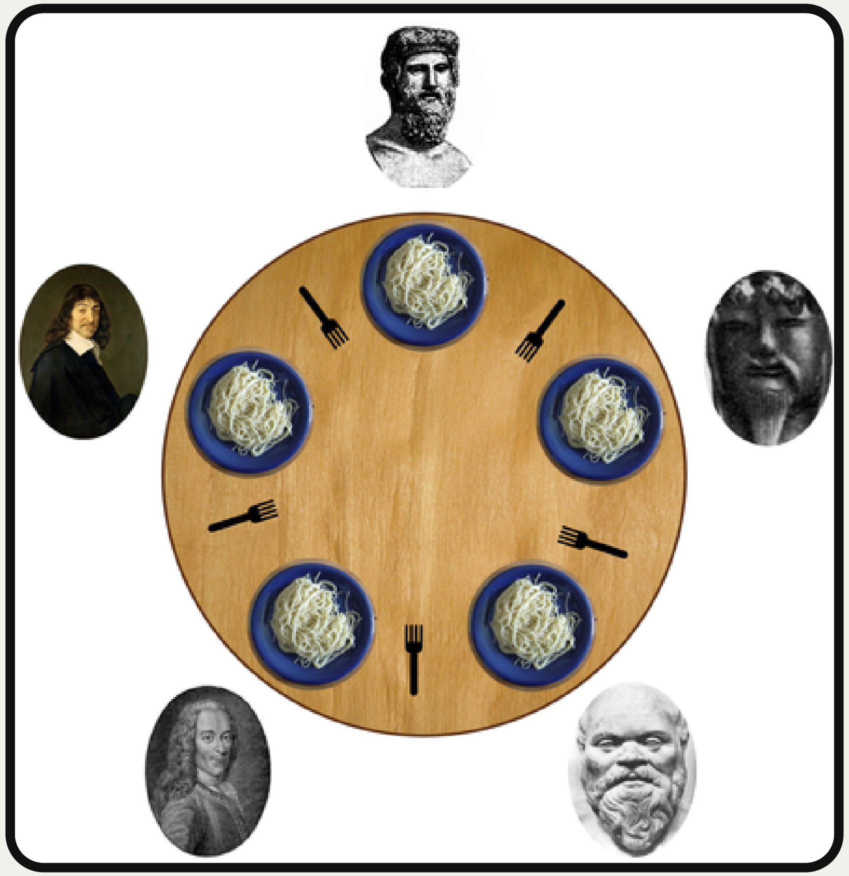
CS110 Lecture 15: Mutexes and Condition Variables

CS110: Principles of Computer Systems

Winter 2021-2022

Stanford University

Instructors: Nick Troccoli and Jerry Cain



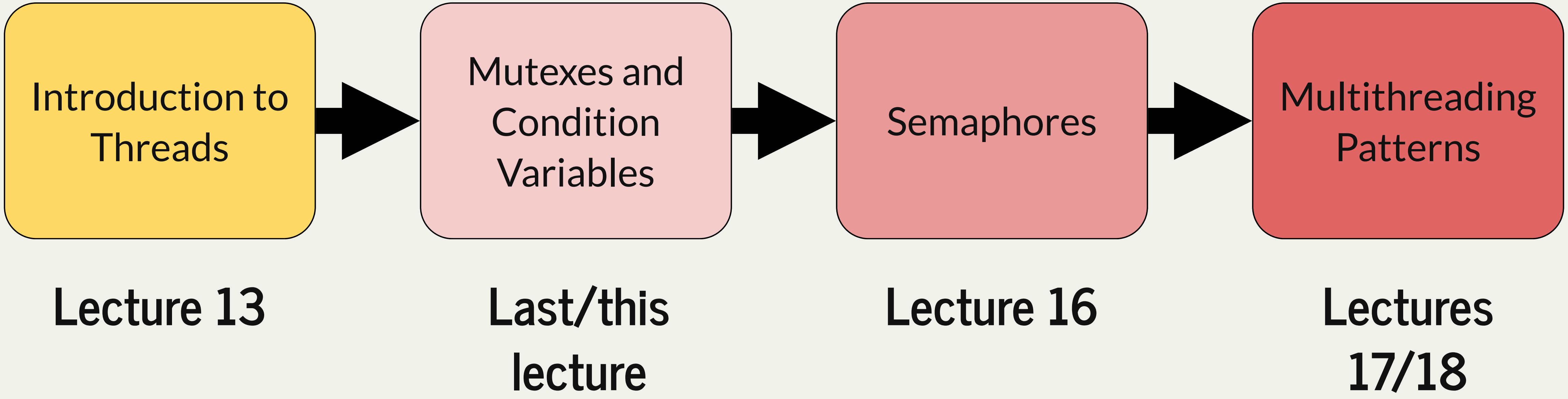
`https://commons.wikimedia.org/wiki/File:An_illus`

