

# CS111, Lecture 13

## Race Conditions and Locks

Optional reading:

Operating Systems: Principles and Practice (2<sup>nd</sup> Edition): Sections 5.2-5.4  
and Section 6.5



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.

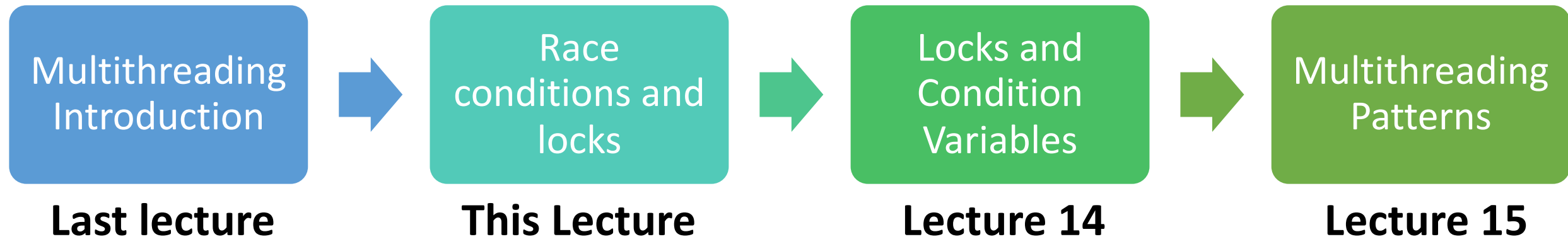
NOTICE RE UPLOADING TO WEBSITES: This content is protected and may not be shared, uploaded, or distributed. (without expressed written permission)

# Announcements

- Updates to lecture attendance policy – now using lecture checkin quizzes
- Midterm details posted (paper exam, review materials coming soon!)
- New Fri. 3:30-4:20PM section added this week, feel free to join!
- Assign3: check out YEAH Hours materials, particularly for 2-process pipelines!

# **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

- Discover some of the pitfalls of threads sharing the same virtual address space
- Understand how to identify critical sections and fix race conditions/deadlock
- Learn how locks can help us limit access to shared resources

# Plan For Today

- **Recap:** Threads
- Threads Share Memory
- Critical Sections
- Mutexes

```
cp -r /afs/ir/class/cs111/lecture-code/lect13 .
```

# Plan For Today

- **Recap: Threads**
- Threads Share Memory
- Critical Sections
- Mutexes

```
cp -r /afs/ir/class/cs111/lecture-code/lect13 .
```

# From Processes to Threads

We can have concurrency *within a single process* using **threads**: independent execution sequences within a single process.

- Threads let us run multiple functions in our program concurrently (e.g. parallelize computation)
- Each thread operates within the same process, so they *share a virtual address space* (!) (globals, heap, pass by reference, etc.)



# C++ Thread

A thread object can be spawned to run the specified function with the given arguments.

```
thread myThread(myFunc, arg1, arg2, ...);
```

- **myFunc**: the function the thread should execute asynchronously
- **args**: a list of arguments (any length, or none) to pass to the function
- **myFunc**'s function's return value is ignored (use pass by reference instead)
- Once initialized with this constructor, the thread may execute at any time!

# C++ Thread

To wait on a thread to finish, use the `.join()` method:

```
thread myThread(myFunc, arg1, arg2);  
...  
// Wait for thread to finish (blocks)  
myThread.join();
```

For multiple threads, we must wait on a specific thread one at a time:

```
thread friends[5];  
...  
for (int i = 0; i < 5; i++) {  
    friends[i].join();  
}
```

# Race Conditions

- Like with processes, threads can execute in unpredictable orderings.
- A **race condition** is an unpredictable ordering of events where some orderings may cause undesired behavior.
- An example where race conditions can occur is with **operator<<**. e.g. **cout** statements could get interleaved!
- To avoid this, use **oslock** and **osunlock** (custom CS111 functions - **#include "ostreamlock.h"**) around streams. They ensure at most one thread has permission to write into a stream at any one time.

```
cout << oslock << "Hello, world!" << endl << osunlock;
```

