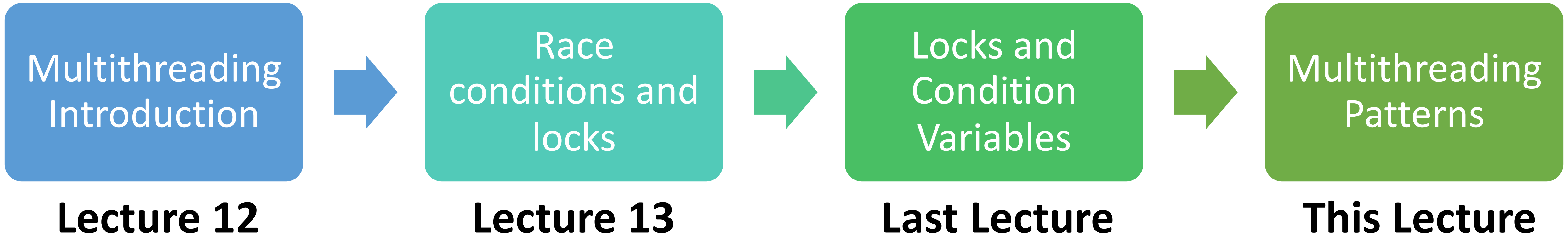
CS111, Lecture 15
Multithreading Patterns

😷 masks 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
Lecture 13
Locks and Condition Variables
Last Lecture
Multithreading Patterns
This Lecture

assign4: implement several multithreaded programs while eliminating race conditions!

Learning Goals

- Get more practice using both mutexes and condition variables to implement synchronization logic.
- Learn about the **monitor** pattern for designing multithreaded code in the simplest way possible, using classes.

Plan For Today

- **Recap:** condition variables and dining philosophers
- **notify_one**
- Unique locks
- Monitor pattern
- **Example:** Bridge Crossing

```
cp -r /afs/ir/class/cs111/lecture-code/lect15 .
```

Plan For Today

- **Recap: condition variables and dining philosophers**
- **notify_one**
- Unique locks
- Monitor pattern
- **Example:** Bridge Crossing

```
cp -r /afs/ir/class/cs111/lecture-code/lect15 .
```

Condition Variable Key Takeaways

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

- We can call **wait(*lock*)** to sleep until another thread signals this condition variable. The condition variable will unlock and re-lock the specified lock for us.
  - This is necessary because we must give up the lock while waiting so another thread may return a permit, but if we unlock before waiting, there is a race condition.
- We can call ***notify_all()*** to send a signal to waiting threads and wake them up.
- We call ***wait(lock)*** in a loop in case we are woken up but must wait longer
  - This could happen if multiple threads are woken up for a single new permit.

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

How did we use these steps to implement the "permits" model for the dining philosophers?

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

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.

Condition Variable Wait

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

cv.wait() must be called in a loop, and does the following:

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

Plan For Today

- **Recap:** condition variables and dining philosophers
- **notify_one**
- Unique locks
- Monitor pattern
- **Example:** Bridge Crossing

```
cp -r /afs/ir/class/cs111/lecture-code/lect15 .
```

notify_all vs. notify_one

Our current design uses **notify_all** to wake up everyone when we go from “no permits” to “permits”.

- We need to wake up everyone because when we put back the first permit, others may put back further permits after us; therefore, we should wake up everyone, just in case.

There is another condition variable method, **notify_one**, that just notifies one waiting thread, not all.

- What would our implementation look like with this instead?

notify_one

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

notify_one

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

We must now notify someone every time a permit is put back. If we just notified one thread when permits is 1, if someone else puts back permits after us, other waiting threads will not be alerted to those.

notify_one

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

In theory, with **notify_one** it's no longer possible to be woken up but not have a permit available.

However, condition variables can have *spurious wakeups* – they wake us up even when not being notified by another thread!   Thus, we should *always* wrap calls to **wait** in a while loop.

Plan For Today

- **Recap:** condition variables and dining philosophers
- **notify_one**
- **Unique locks**
- Monitor pattern
- **Example:** Bridge Crossing

```
cp -r /afs/ir/class/cs111/lecture-code/lect15 .
```

Unique Locks

- It is common to acquire a lock and hold onto it until the end of some scope (e.g. end of function, end of loop, etc.).
- There is a convenient variable type called ***unique_lock*** that when created can automatically lock a mutex, and when destroyed (e.g. when it goes out of scope) can automatically unlock a mutex.
- Particularly useful if you have many paths to exit a function and you must unlock in all paths.

grantPermission

We lock at the beginning of this function and unlock at the end.