`tration_of_the_dining_philosophers_problem.png`

PDF of this presentation

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

Learning About Multithreading



assign5: implement your own multithreaded news aggregator to quickly fetch news from the web!

Learning Goals

- Get more practice using mutexes to prevent race conditions
- Apply the race condition checklist to eliminate race conditions
- Learn how condition variables can let threads signal to each other
- Get practice with the "available permits" resource model

Plan For Today

- **Recap:** Race Conditions and Mutexes
- Dining With Philosophers
- Encoding Resource Constraints
- Condition Variables

Plan For Today

- **Recap: Race Conditions and Mutexes**
- Dining With Philosophers
- Encoding Resource Constraints
- Condition Variables

Mutexes

- We can create a lock-and-key combo by creating a variable of type **mutex**.
- A **mutex** is technically a type of lock; there are others, but we focus just on **mutexes**
- When you create a mutex, it is initially unlocked with the key available
- You call **lock()** on the mutex to attempt to lock it and take the key
- You call **unlock()** on a mutex *if you have ownership of it* and wish to unlock it and return the key. That thread continues normally; one waiting thread (if any) then takes the lock and is scheduled to run.

```
// Assume multiple threads share this same mutex
mutex myLock;

...

myLock.lock();
// only one thread can be executing here at a time
myLock.unlock()
```

Mutexes

When a thread calls **lock()**:

- **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 (not necessarily one waiting longest)

```
// Assume multiple threads share this same mutex
mutex myLock;

...

myLock.lock();
// only one thread can be executing here at a time
myLock.unlock()
```

Mutex Usage

1. Identify a critical section; a section that only one thread should execute at a time.
2. Create a mutex and ***pass it by reference*** to all threads executing that critical section
3. Add a line to lock the mutex at the start of the critical section
4. Add a line to unlock the mutex at the end of the critical section

```
// Assume multiple threads share this same mutex
mutex myLock;

...

myLock.lock();
// only one thread can be executing here at a time
myLock.unlock()
```

Mutex Usage

1. **<u>Identify a critical section; a section that only one thread should execute at a time.</u>**
2. Create a mutex and ***pass it by reference*** to all threads executing that critical section
3. Add a line to lock the mutex at the start of the critical section
4. Add a line to unlock the mutex at the end of the critical section

```cpp
static void sellTickets(size_t id, size_t& remainingTickets) {
    while (true) {

        if (remainingTickets == 0) break;
        remainingTickets--;
        sleep_for(500);  // simulate "selling a ticket"
        cout << oslock << "Thread #" << id << " sold a ticket (" << remainingTickets << " remain)." << endl << osunlock;

    }
    cout << oslock << "Thread #" << id << " sees no remaining tickets to sell and exits." << endl << osunlock;
}
```

Mutex Usage

1. Identify a critical section; a section that only one thread should execute at a time.
2. Create a mutex and *pass it by reference* to all threads executing that critical section
3. Add a line to lock the mutex at the start of the critical section
4. Add a line to unlock the mutex at the end of the critical section