# Plan For Today

- Recap: Threads
- **Threads Share Memory**
- Critical Sections
- Mutexes

```
cp -r /afs/ir/class/cs111/lecture-code/lect13 .
```

# Threads Share Memory

Unlike parent/child processes, threads execute in the same virtual address space

- This means we can e.g. pass parameters by reference and have all threads access/modify them!
- To pass by reference with **thread()**, we must use the special **ref()** function around any reference parameters when we create a thread:

```
static void greeting(size_t& i) {  
    ...  
}  
  
for (size_t i = 0; i < kNumFriends; i++) {  
    friends[i] = thread(greeting, ref(i));  
}
```



# Threads Share Memory

- Here, all threads are referencing the same copy of `i`, which is updated in the **for** loop. It could be that by the time the threads access it, it's already been incremented all the way to 6!
- While in this example we can just pass by copy, we must keep an eye out for the consequences of shared memory.

Let's see another example of the potential for race condition problems.

# Parallelizing Tasks

Threads allow a process to parallelize a program across multiple cores.

- Consider a scenario where we want to sell 250 tickets and have 10 cores
- **Simulation:** let each thread help sell tickets until none are left

# Parallelizing Tasks

**Simulation:** let each thread help sell the 250 tickets until none are left.

```
const size_t kNumTicketAgents = 10;
int main(int argc, const char *argv[]) {
    thread ticketAgents[kNumTicketAgents];
    size_t remainingTickets = 250;

    for (size_t i = 0; i < kNumTicketAgents; i++) {
        ticketAgents[i] = thread(sellTickets, i, ref(remainingTickets));
    }

    for (size_t i = 0; i < kNumTicketAgents; i++) {
        ticketAgents[i].join();
    }
    cout << "Ticket selling done!" << endl;
    return 0;
}
```



**Demo: confused-ticket-  
agents.cc**

# Overselling Tickets

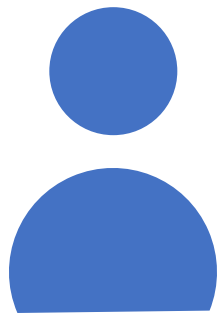
```
static void sellTickets(size_t id, size_t& remainingTickets) {
    while (remainingTickets > 0) {
        sleep_for(500); // simulate "selling a ticket"
        remainingTickets--;
        cout << oslock << "Thread #" << id << " sold a ticket ("
            << remainingTickets << " remain)." << endl << osunlock;
    }
    cout << oslock << "Thread #" << id
        << " sees no remaining tickets to sell and exits." << endl << osunlock;
}
```

What might have caused us to oversell tickets?

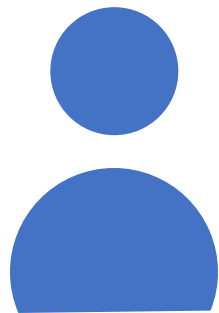
# Race Condition: Overselling Tickets

```
static void sellTickets(size_t id, size_t& remainingTickets) {  
    while (remainingTickets > 0) {  
        sleep_for(500); // simulate "selling a ticket"  
        remainingTickets--;  
        cout << oslock << "Thread #" << id << " sold a ticket ("  
            << remainingTickets << " remain)." << endl << osunlock;  
    }  
    cout << oslock << "Thread #" << id  
    << " sees no remaining tickets to sell and exits." << endl << osunlock;  
}
```

