 11: Multithreading and Condition Variables

- **eat** now relies on that **semaphore** to play the role previously played by **waitForPermission** and **grantPermission**.

```
static void eat(size_t id, mutex& left, mutex& right, semaphore& permits) {
  permits.wait();
  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;
  permits.signal();
  left.unlock();
  right.unlock();
}
```

  - We could switch the order of the last two lines, so that **right.unlock()** precedes **left.unlock()**. Is the switch a good idea? a bad one? or is it really just arbitrary?
  - One student suggested we use a **mutex** to bundle the calls to **left.lock()** and **right.lock()** into a critical region. Is this a solution to the deadlock problem?
  - We could lift the **permits.signal()** call up to appear in between **right.lock()** and the first **cout** statement. Is that valid? Why or why not?

Lecture 11: Multithreading and Condition Variables

- New concurrency pattern!
    - **semaphore::wait** and **semaphore::signal** can be leveraged to support a different form of communication: **thread rendezvous**.
    - Thread rendezvous is a generalization of **thread::join**. It allows one thread to stall —via **semaphore::wait**—until another thread calls **semaphore::signal**, often because the signaling thread just prepared some data that the waiting thread needs before it can continue.
- To illustrate when thread rendezvous is useful, we'll implement a simple program without it, and see how thread rendezvous can be used to repair some of its problems.
    - The program has two meaningful threads of execution: one thread publishes content to a shared buffer, and a second reads that content as it becomes available.
    - The program is a nod to the communication in place between a web server and a browser. The server publishes content over a dedicated communication channel, and the browser consumes that content.
    - The program also reminds me of how two independent processes behave when one writes to a pipe, a second reads from it, and how the write and read processes behave when the pipe is full (in principle, a possibility) or empty.

Lecture 11: Multithreading and Condition Variables

- Consider the following program, where concurrency directives have been intentionally omitted. (The full program is right here.)

```
static void writer(char buffer[]) {
  cout << oslock << "Writer: ready to write." << endl << osunlock;
  for (size_t i = 0; i < 320; i++) { // 320 is 40 cycles around the circular buffer of length 8
    char ch = prepareData();
    buffer[i % 8] = ch;
    cout << oslock << "Writer: published data packet with character '"
         << ch << "'." << endl << osunlock;
  }
}

static void reader(char buffer[]) {
  cout << oslock << "\t\tReader: ready to read." << endl << osunlock;
  for (size_t i = 0; i < 320; i++) { // 320 is 40 cycles around the circular buffer of length 8
    char ch = buffer[i % 8];
    processData(ch);
    cout << oslock << "\t\tReader: consumed data packet " << "with character '"
         << ch << "'." << endl << osunlock;
  }
}

int main(int argc, const char *argv[]) {
  char buffer[8];
  thread w(writer, buffer);
  thread r(reader, buffer);
  w.join();
  r.join();
  return 0;
}
```

Lecture 11: Multithreading and Condition Variables

- Here's what works:
  - Because the **main** thread declares a circular buffer and shares it with both children, the children each agree where content is stored.
  - Think of the buffer as the state maintained by the implementation of **pipe**, or the state maintained by an internet connection between a server and a client.
  - The **writer** thread publishes content to the circular buffer, and the **reader** thread consumes that same content as it's written. Each thread cycles through the buffer the same number of times, and they both agree that **i % 8** identifies the next slot of interest.
- Here's what's broken:
  - Each thread runs more or less independently of the other, without consulting the other to see how much progress it's made.
  - In particular, there's nothing in place to inform the **reader** that the slot it wants to read from has meaningful data in it. It's possible the writer just hasn't gotten that far yet.
  - Similarly, there's nothing preventing the **writer** from advancing so far ahead that it begins to overwrite content that has yet to be consumed by the **reader**.

Lecture 11: Multithreading and Condition Variables

- One solution? Maintain two **semaphore**s.
  - One can track the number of slots that can be written to without clobbering yet-to-be-consumed data. We'll call it **emptyBuffers**, and we'll initialize it to 8.
  - A second can track the number of slots that contain yet-to-be-consumed data that can be safely read. We'll call it **fullBuffers**, and we'll initialize it to 0.
- Here's the new **main** program that declares, initializes, and shares the two **semaphore**s.

```
int main(int argc, const char *argv[]) {
  char buffer[8];
  semaphore fullBuffers, emptyBuffers(8);
  thread w(writer, buffer, ref(fullBuffers), ref(emptyBuffers));
  thread r(reader, buffer, ref(fullBuffers), ref(emptyBuffers));
  w.join();
  r.join();
  return 0;
}
```

- The **writer** thread waits until at least one buffer is empty before writing. Once it writes, it'll increment the full buffer count by one.
- The **reader** thread waits until at least one buffer is full before reading. Once it reads, it increments the empty buffer count by one.

Lecture 11: Multithreading and Condition Variables

- Here are the two new thread routines:

```
static void writer(char buffer[], semaphore& full, semaphore& empty) {
  cout << oslock << "Writer: ready to write." << endl << osunlock;
  for (size_t i = 0; i < 320; i++) { // 320 is 40 cycles around the circular buffer of length 8
    char ch = prepareData();
    empty.wait();   // don't try to write to a slot unless you know it's empty
    buffer[i % 8] = ch;
    full.signal();  // signal reader there's more stuff to read
    cout << oslock << "Writer: published data packet with character '"
         << ch << "'." << endl << osunlock;
  }
}

static void reader(char buffer[], semaphore& full, semaphore& empty) {
  cout << oslock << "\t\tReader: ready to read." << endl << osunlock;
  for (size_t i = 0; i < 320; i++) { // 320 is 40 cycles around the circular buffer of length 8
    full.wait();    // don't try to read from a slot unless you know it's full
    char ch = buffer[i % 8];
    empty.signal(); // signal writer there's a slot that can receive data
    processData(ch);
    cout << oslock << "\t\tReader: consumed data packet " << "with character '"
         << ch << "'." << endl << osunlock;
  }
}
```

- The reader and writer rely on these **semaphore**s to inform the other how much work they can do before being necessarily forced off the CPU.
- Thought question: can we rely on just one **semaphore** instead of two? Why or why not?