

CS111, Lecture 16

Multithreading Patterns 2



masks strongly
recommended

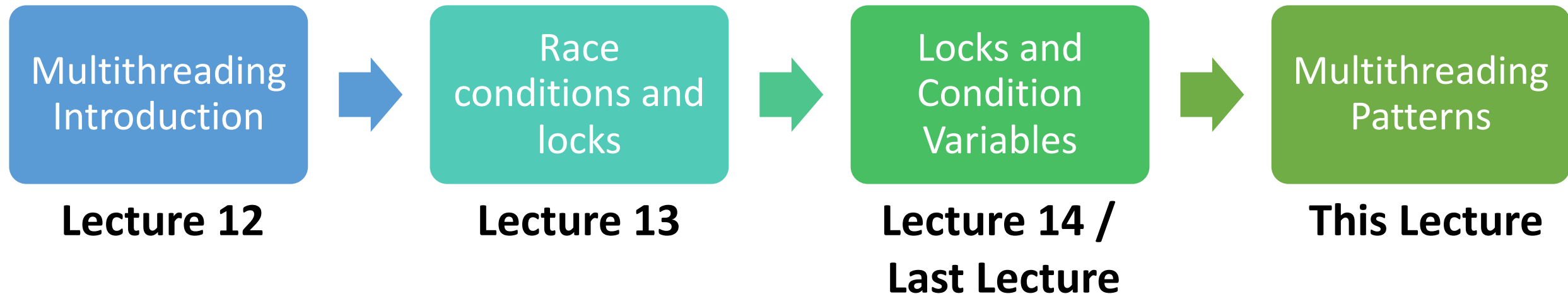
This document is copyright (C) Stanford Computer Science and Nick Troccoli, licensed under Creative Commons Attribution 2.5 License. All rights reserved.

Based on slides and notes created by John Ousterhout, Jerry Cain, Chris Gregg, and others.

NOTICE RE UPLOADING TO WEBSITES: This content is protected and may not be shared, uploaded, or distributed. (without expressed written permission)

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



assign4: implement several multithreaded programs while eliminating race conditions!

Learning Goals

- Understand C++ lambda functions and how they can help us write multithreaded code
- Learn about the **monitor** pattern for designing multithreaded code in the simplest way possible, using classes.

Plan For Today

- **Recap:** Multithreading So Far
- Lambda Functions
- Monitor pattern
- **Example:** Bridge Crossing

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

Plan For Today

- **Recap: Multithreading So Far**
- Lambda Functions
- Monitor pattern
- **Example:** Bridge Crossing

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

Multithreading Key Takeaways

- Synchronization code is powerful – it allows us to run multiple tasks at the same time and coordinate between them.
- Synchronization code is tricky to get right – the coordination can become complex, and sharing data introduces many opportunities for race conditions/
- **Mutexes** and **condition variables** are our 2 main tools to synchronize threads

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

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

Plan For Today

- Recap: Multithreading So Far
- **Lambda Functions**
- Monitor pattern
- Example: Bridge Crossing

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

Lambda Functions

In C++, it's possible to define an “anonymous function” that doesn't have a name, and whose code is just written inline.

- We can spawn threads in this way – instead of specifying a function name, we can write the code we want the thread to run directly inline.

Revisiting Friends

```
static void greeting(size_t i) {  
    cout << oslock << "I am thread " << i << endl << osunlock;  
}
```

...



Revisiting Friends

```
static const size_t kNumFriends = 6;

int main(int argc, char *argv[]) {
    cout << "Let's hear from " << kNumFriends << " threads." << endl;

    thread friends[kNumFriends];
    for (size_t i = 0; i < kNumFriends; i++) {
        friends[i] = thread(greeting, i);
    }

    // Wait for threads
    for (size_t i = 0; i < kNumFriends; i++) {
        friends[i].join();
    }

    cout << "Everyone's said hello!" << endl;
    return 0;
}
```

Friends with Lambda

```
static const size_t kNumFriends = 6;

int main(int argc, char *argv[]) {
    cout << "Let's hear from " << kNumFriends << " threads." << endl;

    thread friends[kNumFriends];
    for (size_t i = 0; i < kNumFriends; i++) {
        friends[i] = thread([](int i) -> void {
            cout << oslock << "I am thread " << i << endl << osunlock;
        }, i);
    }
    ...
}
```

Inline version of:

```
static void greeting(size_t i) {
    cout << oslock << "I am thread " << i << endl << osunlock;
}
```

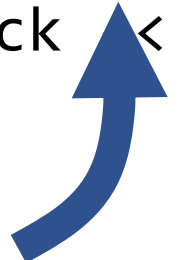
Friends with Lambda

```
static const size_t kNumFriends = 6;

int main(int argc, char *argv[]) {
    cout << "Let's hear from " << kNumFriends << " threads." << endl;

    thread friends[kNumFriends];
    for (size_t i = 0; i < kNumFriends; i++) {
        friends[i] = thread([](int i) -> void {
            cout << oslock << "I am thread " << i << endl << osunlock;
        }, i);
    }
    ...
}
```

