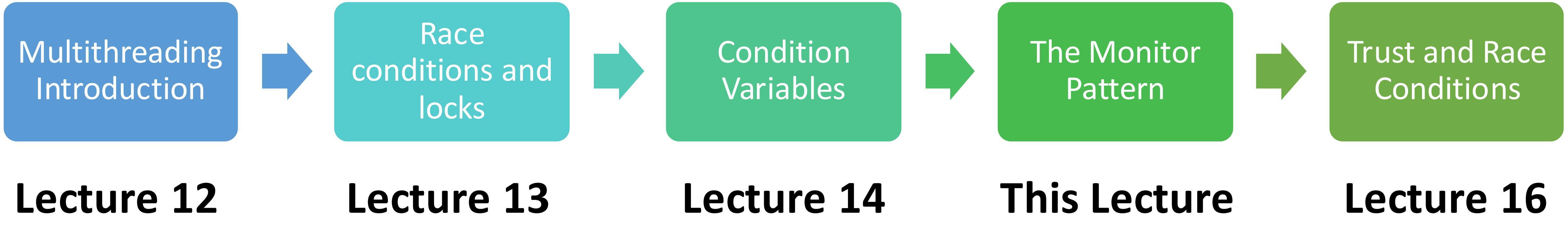
CS111, Lecture 15
The Monitor Pattern

CS111 Topic 3: Multithreading, Part 1

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



assign4: ethics exploration + implementing 2 *monitor pattern classes* for 2 multithreaded programs.

Plan For Today

- **Recap**: mutexes, condition variables and dining philosophers
- Monitor pattern
- **Example:** Bridge Crossing
- Unique Locks
- assign4

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

Plan For Today

- **Recap: mutexes, condition variables and dining philosophers**
- Monitor pattern
- **Example:** Bridge Crossing
- Unique Locks
- assign4

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

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(lock)** on the condition variable to sleep until another thread signals this condition variable (no busy waiting). The condition variable will unlock (at the beginning) and re-lock (at the end) the specified lock for us.
- You call **notify_all** on the condition variable to send a notification to all waiting threads and wake them up.
- Analogy: radio station – broadcast and tune in

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

waitForPermission (Final version)

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

This is the final implementation with the final version of wait() that takes a mutex parameter and which is called in a while loop.

Passing a Lock To CV.wait()

Why do we need to pass our mutex as a parameter to wait()?

- We must release the lock when waiting so someone else can put a permit back (which requires having the lock)
- But if we release the lock before calling wait, someone else could swoop in and put a permit back before we call wait(), meaning we will miss the notification!

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
      permitsLock.unlock();
      // AIR GAP HERE – someone could acquire the lock before we wait!
      permitsCV.wait();      // (note: not final form of wait)
      permitsLock.lock();
    }
    permits--;
    permitsLock.unlock();
}
```

Passing a Lock To CV.wait()

Why do we need to call wait() in a while loop?

- If we are waiting and then woken up by a notification, it's possible by the time we exit wait(), there are no permits, so we must wait again.
- Note: wait() reacquires the lock before returning
- *spurious wakeups* – wakeups up even when not being notified by another thread (!)

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
      permitsCV.wait(permitsLock);
      // by the time we wake up here, all the permits could already be gone!
    }
    permits--;
    permitsLock.unlock();
}
```

Plan For Today

- **Recap**: mutexes, condition variables and dining philosophers
- **Monitor pattern**
- **Example:** Bridge Crossing
- Unique Locks
- assign4

```
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**
  - That lock is required to access any of those variables

Monitor Design Pattern

The monitor pattern is a design pattern for writing multithreaded code, where we associate a single lock with a collection of related variables, e.g. a class.

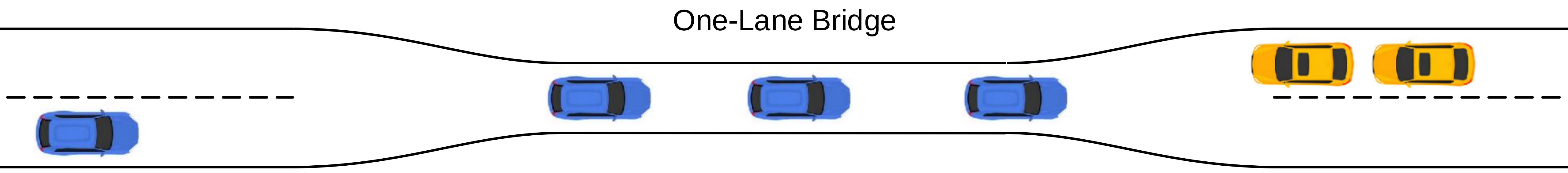
- For a multithreaded program, we can define a class that encapsulates the key multithreading logic and make an instance of it in our program.
- This class will have 1 mutex instance variable, and in all its methods we'll lock and unlock it as needed when accessing our shared state, so multiple threads can call the methods
- We can add any other state or condition variables we need as well – but the key idea is there is **one mutex** protecting access to all shared state, and which is locked/unlocked in the class methods that use the shared state.

Plan For Today

- **Recap**: mutexes, condition variables and dining philosophers
- Monitor pattern
- **Example: Bridge Crossing**
- Unique Locks
- assign4

```
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. Cars will arrive at the bridge at various points in time.
- 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

Demo: `car-simulation-no-monitor-soln`

Bridge Crossing

A car thread would execute one of these two functions:

```
static void cross_bridge_east(size_t id) {
    approach_bridge(); // sleep
    // TODO: wait until no cars going westbound
    driveAcross(); // sleep
    // now we have crossed
}

static void cross_bridge_west(size_t id) {
    approach_bridge(); // sleep
    // TODO: wait until no cars going eastbound
    driveAcross(); // sleep
    // now we have crossed
}
```

Arriving Eastbound

Key task: a thread needs to wait for it to be clear to cross.

