CS111, Lecture 14
Locks and Condition Variables

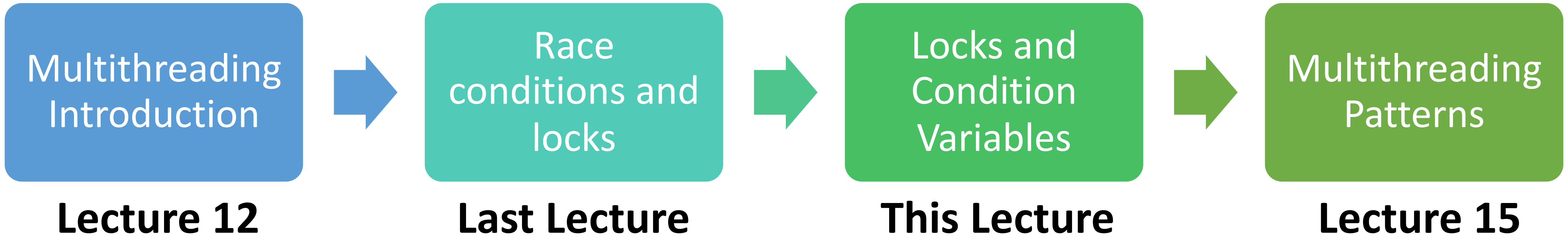
Optional reading:

Operating Systems: Principles and Practice (2nd Edition): Sections 5.2-5.4 and Section 6.5

😷 masks strongly recommended

Topic 3: Multithreading - How can we have concurrency within a single process? How does the operating system support this?

CS111 Topic 3: Multithreading, Part 1


Multithreading Introduction
Lecture 12
Race conditions and locks
Last Lecture
Locks and Condition Variables
This Lecture
Multithreading Patterns
Lecture 15

assign4: implement several multithreaded programs while eliminating race conditions!

Learning Goals

- Get more practice using mutexes to prevent race conditions
- Learn about ways to add constraints to our programs to prevent deadlock
- Learn how condition variables can let threads signal to each other and wait for conditions to become true

Plan For Today

- **Recap:** mutexes
- Deadlock
- Dining Philosophers
- Encoding resource constraints
- Condition Variables

```
cp -r /afs/ir/class/cs111/lecture-code/lect14 .
```

Plan For Today

- **Recap: mutexes**
- Deadlock
- Dining Philosophers
- Encoding resource constraints
- Condition Variables

```
cp -r /afs/ir/class/cs111/lecture-code/lect14 .
```

Mutexes

A **mutex** (”mutual exclusion”) is a variable type that lets us enforce the pattern of only 1 thread having access to something at a time.

- You make a mutex for each distinct thing you need to limit access to.
- You call **lock()** on the mutex to attempt to take the lock
- You call **unlock()** on the mutex when you are done to give the lock back

Mutexes

1. Identify a critical section; section that only 1 thread should execute at a time.
2. Create a mutex and share it among all threads executing that critical section
3. Lock the mutex at the start of the critical section
4. Unlock the mutex at the end of the critical section

Ticket Agents

```
static void sellTickets(size_t id, size_t& remainingTickets, mutex&
counterLock) {
    while (true) {
        counterLock.lock();  // only 1 thread can proceed at a time
        if (remainingTickets == 0) {
            counterLock.unlock(); // must give up lock before exiting
            break;
        }
        size_t myTicket = remainingTickets;
        remainingTickets--;
        counterLock.unlock(); // once thread passes here, another can go
        sleep_for(500); // simulate "selling a ticket"
        ...
```

Plan For Today

- **Recap:** mutexes
- **Deadlock**
- Dining Philosophers
- Encoding resource constraints
- Condition Variables

```
cp -r /afs/ir/class/cs111/lecture-code/lect14 .
```

Deadlock

Deadlock occurs when multiple threads are all blocked, waiting on a resource owned by one of the other threads. None can make progress! Example:

Thread A	Thread B
`mutex1.lock();`	`mutex2.lock();`
`mutex2.lock();`	`mutex1.lock();`
`...`	`...`

E.g. if thread A executes 1 line, then thread B executes 1 line, deadlock!

One prevention technique - prevent circularities: all threads request resources in the same order (e.g., always lock mutex1 before mutex2.)

