CS110 Lecture 11: Condition Variables and Semaphores

Principles of Computer Systems

Winter 2020

Stanford University

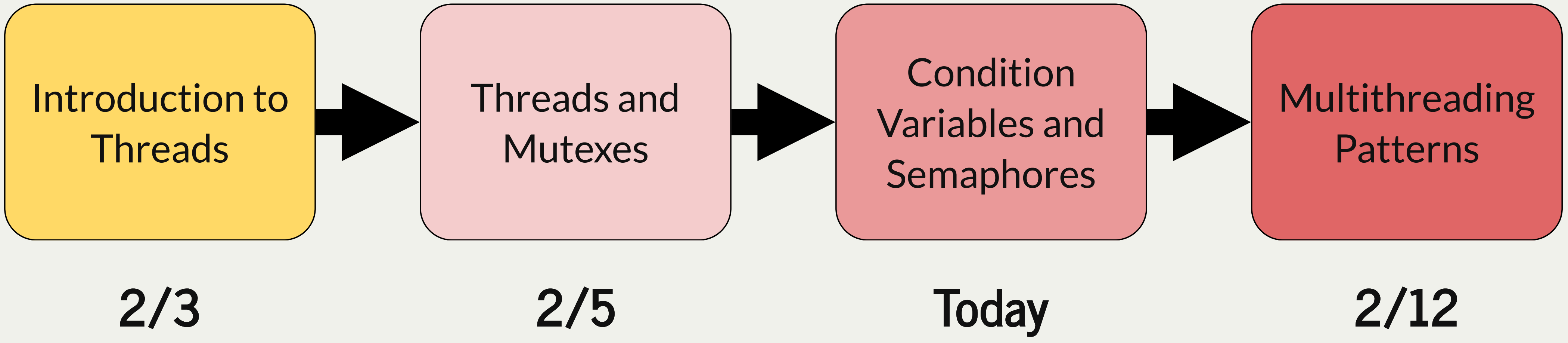
Computer Science Department

Instructors: Chris Gregg and
Nick Troccoli

PDF of this presentation

<u>**CS110 Topic 3:**</u> How can we have concurrency within a single process?

Learning About Processes



Today’s Learning Goals

- Learn how condition variables can let threads signal to each other
- Get practice with the "available permits" resource model
- Learn what a semaphore is and how it is implemented

Plan For Today

- **Recap:** Race Conditions and Mutexes
- **Recap:** Dining With Philosophers
- Encoding Resource Constraints
- Condition Variables
- **Break:** Announcements
- Semaphores
- Thread Coordination
- Example: Reader-Writer

Plan For Today

- **Recap:** Race Conditions and Mutexes
- **Recap:** Dining With Philosophers
- Encoding Resource Constraints
- Condition Variables
- **Break:** Announcements
- Semaphores
- Thread Coordination
- Example: Reader-Writer

Race Conditions and Mutexes

- 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

```
// 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;
}
```

Race Conditions and Mutexes

There is a *race condition* here!

- **Problem:** multiple threads could access remainingImages between lines 2 and 4.

```
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.

Race Conditions and Mutexes

- C++ statements are not inherently atomic - anything that takes more than 1 assembly instruction could be interleaved or interrupted.
- E.g. even **remainingImages--** takes multiple assembly instructions:

```
// gets remainingImages
0x0000000000401a9b <+36>:    mov    -0x20(%rbp),%rax
0x0000000000401a9f <+40>:    mov    (%rax),%eax

// Decrements by 1
0x0000000000401aa1 <+42>:    lea    -0x1(%rax),%edx

// Saves updated value
0x0000000000401aa4 <+45>:    mov    -0x20(%rbp),%rax
0x0000000000401aa8 <+49>:    mov    %edx,(%rax)
```

- Each core has its own registers that it has to read from
- Each thread makes a local copy of the variable before operating on it
- **Problem:** What if multiple threads do this simultaneously? They all think there's only 128 images remaining and process 128 at the same time!

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
- 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 it's 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

```
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 Can Be Bottlenecks

Goal: keep critical sections as small as possible to maximize concurrent execution

Note: we don't need to lock around printing out remainingImages - reading a size_t has no risk of corruption

