CS110 Lecture 12: Semaphores and Multithreading Patterns

Principles of Computer Systems

Winter 2020

Stanford University

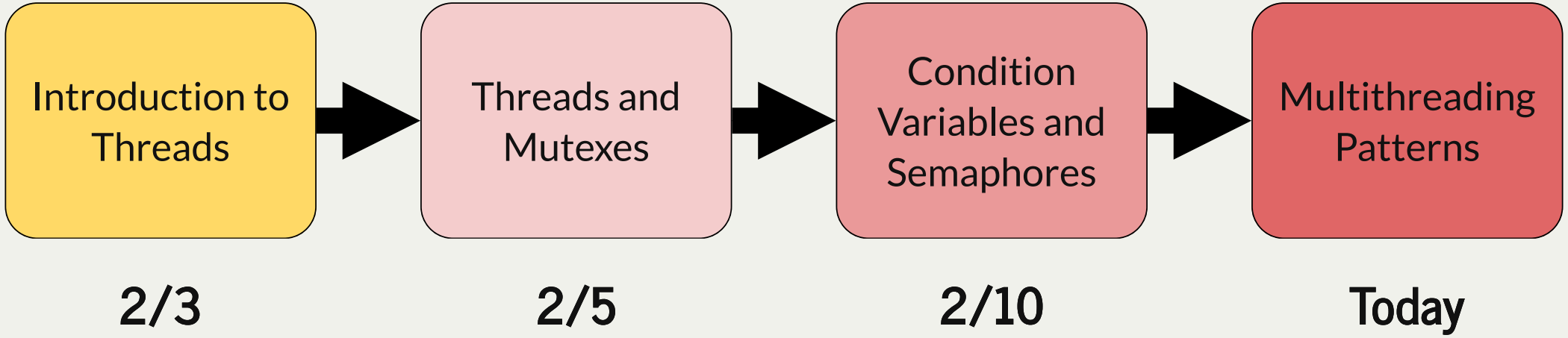
Computer Science Department

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



Today's Learning Goals

- Learn how a semaphore generalizes the "permits pattern" we previously saw
- Review the different concurrency directives (mutex, condition variable, semaphore)
- Learn how to apply semaphores to coordinate threads in different ways

Plan For Today

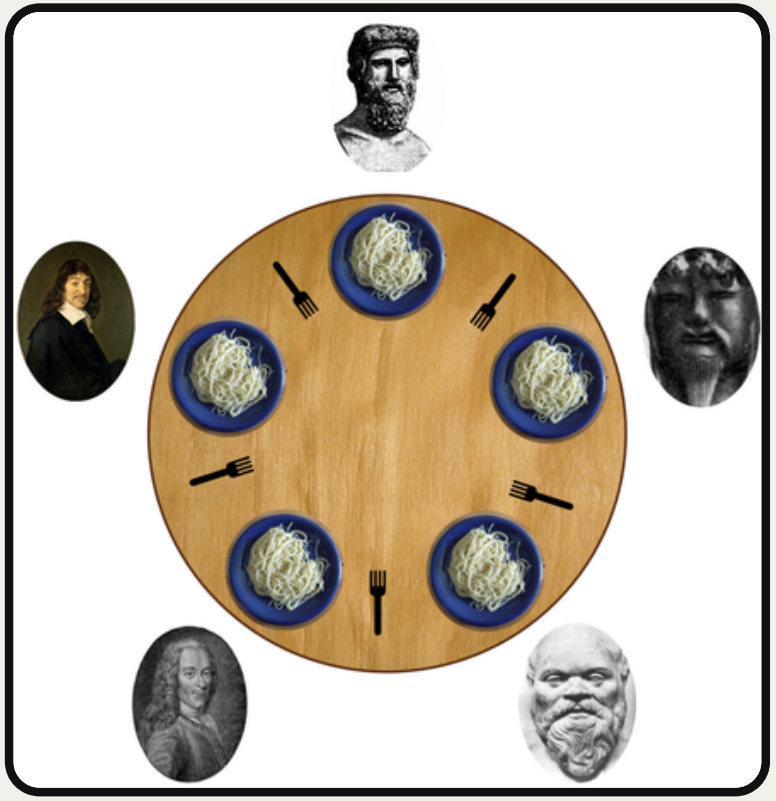
- **Recap:** Dining With Philosophers
- Semaphores
- Thread Coordination
- **Break:** Announcements
- Example: Reader-Writer
- Example: Mythbusters

Plan For Today

- **Recap:** Dining With Philosophers
- Semaphores
- Thread Coordination
- **Break:** Announcements
- Example: Reader-Writer
- Example: Mythbusters

Dining Philosophers Problem

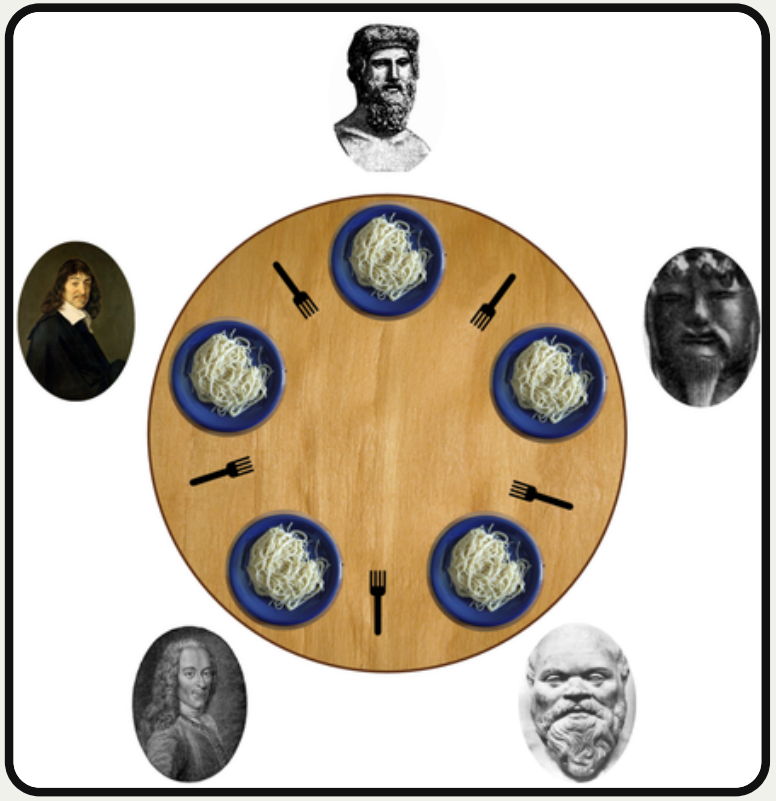
- This is a canonical multithreading example of the potential for deadlock and how to avoid it.



