CS 111 Project 3
Synchronization
Outline

1. Review of mutexes and condition variables
2. Part 1: CalTrain Automation
3. Part 2: Party Introductions
Outline

1. Review of mutexes and condition variables
2. Part 1: CalTrain Automation
3. Part 2: Party Introductions
Review: Mutexes

- a.k.a. “locks”
- Create “critical sections” of code
  - Only one thread can lock the mutex, other threads must wait until unlocked
- Needed to protect reads and writes to shared variables that are not thread-safe
Why mutexes?

- Consider simple line of code: `i--;`
  - Here `i` is an `int`
- What happens if two threads run this same line of code concurrently?
- `i--` can compile into 3 distinct instructions
  - Load the current value of `i` into register
  - Decrement value of `i` in register
  - Store value `i` in register back into memory
Why mutexes?

1. **Thread 1** loads $i = 5$ into register
2. **Thread 1** decrements $i$ to 4 in register
3. **Thread 1** preempted
4. 
5. 
6. 
7. 
8. **Thread 1** stores $i = 4$ back into memory

1. 
2. 
3. 
4. **Thread 2** loads $i = 5$ into register
5. **Thread 2** decrements $i = 4$ in register
6. **Thread 2** stores $i = 4$ back into memory
7. **Thread 2** preempted

Result: even though $i--;$ was called twice, the final value of $i$ is 4 and not 3
Why mutexes?

- Simple fix: require that thread decrementing i be holding mutex
  - i.e., `m.lock(); i--; m.unlock();`
Review: lock_guard

- Convenience class that wraps around a **mutex**
- Constructor takes in *unlocked* **mutex** and **locks** it
- Destructor **unlocks** **mutex** that was given to constructor
Usefulness of lock_guard

```cpp
// Assume shared mutex "m"

m.lock();
if (shared_variable == 1) {
    return val_1;
} else if (shared_variable == 2) {
    return val_2;
} else if (shared_variable == 3) {
    return val_3;
} else {
    return val_4;
}
```

```cpp
// Assume shared mutex "m"

m.lock();
if (shared_variable == 1) {
    m.unlock();
    return val_1;
} else if (shared_variable == 2) {
    m.unlock();
    return val_2;
} else if (shared_variable == 3) {
    m.unlock();
    return val_3;
} else {
    m.unlock();
    return val_4;
}
```
Usefulness of lock_guard: put function signature

```cpp
// Assume shared mutex "m"

std::lock_guard<std::mutex> lg(m); // m is now locked!
if (shared_variable == 1) {
    return val_1;
} else if (shared_variable == 2) {
    return val_2;
} else if (shared_variable == 3) {
    return val_3;
} else {
    return val_4;
}
// m gets unlocked since lg goes out of scope!
```
Review: Condition Variables

- Used to block threads when they are waiting for some work to be done
  - Review: What is busy waiting, and why do we avoid it?
- Important methods
  - `wait(std::mutex& m)`: Takes in locked mutex, atomically unlocks mutex and blocks thread
    - Before thread emerges from `wait()` it re-locks mutex
  - `notify_one/all()`: Wakes one/all threads blocked on `wait()`
- Important: Threads that call `wait()` must be woken via `notify_one/all()`!
  - In other words, waiting threads don’t automatically wake when condition changes
Code example: semaphores

- Alternative synchronization primitive
- A semaphore helps threads keep track of a shared resource in a thread-safe manner
- Stores an internal counter that represents the current amount of this resource
  - e.g., the number of items in a shared queue
- Has two functions:
  - `wait()`: Thread blocks until internal counter is *positive*, then decrements counter by 1
  - `signal()`: Thread increments internal counter and wakes any threads blocking on `wait()`
Semaphore example

```cpp
std::queue<std::function<void(void)>> function_queue;
std::mutex function_queue_lock;
semaphore num_functions_in_queue;

void schedule_func(std::function<void(void)> func) {
    std::lock_guard<std::mutex> lg(function_queue_lock);
    function_queue.push(func);
    num_functions_in_queue.signal();
}

void worker_main() {
    num_functions_in_queue.wait();
    function_queue_lock.lock();
    std::function<void(void)> func = function_queue.front();
    function_queue.pop();
    function_queue_unlock();
    func();
}
```
Semaphore implementation

class Semaphore {
  public:
    Semaphore(int count = 0) : count(count) {}  
    void wait() {
      std::lock_guard<std::mutex> lg(lock);
      while (count <= 0) cv.wait(lock);
      count--;
    }
    void signal() {
      std::lock_guard<std::mutex> lg(lock);
      count++;
      if (count > 0) cv.notify_all();
    }
  private:
    int count;
    std::mutex lock;
    std::condition_variable_any cv;
};
Outline

1. Review of mutexes and condition variables
2. Part 1: CalTrain Automation
3. Part 2: Party Introductions
CalTrain Automation

- You will program the behavior of trains and passengers
  - Each is represented by a single thread
- Passengers wait for arrival of train with open seats and then board
- Trains arrive at station, wait until fully boarded or no more waiting passengers
CalTrain Automation

- **Train**: calls `load_train(int available)` when it arrives to the station
  - `int available` is the number of open seats on train
  - Blocks until either
    - i. No more seats are available and *all passengers are seated*
    - ii. No more passengers are waiting at station
- **Passenger**: calls `wait_for_train()` when it arrives to station, calls `seated()` once sitting down
  - `wait_for_train()` blocks until a train arrives with open seats
    - i. i.e., `load_train` is called by train thread
  - `seated()` is called by passenger at some point after `wait_for_train()` returns
    - i. Time in between represents boarding time for passenger
CalTrain Automation

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

1. Review of mutexes and condition variables
2. Part 1: CalTrain Automation
3. Part 2: Party Introductions
Party Introductions

- You will program a mechanism for partygoers to meet someone of a specified gender
  - Gender represented by non-negative integer in \( \{0, \ldots, \text{MAX\_GENDER}\} \)
- Partygoers have their own name and gender `my\_gender`, specify gender `gender\_to\_meet`
  - Wait until another partygoer of gender `gender\_to\_meet` who wants to meet `my\_gender` arrives
- Matched partygoers receive the name of their match
Party Introductions

- One function to implement
  - `std::string meet(std::string &my_name, int my_gender, int other_gender);

- Example
  - Bob calls `meet("Bob", 0, 1);
    - Must wait until someone calls `meet` with `my_gender == 1` and `other_gender == 0`
  - Alice calls `meet("Alice", 1, 0);
  - Bob returns from `meet` with value "Alice"
  - Alice returns from `meet` with value "Bob"
Party Introductions

- Make sure that both gender values in `meet` are matched
  - Bob is of gender 3, wants to meet someone of gender 4
  - Alice is of gender 4, wants to meet someone of gender 2
  - These two partygoers should not be matched!
- Matches must be mutual
  - If Bob receives Alice’s name, then Alice receives Bob’s name
- Matches must occur in parallel
  - If Bob is waiting for a match, shouldn’t prevent others from being matched
General Guidelines

- Your code will not call the functions you implement
  - Our test harness will spawn threads and invoke your functions
- **You must** utilize the *monitor-style* design paradigm discussed in lecture
  - Exactly one *mutex* per object instance! (i.e., per Station or Party)
  - ex) *Pipe* class from lecture
- Think before you code
  - Not many lines of code required, more time for design, brainstorming
- Simplicity is crucial
  - We’ll take points off from overly-complex solutions
- **Station** and **Party** class instances should operate independently