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

    counterLock.lock();
    if (remainingImages == 0) {
      counterLock.unlock();
      break;
    } else {
      myImage = remainingImages;
      remainingImages--;
      counterLock.unlock();

      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;
}
```

How Do Mutexes Work?

Hardware provides atomic memory operations to build on top of: e.g. "compare and swap"

- *cas old*, *new*, *addr* - instruction that says if addr == old, set addr to new
- Idea: use this as a single bit to see if the lock is held - if not, take it - if it is, 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

Caches add an additional challenge:

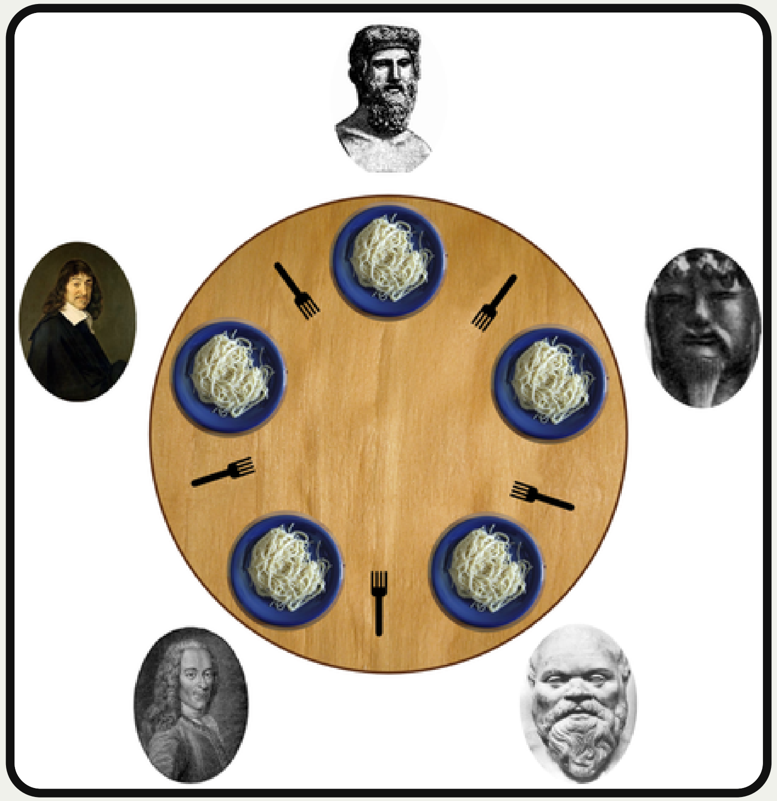
- Each core has its own cache
- Writes are typically write-back (write to higher cache level when line is evicted), not write-through (always write to main memory) for performance
- Caches are *coherent* -- if one core writes to a cache line that is also in another core's cache, the other core's cache line is invalidated: this can become a performance problem

Plan For Today

- **Recap:** Race Conditions and Mutexes
- **Recap:** Dining With Philosophers
- Encoding Resource Constraints
- Condition Variables
- **Break:** Announcements