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

grantPermission

We lock at the beginning of this function and unlock at the end.

```
static void grantPermission(size_t& permits,
condition_variable_any& permitsCV, mutex& permitsLock) {
    unique_lock<mutex> uniqueLock(permitsLock);
    permits++;
    if (permits == 1) permitsCV.notify_all();
}
```

Auto-locks permitsLock here

grantPermission

We lock at the beginning of this function and unlock at the end.

```
static void grantPermission(size_t& permits,
condition_variable_any& permitsCV, mutex& permitsLock) {
    unique_lock<mutex> uniqueLock(permitsLock);
    permits++;
    if (permits == 1) permitsCV.notify_all();
}
```

Auto-unlocks permitsLock
here (goes out of scope)

waitForPermission

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

waitForPermission

```
static void waitForPermission(size_t& permits,
condition_variable_any& permitsCV, mutex& permitsLock) {
    unique_lock<mutex> uniqueLock(permitsLock);
    while (permits == 0) {
        permitsCV.wait(uniqueLock);
    }
    permits--;
}
```

Auto-locks permitsLock here

waitForPermission

```
static void waitForPermission(size_t& permits,
condition_variable_any& permitsCV, mutex& permitsLock) {
    unique_lock<mutex> uniqueLock(permitsLock);
    while (permits == 0) {
        permitsCV.wait(uniqueLock);
    }
    permits--;
}
```

Use it with CV instead of original lock (it has wrapper methods for manually locking/unlocking!)

waitForPermission

```
static void waitForPermission(size_t& permits,
condition_variable_any& permitsCV, mutex& permitsLock) {
    unique_lock<mutex> uniqueLock(permitsLock);
    while (permits == 0) {
        permitsCV.wait(uniqueLock);
    }
    permits--;
}
```

Auto-unlocks permitsLock here (goes out of scope)

Plan For Today

- **Recap:** condition variables and dining philosophers
- **notify_one**
- Unique locks
- **Monitor pattern**
- **Example:** Bridge Crossing

```
cp -r /afs/ir/class/cs111/lecture-code/lect15 .
```

Multithreading Patterns

- Writing synchronization code is *hard* – difficult to reason about, bugs are tricky if they are hard to reproduce
- E.g. how many locks should we use for a given program?
  - Just one? Doesn't allow for much concurrency
  - One lock per shared variable? Very hard to manage, gets complex, inefficient
- Like with dining philosophers, we must consider many scenarios and have lots of state to track and manage
- **One design idea to help:** the "monitor" design pattern - associate a single lock with a collection of related variables, e.g. a **class**

Monitor Design Pattern

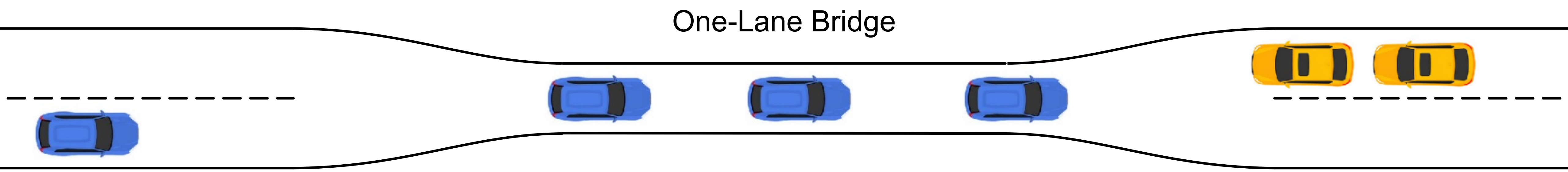
- A shared data structure
- A collection of procedures (methods)
- One lock that must be held whenever accessing the shared data (typically each procedure acquires the lock at the very beginning and releases the lock before returning).
- One or more condition variables used for waiting.