https://commons.wikimedia.org/wiki/File:An_illustration_of_the_dining_philosophers_problem.png

Dining Philosophers Problem

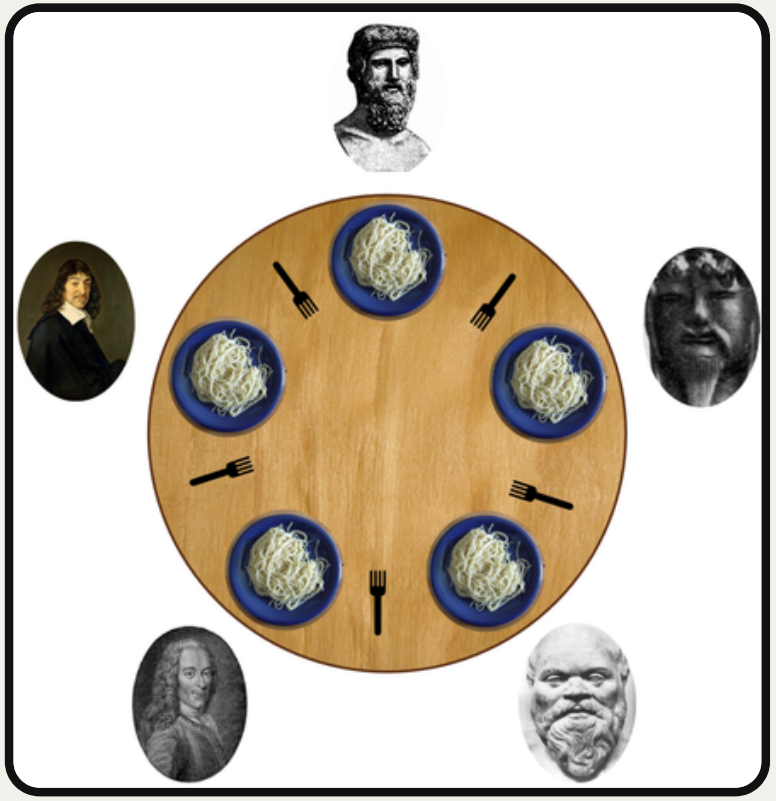
- This is a canonical multithreading example of the potential for deadlock and how to avoid it.
- Five philosophers sit around a **circular table**, eating spaghetti



https://commons.wikimedia.org/wiki/File:An_illustration_of_the_dining_philosophers_problem.png

Dining Philosophers Problem

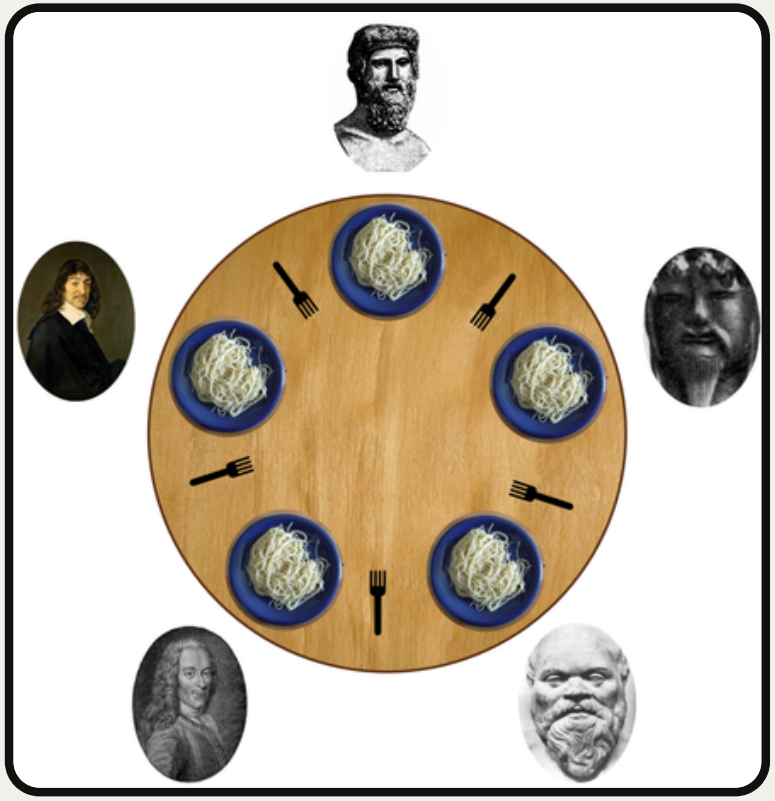
- This is a canonical multithreading example of the potential for deadlock and how to avoid it.
- Five philosophers sit around a **circular table**, eating spaghetti
- There is **one fork** for each of them



https://commons.wikimedia.org/wiki/File:An_illustration_of_the_dining_philosophers_problem.png

