CS110 Lecture 10: Threads and Mutexes

Principles of Computer Systems

Winter 2020

Stanford University

Computer Science Department

Instructors: Chris Gregg and

Nick Troccoli

PDF of this presentation

CS110 Topic 3: How can we have concurrency within a single process?

Learning About Processes

Introduction to Threads → Mutexes and Condition Variables → Condition Variables and Semaphores → Multithreading Patterns

Introduction to Threads	Mutexes and Condition Variables	Condition Variables and Semaphores	Multithreading Patterns
2/3	Today	2/10	2/12

Today's Learning Goals

- Discover some of the pitfalls of threads sharing the same virtual address space
- Learn how locks can help us limit access to shared resources
- Get practice using condition variables to wait for signals from other threads

Plan For Today

- **Recap:** Threads in C++
- Races When Accessing Shared Data
- Introducing Mutexes
- **Break:** Announcements
- Dining With Philosophers

Plan For Today

- **Recap:** Threads in C++
- Races When Accessing Shared Data
- Introducing Mutexes
- **Break:** Announcements
- Dining With Philosophers

Threads

A **thread** is an independent execution sequence within a single process.

- Most common: assign each thread to execute a single function in parallel
- Each thread operates within the same process, so they *share global data* (!) (text, data, and heap segments)
- They each have their own stack (e.g. for calls within a single thread)
- Execution alternates between threads as it does for processes
- Many similarities between threads and processes; in fact, threads are often called **lightweight processes**.

Threads vs. Processes

Processes:

- isolate virtual address spaces (good: security and stability, bad: harder to share info)
- can run external programs easily (fork-exec) (good)
- harder to coordinate multiple tasks within the same program (bad)

Threads:

- share virtual address space (bad: security and stability, good: easier to share info)
- can't run external programs easily (bad)
- easier to coordinate multiple tasks within the same program (good)

C++ thread

A thread object can be spawned to run the specified function with the given arguments.

```
thread myThread(myFunc, arg1, arg2, ...);
```

- **myFunc:** the function the thread should execute asynchronously
- **args:** a list of arguments (any length, or none) to pass to the function upon execution
- Once initialized with this constructor, the thread may execute at any time!

To pass objects by reference to a thread, use the **ref()** function:

```
void myFunc(int& x, int& y) {...}

thread myThread(myFunc, ref(arg1), ref(arg2));
```

C++ thread

We can make an array of threads as follows:

```
// declare array of empty thread handles
thread friends[5];

// Spawn threads
for (size_t i = 0; i < 5; i++) {
        friends[i] = thread(myFunc, arg1, arg2);
}
```

We can also initialize an array of threads as follows (note the loop by reference):

```
thread friends[5];
for (thread& currFriend : friends) {
    currFriend = thread(myFunc, arg1, arg2);
}
```

C++ thread

To wait on a thread to finish, use the **.join()** method:

```
thread myThread(myFunc, arg1, arg2);

... // do some work

// Wait for thread to finish (blocks)
myThread.join();
```

For multiple threads, we must wait on a specific thread one at a time:

```
thread friends[5];
// spawn here
// now we wait for each to finish
for (size_t i = 0; i < 5; i++) {
        friends[i].join();
}
```

Thread Safety

A *thread-safe* function is one that will always execute correctly, even when called concurrently from multiple threads.

- **printf** is thread-safe, but **operator<<** is *not*. This means e.g. **cout** statements could get interleaved!
- To avoid this, use **oslock** and **osunlock** (custom CS110 functions - **#include "ostreamlock.h"**) around streams. They ensure at most one thread has permission to write into a stream at any one time.

```
cout << oslock << "Hello, world!" << endl << osunlock;
```