- Semaphores
- Thread Coordination
- Example: Reader-Writer

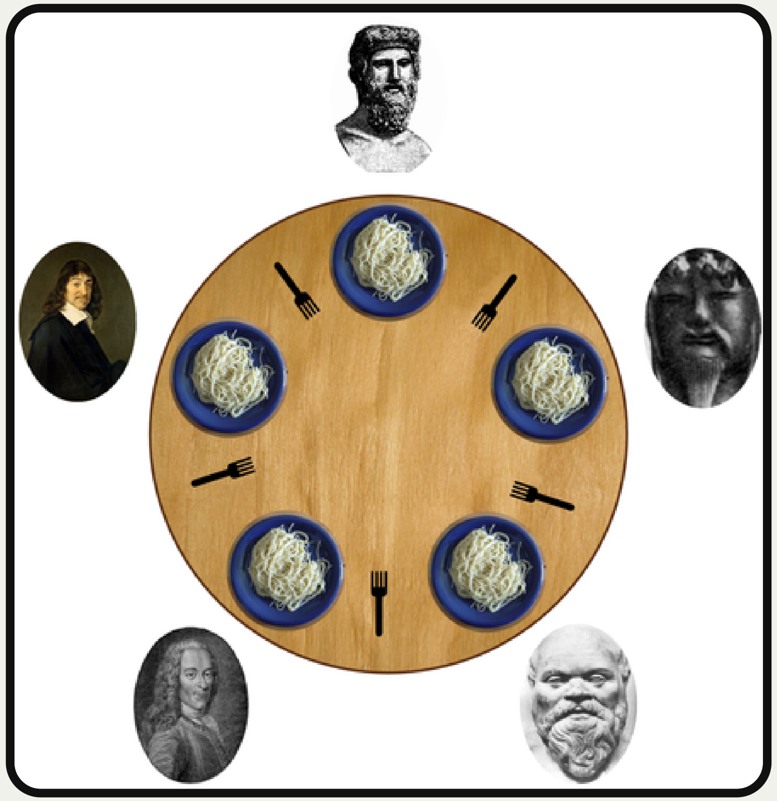
Dining Philosophers Problem

- 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

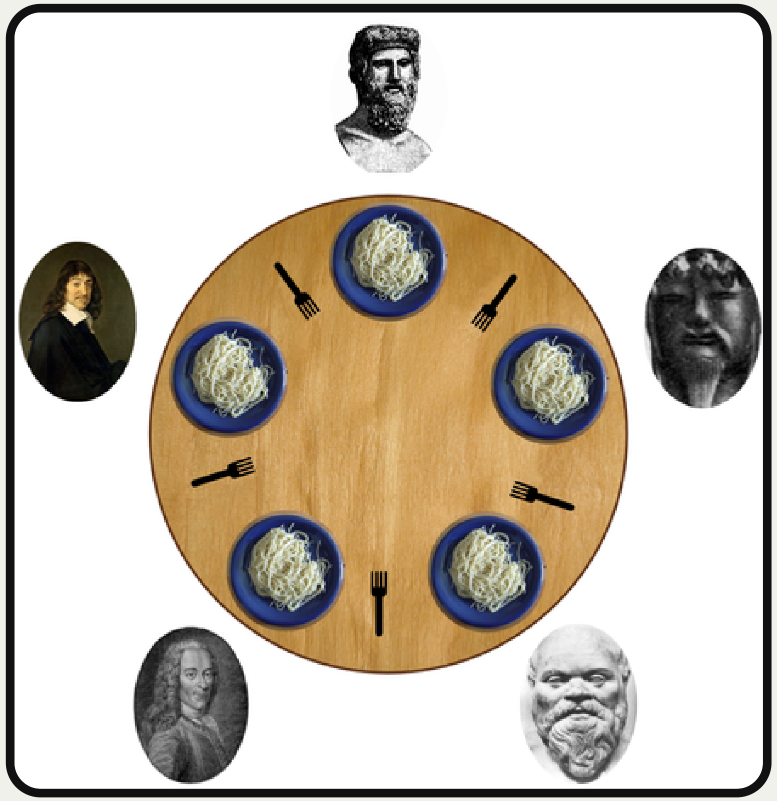
- This is a canonical multithreading example of the potential for deadlock and how to avoid it.
- Five philosophers sit around a **circular table**, eating spaghetti



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

Dining Philosophers Problem

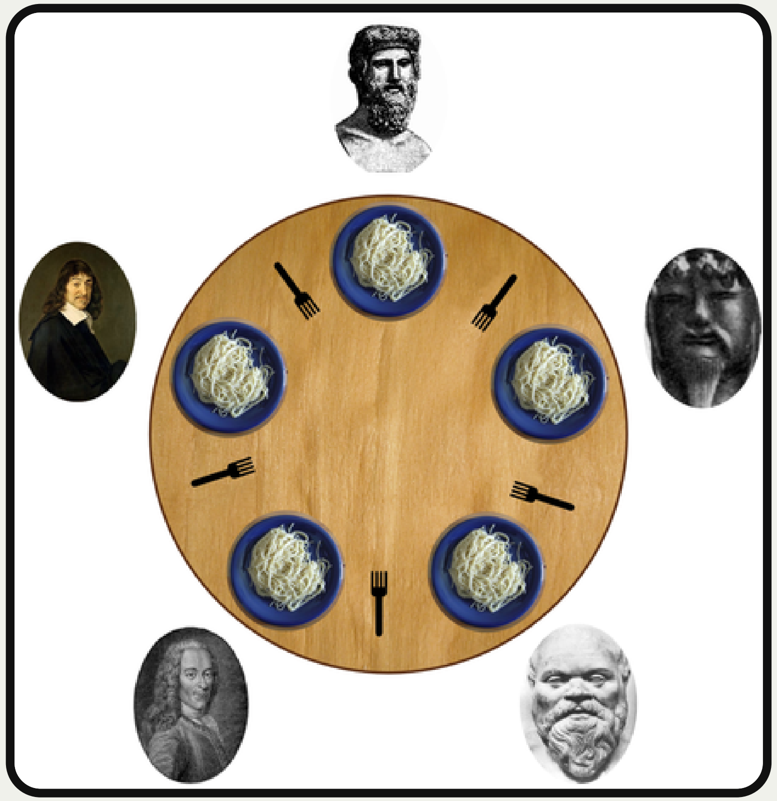
- This is a canonical multithreading example of the potential for deadlock and how to avoid it.
- Five philosophers sit around a **circular table**, eating spaghetti
- There is **one fork** for each of them



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.

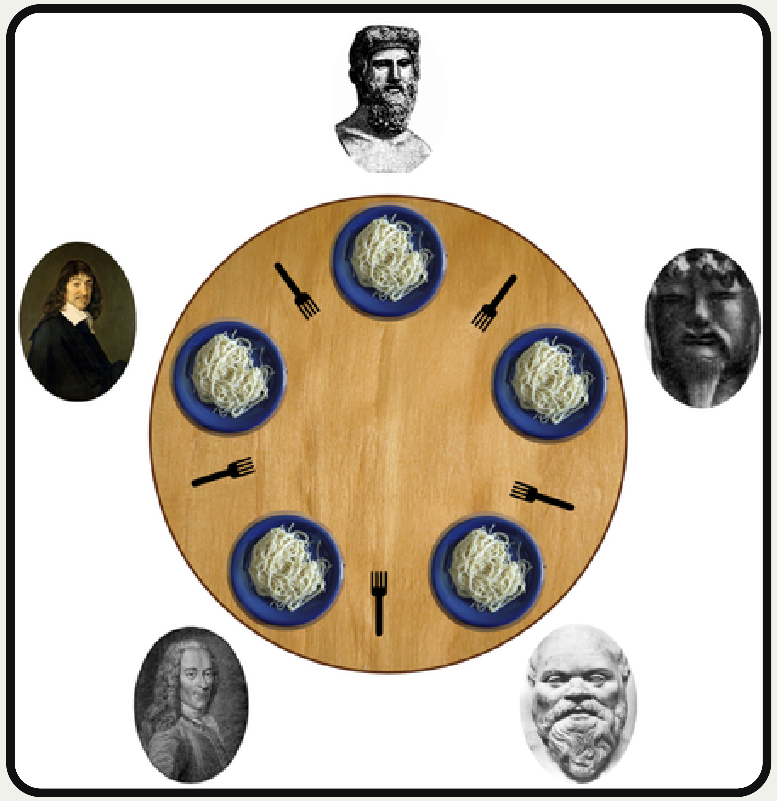


https://commons.wikimedia.org/wiki/File:An_illus

tration_of_the_dining_philosophers_problem.png

Dining Philosophers Problem

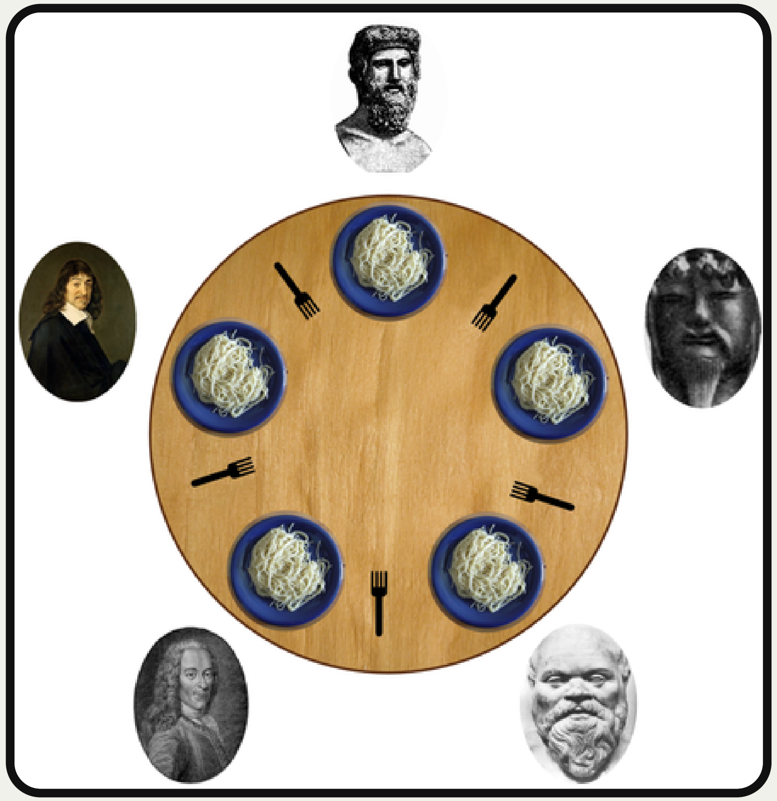
- This is a canonical multithreading example of the potential for deadlock and how to avoid it.
- Five philosophers sit around a **circular table**, eating spaghetti
- There is **one fork** for each of them