Dining Philosophers Problem

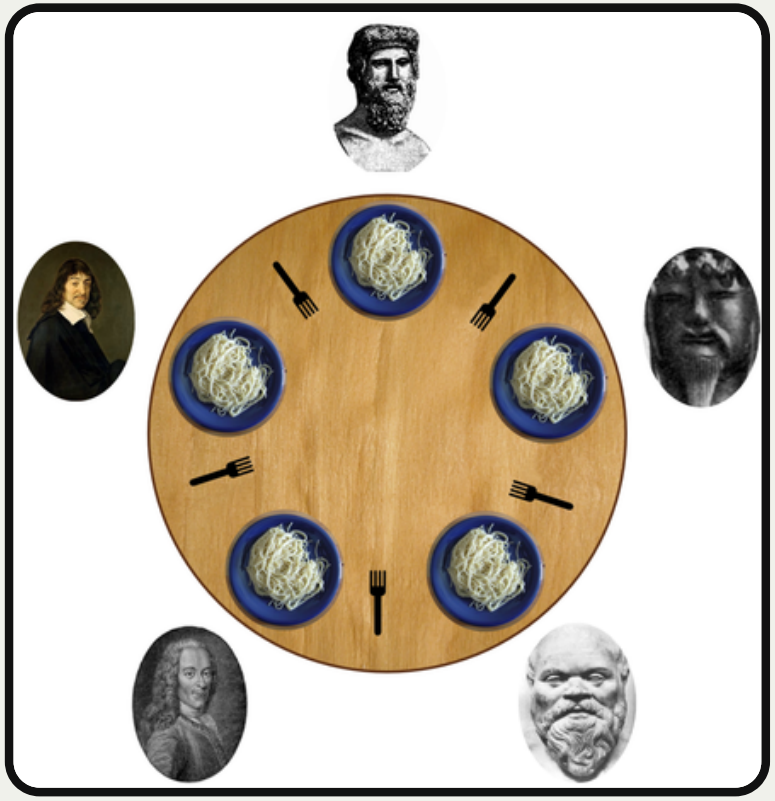
- This is a canonical multithreading example of the potential for deadlock and how to avoid it.
- Five philosophers sit around a **circular table**, eating spaghetti
- There is **one fork** for each of them
- Each philosopher **thinks, then eats**, and repeats this **three times** for their three daily meals.



https://commons.wikimedia.org/wiki/File:An_illustration_of_the_dining_philosophers_problem.png

Dining Philosophers Problem

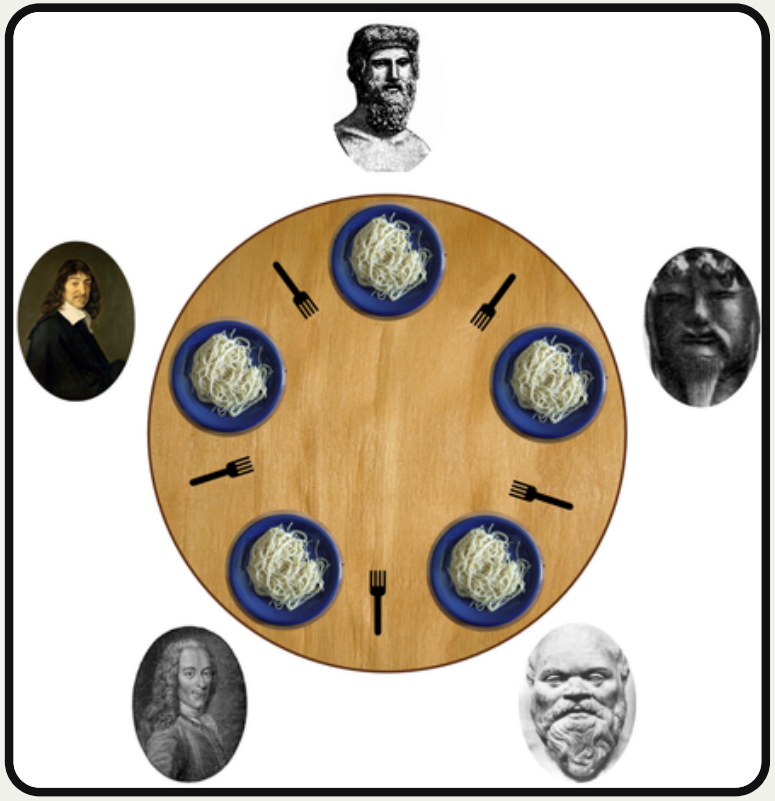
- This is a canonical multithreading example of the potential for deadlock and how to avoid it.
- Five philosophers sit around a **circular table**, eating spaghetti
- There is **one fork** for each of them
- Each philosopher **thinks, then eats**, and repeats this **three times** for their three daily meals.
- **To eat**, a philosopher must grab the fork on their left *and* the fork 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 return to thinking for a while.



https://commons.wikimedia.org/wiki/File:An_illustration_of_the_dining_philosophers_problem.png

Dining Philosophers Problem

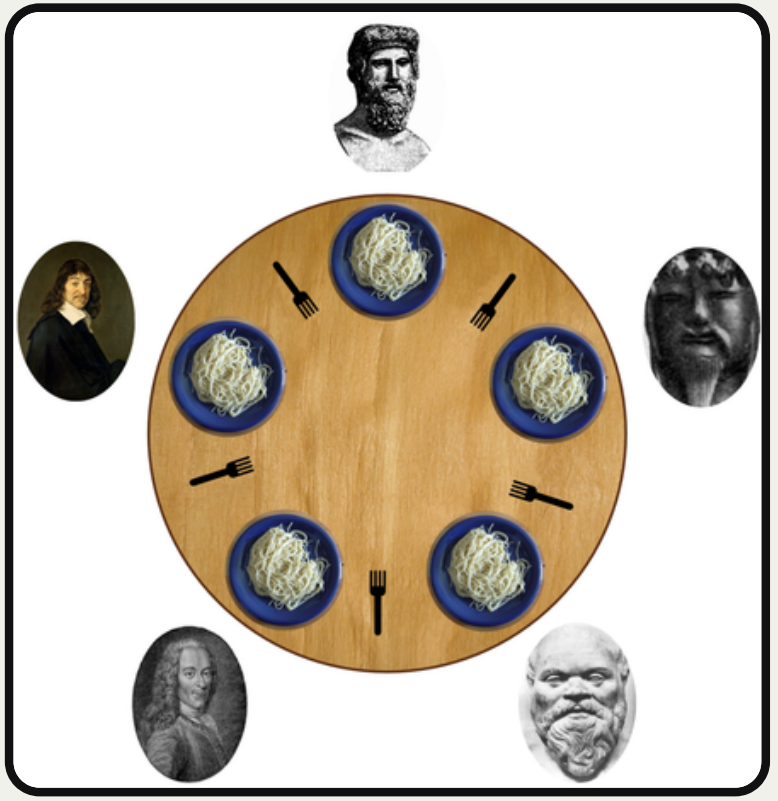
- This is a canonical multithreading example of the potential for deadlock and how to avoid it.
- Five philosophers sit around a **circular table**, eating spaghetti
- There is **one fork** for each of them
- Each philosopher **thinks, then eats**, and repeats this **three times** for their three daily meals.
- **To eat**, a philosopher must grab the fork on their left *and* the fork 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 return to thinking for a while.
- **To think**, the a philosopher keeps to themselves for some amount of time. Sometimes they think for a long time, and sometimes they barely think at all.



