CS111, Lecture 16
The Monitor Pattern
Topic 3: Multithreading - How can we have concurrency within a single process? How does the operating system support this?

Lecture 12: Multithreading Introduction
Lecture 13: Race conditions and locks
Lecture 14: Condition Variables
Lecture 15: Trust, Race Conditions and Multithreading Patterns
This Lecture: The Monitor Pattern

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

• Monitor pattern
• **Example:** Bridge Crossing
• Unique Locks
• assign4

```bash
cp -r /afs/ir/class/cs111/lecture-code/lect16 .
```
Plan For Today

• Monitor pattern
• Example: Bridge Crossing
• Unique Locks
• assign4

cp -r /afs ir/class/cs111/lecture-code/lect16
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

• Monitor pattern

• Example: Bridge Crossing

• Unique Locks

• assign4

```bash
cp -r /afs/ir/class/cs111/lecture-code/lect16 .
```
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
A car thread would execute one of these two functions:

```c
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".*
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)

```c
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 or text CS111 to 22333 once to join.
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

```c
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

- Monitor pattern
- **Example**: Bridge Crossing
- **Unique Locks**
- assign4

```bash
cp -r /afs(ir/class/cs111/lecture-code/lect16 .
```
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.
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.

```cpp
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 permitsLock here
We lock at the beginning of this function and unlock at the end.

```cpp
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 permitsLock here (goes out of scope)**
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();
}
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 permitsLock here
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!)
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 permitsLock here (goes out of scope)
Plan For Today

• Monitor pattern
• **Example:** Bridge Crossing
• Unique Locks
• assign4

```
  cp -r /afs/ir/class/cs111/lecture-code/lect16 .
```
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!
Recap

• Monitor pattern
• **Example:** Bridge Crossing
• Unique Locks
• assign4

Lecture 16 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 run and switch between threads?

```bash
cp -r /afs(ir/class/cs111/lecture-code/lect16
```