- Each philosopher **thinks, then eats**, and repeats this **three times** for their three daily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

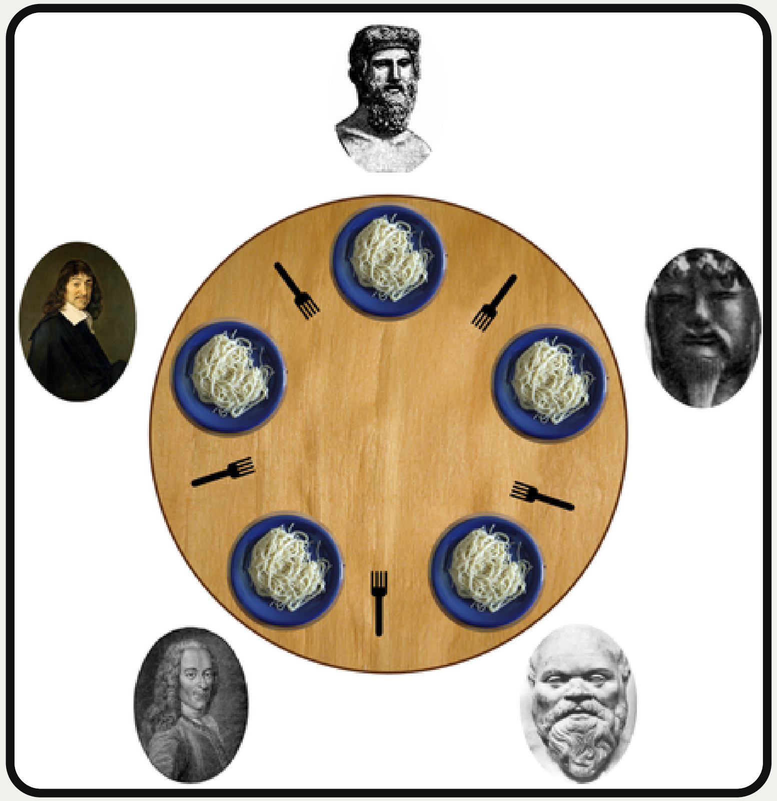
- This is a canonical multithreading example of the potential for deadlock and how to avoid it.
- Five philosophers sit around a **circular table**, eating spaghetti
- There is **one fork** for each of them
- Each philosopher **thinks, then eats**, and repeats this **three times** for their three daily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

Dining Philosophers Problem

Goal: we must encode resource constraints into our program.

Example: how many philosophers can hold a fork at the same time? **One**.

How can we encode this into our program? Let's make a mutex for each fork.

Dining Philosophers Problem

Goal: we must encode resource constraints into our program.

Example: how many philosophers can hold a fork at the same time? **One**.

How can we encode this into our program? Let's make a mutex for each fork.

- Each philosopher either holds a fork or doesn't.

Dining Philosophers Problem

Goal: we must encode resource constraints into our program.

Example: how many philosophers can hold a fork at the same time? **One**.

How can we encode this into our program? Let's make a mutex for each fork.

- Each philosopher either holds a fork or doesn't.
- A philosopher grabs a fork by locking that mutex. If the fork is available, the philosopher continues. Otherwise, it blocks until the fork becomes available and it can have it.

Dining Philosophers Problem

Goal: we must encode resource constraints into our program.

Example: how many philosophers can hold a fork at the same time? **One**.

How can we encode this into our program? Let's make a mutex for each fork.

- Each philosopher either holds a fork or doesn't.
- A philosopher grabs a fork by locking that mutex. If the fork is available, the philosopher continues. Otherwise, it blocks until the fork becomes available and it can have it.
- A philosopher puts down a fork by unlocking that mutex.

Dining Philosophers Problem

Goal: we must encode resource constraints into our program.

Example: how many philosophers can hold a fork at the same time? **One**.

How can we encode this into our program? Let's make a mutex for each fork.

- Each philosopher either holds a fork or doesn't.
- A philosopher grabs a fork by locking that mutex. If the fork is available, the philosopher continues. Otherwise, it blocks until the fork becomes available and it can have it.
- A philosopher puts down a fork by unlocking that mutex.