https://commons.wikimedia.org/wiki/File:An_illustration_of_the_dining_philosophers_problem.png

Dining Philosophers Problem

- This is a canonical multithreading example of the potential for deadlock and how to avoid it.
- Five philosophers sit around a **circular table**, eating spaghetti
- There is **one fork** for each of them
- Each philosopher **thinks, then eats**, and repeats this **three times** for their three daily meals.
- **To eat**, a philosopher must grab the fork on their left *and* the fork 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 return to thinking for a while.
- **To think**, the a philosopher keeps to themselves for some amount of time. Sometimes they think for a long time, and sometimes they barely think at all.
- Let's take our first attempt. (The full program is right here.)



https://commons.wikimedia.org/wiki/File:An_illustration_of_the_dining_philosophers_problem.png

Dining Philosophers Problem

A philosopher thinks, then eats, and repeats this three times.

- **think** is modeled as sleeping the thread for some amount of time

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

Dining Philosophers Problem

A philosopher thinks, then eats, and repeats this three times.

- **eat** is modeled as grabbing the two forks, sleeping for some amount of time, and putting the forks down.

```
1 static void eat(size_t id, mutex& left, mutex& right) {
2   left.lock();
3   right.lock();
4   cout << oslock << id << " starts eating om nom nom nom." << endl << osunlock;
5   sleep_for(getEatTime());
6   cout << oslock << id << " all done eating." << endl << osunlock;
7   left.unlock();
8   right.unlock();
9 }
```

Dining Philosophers Problem

A philosopher thinks, then eats, and repeats this three times.

- **think** is modeled as waiting for permission, then grabbing the two forks, sleeping for some amount of time, putting the forks down and finally granting permission for another to eat.

```
1  static void eat(size_t id, mutex& left, mutex& right, size_t& permits, mutex& permitsLock) {
2    waitForPermission(permits, permitsLock);
3
4    left.lock();
5    right.lock();
6    cout << oslock << id << " starts eating om nom nom nom." << endl << osunlock;
7    sleep_for(getEatTime());
8    cout << oslock << id << " all done eating." << endl << osunlock;
9
10   grantPermission(permits, permitsLock);
11
12   left.unlock();
13   right.unlock();
14 }
```

grantPermission

To return a permit, increment by 1 and continue

```
1 static void grantPermission(size_t& permits, mutex& permitsLock) {
2   permitsLock.lock();
3   permits++;
4   permitsLock.unlock();
5 }
```

waitForPermission

- How do we implement **waitForPermission**?
- Recall:
  - "If there are permits available (count > 0) then decrement by 1 and continue"
  - "If there are no permits available (count == 0) then block until a permit is available"

waitForPermission

- How do we implement **waitForPermission**?
- Recall:
  - "If there are permits available (count > 0) then decrement by 1 and continue"
  - "If there are no permits available (count == 0) then block until a permit is available"

```
1  static void waitForPermission(size_t& permits, mutex& permitsLock) {
2    while (true) {
3      permitsLock.lock();
4      if (permits > 0) break;
5      permitsLock.unlock();
6      sleep_for(10);
7    }
8    permits--;
9    permitsLock.unlock();
10 }
```

waitForPermission

- How do we implement **waitForPermission**?
- Recall:
  - "If there are permits available (count > 0) then decrement by 1 and continue"
  - "If there are no permits available (count == 0) then block until a permit is available"

```
1  static void waitForPermission(size_t& permits, mutex& permitsLock) {
2    while (true) {
3      permitsLock.lock();
4      if (permits > 0) break;
5      permitsLock.unlock();
6      sleep_for(10);
7    }
8    permits--;
9    permitsLock.unlock();
10 }
```

Problem: this is busy waiting!

It would be nice if....someone could let us know when they return their permit. Then, we can sleep until this happens.

Plan For Today

A **condition variable** is a variable that can be shared across threads and used for one thread to notify to another thread when something happens. A thread can also use this to *wait* until it is notified by another thread.

Plan For Today

A **condition variable** is a variable that can be shared across threads and used for one thread to notify to another thread when something happens. A thread can also use this to *wait* until it is notified by another thread.

```
class condition_variable_any {
public:
   void wait(mutex& m);
   template <typename Pred> void wait(mutex& m, Pred pred);
   void notify_one();
   void notify_all();
};
```

Plan For Today

A **condition variable** is a variable that can be shared across threads and used for one thread to notify to another thread when something happens. A thread can also use this to *wait* until it is notified by another thread.

```
class condition_variable_any {
public:
   void wait(mutex& m);
   template <typename Pred> void wait(mutex& m, Pred pred);
   void notify_one();
   void notify_all();
};
```

- We can call **wait** to sleep until another thread signals this condition variable.