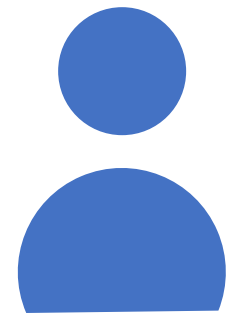
**remainingTickets = 1**



Thread #1



Thread #2



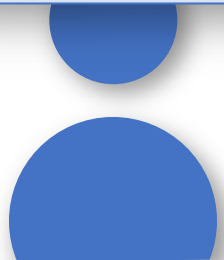
Thread #3

# Race Condition: Overselling Tickets

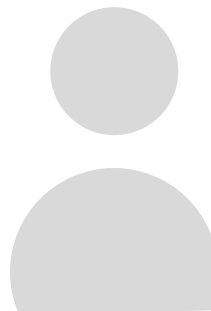
```
static void sellTickets(size_t id, size_t& remainingTickets) {  
    while (remainingTickets > 0) {  
        sleep_for(500); // simulate "selling a ticket"  
        remainingTickets--;  
        cout << oslock << "Thread #" << id << " sold a ticket ("  
            << remainingTickets << " remain)." << endl << osunlock;  
    }  
    cout << oslock << "Thread #" << id  
        << " sees no remaining tickets to sell and exits." << endl << osunlock;  
}
```

**remainingTickets = 1**

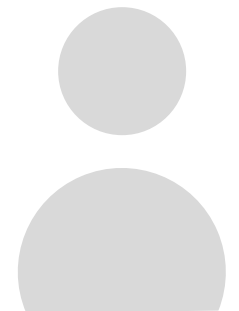
Are there tickets  
to sell? Yep!



Thread #1



Thread #2



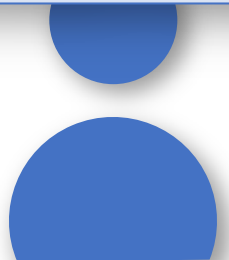
Thread #3

# Race Condition: Overselling Tickets

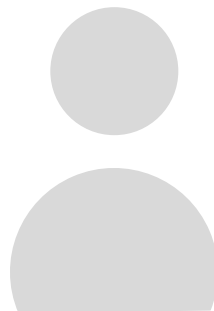
```
static void sellTickets(size_t id, size_t& remainingTickets) {  
    while (remainingTickets > 0) {  
        sleep_for(500); // simulate "selling a ticket"  
        remainingTickets--;  
        cout << oslock << "Thread #" << id << " sold a ticket ("  
            << remainingTickets << " remain)." << endl << osunlock;  
    }  
    cout << oslock << "Thread #" << id  
        << " sees no remaining tickets to sell and exits." << endl << osunlock;  
}
```

**remainingTickets = 1**

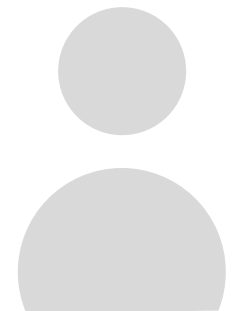
Are there tickets  
to sell? Yep!



Thread #1



Thread #2



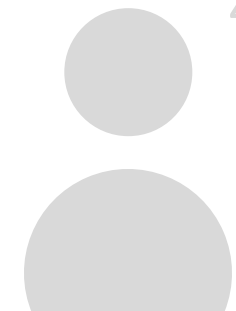
Thread #3

# Race Condition: Overselling Tickets

```
static void sellTickets(size_t id, size_t& remainingTickets) {  
    while (remainingTickets > 0) {  
        sleep_for(500); // simulate "selling a ticket"  
        remainingTickets--;  
        cout << oslock << "Thread #" << id << " sold a ticket ("  
            << remainingTickets << " remain)." << endl << osunlock;  
    }  
    cout << oslock << "Thread #" << id  
    << " sees no remaining tickets to sell and exits." << endl << osunlock;  
}
```

**remainingTickets = 1**

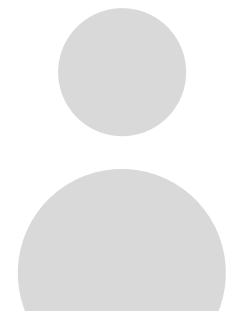
Are there tickets  
to sell? Yep!