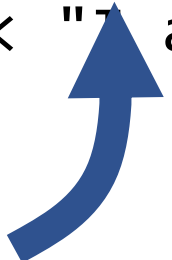
**Empty (for now)
square brackets**



Friends with Lambda

```
static const size_t kNumFriends = 6;
```

```
int main(int argc, char *argv[]) {  
    cout << "Let's hear from " << kNumFriends << " threads." << endl;  
  
    thread friends[kNumFriends];  
    for (size_t i = 0; i < kNumFriends; i++) {  
        friends[i] = thread([](int i) -> void {  
            cout << oslock << " am thread " << i << endl << osunlock;  
        }, i);  
    }  
    ...  
}
```



Parameter list


Friends with Lambda

```
static const size_t kNumFriends = 6;

int main(int argc, char *argv[]) {
    cout << "Let's hear from " << kNumFriends << " threads." << endl;

    thread friends[kNumFriends];
    for (size_t i = 0; i < kNumFriends; i++) {
        friends[i] = thread([](int i) -> void {
            cout << oslock << "I am thread " << i << endl << osunlock;
        }, i);
    }
    ...
}
```

Return type




Friends with Lambda

```
static const size_t kNumFriends = 6;

int main(int argc, char *argv[]) {
    cout << "Let's hear from " << kNumFriends << " threads." << endl;

    thread friends[kNumFriends];
    for (size_t i = 0; i < kNumFriends; i++) {
        friends[i] = thread([](int i) -> void {
            cout << oslock << "I am thread " << i << endl << osunlock;
        }, i);
    }
    ...
}
```

Function body




Friends with Lambda

```
static const size_t kNumFriends = 6;

int main(int argc, char *argv[]) {
    cout << "Let's hear from " << kNumFriends << " threads." << endl;

    thread friends[kNumFriends];
    for (size_t i = 0; i < kNumFriends; i++) {
        friends[i] = thread([](int i) -> void {
            cout << oslock << "I am thread " << i << endl << osunlock;
        }, i);
    }
    ...
}
```



**Parameter values passed
into thread constructor
(same as before)**

Lambda Functions

The real power of lambdas come when a function takes another function as a parameter, and we can specify the function parameter as a lambda.

Lambda Functions

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

This while loop pattern is so common that there is another convenience form of **wait** that also includes the loop.

- There is a second parameter which is a *function*: it should return true when we wish to stop repeatedly waiting for a notification.

CV Wait With Lambda

```
void condition_variable_any::wait(mutex& m, Function f) {  
    while (!f()) wait(m);  
}
```

Functions like **wait** take in a function as a parameter, and they may call the function later. When they do, they don't know what parameters to call it with since they don't know what the function is. So it cannot take in any parameters.

Lambda Functions

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    permitsCV.wait(permitsLock, []() -> bool {
        return permits > 0; // how do we access permits without parameters??
    });
    permits--;
    permitsLock.unlock();
}
```

The [] can contain a list of variables we want to *capture* from the surrounding code. We can capture by reference or by copy. Anything we capture can be used in the lambda body!

Lambda Functions

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    permitsCV.wait(permitsLock, [&permits]() -> bool {
        return permits > 0;
    });
    permits--;
    permitsLock.unlock();
}
```

The [] can contain a list of variables we want to *capture* from the surrounding code. We can capture by reference or by copy. Anything we capture can be used in the lambda body!

Lambda Functions

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    permitsCV.wait(permitsLock, [&permits]() -> bool {
        return permits > 0;
    });
    permits--;
    permitsLock.unlock();
}
```

Capturing is one of the most powerful uses of lambda functions. It means we can define functions that take no parameters, but which can reference values in their surrounding scope.

Plan For Today

- Recap: Multithreading So Far
- Lambda Functions
- **Monitor pattern**
- Example: Bridge Crossing

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

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

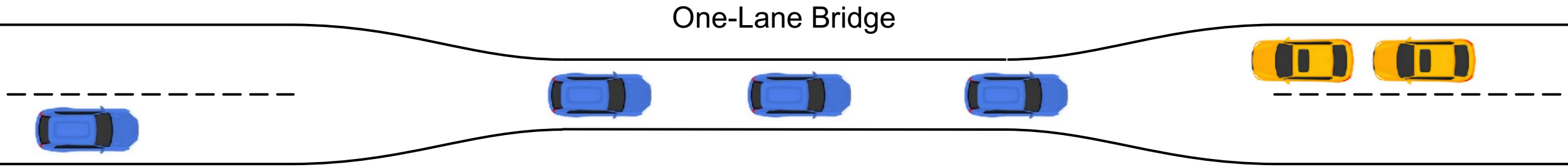
- For a multithreaded program, we can define a class that encapsulates the key multithreading logic and make an instance of it in our program.
- This class will have 1 mutex instance variable, and in all its methods we'll lock and unlock it as needed when accessing our shared state, so multiple threads 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: Multithreading So Far
- Lambda Functions
- Monitor pattern
- **Example: Bridge Crossing**

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

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& car : cars) car.start();  
    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!

Recap

- **Recap:** Multithreading So Far
- Lambda Functions
- 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 run and switch between threads?

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