Plan For Today

A **condition variable** is a variable that can be shared across threads and used for one thread to notify to another thread when something happens. A thread can also use this to *wait* until it is notified by another thread.

```
class condition_variable_any {
public:
   void wait(mutex& m);
   template <typename Pred> void wait(mutex& m, Pred pred);
   void notify_one();
   void notify_all();
};
```

- We can call **wait** to sleep until another thread signals this condition variable.
- We can call **notify_all** to send a signal to waiting threads.

grantPermission

Full program: here

- For **grantPermission**, we must signal when we make permits go from 0 to 1.

```
1 static void grantPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
2   m.lock();
3   permits++;
4   if (permits == 1) cv.notify_all();
5   m.unlock();
6 }
7
```

waitForPermission

Full program: here

- For **waitForPermission**, if no permits are available we must wait until one becomes available.

```
1 static void waitForPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
2   m.lock();
3   while (permits == 0) cv.wait(m);
4   permits--;
5   m.unlock();
6 }
```

Here's what **cv.wait** does:

- it puts the caller to sleep *and* unlocks the given lock, all atomically (it sleeps, then unlocks)
- it wakes up when the cv is signaled
- upon waking up, it tries to acquire the given lock (and blocks until it's able to do so)
- then, cv.wait returns

waitForPermission

Full program: here

- An alternate form of **wait** takes a lambda function that returns true when we should **stop looping** around cv.wait.

```
1 static void waitForPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
2   m.lock();
3   // while (permits == 0) cv.wait(m);
4   cv.wait(m, [&permits] { return permits > 0; });
5   permits--;
6   m.unlock();
7 }
```

waitForPermission

Full program: here

- An alternate form of **wait** takes a lambda function that returns true when we should **stop looping** around cv.wait.

```
1 static void waitForPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
2   m.lock();
3   // while (permits == 0) cv.wait(m);
4   cv.wait(m, [&permits] { return permits > 0; });
5   permits--;
6   m.unlock();
7 }
```

- Here's how this is implemented under the hood:

waitForPermission

Full program: here

- An alternate form of **wait** takes a lambda function that returns true when we should **stop looping** around cv.wait.

```
1 static void waitForPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
2   m.lock();
3   // while (permits == 0) cv.wait(m);
4   cv.wait(m, [&permits] { return permits > 0; });
5   permits--;
6   m.unlock();
7 }
```

- Here's how this is implemented under the hood:

```
1 template <Predicate pred>
2 void condition_variable_any::wait(mutex& m, Pred pred) {
3   while (!pred()) wait(m);
4 }
5
```

Lock Guards

- 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.)
- Here's how we could use it in waitForPermission and grantPermission:

```
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();
}
```

Plan For Today

- **Recap:** Dining With Philosophers
- Semaphores
- Thread Coordination
- **Break:** Announcements
- Example: Reader-Writer
- Example: Mythbusters

Semaphore

This "permission slip" pattern with signaling is a very common pattern:

- Have a **counter**, **mutex** and **condition_variable_any** to track some permission slips
- Thread-safe way to grant permission and to wait for permission (aka sleep)
- But, it's cumbersome to need 3 variables to implement this - is there a better way?
- A *semaphore* is a single variable type that encapsulates all this functionality

Semaphore

A semaphore is a variable type that lets you manage a count of finite resources.

- You initialize the semaphore with the count of resources to start with
- You can request permission via **semaphore::wait()** - aka waitForPermission
- You can grant permission via **semaphore::signal()** - aka grantPermission
- This is a standard definition, but not included in the C++ standard libraries. Why? Perhaps because it's easily built in terms of other supported constructs.

```
1  class semaphore {
2   public:
3    semaphore(int value = 0);
4    void wait();
5    void signal();
6
7   private:
8    int value;
9    std::mutex m;
10   std::condition_variable_any cv;
11 }
```

Semaphore

A semaphore is a variable type that lets you manage a count of finite resources.

- You initialize the semaphore with the count of resources to start with

```
semaphore permits(5); // this will allow five permits
```

- When a thread wants to use a permit, it first **`wait`**s for the permit, and then **`signal`**s when it is done using a permit:

```
permits.wait(); // if five other threads currently hold permits, this will block

// only five threads can be here at once

permits.signal(); // if other threads are waiting, a permit will be available
```

- A **`mutex`** is kind of like a special case of a semaphore with one permit, but you should use a **`mutex`** in that case as it is simpler and more efficient. Additionally, the benefit of a mutex is that it can *only* be released by the lock-holder.

Semaphore - wait

A semaphore is a variable type that lets you manage a count of finite resources.

- You can request permission via **semaphore::wait()** - aka waitForPermission

```
1  class semaphore {
2   public:
3    semaphore(int value = 0);
4    void wait();
5    void signal();
6
7   private:
8    int value;
9    std::mutex m;
10   std::condition_variable_any cv;
11 }
```

```
1 void semaphore::wait() {
2   lock_guard<mutex> lg(m);
3   cv.wait(m, [this]{ return value > 0; });
4   value--;
5 }
```

- Note: we can't capture **value** directly in a lambda (functions aren't normally entitled to private object state), so instead we must capture **this** (a reference to ourself) and access **value** that way.

Semaphore - signal

A semaphore is a variable type that lets you manage a count of finite resources.

- You can grant permission via **semaphore::signal()** - aka grantPermission

```
1  class semaphore {
2   public:
3    semaphore(int value = 0);
4    void wait();
5    void signal();
6
7   private:
8    int value;
9    std::mutex m;
10   std::condition_variable_any cv;
11 }
```

```
1 void semaphore::signal() {
2   lock_guard<mutex> lg(m);
3   value++;
4   if (value == 1) cv.notify_all();
5 }
```

And Now....We Eat!

Here's our final version of the **dining-philosophers**.