Thread #1



Thread #2



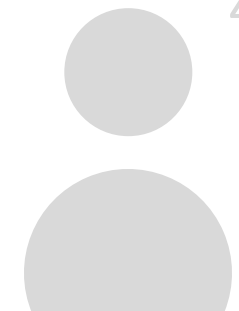
Thread #3

# Race Condition: Overselling Tickets

```
static void sellTickets(size_t id, size_t& remainingTickets) {  
    while (remainingTickets > 0) {  
        sleep_for(500); // simulate "selling a ticket"  
        remainingTickets--;  
        cout << oslock << "Thread #" << id << " sold a ticket ("  
            << remainingTickets << " remain)." << endl << osunlock;  
    }  
    cout << oslock << "Thread #" << id  
        << " sees no remaining tickets to sell and exits." << endl << osunlock;  
}
```

**remainingTickets = 1**

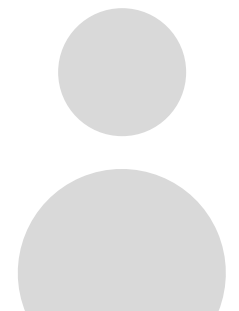
Are there tickets  
to sell? Yep!



Thread #1



Thread #2



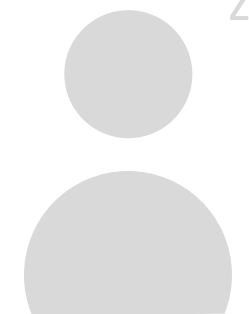
Thread #3

# Race Condition: Overselling Tickets

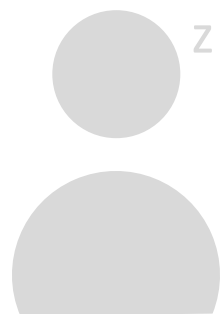
```
static void sellTickets(size_t id, size_t& remainingTickets) {  
    while (remainingTickets > 0) {  
        sleep_for(500); // simulate "selling a ticket"  
        remainingTickets--;  
        cout << oslock << "Thread #" << id << " sold a ticket ("  
            << remainingTickets << " remain)." << endl << osunlock;  
    }  
    cout << oslock << "Thread #" << id  
    << " sees no remaining tickets to sell and exits." << endl << osunlock;  
}
```

**remainingTickets = 1**

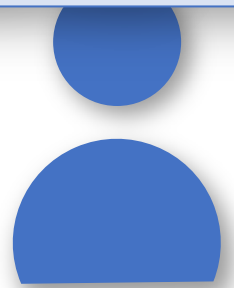
Are there tickets to sell? Yep!



Thread #1



Thread #2



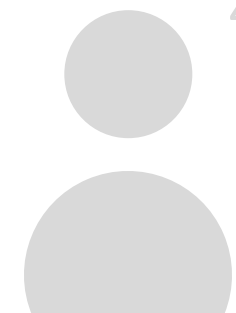
Thread #3



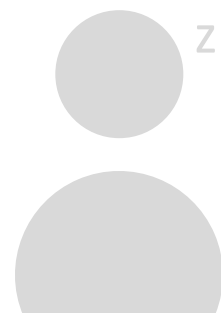
# Race Condition: Overselling Tickets

```
static void sellTickets(size_t id, size_t& remainingTickets) {  
    while (remainingTickets > 0) {  
        sleep_for(500); // simulate "selling a ticket"  
        remainingTickets--;  
        cout << oslock << "Thread #" << id << " sold a ticket ("  
            << remainingTickets << " remain)." << endl << osunlock;  
    }  
    cout << oslock << "Thread #" << id  
    << " sees no remaining tickets to sell and exits." << endl << osunlock;  
}
```

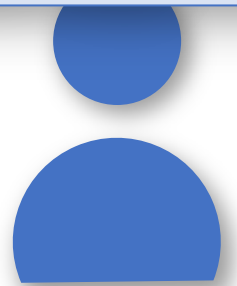
**remainingTickets = 1**



Thread #1



Thread #2



Thread #3

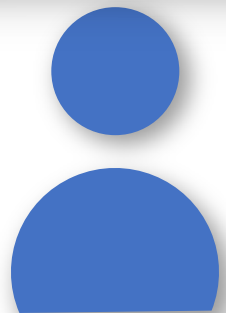
Are there tickets to sell? Yep!

