CS 111 Assignment 4: Synchronization

Synchronization

- **What is it?**
  - Mechanisms to coordinate interactions between “things” executing concurrently
- **Why is it important?**
  - **Performance**: we often want to enable concurrent execution to increase the performance of our systems
  - **Correctness**: when dealing with concurrent execution, without synchronization we can have “incorrect” programs (e.g. data races)
- **How do we synchronize?**
  - Lots of ways! Two core synchronization mechanisms with threading are mutexes and condition variables.
- **Why do we need condition variables? Are mutexes not enough?**
  - They were designed for different purposes (more on this later)!

Mutexes

- **Purpose:** provide mutually exclusive access to a shared resource.
  - Shared resource - e.g. threads share the same memory
  - Mutually exclusive access – only one thread can access the shared resource at a time (no data races!)
- **Mechanics:**
  - `mutex.lock()`: Locks the mutex (or blocks the calling thread until the lock is acquired). Note: this can lead to deadlock.
  - `mutex.unlock()`: Releases the mutex.
- **Best practices:**
  - Identify where there's shared data being accessed – e.g. multiple threads accessing the same data AND at least 1 writer (if ALL only read this is ok). Lock around this.
  - Keep critical sections as small as possible (for performance). Can start with large critical sections when trying to make program "correct".

Unique Locks

- **Constructor and destructor automatically lock and unlock mutex**
  - Can’t accidentally forget to unlock!

- **Without unique_lock:**

```
void foo()
{
    mutex.lock();
    ...
    if (...) {
        mutex.unlock();
        return;
    }
    ...
    mutex.unlock();
}
```

- **With unique_lock:**

```
void foo()
{
    unique_lock<mutex> lock(mutex);
    ...
    if (...) {
        return;
    }
    ...
}
```

Condition Variables

- **Purpose:** designed specifically for use as a notification mechanism. E.g. imagine you're a developer writing multithreaded code and you only want some threads to execute code once a particular condition holds…
- **Mechanics**:
  - `cv.wait(lk)`: Atomically unlocks `lk`, blocks current thread until notified (or spurious wakeup). Acquires `lk` when woken up.
  - `cv.notify_all()`: Unblocks ALL threads waiting on `cv`.

```
void foo()
{
    mutex.lock();

    ...   // thread holds lock

    // thread releases lock
    cv.wait(mutex);
    // thread reacquires lock

    ...   // thread holds lock

    mutex.unlock();
}
```

Condition Variable Strategy

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

Next slide goes over the mechanics of how to do 1. and 2. properly!

Condition Variables

- **Best practices:**
  - CP.42: Don't wait without a condition (handling spurious wakeup):
    - Either you can check the condition in a while loop…
    - Or you can use a lambda!
- **General workflow (mechanics):**
  1. Define a "condition", e.g. bool dataIsReady, or count == something
  2. Use a mutex to guard access to the variable(s) used in the condition
  3. Use this same mutex in cv method calls

Some shared state between threads!

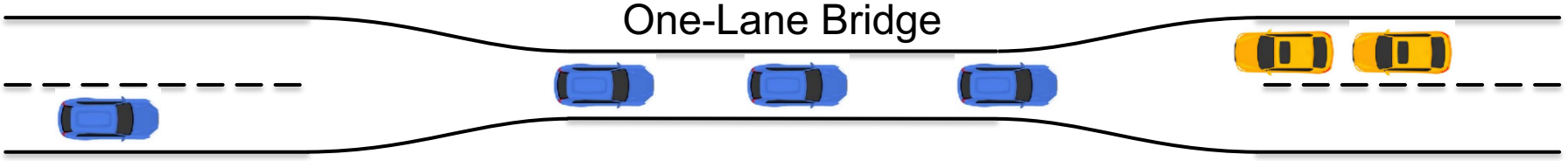
```
bool stop_waiting() {
    return true;
}

while (!stop_waiting()) {
    cv.wait(lock);
)
```

Is equivalent to…

```
cv.wait(lock, []{return true;});
cv.wait(lock, []() -> bool {return true;});
```

Bridge Problem



- **All cars on bridge must be travelling in the same direction**
- **Any number of cars can be on the bridge at once**

Code for Cars

- **Eastbound cars:**

```
bridge.arrive_eastbound();
... drive across bridge ...
bridge.leave_eastbound();
```

- **Westbound cars:**

```
bridge.arrive_westbound();
... drive across bridge ...
bridge.leave_westbound();
```

State Variables

- **What information is needed to know whether an eastbound car can enter the bridge?**