- We replace **size_t**, **mutex**, and **condition_variable_any** with a single **semaphore**.
- It updates the thread constructors to accept a single reference to that **semaphore**.

```
1  static void philosopher(size_t id, mutex& left, mutex& right, semaphore& permits) {
2    for (size_t i = 0; i < 3; i++) {
3      think(id);
4      eat(id, left, right, permits);
5    }
6  }
7
8  int main(int argc, const char *argv[]) {
9    // NEW
10   semaphore permits(4);
11
12   mutex forks[5];
13   thread philosophers[5];
14   for (size_t i = 0; i < 5; i++) {
15     mutex& left = forks[i], & right = forks[(i + 1) % 5];
16     philosophers[i] = thread(philosopher, i, ref(left), ref(right), ref(permits));
17   }
18   for (thread& p: philosophers) p.join();
19   return 0;
20 }
```

And Now....We Eat!

eat now relies on the **semaphore** instead of calling **waitForPermission** and **grantPermission**.

```
1  static void eat(size_t id, mutex& left, mutex& right, semaphore& permits) {
2    // NEW
3    permits.wait();
4
5    left.lock();
6    right.lock();
7    cout << oslock << id << " starts eating om nom nom nom." << endl << osunlock;
8    sleep_for(getEatTime());
9    cout << oslock << id << " all done eating." << endl << osunlock;
10
11   // NEW
12   permits.signal();
13
14   left.unlock();
15   right.unlock();
16 }
```

Thought Questions:

- Could/should we switch the order of lines 14-15, so that **right.unlock()** precedes **left.unlock()**?
- Instead of a semaphore, could we use a **mutex** to bundle the calls to **left.lock()** and **right.lock()** into a critical region?
- Could we call **permits.signal()** in between **right.lock()** and the first **cout** statement?

And Now....We Eat!

```
1  static void eat(size_t id, mutex& left, mutex& right, semaphore& permits) {
2    permits.wait();
3    left.lock();
4    right.lock();
5    cout << oslock << id << " starts eating om nom nom nom." << endl << osunlock;
6    sleep_for(getEatTime());
7    cout << oslock << id << " all done eating." << endl << osunlock;
8    permits.signal();
9    left.unlock();
10   right.unlock();
11 }
```

Thought Questions:

- Could/should we switch the order of lines 14-15, so that **right.unlock()** precedes **left.unlock()**?
  - Yes, but it is arbitrary
- Instead of a semaphore, could we use a **mutex** to bundle the calls to **left.lock()** and **right.lock()** into a critical region? Yes!
- Could we call **permits.signal()** in between **right.lock()** and the first **cout** statement?
  - Yes, but others will still have to wait for forks

More On Semaphores

Question: what would a **`semaphore`** initialized with 0 mean?

```
semaphore permits(0);
```

More On Semaphores

Question: what would a **`semaphore`** initialized with 0 mean?

```
semaphore permits(0);
```

- In this case, we don't have *any* permits!

More On Semaphores

Question: what would a **`semaphore`** initialized with 0 mean?

```
semaphore permits(0);
```

- In this case, we don't have *any* permits!
- So, **`permits.wait()`** *always* has to wait for a signal, and will never stop waiting until that signal is received.

More On Semaphores

Question: what would a **`semaphore`** initialized with 0 mean?

```
semaphore permits(0);
```

- In this case, we don't have *any* permits!
- So, **`permits.wait()`** *always* has to wait for a signal, and will never stop waiting until that signal is received.

Question: what would a **`semaphore`** initialized with a negative number mean?

More On Semaphores

Question: what would a **`semaphore`** initialized with 0 mean?

```
semaphore permits(0);
```

- In this case, we don't have *any* permits!
- So, **`permits.wait()`** *always* has to wait for a signal, and will never stop waiting until that signal is received.

Question: what would a **`semaphore`** initialized with a negative number mean?

```
semaphore permits(-9);
```

More On Semaphores

Question: what would a **`semaphore`** initialized with 0 mean?

```
semaphore permits(0);
```

- In this case, we don't have *any* permits!
- So, **`permits.wait()`** *always* has to wait for a signal, and will never stop waiting until that signal is received.

Question: what would a **`semaphore`** initialized with a negative number mean?

```
semaphore permits(-9);
```

- In this case, the semaphore would have to *reach 1* before the wait would stop waiting. You might want to wait until a bunch of threads finished before a final thread is allowed to continue.

More On Semaphores

Negative semaphores example (full program here):

```
1  void writer(int i, semaphore &s) {
2      cout << oslock << "Sending signal " << i << endl << osunlock;
3      s.signal();
4  }
5
6  void read_after_ten(semaphore &s) {
7      s.wait();
8      cout << oslock << "Got enough signals to continue!" << endl << osunlock;
9  }
10
11 int main(int argc, const char *argv[]) {
12     semaphore negSemaphore(-9);
13     thread readers[10];
14     for (size_t i = 0; i < 10; i++) {
15         readers[i] = thread(writer, i, ref(negSemaphore));
16     }
17     thread r(read_after_ten, ref(negSemaphore));
18     for (thread &t : readers) t.join();
19     r.join();
20     return 0;
21 }
```

Plan For Today

- **Recap:** Dining With Philosophers
- Semaphores
- Thread Coordination
- **Break:** Announcements
- Example: Reader-Writer
- Example: Mythbusters

Thread Coordination

- **semaphore::wait** and **semaphore::signal** can be used to support **thread rendezvous**.
- Thread rendezvous allows one thread to stall—via **semaphore::wait**—until another thread calls **semaphore::signal**, e.g. the signaling thread prepared some data that the waiting thread needs to continue.
- Generalization of **thread::join**

Plan For Today

- **Recap:** Dining With Philosophers
- Semaphores
- Thread Coordination
- **Break:** Announcements
- Example: Reader-Writer
- Example: Mythbusters

Announcements

Midterm This Friday

- Review Session Materials Posted
- Reference Sheet Posted

Assignment 5 Out Tomorrow

- Focus on assign4 and studying between now and the midterm :-)

Mid-Lecture Checkin

We can now answer the following questions:

- What does a condition variable do?
- How is a semaphore implemented?
- What concurrency patterns are unlocked by initializing semaphores with positive numbers? Negative numbers? Zero?

Plan For Today

- **Recap:** Dining With Philosophers
- Semaphores
- Thread Coordination
- **Break:** Announcements
- Example: Reader-Writer
- Example: Mythbusters

Reader-Writer

Let's implement a program that requires thread rendezvous with semaphores. First, we'll look at a version **without** semaphores to see why they are necessary.

Reader-Writer

Let's implement a program that requires thread rendezvous with semaphores. First, we'll look at a version **without** semaphores to see why they are necessary.

- The **reader-writer** pattern/program spawns 2 threads: one writer (publishes content to a shared buffer) and one reader (reads from shared buffer when content is available)

Reader-Writer

Let's implement a program that requires thread rendezvous with semaphores. First, we'll look at a version **without** semaphores to see why they are necessary.