# Race Condition: Overselling Tickets

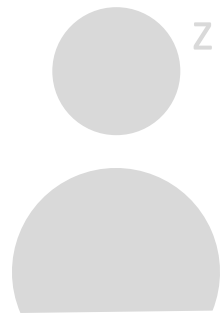
```
static void sellTickets(size_t id, size_t& remainingTickets) {  
    while (remainingTickets > 0) {  
        sleep_for(500); // simulate "selling a ticket"  
        remainingTickets--;  
        cout << oslock << "Thread #" << id << " sold a ticket ("  
            << remainingTickets << " remain)." << endl << osunlock;  
    }  
    cout << oslock << "Thread #" << id  
        << " sees no remaining tickets to sell and exits." << endl << osunlock;  
}
```

Let's sell a ticket!

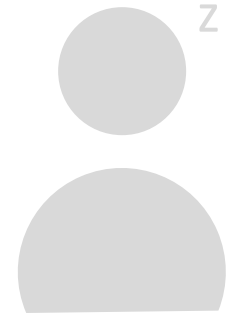
**remainingTickets = 0**



Thread #1



Thread #2



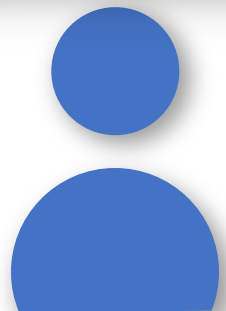
Thread #3

# Race Condition: Overselling Tickets

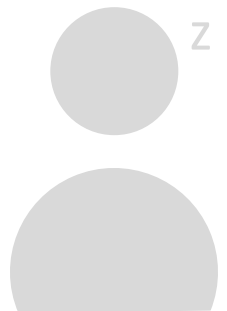
```
static void sellTickets(size_t id, size_t& remainingTickets) {  
    while (remainingTickets > 0) {  
        sleep_for(500); // simulate "selling a ticket"  
        remainingTickets--;  
        cout << oslock << "Thread #" << id << " sold a ticket ("  
            << remainingTickets << " remain)." << endl << osunlock;  
    }  
    cout << oslock << "Thread #" << id  
        << " sees no remaining tickets to sell and exits." << endl << osunlock;  
}
```

Let's sell a ticket!

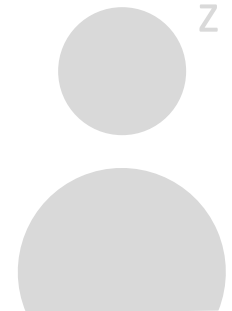
**remainingTickets = 0**



Thread #1



Thread #2



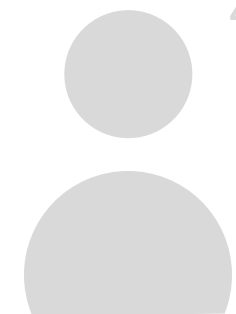
Thread #3

# Race Condition: Overselling Tickets

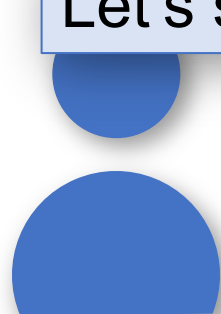
```
static void sellTickets(size_t id, size_t& remainingTickets) {  
    while (remainingTickets > 0) {  
        sleep_for(500); // simulate "selling a ticket"  
        remainingTickets--;  
        cout << oslock << "Thread #" << id << " sold a ticket ("  
            << remainingTickets << " remain)." << endl << osunlock;  
    }  
    cout << oslock << "Thread #" << id  
        << " sees no remaining tickets to sell and exits." << endl << osunlock;  
}
```

**remainingTickets = <really large number>**

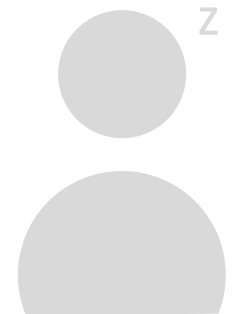
Let's sell a ticket!