(It turns out that the mutex type *can't* be passed by copy in C++; since it doesn't make sense to make a copy of a mutex).

```
int main(int argc, const char *argv[]) {
    thread ticketAgents[kNumTicketAgents];
    size_t remainingTickets = 250;
    mutex counterLock;

    for (size_t i = 0; i < kNumTicketAgents; i++) {
        ticketAgents[i] = thread(sellTickets, i, ref(remainingTickets), ref(counterLock));
    }
    ...
}
```

Mutex Usage

1. Identify a critical section; a section that only one thread should execute at a time.
2. Create a mutex and ***pass it by reference*** to all threads executing that critical section
3. **<u>Add a line to lock the mutex at the start of the critical section</u>**
4. Add a line to unlock the mutex at the end of the critical section

```cpp
static void sellTickets(size_t id, size_t& remainingTickets, mutex& counterLock) {
    while (true) {
        size_t myTicket;

        counterLock.lock();
        if (remainingTickets == 0) {
            counterLock.unlock();
            break;
        } else {
            myTicket = remainingTickets;
            remainingTickets--;
            counterLock.unlock();
        }

        sleep_for(500);  // simulate "selling a ticket"
        cout << oslock << "Thread #" << id << " sold a ticket (" << myTicket << " remain)." << endl << osunlock;
    }
    ...
}
```

Mutex Usage

1. Identify a critical section; a section that only one thread should execute at a time.
2. Create a mutex and ***pass it by reference*** to all threads executing that critical section
3. Add a line to lock the mutex at the start of the critical section
4. **Add a line to unlock the mutex at the end of the critical section**

```
static void sellTickets(size_t id, size_t& remainingTickets, mutex& counterLock) {
    while (true) {
        size_t myTicket;

        counterLock.lock();
        if (remainingTickets == 0) {
            counterLock.unlock();
            break;
        } else {
            myTicket = remainingTickets;
            remainingTickets--;
            counterLock.unlock();
        }

        sleep_for(500);  // simulate "selling a ticket"
        cout << oslock << "Thread #" << id << " sold a ticket (" << myTicket << " remain)." << endl << osunlock;
    }
    ...
}
```

Deadlock

Deadlock is a situation where a thread or threads rely on mutually blocked-on resources that will never become available.

- **Example**: a thread must acquire a lock before proceeding. We forget to call unlock somewhere, so one thread keeps the lock forever while others are stuck waiting for the lock forever.

```
static void sellTickets(size_t id, size_t& remainingTickets, mutex& counterLock) {
    while (true) {
        size_t myTicket;

        counterLock.lock();
        if (remainingTickets == 0) {
            break;
        } else {
            myTicket = remainingTickets;
            remainingTickets--;
            counterLock.unlock();
        }
        ...
    }
    ...
}
```

The Race Condition Checklist

☐ **Identify shared data that may be modified concurrently.** What shared data is used across threads, passed by reference or globally?

☐ **Document and confirm an ordering of events that causes unexpected behavior**. What assumptions are made in the code that can be broken by certain orderings?

☐ **Use concurrency directives to force expected orderings and add constraints.** How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s)?

The Race Condition Checklist

☐ **Identify shared data that may be modified concurrently.** What shared data is used across threads, passed by reference or globally? ***the ticket count.***

☐ **Document and confirm an ordering of events that causes unexpected behavior.** What assumptions are made in the code that can be broken by certain orderings?

☐ **Use concurrency directives to force expected orderings and add constraints.** How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s)?

The Race Condition Checklist

☑ **Identify shared data that may be modified concurrently.** What shared data is used across threads, passed by reference or globally? ***the ticket count.***

☐ **Document and confirm an ordering of events that causes unexpected behavior**. What assumptions are made in the code that can be broken by certain orderings? ***one thread will check-and-sell a ticket at a time.***

☐ **Use concurrency directives to force expected orderings and add constraints.** How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s)?