- The **reader-writer** pattern/program spawns 2 threads: one writer (publishes content to a shared buffer) and one reader (reads from shared buffer when content is available)
- Common pattern! E.g. web server publishes content over a dedicated communication channel, and the web browser consumes that content.

Reader-Writer

Let's implement a program that requires thread rendezvous with semaphores. First, we'll look at a version **without** semaphores to see why they are necessary.

- The **reader-writer** pattern/program spawns 2 threads: one writer (publishes content to a shared buffer) and one reader (reads from shared buffer when content is available)
- Common pattern! E.g. web server publishes content over a dedicated communication channel, and the web browser consumes that content.
- More complex version: multiple readers, similar to how a web server handles many incoming requests (puts request in buffer, readers each read and process requests)

Reader-Writer

Let's implement a program that requires thread rendezvous with semaphores. First, we'll look at a version **without** semaphores to see why they are necessary.

- The **reader-writer** pattern/program spawns 2 threads: one writer (publishes content to a shared buffer) and one reader (reads from shared buffer when content is available)
- Common pattern! E.g. web server publishes content over a dedicated communication channel, and the web browser consumes that content.
- More complex version: multiple readers, similar to how a web server handles many incoming requests (puts request in buffer, readers each read and process requests)
- Demo (confused-reader-writer.cc)

Reader-Writer

```
1  static void writer(char buffer[]) {
2    cout << oslock << "Writer: ready to write." << endl << osunlock;
3    for (size_t i = 0; i < 320; i++) { // 320 is 40 cycles around the circular buffer of length 8
4      char ch = prepareData();
5      buffer[i % 8] = ch;
6      cout << oslock << "Writer: published data packet with character '"
7           << ch << "'." << endl << osunlock;
8    }
9  }
10
11 static void reader(char buffer[]) {
12   cout << oslock << "\t\tReader: ready to read." << endl << osunlock;
13   for (size_t i = 0; i < 320; i++) { // 320 is 40 cycles around the circular buffer of length 8
14     char ch = buffer[i % 8];
15     processData(ch);
16     cout << oslock << "\t\tReader: consumed data packet " << "with character '"
17          << ch << "'." << endl << osunlock;
18   }
19 }
20
21 int main(int argc, const char *argv[]) {
22   char buffer[8];
23   thread w(writer, buffer);
24   thread r(reader, buffer);
25   w.join();
26   r.join();
27   return 0;
28 }
29
```

The full program is right here

Reader-Writer

Reader-Writer

- Both threads share the same buffer, so they agree where content is stored (think of buffer like state for a pipe or a connection between client and server)

Reader-Writer

- Both threads share the same buffer, so they agree where content is stored (think of buffer like state for a pipe or a connection between client and server)
- The **writer** 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.

Reader-Writer

- Both threads share the same buffer, so they agree where content is stored (think of buffer like state for a pipe or a connection between client and server)
- The **writer** 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.
- **Problem:** each thread runs independently, without knowing how much progress the other has made.
  - Example: no way for the **reader** to know 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
  - Example: the **writer** could loop around and overwrite content that the **reader** has not yet consumed.

Reader-Writer Constraints

Goal: we must encode resource constraints into our program.

What constraint(s) should we add to our program?

How can we model these constraint(s)?

Reader-Writer Constraints

Goal: we must encode resource constraints into our program.

What constraint(s) should we add to our program?

- A reader should not read until something is available to read

How can we model these constraint(s)?

Reader-Writer Constraints

Goal: we must encode resource constraints into our program.

What constraint(s) should we add to our program?

- A reader should not read until something is available to read
- A writer should not write until there is space available to write

How can we model these constraint(s)?

Reader-Writer Constraints

Goal: we must encode resource constraints into our program.

What constraint(s) should we add to our program?

- A reader should not read until something is available to read
- A writer should not write until there is space available to write

How can we model these constraint(s)?

- One semaphore to manage open slots

Reader-Writer Constraints

Goal: we must encode resource constraints into our program.

What constraint(s) should we add to our program?

- A reader should not read until something is available to read
- A writer should not write until there is space available to write

How can we model these constraint(s)?

- One semaphore to manage open slots
- One semaphore to manage readable slots

Reader-Writer Constraints

What might this look like in code?

- 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.
- Let's try it!

Plan For Today

- **Recap:** Dining With Philosophers
- Semaphores
- Thread Coordination
- **Break:** Announcements
- Example: Reader-Writer
- Example: Mythbusters

semaphore

- Implementing **myth-buster**!
  - The **myth-buster** is a command line utility that polls all 16 **myth** machines to determine which is the least loaded.
    - By least loaded, we mean the **myth** machine that's running the fewest number of CS110 student processes.
    - Our **myth-buster** application is representative of the type of thing load balancers (e.g. **myth.stanford.edu**, **www.facebook.com**, or **www.netflix.com**) run to determine which internal server your request should forward to.
  - The overall architecture of the program looks like that below. We'll present various ways to implement **compileCS110ProcessCountMap.**

```
static const char *kCS110StudentIDsFile = "studentsunets.txt";
int main(int argc, char *argv[]) {
  unordered_set<string> cs110Students;
  readStudentFile(cs110Students, argv[1] != NULL ? argv[1] : kCS110StudentIDsFile);
  map<int, int> processCountMap;
  compileCS110ProcessCountMap(cs110Students, processCountMap);
  publishLeastLoadedMachineInfo(processCountMap);
  return 0;
}
```

semaphore

- Implementing **myth-buster**!

```
static const char *kCS110StudentIDsFile = "studentsunets.txt";
int main(int argc, char *argv[]) {
  unordered_set<string> cs110Students;
  readStudentFile(cs110Students, argv[1] != NULL ? argv[1] : kCS110StudentIDsFile);
  map<int, int> processCountMap;
  compileCS110ProcessCountMap(cs110Students, processCountMap);
  publishLeastLoadedMachineInfo(processCountMap);
  return 0;
}
```