First cut at bridge.hh

```
class Bridge {
public:
    Bridge();
    void arrive_eastbound();
    void leave_eastbound();
    void arrive_westbound();
    void leave_westbound();

private:
    // Synchronizes access to all info in
    // this object.
    std::mutex bridgeLock;

    // Number of cars currently crossing in
    // each direction (only one of
    // these can be nonzero at any given
    // time).
    size_t crossing_eastbound = 0;
    size_t crossing_westbound = 0;
}
```

Maintain state variables

```
void Bridge::arrive_eastbound() {
    unique_lock<mutex> lock(bridgeLock);

    /* Wait until safe to cross */

    crossing_eastbound++;
}
```

```
void Bridge::leave_eastbound() {
    unique_lock<mutex> lock(bridgeLock);
    crossing_eastbound--;

    /* Maybe wake up westbound cars */
}
```

Final bridge.hh

```
class Bridge {
public:
    Bridge();
    void arrive_eastbound();
    void leave_eastbound();
    void arrive_westbound();
    void leave_westbound();

private:
    // Synchronizes access to all info in
    // this object.
    std::mutex bridgeLock;

    // Number of cars currently crossing in
    // each direction (only one of
    // these can be nonzero at any given
    // time).
    size_t crossing_eastbound = 0;
    size_t crossing_westbound = 0;

    // Signaled to indicate that there are
    // no longer any cars crossing
    // in the eastbound direction.
    std::condition_variable_any done_eastbound;

    // Same for the westbound direction.
    std::condition_variable_any done_westbound;
};
```

Final bridge.cc

```
void Bridge::arrive_eastbound() {
    unique_lock<mutex> lock(bridgeLock);

    /* Wait until safe to cross */
    done_westbound.wait(lock, [this]() -> bool {
        return crossing_westbound == 0;
    });

    crossing_eastbound++;
}
```

```
void Bridge::leave_eastbound() {
    unique_lock<mutex> lock(bridgeLock);
    crossing_eastbound--;

    /* Maybe wake up westbound cars */
    if (crossing_eastbound == 0) {
        done_eastbound.notify_all();
    }
}
```

```
// same for arrive_westbound / leave_westbound
```

CalTrain Automation

- **Train:**
  - Calls `load_train(int available)` when it arrives at the station
  - `available` is the number of empty seats on the train.
  - Must block until either
    - No more seats are available and all passengers are seated, or
    - No more passengers are waiting at the station and all passengers are seated
- **Passenger:**
  - Calls `wait_for_train()` when it arrives at the station
  - `wait_for_train()` blocks until a train arrives with open seats
    - i.e. `load_train()` is called by a train thread
  - When `wait_for_train` returns, passenger starts boarding
  - Calls `boarded()` once passenger is safely in seat.

CalTrain Automation

- **Don’t overbook train!**
  - Once passenger starts boarding, that means one fewer seat available
- **Wait for passengers to sit down!**
  - A passenger boarding has not sat down until it calls `boarded()`
  - No available seats does not mean all passengers are seated
- **Passengers must be able to board concurrently!**
  - Don’t board passengers one at a time

Party Introductions

- **Program a mechanism that pairs people up at a party based on their Zodiac signs and preferences**

- **Each person invokes:**
  `std::string meet(std::string &my_name, int my_sign, int other_sign)`
  - `my_name`: name of person
  - `my_sign`: person's Zodiac sign (integer in {0 … 11})
  - `other_sign`: Zodiac sign they'd like to meet
  - Most block until a suitable match is available, then return the name of the matching person

Party Introductions

- **Matches must be mutual:**
  - Suppose Bob has sign 3, wants to meet sign 6
  - If Alice has sign 6 and wants to meet sign 3, then they can match
  - Each person must receive the other person's name
  - If Casey has sign 6 and wants to meet sign 4, cannot match with Bob
- **Matches must occur in parallel:**
  - If Bob is waiting for a match, shouldn't prevent others from being matched

Party Introductions

- **bool has_matching_waiting_guest(int my_sign, int other_sign);**
  - Assumes caller has mutex_ (as do all of these functions)
- **WaitingGuest& get_matching_waiting_guest(int my_sign, int other_sign);**
  - Removes the first guest that matches from the waiting guests. Store result in a variable of type WaitingGuest&

```
WaitingGuest& match = get_matching_waiting_guest(...);
```

- **void add_waiting_guest(int my_sign, int other_sign, WaitingGuest& guest);**
  - This keeps track of the order, so the first guest that calls this function will be the first guest returned by get_matching_waiting_guest (among all guests that match)

General Guidelines

- **Your code will not call the methods you implement**
  - Our test harness will spawn threads, construct objects, and invoke your methods
- **You must use the monitor style discussed in lecture**
  - Exactly one mutex per Station or Party object
- **It should be possible have multiple Station or Party objects**
  - Each operates independently (with its own mutex)
- **Simplicity is crucial:**
  - If it's complex, it probably won't work