Thread #1



Thread #2



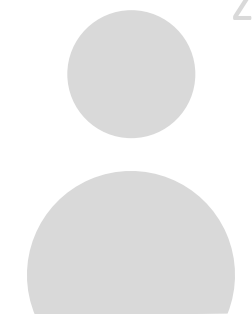
Thread #3

# Race Condition: Overselling Tickets

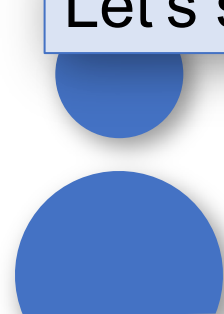
```
static void sellTickets(size_t id, size_t& remainingTickets) {  
    while (remainingTickets > 0) {  
        sleep_for(500); // simulate "selling a ticket"  
        remainingTickets--;  
        cout << oslock << "Thread #" << id << " sold a ticket ("  
            << remainingTickets << " remain)." << endl << osunlock;  
    }  
    cout << oslock << "Thread #" << id  
    << " sees no remaining tickets to sell and exits." << endl << osunlock;  
}
```

**remainingTickets = <really large number>**

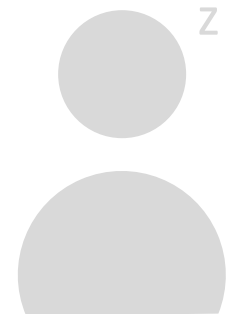
Let's sell a ticket!



Thread #1



Thread #2



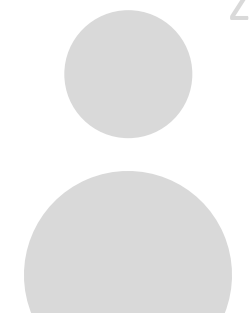
Thread #3

# Race Condition: Overselling Tickets

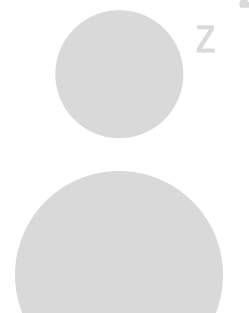
```
static void sellTickets(size_t id, size_t& remainingTickets) {  
    while (remainingTickets > 0) {  
        sleep_for(500); // simulate "selling a ticket"  
        remainingTickets--;  
        cout << oslock << "Thread #" << id << " sold a ticket ("  
            << remainingTickets << " remain)." << endl << osunlock;  
    }  
    cout << oslock << "Thread #" << id  
        << " sees no remaining tickets to sell and exits." << endl << osunlock;  
}
```

**remainingTickets = <really large number - 1>**

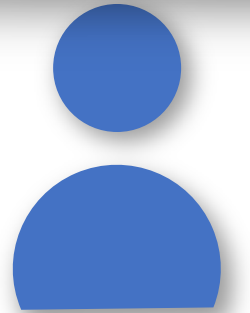
Let's sell a ticket!



Thread #1



Thread #2



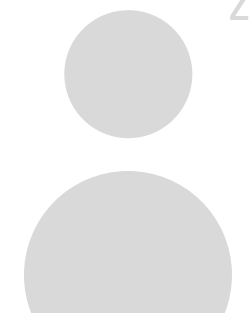
Thread #3

# Race Condition: Overselling Tickets

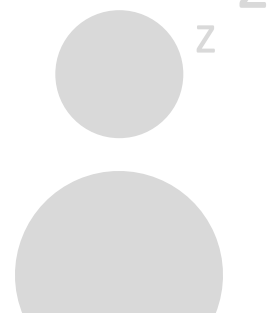
```
static void sellTickets(size_t id, size_t& remainingTickets) {  
    while (remainingTickets > 0) {  
        sleep_for(500); // simulate "selling a ticket"  
        remainingTickets--;  
        cout << oslock << "Thread #" << id << " sold a ticket ("  
            << remainingTickets << " remain)." << endl << osunlock;  
    }  
    cout << oslock << "Thread #" << id  
    << " sees no remaining tickets to sell and exits." << endl << osunlock;  
}
```

**remainingTickets = <really large number - 1>**

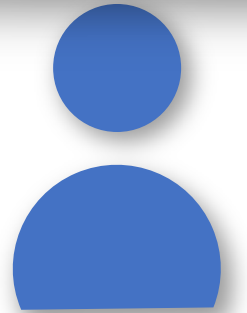
Let's sell a ticket!