The Race Condition Checklist

☑ **Identify shared data that may be modified concurrently.** What shared data is used across threads, passed by reference or globally? ***the ticket count.***

☑ **Document and confirm an ordering of events that causes unexpected behavior.** What assumptions are made in the code that can be broken by certain orderings? ***one thread will check-and-sell a ticket at a time.***

☐ **Use concurrency directives to force expected orderings and add constraints.** How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s)? ***add a mutex that must be acquired before checking-and-selling a ticket.***

The Race Condition Checklist

☑ **Identify shared data that may be modified concurrently.** What shared data is used across threads, passed by reference or globally? ***the ticket count.***

☑ **Document and confirm an ordering of events that causes unexpected behavior**. What assumptions are made in the code that can be broken by certain orderings? ***one thread will check-and-sell a ticket at a time.***

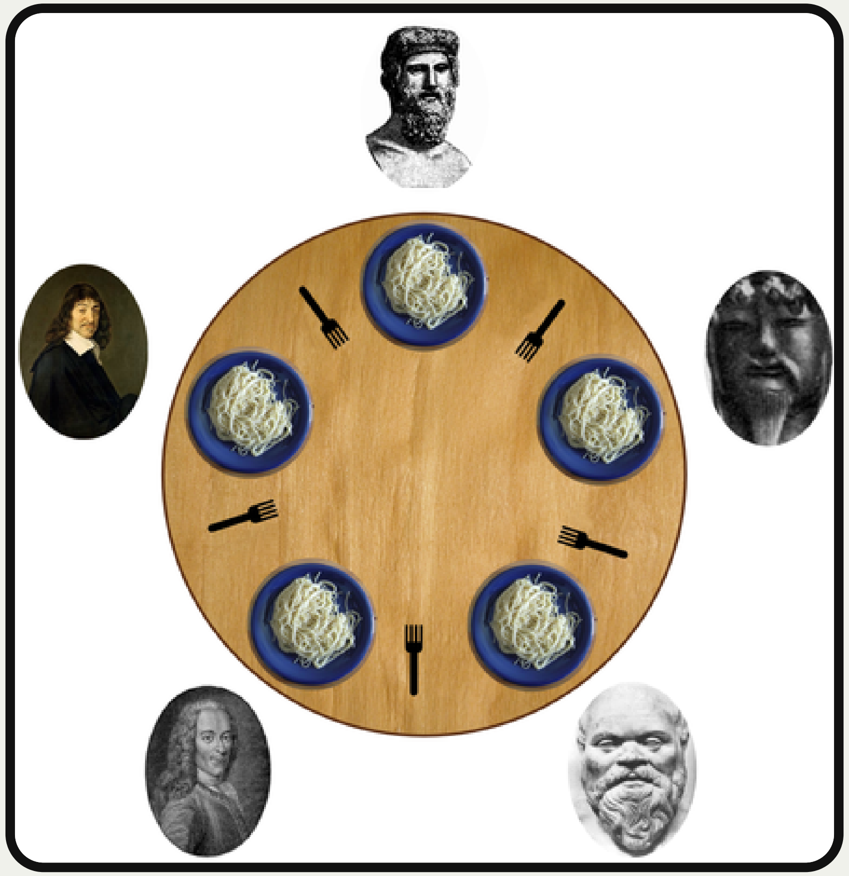
☑ **Use concurrency directives to force expected orderings and add constraints**. How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s)? ***add a mutex that must be acquired before checking-and-selling a ticket.***

Plan For Today

- **Recap:** Race Conditions and Mutexes
- **Dining With Philosophers**
- Encoding Resource Constraints
- Condition Variables

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. Then 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-with-deadlock.cc