E.g. car going eastbound:

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

“Waiting for an event to happen” -> condition variable!

For going east, we are waiting for the event ”no more cars are going westbound”.

State

What variables do we need to create to share across threads?

- 1 mutex to lock shared state
- Condition variable (for waiting to go east)
- ?? (for going east)
- Condition variable (for waiting to go west)
- ?? (for going west)

```
static void cross_bridge_east(size_t id) {
    approach_bridge(); // sleep
    // TODO: wait until no cars going westbound
    driveAcross(); // sleep
    // now we have crossed
}

static void cross_bridge_west(size_t id) {
    approach_bridge(); // sleep
    // TODO: wait until no cars going eastbound
    driveAcross(); // sleep
    // now we have crossed
}
```

Respond on PollEv:
pollev.com/cs111

What last two pieces of state/shared variables do we need?

Nobody has responded yet.

Hang tight! Responses are coming in.

State

What variables do we need to create to share across threads?

- 1 mutex to lock shared state
- Condition variable (for waiting to go east)
- Counter of cars crossing east
- Condition variable (for waiting to go west)
- Counter of cars crossing west

```
static void cross_bridge_east(size_t id) {
    approach_bridge(); // sleep
    // TODO: wait until no cars going westbound
    driveAcross(); // sleep
    // now we have crossed
}

static void cross_bridge_west(size_t id) {
    approach_bridge(); // sleep
    // TODO: wait until no cars going eastbound
    driveAcross(); // sleep
    // now we have crossed
}
```

Live Coding: Bridge Crossing

Plan For Today

- **Recap**: mutexes, condition variables and dining philosophers
- Monitor pattern
- **Example:** Bridge Crossing
- **Unique Locks**
- assign4

```
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.

leave_eastbound

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

```
void Bridge::leave_eastbound(size_t id) {
    bridge_lock.lock();
    n_crossing_eastbound--;
    if (n_crossing_eastbound == 0) {
        none_crossing_eastbound.notify_all();
    }
    print(id, "crossed", true);
    bridge_lock.unlock();
}
```

leave_eastbound

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

```
void Bridge::leave_eastbound(size_t id) {
    unique_lock<mutex> lock(bridge_lock);
    n_crossing_eastbound--;
    if (n_crossing_eastbound == 0) {
        none_crossing_eastbound.notify_all();
    }
    print(id, "crossed", true);
}
```

Auto-locks lock here

leave_eastbound

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

```
void Bridge::leave_eastbound(size_t id) {
    unique_lock<mutex> lock(bridge_lock);
    n_crossing_eastbound--;
    if (n_crossing_eastbound == 0) {
        none_crossing_eastbound.notify_all();
    }
    print(id, "crossed", true);
}
```

Auto-unlocks lock here (goes out of scope)

arrive_eastbound

```
void Bridge::arrive_eastbound(size_t id) {
    bridge_lock.lock();
    print(id, "arrived", true);
    while (n_crossing_westbound > 0) {
        none_crossing_westbound.wait(bridge_lock);
    }
    n_crossing_eastbound++;
    print(id, "crossing", true);
    bridge_lock.unlock();
}
```

arrive_eastbound

```
void Bridge::arrive_eastbound(size_t id) {
    unique_lock<mutex> lock(bridge_lock);
    print(id, "arrived", true);
    while (n_crossing_westbound > 0) {
        none_crossing_westbound.wait(lock);
    }
    n_crossing_eastbound++;
    print(id, "crossing", true);
}
```

Auto-locks lock here

arrive_eastbound

```
void Bridge::arrive_eastbound(size_t id) {
    unique_lock<mutex> lock(bridge_lock);
    print(id, "arrived", true);
    while (n_crossing_westbound > 0) {
        none_crossing_westbound.wait(lock);
    }
    n_crossing_eastbound++;
    print(id, "crossing", true);
}
```

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

arrive_eastbound

```
void Bridge::arrive_eastbound(size_t id) {
    unique_lock<mutex> lock(bridge_lock);
    print(id, "arrived", true);
    while (n_crossing_westbound > 0) {
        none_crossing_westbound.wait(lock);
    }
    n_crossing_eastbound++;
    print(id, "crossing", true);
}
```

Auto-unlocks lock here (goes out of scope)

Plan For Today

- **Recap:** mutexes, condition variables and dining philosophers
- Monitor pattern
- **Example:** Bridge Crossing
- Unique Locks
- **assign4**

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

Assign4

Assign4: ethics exploration + implementing 2 *monitor pattern classes* for 2 multithreaded programs.

- Data structures can be used to store condition variables or state
- Structs also helpful to bundle state together and make multiple instances of structs
- Note: when you add elements to C++ data structures (e.g. vector, queue, set, map) it inserts *copies*.
- **condition variables cannot be copied**. E.g. cannot create a condition variable and push onto vector.
- **For two above bullets:** consider how pointers can help!
- **Types:** make sure to use **condition_variable_any**, and only **notify_all** for condition variables (there's also **notify_one**, but it's not necessary for assign4)

Recap

- **Recap:** mutexes, condition variables and dining philosophers
- Monitor pattern
- **Example:** Bridge Crossing
- Unique Locks
- assign4

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: race conditions, trust and operating systems

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