Thread #1



Thread #2



Thread #3

# Race Condition: Overselling Tickets

There is a *race condition* here! Threads could interrupt each other in between checking for remaining tickets and selling them.

```
static void sellTickets(size_t id, size_t& remainingTickets) {  
    while (remainingTickets > 0) {  
        sleep_for(500); // simulate "selling a ticket"  
        remainingTickets--;  
        ...  
    }  
    ...  
}
```

- If thread A sees tickets remaining and commits to selling a ticket, another thread B could come in and sell that same ticket before thread A does.
- This can happen because this portion of code isn't *atomic*.



# Race Condition: Overselling Tickets

If thread A sees tickets remaining and commits to selling a ticket, another thread B could come in and sell that same ticket before thread A does.

```
static void sellTickets(size_t id, size_t& remainingTickets) {  
    while (remainingTickets > 0) {  
        sleep_for(500); // simulate "selling a ticket"  
        remainingTickets--;  
        ...  
    }  
    ...  
}
```

- Atomicity: externally, the code has either executed or not; external observers do not see any intermediate states mid-execution.
- We want a thread to do the entire check-and-sell operation **uninterrupted** by other threads.

# Atomicity

- C++ statements aren't inherently atomic.
- Even single C++ statements like **remainingTickets--** take multiple operations and could be interrupted in the middle. (multiple assembly instructions to get value, decrement value, and save updated value).
- Even if we altered the code as below, it still wouldn't fix the problem:

```
static void sellTickets(size_t id, size_t& remainingTickets) {  
    while (remainingTickets-- > 0) {  
        sleep_for(500); // simulate "selling a ticket"  
        ...  
    }  
}
```

**It would be nice if we could allow only one thread at a time to execute a region of code.**

# Plan For Today

- Recap: Threads
- Threads Share Memory
- **Critical Sections**
- Mutexes

```
cp -r /afs/ir/class/cs111/lecture-code/lect13 .
```

# Critical Section

A **critical section** is a section of code that should be executed by only one thread at a time.

```
static void sellTickets(size_t id, size_t& remainingTickets) {
    while (remainingTickets > 0) {
        sleep_for(500); // simulate "selling a ticket"
        remainingTickets--;
        cout << oslock << "Thread #" << id << " sold a ticket ("
             << remainingTickets << " remain)." << endl << osunlock;
    }
    cout << oslock << "Thread #" << id
         << " sees no remaining tickets to sell and exits." << endl << osunlock;
}
```

What should we make a critical section? **Key:** keep them as small as possible to protect performance.

# Critical Section

A **critical section** is a section of code that should be executed by only one thread at a time.

```
static void sellTickets(size_t id, size_t& remainingTickets) {  
    while (remainingTickets > 0) {  
        sleep_for(500); // simulate "selling a ticket"  
        remainingTickets--;  
        cout << oslock << "Thread #" << id << " sold a ticket ("  
            << remainingTickets << " remain)." << endl << osunlock;  
    }  
    cout << oslock << "Thread #" << id  
    << " sees no remaining tickets to sell and exits." << endl << osunlock;  
}
```

What should we make a critical section? **Key:** keep them as small as possible to protect performance.

# Critical Section

A **critical section** is a section of code that should be executed by only one thread at a time.

```
static void sellTickets(size_t id, size_t& remainingTickets) {
    while (true) {
        if (remainingTickets == 0) break;
        sleep_for(500); // simulate "selling a ticket"
        remainingTickets--;
        cout << oslock << "Thread #" << id << " sold a ticket ("
            << remainingTickets << " remain)." << endl << osunlock;
    }
    cout << oslock << "Thread #" << id
        << " sees no remaining tickets to sell and exits." << endl << osunlock;
}
```

What should we make a critical section? **Key:** keep them as small as possible to protect performance.