Threads Share Memory

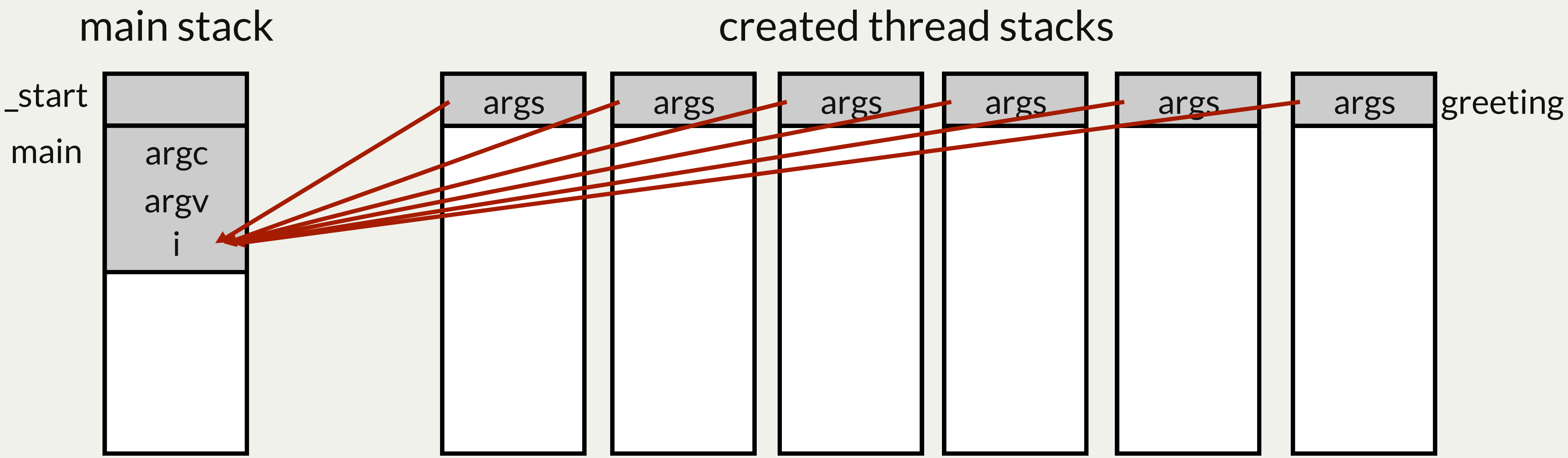
```
static void greeting(size_t& i) {
    cout << oslock << "Hello, world! I am thread " << i << endl << osunlock;
}

static const size_t kNumFriends = 6;
int main(int argc, char *argv[]) {
  cout << "Let's hear from " << kNumFriends << " threads." << endl;

  thread friends[kNumFriends]; // declare array of empty thread handles

  // Spawn threads
  for (size_t i = 0; i < kNumFriends; i++) {
      friends[i] = thread(greeting, ref(i));
  }

  // Wait for threads
  for (size_t i = 0; i < kNumFriends; i++) {
     friends[i].join();
  }

  cout << "Everyone's said hello!" << endl;
  return 0;
}
```

Output

```
$ ./friends
Let's hear from 6 threads.
Hello, world! I am thread 2
Hello, world! I am thread 2
Hello, world! I am thread 3
Hello, world! I am thread 5
Hello, world! I am thread 5
Hello, world! I am thread 6
Everyone's said hello!
```

Threads Share Memory

```
for (size_t i = 0; i < kNumFriends; i++) {
    friends[i] = thread(greeting, ref(i));
}
```



Solution: pass a copy of i (not by reference) so it does not change.

Plan For Today

- **Recap:** Threads in C++
- **Races When Accessing Shared Data**
- Introducing Mutexes
- **Break:** Announcements
- Dining With Philosophers

Thread-Level Parallelism

- Threads allow a process to parallelize a problem across multiple cores
- Consider a scenario where we want to process 250 images and have 10 cores
- **Simulation**: let each thread help process images until none are left
- Let's jump to a demo to see how this works

```
// images.cc
int main(int argc, const char *argv[]) {
  thread processors[10];
  size_t remainingImages = 250;
  for (size_t i = 0; i < 10; i++)
    processors[i] = thread(process, 101 + i, ref(remainingImages));
  for (thread& proc: processors) proc.join();
  cout << "Images done!" << endl;
  return 0;
}
```

Thread-Level Parallelism

There is a *race condition* here!

- **Problem:** threads could interrupt each other in between lines 2 and 3.

```
1 static void process(size_t id, size_t& remainingImages) {
2     while (remainingImages > 0) {
3         sleep_for(500);  // simulate "processing image"
4         remainingImages--;
5         ...
6     }
7     ...
8 }
```

- **Why is this?** It's because **remainingImages > 0** test and **remainingImages--** aren't atomic
- Atomicity: externally, the code has either executed or not; external observers do not see any intermediate states mid-execution
- If a thread evaluates **remainingImages > 0** to be **true** and commits to processing an image, another thread could come in and claim that same image before this thread processes it.