```
static void philosopher(size_t id, mutex& left, mutex& 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;
}
```

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;
}

static void eat(size_t id, mutex& left, mutex& right) {
  ...
}

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

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.

```
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();
}
```

Food For Thought

Food For Thought

What if: all philosophers grab their left fork and then go off the CPU?

Food For Thought

What if: all philosophers grab their left fork and then go off the CPU?

- **deadlock!** All philosophers will wait on their right fork, which will never become available.

Food For Thought

What if: all philosophers grab their left fork and then go off the CPU?

- **deadlock!** All philosophers will wait on their right fork, which will never become available.
- **Testing our hypothesis:** insert a **sleep_for** call on line 3, between getting left fork and right fork

Food For Thought

What if: all philosophers grab their left fork and then go off the CPU?

- **deadlock!** All philosophers will wait on their right fork, which will never become available.
- **Testing our hypothesis:** insert a **sleep_for** call on line 3, between getting left fork and right fork
- We should be able to insert a **sleep_for** call anywhere in a thread routine and have no concurrency issues.

Food For Thought

What if: all philosophers grab their left fork and then go off the CPU?

- **deadlock!** All philosophers will wait on their right fork, which will never become available.
- **Testing our hypothesis:** insert a **sleep_for** call on line 3, between getting left fork and right fork
- We should be able to insert a **sleep_for** call anywhere in a thread routine and have no concurrency issues.