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[kNumForks];
  thread philosophers[kNumPhilosophers];
  for (size_t i = 0; i < kNumPhilosophers; i++) {
    mutex& left = forks[i];
    mutex& right = forks[(i + 1) % kNumPhilosophers];
    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();
}
```

Spoiler: there is a race condition here that leads to deadlock.

The Race Condition Checklist

☐ **Identify shared data that may be modified concurrently.** What shared data is used across threads, passed by reference or globally?

☐ **Document and confirm an ordering of events that causes unexpected behavior**. What assumptions are made in the code that can be broken by certain orderings?

☐ **Use concurrency directives to force expected orderings and add constraints.** How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s)?

The Race Condition Checklist

☐ **Identify shared data that may be modified concurrently.** What shared data is used across threads, passed by reference or globally? ***the "forks" (mutexes).***

☐ **Document and confirm an ordering of events that causes unexpected behavior.** What assumptions are made in the code that can be broken by certain orderings?

☐ **Use concurrency directives to force expected orderings and add constraints.** How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s)?

The Race Condition Checklist

☑ **Identify shared data that may be modified concurrently.** What shared data is used across threads, passed by reference or globally?

☐ **Document and confirm an ordering of events that causes unexpected behavior.** What assumptions are made in the code that can be broken by certain orderings?

☐ **Use concurrency directives to force expected orderings and add constraints.** How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s)?

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();
}
```

Discuss with your neighbor: come up with an ordering of events that causes deadlock.

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 }
```

The Race Condition Checklist

☑ **Identify shared data that may be modified concurrently.** What shared data is used across threads, passed by reference or globally?

☐ **Document and confirm an ordering of events that causes unexpected behavior.** What assumptions are made in the code that can be broken by certain orderings? ***we are assuming that someone is always able to pick up 2 forks.***

☐ **Use concurrency directives to force expected orderings and add constraints.** How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s)?

The Race Condition Checklist

☑ **Identify shared data that may be modified concurrently.** What shared data is used across threads, passed by reference or globally?

☑ **Document and confirm an ordering of events that causes unexpected behavior.** What assumptions are made in the code that can be broken by certain orderings? ***we are assuming that someone is always able to pick up 2 forks.***

☐ **Use concurrency directives to force expected orderings and add constraints.** How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s)?

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
- Dining With Philosophers
- **Encoding Resource Constraints**
- Condition Variables

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.

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

  mutex forks[5];
  thread philosophers[5];
  for (size_t i = 0; i < 5; i++) {
    mutex& left = forks[i];
    mutex& 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;
}
```

dining-philosophers-with-busy-waiting.cc

Tickets, Please...

- Each philosopher takes two additional parameters as a result.
- The implementation of **think** does not change, as it does not use permits.

```
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);
  }
}
```

dining-philosophers-with-busy-waiting.cc

Tickets, Please...

- The implementation of **eat** changes:
  - Before eating, the philosopher must get a permit
  - After eating, the philosopher must return their permit.

```
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();
}
```

dining-philosophers-with-busy-waiting.cc

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();
}
```

dining-philosophers-with-busy-waiting.cc

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! We are unnecessarily and arbitrarily using CPU time to check when a permit is available.

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
- Dining With Philosophers
- Encoding Resource Constraints
- **Condition Variables**

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. Conversely, 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

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

```
int main(int argc, const char *argv[]) {
  mutex forks[kNumForks];

  size_t permits = kNumForks - 1;
  mutex permitsLock;

  // NEW
  condition_variable_any permitsCV;

  thread philosophers[kNumPhilosophers];
  for (size_t i = 0; i < kNumPhilosophers; i++) {
    mutex& left = forks[i];
    mutex& right = forks[(i + 1) % kNumForks];
    philosophers[i] = thread(philosopher, i, ref(left), ref(right),
      ref(permits), ref(permitsCV), ref(permitsLock));
  }
  for (thread& p: philosophers) p.join();
  return 0;
}
```

dining-philosophers-with-cv-wait.cc

grantPermission

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

```
static void grantPermission(size_t& permits, condition_variable_any& permitsCV, mutex& pe
  permitsLock.lock();
  permits++;
  if (permits == 1) permitsCV.notify_all();
  permitsLock.unlock();
}
```

dining-philosophers-with-cv-wait.cc

waitForPermission

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

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

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

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

`dining-philosophers-with-cv-wait.cc`

Recap

- **Recap:** Race Conditions and Mutexes
- Dining With Philosophers
- Encoding Resource Constraints
- Condition Variables

Next time: more condition variables and introducing semaphores