  - **readStudentFile** updates **cs110Students** to house the SUNet IDs of all students currently enrolled in CS110. There's nothing interesting about its implementation, so I don't even show it (though you can see its implementation right here).
  - **compileCS110ProcessCountMap** is more interesting, since it uses networking—our first networking example!—to poll all 16 **myth**s and count CS110 student processes.
  - **processCountMap** is updated to map **myth** numbers (e.g. 61) to process counts (e.g. 9).
  - **publishLeastLoadedMachineInfo** traverses **processCountMap** and and identifies the least loaded **myth**.

semaphore

- The networking details are hidden and packaged in a library routine with this prototype:

```
int getNumProcesses(int num, const unordered_set<std::string>& sunetIDs);
```

- **num** is the myth number (e.g. 54 for **myth54**) and **sunetIDs** is a hashset housing the SUNet IDs of all students currently enrolled in CS110 (according to our **/usr/class/cs110/repos/assign4** directory).
- Here is the sequential implementation of a **compileCS110ProcessCountMap**, which is very brute force and CS106B-ish:

```
static const int kMinMythMachine = 51;
static const int kMaxMythMachine = 66;
static void compileCS110ProcessCountMap(const unordered_set<string>& sunetIDs,
                                        map<int, int>& processCountMap) {
  for (int num = kMinMythMachine; num <= kMaxMythMachine; num++) {
    int numProcesses = getNumProcesses(num, sunetIDs);
    if (numProcesses >= 0) {
      processCountMap[num] = numProcesses;
      cout << "myth" << num << " has this many CS110-student processes: " << numProcesses <<
    }
  }
}
```

semaphore

- Here are two sample runs of **myth-buster-sequential**, which polls each of the **myth**s in sequence (i.e. without concurrency).

```
poohbear@myth61$ time ./myth-buster-sequential
myth51 has this many CS110-student processes: 62
myth52 has this many CS110-student processes: 133
myth53 has this many CS110-student processes: 116
myth54 has this many CS110-student processes: 90
myth55 has this many CS110-student processes: 117
myth56 has this many CS110-student processes: 64
myth57 has this many CS110-student processes: 73
myth58 has this many CS110-student processes: 92
myth59 has this many CS110-student processes: 109
myth60 has this many CS110-student processes: 145
myth61 has this many CS110-student processes: 106
myth62 has this many CS110-student processes: 126
myth63 has this many CS110-student processes: 317
myth64 has this many CS110-student processes: 119
myth65 has this many CS110-student processes: 150
myth66 has this many CS110-student processes: 133
Machine least loaded by CS110 students: myth51
Number of CS110 processes on least loaded machine: 62
poohbear@myth61$
```

```
poohbear@myth61$ time ./myth-buster-sequential
myth51 has this many CS110-student processes: 59
myth52 has this many CS110-student processes: 135
myth53 has this many CS110-student processes: 112
myth54 has this many CS110-student processes: 89
myth55 has this many CS110-student processes: 107
myth56 has this many CS110-student processes: 58
myth57 has this many CS110-student processes: 70
myth58 has this many CS110-student processes: 93
myth59 has this many CS110-student processes: 107
myth60 has this many CS110-student processes: 145
myth61 has this many CS110-student processes: 105
myth62 has this many CS110-student processes: 126
myth63 has this many CS110-student processes: 314
myth64 has this many CS110-student processes: 119
myth65 has this many CS110-student processes: 156
myth66 has this many CS110-student processes: 144
Machine least loaded by CS110 students: myth56
Number of CS110 processes on least loaded machine: 58
poohbear@myth61$
```

- Each call to **getNumProcesses** is slow (about half a second), so 16 calls adds up to about 16 times that. Each of the two runs took about 5 seconds.

semaphore

- Each call to **getNumProcesses** spends most of its time off the CPU, waiting for a network connection to be established.
- Idea: poll each **myth** machine in its own thread of execution. By doing so, we'd align the dead times of each **getNumProcesses** call, and the total execution time will plummet.

```
static void countCS110Processes(int num, const unordered_set<string>& sunetIDs,
                                map<int, int>& processCountMap, mutex& processCountMapLock,
                                semaphore& permits) {
  int count = getNumProcesses(num, sunetIDs);
  if (count >= 0) {
    lock_guard<mutex> lg(processCountMapLock);
    processCountMap[num] = count;
    cout << "myth" << num << " has this many CS110-student processes: " << count << endl;
  }
  permits.signal(on_thread_exit);
}

static void compileCS110ProcessCountMap(const unordered_set<string> sunetIDs,
                                        map<int, int>& processCountMap) {
  vector<thread> threads;
  mutex processCountMapLock;
  semaphore permits(8); // limit the number of threads to the number of CPUs
  for (int num = kMinMythMachine; num <= kMaxMythMachine; num++) {
    permits.wait();
    threads.push_back(thread(countCS110Processes, num, ref(sunetIDs),
                             ref(processCountMap), ref(processCountMapLock), ref(permits)));
  }
  for (thread& t: threads) t.join();
}
```

semaphore

- Here are key observations about the code on the prior slide:
  - Polling the **myth**s concurrently means updating **processCountMap** concurrently. That means we need a **mutex** to guard access to **processCountMap**.
  - The implementation of **compileCS110ProcessCountMap** wraps a **thread** around each call to **getNumProcesses** while introducing a **semaphore** to limit the number of threads to a reasonably small number.
  - Note we use an overloaded version of **signal**. This one accepts the **on_thread_exit** tag as its only argument.
    - Rather than signaling the **semaphore** right there, this version schedules the **signal** to be sent after the entire thread routine has exited, as the **thread** is being destroyed.
    - That's the correct time to really **signal** if you're using the **semaphore** to track the number of active threads.
  - This new version, called **myth-buster-concurrent**, runs in about 0.75 seconds. That's a substantial improvement.
  - The full implementation of **myth-buster-concurrent** sits right here.

Recap

- **Recap:** Dining With Philosophers
- Semaphores
- Thread Coordination
- **Break:** Announcements
- Example: Reader-Writer
- Example: Mythbusters

Next time: a trip to the ice cream store