```
 1 static void eat(size_t id, mutex& left, mutex& right) {
 2   left.lock();
 3   sleep_for(5000);  // artificially force off the processor
 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   left.unlock();
 9   right.unlock();
10 }
```

Race Conditions and Deadlock

When coding with threads, you need to ensure that:

- there are **never** any race conditions
- there's **zero** chance of deadlock; otherwise a subset of threads are forever starved
- **Race conditions** can generally be solved with **mutexes**.
    - We 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. **What does this look like?**

Plan For Today

- **Recap:** Race Conditions and Mutexes
- **Recap:** Dining With Philosophers
- Encoding Resource Constraints
- Condition Variables
- **Break:** Announcements
- Semaphores
- Thread Coordination
- Example: Reader-Writer

Race Conditions and Deadlock

Race Conditions and Deadlock

Goal: we must encode resource constraints into our program.

Example: how many philosophers can *try to* eat at the same time? **Four**.

How can we encode this into our program?

What does this look like in code?

Race Conditions and Deadlock

Goal: we must encode resource constraints into our program.

Example: how many philosophers can *try to* eat at the same time? **Four**.

- *Alternative:* how many philosophers can *eat* at the same time? **Two.**

How can we encode this into our program?

What does this look like in code?

Race Conditions and Deadlock

Goal: we must encode resource constraints into our program.

Example: how many philosophers can *try to* eat at the same time? **Four**.

- *Alternative:* how many philosophers can *eat* at the same time? **Two.**
- Why might the first one be better? Imposes less bottlenecking while still solving the issue.

How can we encode this into our program?

What does this look like in code?

Race Conditions and Deadlock

Goal: we must encode resource constraints into our program.

Example: how many philosophers can *try to* eat at the same time? **Four**.

- *Alternative:* how many philosophers can *eat* at the same time? **Two.**
- Why might the first one be better? Imposes less bottlenecking while still solving the issue.

How can we encode this into our program?

- let's add another shared variable representing a count of "permits" or "tickets" available.

What does this look like in code?

Race Conditions and Deadlock

Goal: we must encode resource constraints into our program.

Example: how many philosophers can *try to* eat at the same time? **Four**.

- *Alternative:* how many philosophers can *eat* at the same time? **Two.**
- Why might the first one be better? Imposes less bottlenecking while still solving the issue.

How can we encode this into our program?

- let's add another shared variable representing a count of "permits" or "tickets" available.
- In order to try to eat (aka grab forks at all) a philosopher must get a permit

What does this look like in code?

Race Conditions and Deadlock

Goal: we must encode resource constraints into our program.

Example: how many philosophers can *try to* eat at the same time? **Four**.

- *Alternative:* how many philosophers can *eat* at the same time? **Two**.
- Why might the first one be better? Imposes less bottlenecking while still solving the issue.

How can we encode this into our program?

- let's add another shared variable representing a count of "permits" or "tickets" available.
- In order to try to eat (aka grab forks at all) a philosopher must get a permit
- Once done eating, a philosopher must return their permit

What does this look like in code?

Race Conditions and Deadlock

Goal: we must encode resource constraints into our program.

Example: how many philosophers can *try to* eat at the same time? **Four**.

- *Alternative:* how many philosophers can *eat* at the same time? **Two.**
- Why might the first one be better? Imposes less bottlenecking while still solving the issue.

How can we encode this into our program?

- let's add another shared variable representing a count of "permits" or "tickets" available.
- In order to try to eat (aka grab forks at all) a philosopher must get a permit
- Once done eating, a philosopher must return their permit

What does this look like in code?

- If there are permits available (count > 0) then decrement by 1 and continue

Race Conditions and Deadlock

Goal: we must encode resource constraints into our program.

Example: how many philosophers can *try to* eat at the same time? **Four**.

- *Alternative:* how many philosophers can *eat* at the same time? **Two.**
- Why might the first one be better? Imposes less bottlenecking while still solving the issue.

How can we encode this into our program?

- let's add another shared variable representing a count of "permits" or "tickets" available.
- In order to try to eat (aka grab forks at all) a philosopher must get a permit
- Once done eating, a philosopher must return their permit

What does this look like in code?

- 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

Race Conditions and Deadlock

Goal: we must encode resource constraints into our program.

Example: how many philosophers can *try to* eat at the same time? **Four**.

- *Alternative:* how many philosophers can *eat* at the same time? **Two**.
- Why might the first one be better? Imposes less bottlenecking while still solving the issue.

How can we encode this into our program?

- let's add another shared variable representing a count of "permits" or "tickets" available.
- In order to try to eat (aka grab forks at all) a philosopher must get a permit
- Once done eating, a philosopher must return their permit

What does this look like in code?

- 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
- To return a permit, increment by 1 and continue

Tickets, Please...

- Let's add a new variable in main called **permits**, and a lock for it called **permitsLock**, so that we can update it without race conditions.
- We pass these to each philosopher by reference.
- The full program can be found right here.