Plan For Today

- **Recap:** condition variables and dining philosophers
- **notify_one**
- Unique locks
- Monitor pattern
- **Example: Bridge Crossing**

```
cp -r /afs/ir/class/cs111/lecture-code/lect15 .
```

Bridge Crossing



Let’s write a program that simulates cars crossing a one-lane bridge.

- We will have each car represented by a thread, and they must coordinate as though they all need to cross the bridge.
- A car can be going either east or west
- All cars on bridge must be travelling in the same direction
- Any number of cars can be on the bridge at once
- A car from the other direction can only go once the coast is clear

Bridge Crossing

```
int main(int argc, const char *argv[]) {
    Bridge bridge;
    thread cars[kNumCars];
    for (size_t i = 0; i < kNumCars; i++) {
        if (flipCoin()) {
            cars[i] = thread(crossBridgeEast, i, ref(bridge));
        } else {
            cars[i] = thread(crossBridgeWest, i, ref(bridge));
        }
    }
    for (thread& car : cars) car.join();
    return 0;
}
```

Bridge Crossing

```
int main(int argc, const char *argv[]) {
    Bridge bridge;
    thread cars[kNumCars];
    for (size_t i = 0; i < kNumCars; i++) {
        if (flipCoin()) {
            cars[i] = thread(crossBridgeEast, i, ref(bridge));
        } else {
            cars[i] = thread(crossBridgeWest, i, ref(bridge));
        }
    }
    for (thread& ca
    return 0;
}
```

Wouldn’t it be cool if, instead of making all these CVs/locks/etc and managing them directly in our program, we had a variable type that would manage them internally?

Bridge Crossing

```
int main(int argc, const char *argv[]) {
    Bridge bridge;
    thread cars[kNumCars];
    for (size_t i = 0; i < kNumCars; i++) {
        if (flipCoin()) {
            cars[i] =
        } else {
            cars[i] =
        }
    }
    for (thread& car
    return 0;
}
```

Imagine a variable type **Bridge** that you could have manage the following:

- “I need to cross!” – would block for you until you’re able to cross in a given direction.
- ”I’m done crossing!” – would automatically manage things to potentially allow cars going the other direction to proceed.

Bridge Crossing

Each car thread would run a function like this – the concurrency is managed internally inside the bridge variable!

```
static void crossBridgeEast(size_t id, Bridge& bridge) {
    approachBridge(); // sleep
    bridge.arrive_eastbound(id);
    driveAcross(); // sleep
    bridge.leave_eastbound(id);
}
```

Bridge Crossing

Each car thread would run a function like this – the concurrency is managed internally inside the bridge variable!

```
static void crossBridgeWest(size_t id, Bridge& bridge) {
    approachBridge(); // sleep
    bridge.arrive_westbound(id);
    driveAcross(); // sleep
    bridge.leave_westbound(id);
}
```

arrive_westbound/eastbound would need to know if we are able to cross, and either return immediately or block.

Bridge Crossing

Each car thread would run a function like this – the concurrency is managed internally inside the bridge variable!

```
static void crossBridgeWest(size_t id, Bridge& bridge) {
    approachBridge(); // sleep
    bridge.arrive_westbound(id);
    driveAcross(); // sleep
    bridge.leave_westbound(id);
}
```

leave_westbound/eastbound would need to be able to communicate with other threads who are waiting to cross in the other direction.

Demo: Starter Code

car-simulation.cc and bridge.hh/bridge.cc

Arriving Eastbound

arrive_eastbound needs to wait for it to be clear for the car to cross, and then let it cross.

- If other cars are already crossing eastbound, they can go
- If other cars are already crossing *westbound*, we have to wait

“Waiting for an event to happen” -> 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

Arriving Westbound

arrive_westbound needs to wait for it to be clear for the car to cross, and then let it cross.

- If other cars are already crossing westbound, they can go
- If other cars are already crossing *eastbound*, we have to wait

"Waiting for an event to happen" -> condition variable!

Plan For Today

- **Recap:** condition variables and dining philosophers
- **notify_one**
- Unique locks
- Monitor pattern
- **Example:** Bridge Crossing

Lecture 15 takeaway: The monitor pattern combines procedures and state into a class for easier management of synchronization. Then threads can call its thread-safe methods!

Next time: how does the OS schedule and switch between threads?

```
cp -r /afs/ir/class/cs111/lecture-code/lect15 .
```