Another – limit number of threads competing for a shared resource

Plan For Today

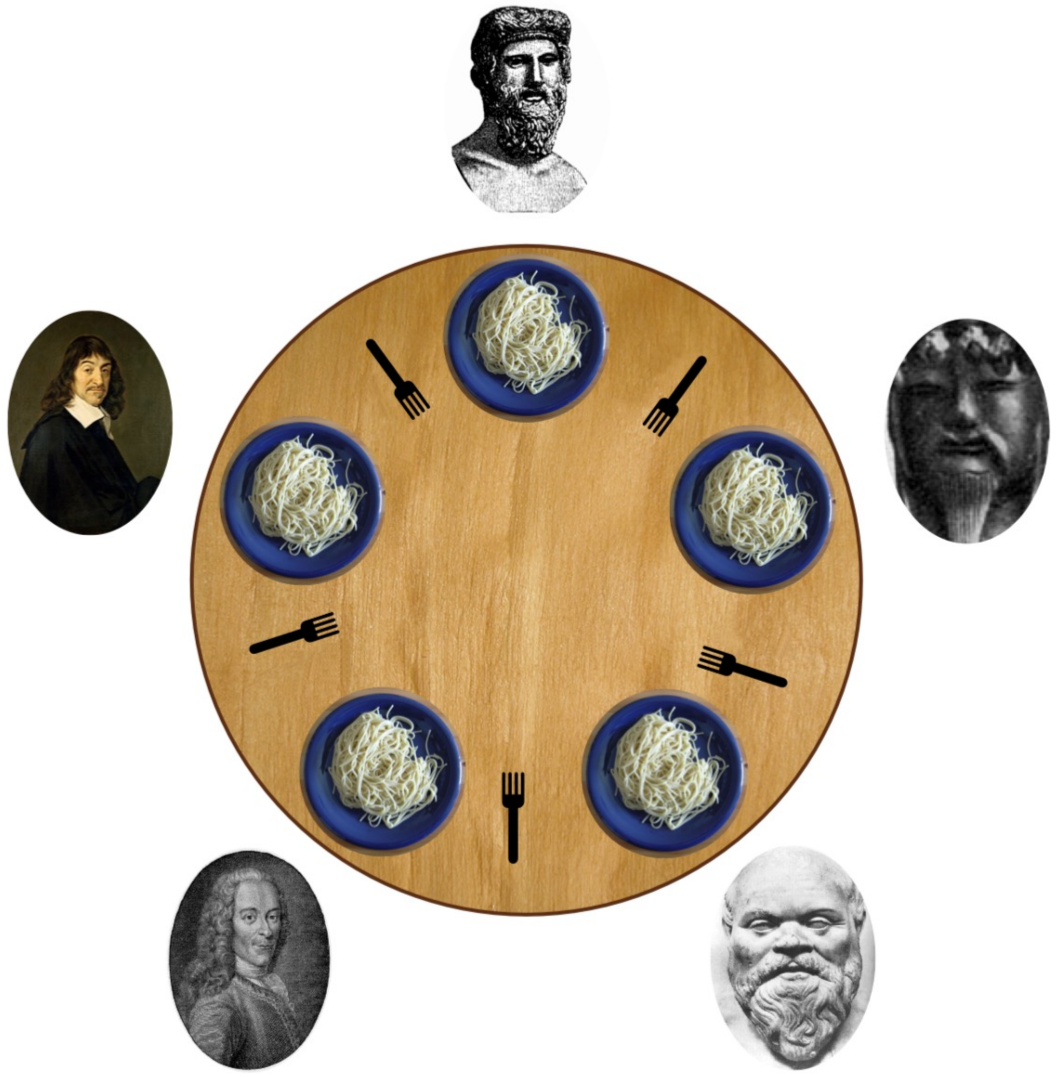
- Recap: mutexes
- Deadlock
- **Dining Philosophers**
- Encoding resource constraints
- Condition Variables

```
cp -r /afs/ir/class/cs111/lecture-code/lect14 .
```

Deadlock Example: Dining Philosophers Simulation

- 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**, a philosopher keeps to themselves for some amount of time. Sometimes they think for a long time, and sometimes they barely think at all.

Dining Philosophers



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

Dining Philosophers

Goal: we must encode resource constraints into our program.

Example: for a given fork, how many philosophers can use it at a time? **One**.

How can we encode this into our program? Make a mutex for each fork.

Dining Philosophers

```
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++) {
        philosophers[i] = thread(philosopher, i,
                           ref(forks[i]),
                           ref(forks[(i + 1) % kNumPhilosophers]));
    }
    for (thread& p: philosophers) p.join();
    return 0;
}
```

Dining Philosophers

A philosopher thinks and eats, and repeats this 3 times.

```
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

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

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 << osu
    sleep_for(getEat
    cout << oslock <
         << osunlock
    left.unlock();
    right.unlock();
}
```

Spoiler: there is a race condition here that leads to **deadlock – deadlock** occurs when multiple threads are all blocked, waiting on a resource owned by one of the other blocked threads. When could this happen?

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 in between grabbing the two forks
- We should be able to insert a **sleep_for** call anywhere in a thread routine and have no concurrency issues. Let's try it!

`dining-philosophers-with-deadlock.cc`

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 in between grabbing the two forks
- We should be able to insert a **sleep_for** call anywhere in a thread routine and have no concurrency issues. Let's try it!
- We (incorrectly) assumed that at least one philosopher is always able to pick up both of their forks. How can we fix this?

dining-philosophers-with-deadlock.cc

Race Conditions and Deadlock

In multithreaded programs, we need to ensure that:

there are **never** race conditions

- we can generally solve race conditions with **mutexes.** Use them to mark the boundaries of critical sections to limit them to 1 thread at a time.

there's **zero** chance of deadlock (otherwise some or all threads are starved)

- we can solve deadlock by requesting resources in the same order and by limiting the number of threads competing for a shared resource.