```
int main(int argc, const char *argv[]) {
  // NEW
  size_t permits = 4;
  mutext permitsLock;

  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), ref(permitsLock));
  }
  for (thread& p: philosophers) p.join();
  return 0;
}
```

Tickets, Please...

- Each philosopher takes two additional parameters as a result.
- The implementation of **think** does not change, as it does not use permits.
- The full program can be found right here.

```cpp
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);
  }
}
```

Tickets, Please...

- The implementation of **eat** changes:
  - Before eating, the philosopher must get a permit
  - After eating, the philosopher must return their permit.
- The full program can be found right here.

```
static void eat(size_t id, mutex& left, mutex& right, size_t& permits, mutex& permitsLock) {
  // NEW
  waitForPermission(permits, permitsLock);

  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;

  // NEW
  grantPermission(permits, permitsLock);

  left.unlock();
  right.unlock();
}
```

grantPermission

- How do we implement **grantPermission**?
- Recall: "To return a permit, increment by 1 and continue"

```
static void grantPermission(size_t& permits, mutex& permitsLock) {
  permitsLock.lock();
  permits++;
  permitsLock.unlock();
}
```

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"

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

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"

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

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

- **Recap:** Race Conditions and Mutexes
- **Recap:** Dining With Philosophers
- Encoding Resource Constraints
- Condition Variables
- **Break:** Announcements
- Semaphores
- Thread Coordination
- Example: Reader-Writer

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.

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"

Idea:

- when someone returns a permit and it is the only one now available, signal.
- if we need a permit but there are none available, wait.

Condition Variables

Full program: here

- Now we must create a condition variable to pass by reference to all threads.

```cpp
int main(int argc, const char *argv[]) {
  size_t permits = 4;
  mutex forks[5], m;

  // NEW
  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;
}
```

grantPermission

Full program: here

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

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

```

waitForPermission

Full program: here

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

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

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

- it puts the caller to sleep *and* unlocks the given lock, all atomically
- 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

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--;
}
```

Plan For Today

- **Recap:** Race Conditions and Mutexes
- **Recap:** Dining With Philosophers
- Encoding Resource Constraints
- Condition Variables
- **Break:** Announcements
- Semaphores
- Thread Coordination
- Example: Reader-Writer

Announcements

Midterm This Friday

- Midterm info webpage with practice materials, BlueBook download: cs110.stanford.edu/exams/midterm/
- Please notify us of any OAE accommodations by **today**
- 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 **today**.
- Review Session **tonight 7-8:30PM in Hewlett 201** (recorded)

Plan For Today

- **Recap:** Race Conditions and Mutexes
- **Recap:** Dining With Philosophers
- Encoding Resource Constraints
- Condition Variables
- **Break:** Announcements
- Semaphores
- Thread Coordination
- Example: Reader-Writer

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();
}
```

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?

semaphore

- The **`semaphore`** class is not built in to C++, but it is a useful way to generalize the "permits" idea. We will link against our version of a semaphore for this class, but you should understand how it is built.
- Using a **`semaphore`** is straightforward: you first declare a semaphore with a number of permits you would like:

```
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

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

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

semaphore

- 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.
- We will see an example of this shortly.

- What about a *negative initializer* for a semaphore?

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

semaphore

- What about a *negative initializer* for a semaphore?

```
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. Example (full program here):

```
void writer(int i, semaphore &s) {
    cout << oslock << "Sending signal " << i << endl << osunlock;
    s.signal();
}

void read_after_ten(semaphore &s) {
    s.wait();
    cout << oslock << "Got enough signals to continue!" << endl << osunlock;
}

int main(int argc, const char *argv[]) {
    semaphore negSemaphore(-9);
    thread readers[10];
    for (size_t i = 0; i < 10; i++) {
        readers[i] = thread(writer, i, ref(negSemaphore));
    }
    thread r(read_after_ten, ref(negSemaphore));
    for (thread &t : readers) t.join();
    r.join();
    return 0;
}
```

Plan For Today

- **Recap:** Race Conditions and Mutexes
- **Recap:** Dining With Philosophers
- Encoding Resource Constraints
- Condition Variables
- **Break:** Announcements
- Semaphores
- Thread Coordination
- Example: Reader-Writer

semaphore

- 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.

semaphore

- 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;
}
```

Plan For Today

- **Recap:** Race Conditions and Mutexes
- **Recap:** Dining With Philosophers
- Encoding Resource Constraints
- Condition Variables
- **Break:** Announcements
- Semaphores
- Thread Coordination
- Example: Reader-Writer

semaphore

- 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**.

semaphore

- 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.

semaphore

- 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?

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:** Race Conditions and Mutexes
- **Recap:** Dining With Philosophers
- Encoding Resource Constraints
- Condition Variables
- **Break:** Announcements
- Semaphores
- Thread Coordination
- Example: Reader-Writer

Next time: more threads