Why Test and Decrement Is REALLY NOT Thread-Safe

- C++ statements aren't inherently atomic. Virtually all C++ statements—even ones as simple as **remainingImages--**—compile to multiple assembly code instructions.
- Assembly code instructions are atomic, but C++ statements are not.
- **g++** on the myths compiles **remainingImages--** to five assembly code instructions, as with:

```
0x0000000000401a9b <+36>:    mov    -0x20(%rbp),%rax
0x0000000000401a9f <+40>:    mov    (%rax),%eax
0x0000000000401aa1 <+42>:    lea    -0x1(%rax),%edx
0x0000000000401aa4 <+45>:    mov    -0x20(%rbp),%rax
0x0000000000401aa8 <+49>:    mov    %edx,(%rax)
```

- The first two lines drill through the **remainingImages** reference to load a copy of the **remainingImages** held on **main**'s stack. The third line decrements that copy, and the last two write the decremented copy back to the **remainingImages** variable held on **main**'s stack.
- The ALU operates on registers, but registers are private to a core, so the variable needs to be loaded from and stored to memory.
  - Each thread makes a local copy of the variable before operating on it
  - What if multiple threads all load the variable at the same time: they all think there's only 128 images remaining and process 128 at the same time

Plan For Today

- **Recap:** Threads in C++
- Races When Accessing Shared Data
- **Introducing Mutexes**
- **Break:** Announcements
- Dining With Philosophers

Mutex

A mutex is a variable type that represents something like a "locked door".

https://www.flickr.com/photos/ofsmallthings/8220574255

You can **lock** the door:

- if it's unlocked, you go through the door and lock it

- if it's locked, you *wait for it to unlock first*

If you most recently locked the door, you can **unlock** the door:

- door is now unlocked, another may go in now

Mutex - Mutual Exclusion

- A mutex is a type used to enforce *mutual exclusion*, i.e., a critical section
- Mutexes are often called locks
  - To be very precise, mutexes are one kind of lock, there are others (read/write locks, reentrant locks, etc.), but we can just call them locks in this course, usually "lock" means "mutex"
- When a thread locks a mutex
  - If the lock is unlocked the thread takes the lock and continues execution
  - If the lock is locked, the thread blocks and waits until the lock is unlocked
  - If multiple threads are waiting for a lock they all wait until lock is unlocked, one receives lock
- When a thread unlocks a mutex
  - It continues normally; one waiting thread (if any) takes the lock and is scheduled to run
- This is a subset of the C++ mutex abstraction: nicely simple! How can we use this in our buggy program?

```
class mutex {
public:
  mutex();          // constructs the mutex to be in an unlocked state
  void lock();      // acquires the lock on the mutex, blocking until it's unlocked
  void unlock();    // releases the lock and wakes up another threads trying to lock it
};
```

Critical Sections With Mutexes

- **main** instantiates a mutex, which it passes (by reference!) to invocations of **process.**
- The **process** code uses this lock to protect **remainingImages**.
- Note we need to unlock on line 5 -- in complex code forgetting this is an easy bug

```
1 static void process(size_t id, size_t& remainingImages, mutex& counterLock) {
2   while (true) {
3     counterLock.lock();
4     if (remainingImages == 0) {
5       counterLock.unlock();
6       break;
7     }
8     processImage(remainingImages);
9     remainingImages--;
10    cout << oslock << "Thread#" << id << " processed an image (" << remainingImages
11     << " remain)." << endl << osunlock;
12    counterLock.unlock();
13  }
14  cout << oslock << "Thread#" << id << " sees no remaining images and exits."
15  << endl << osunlock;
16 }
17
18 // Create single mutex in main, pass by reference
```