Plan For Today

- **Recap:** mutexes
- Deadlock
- Dining Philosophers
- **Encoding resource constraints**
- Condition Variables

```
cp -r /afs/ir/class/cs111/lecture-code/lect14 .
```

Encoding Resource Constraints

Goal: we must encode resource constraints into our program.

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

- *Alternatively:* 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?

Have a counter of "permits". Initially 4. A philosopher must have a permit (decrement counter or wait) to try to eat. Once done eating, a philosopher returns its permit (increment counter).

Tickets, Please...

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

    size_t permits = kNumForks - 1;
    mutex permitsLock;

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

Tickets, Please...

A philosopher thinks and eats, and repeats this 3 times.

```
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...

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

    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;
    grantPermission(permits, permitsLock);
    left.unlock();
    right.unlock();
}
```

grantPermission

To put a permit back, increment the counter by 1 and continue.

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

waitForPermission

- If there are permits, decrement the counter by 1 and continue
- If there aren’t permits, wait for a permit, then decrement by 1 and continue

```
static void waitForPermission(size_t& permits, mutex&
permitsLock) {
  while (true) {
    permitsLock.lock();
    if (permits > 0) break;
    permitsLock.unlock();
    // wait a little while (how??)
  }
  permits--;
  permitsLock.unlock();
}
```

waitForPermission

- If there are permits, decrement the counter by 1 and continue
- If there aren't permits, wait for a permit, then decrement by 1 and continue

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

This is called busy waiting (bad). 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:** mutexes
- Deadlock
- Dining Philosophers
- Encoding resource constraints
- **Condition Variables**

```
cp -r /afs/ir/class/cs111/lecture-code/lect14 .
```

Condition Variables

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

- You make one for each distinct event you need to wait / notify for.
- We can call **wait** on the condition variable to sleep until another thread signals this condition variable.
- You call **notify_all** on the condition variable to send a notification to all waiting threads and wake them up.

Condition Variables

1. Identify a single kind of event that we need to wait / notify for
2. Ensure there is proper state to check if the event has happened
3. Create a condition variable and share it among all threads either waiting for that event to happen or triggering that event
4. Identify who will notify that this happens, and have them notify via the condition variable
5. Identify who will wait for this to happen, and have them wait via the condition variable

Condition Variables

1. Identify a single kind of event that we need to wait / notify for
2. Ensure there is proper state to check if the event has happened
3. Create a condition variable and share it among all threads either waiting for that event to happen or triggering that event
4. Identify who will notify that this happens, and have them notify via the condition variable
5. Identify who will wait for this to happen, and have them wait via the condition variable

The event here is ”some permits are again available”.

Condition Variables

1. Identify a single kind of event that we need to wait / notify for
2. Ensure there is proper state to check if the event has happened
3. Create a condition variable and share it among all threads either waiting for that event to happen or triggering that event
4. Identify who will notify that this happens, and have them notify via the condition variable
5. Identify who will wait for this to happen, and have them wait via the condition variable

We can check whether there are permits now available by checking the permits count.

Condition Variables

1. Identify a single kind of event that we need to wait / notify for
2. Ensure there is proper state to check if the event has happened
3. Create a condition variable and share it among all threads either waiting for that event to happen or triggering that event
4. Identify who will notify that this happens, and have them notify via the condition variable
5. Identify who will wait for this to happen, and have them wait via the condition variable

Condition Variables

```
int main(int argc, const char *argv[]) {
    mutex forks[kNumForks];
    size_t permits = kNumForks - 1;
    mutex permitsLock;
    condition_variable_any permitsCV;

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

Condition Variables

1. Identify a single kind of event that we need to wait / notify for
2. Ensure there is proper state to check if the event has happened
3. Create a condition variable and share it among all threads either waiting for that event to happen or triggering that event
4. Identify who will notify that this happens, and have them notify via the condition variable
5. Identify who will wait for this to happen, and have them wait via the condition variable

When someone returns a permit and there were no permits available previously, notify all.

grantPermission

We must notify all once permits have become available again to wake up waiting threads.

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

Condition Variables

1. Identify a single kind of event that we need to wait / notify for
2. Ensure there is proper state to check if the event has happened
3. Create a condition variable and share it among all threads either waiting for that event to happen or triggering that event
4. Identify who will notify that this happens, and have them notify via the condition variable
5. Identify who will wait for this to happen, and have them wait via the condition variable

If we need a permit but there are none available, wait.

waitForPermission

If no permits are available, we must wait until one becomes available.

Key Idea: we must give up ownership of the lock when we wait, so that someone else can put a permit back.

```
static void waitForPermission(size_t& permits,
condition_variable_any& permitsCV, mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
      permitsLock.unlock();
      permitsCV.wait();     // (note: not final form of wait)
      permitsLock.lock();
    }
    permits--;
    permitsLock.unlock();
}
```

Recap

- **Recap:** mutexes
- Deadlock
- Dining Philosophers
- Encoding resource constraints
- Condition Variables

Lecture 14 takeaway: Condition variables let us wait on an event to occur and notify other threads that an event has occurred, all without busy waiting.

Next time: multithreading patterns

```
cp -r /afs/ir/class/cs111/lecture-code/lect14 .
```