Critical Sections Can Be Bottlenecks

- The way we've set it up, only one thread agent can process an image at a time!
- We can do better: serialize deciding which image to process and parallelize the actual processing
- Keep your critical sections as small as possible!

```
static void process(size_t id, size_t& remainingImages, mutex& counterLock) {
  while (true) {
    size_t myImage;

    counterLock.lock();    // Start of critical section
    if (remainingImages == 0) {
      counterLock.unlock(); // Rather keep it here, easier to check
      break;
    } else {
      myImage = remainingImages;
      remainingImages--;
      counterLock.unlock(); // end of critical section

      processImage(myImage);
      cout << oslock << "Thread#" << id << " processed an image (" << remainingImages
      << " remain)." << endl << osunlock;
    }
  }
  cout << oslock << "Thread#" << id << " sees no remaining images and exits."
  << endl << osunlock;
}
```

Problems That Might Arise

- What if **processImage** can return an error?
  - E.g., what if we need to distinguish allocating an image and processing it
  - A thread can grab the image by decrementing **remainingImages** but if it fails there's no way for another thread to retry
  - Because these are threads, if one thread has a SEGV the whole process will fail
  - A more complex approach might be to maintain an actual queue of images and allow threads (in a critical section) to push things back into the queue
- What if image processing times are *highly* variable (e.g, one image takes 100x as long as the others)?
  - Might scan images to estimate execution time and try more intelligent scheduling
- What if there's a bug in your code, such that sometimes processImage randomly enters an infinite loop?
  - Need a way to reissue an image to an idle thread
  - An infinite loop of course shouldn't occur, but when we get to networks sometimes execution time can vary by 100x for reasons outside our control

Some Types of Mutexes

- Standard **mutex**: what we've seen
  - If a thread holding the lock tries to re-lock it, deadlock
- **recursive_mutex**
  - A thread can lock the mutex multiple times, and needs to unlock it the same number of times to release it to other threads
- **timed_mutex**
  - A thread can **try_lock_for** / **try_lock_until**: if time elapses, don't take lock
  - Deadlocks if same thread tries to lock multiple times, like standard mutex
- In this class, we'll focus on just regular **mutex**

How Do Mutexes Work?

- Something we've seen a few times is that you can't read and write a variable atomically
  - But a mutex does so! If the lock is unlocked, lock it
- How does this work with caches?
  - Each core has its own cache
  - Writes are typically write-back (write to higher cache level when line is evicted), not write-through (always write to main memory) for performance
  - Caches are *coherent* -- if one core writes to a cache line that is also in another core's cache, the other core's cache line is invalidated: this can become a performance problem
- Hardware provides atomic memory operations, such as compare and swap
  - cas old, new, addr
    - If addr == old, set addr to new
  - Use this as a single bit to see if the lock is held and if not, take it
  - If the lock is held already, then enqueue yourself (in a thread safe way) and tell kernel to sleep you
  - When a node unlocks, it clears the bit and wakes up a thread

Plan For Today

- **Recap:** Threads in C++
- Races When Accessing Shared Data
- Introducing Mutexes
- **Break: Announcements**
- Dining With Philosophers

Announcements

Midterm Next Friday

- Midterm info webpage with practice materials, BlueBook download: cs110.stanford.edu/exams/midterm/
- Please notify us of any OAE accommodations by **this Monday**
- We use BlueBook, computerized testing software you will run on your laptop. If you don't have a laptop to use, let us know by **this Monday**.
- Covers through this week + assign4
- Limited power outlets for laptops
- You are allowed one back/front page of 8.5 x 11in paper for any notes you would like to bring in. We will also provide references in the exam itself as needed.

Plan For Today

- **Recap:** Threads in C++
- Races When Accessing Shared Data
- Introducing Mutexes
- **Break:** Announcements
- **Dining With Philosophers**

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 themselves as they think. Sometime they think for a long time, and sometimes they barely think at all.
      - After each philosopher has thought for a while, they proceed to eat one of their three daily meals. In order to eat, they must grab hold of two forks—one on their left, then one on their right. With two forks in hand, they chow on spaghetti to nourish their big, philosophizing brain. When they're full, they put down the forks in the same order they 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 their left, and is then forced off the processor because their 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 a 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. 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.

- 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.

Recap

- **Recap:** Threads in C++
- Races When Accessing Shared Data
- Introducing Mutexes
- **Break:** Announcements
- Dining With Philosophers

Next time: more about concurrency directives