# Critical Section


A **critical section** is a section of code that should be executed by only one thread at a time.

```
static void sellTickets(size_t id, size_t& remainingTickets) {
    while (true) {
        if (remainingTickets == 0) break;
        size_t myTicket = remainingTickets;
        remainingTickets--;
        sleep_for(500); // simulate "selling a ticket"
        cout << oslock << "Thread #" << id << " sold a ticket ("
            << myTicket - 1 << " remain)." << endl << osunlock;
    }
    cout << oslock << "Thread #" << id
        << " sees no remaining tickets to sell and exits." << endl << osunlock;
}
```



# Critical Section

A **critical section** is a section of code that should be executed by only one thread at a time.

```
static void sellTickets(size_t id, size_t& remainingTickets) {
    while (true) {
         // only 1 thread can proceed at a time
        if (remainingTickets == 0) break;
        size_t myTicket = remainingTickets;
        remainingTickets--;
        // once thread passes here, another can go
        sleep_for(500); // simulate "selling a ticket"
        cout << oslock << "Thread #" << id << " sold a ticket ("
            << myTicket - 1 << " remain)." << endl << osunlock;
    }
    cout << oslock << "Thread #" << id
        << " sees no remaining tickets to sell and exits." << endl << osunlock;
}
```

# Plan For Today

- Recap: Threads
- Threads Share Memory
- Critical Sections
- **Mutexes**

```
cp -r /afs/ir/class/cs111/lecture-code/lect13 .
```

# Mutexes

A **mutex** (“mutual exclusion”) is a variable type that lets us enforce this pattern of only 1 thread having access to something.

- Also known as a *lock* (there are other types of locks as well)
- A way to add a *constraint* to your program: “only one thread may access or execute this at a time”.
- Initially unlocked
- You make a mutex for each distinct thing you need to limit access to.
- Owned by one thread at a time
- You call **lock()** on the mutex to attempt to take the lock
- You call **unlock()** on the mutex when you are done to give the lock back

# 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

# 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

# Mutexes

```
int main(int argc, const char *argv[]) {
    thread ticketAgents[kNumTicketAgents];
    size_t remainingTickets = 250;
    mutex counterLock;

    for (size_t i = 0; i < kNumTicketAgents; i++) {
        ticketAgents[i] = thread(sellTickets, i, ref(remainingTickets),
ref(counterLock));
    }
    ...
}
```

# 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

# Mutexes

**Step 3:** Lock the mutex at the start of the critical section

```
static void sellTickets(size_t id, size_t& remainingTickets, mutex&
counterLock) {
    while (true) {
        counterLock.lock(); // only 1 thread can proceed at a time
        if (remainingTickets == 0) break;
        size_t myTicket = remainingTickets;
        remainingTickets--;
        // once thread passes here, another can go
        sleep_for(500); // simulate "selling a ticket"
        cout << oslock << "Thread #" << id << " sold a ticket ("
            << myTicket - 1 << " remain)." << endl << osunlock;
    }
    ...
}
```



# Mutexes

When a thread calls lock():

- If the lock is unlocked: the thread now owns the lock and continues execution
- If the lock is locked: the thread blocks and waits until the lock is unlocked
- If multiple threads are waiting for a lock: they all wait until it's unlocked, one receives lock (not necessarily one waiting longest)

```
static void sellTickets(size_t id, size_t& remainingTickets, mutex&
counterLock) {
    while (true) {
        counterLock.lock(); // only 1 thread can proceed at a time
        if (remainingTickets == 0) break;
        size_t myTicket = remainingTickets;
        remainingTickets--;
        // once thread passes here, another can go
        sleep_for(500); // simulate "selling a ticket"
        cout << oslock << "Thread #" << id << " sold a ticket ("
            << myTicket - 1 << " remain)." << endl << osunlock;
    }
    ...
}
```

# Mutexes

**Step 4:** Unlock the mutex at the end of the critical section

Calling **unlock** lets another waiting thread (if any) take ownership of the lock

```
static void sellTickets(size_t id, size_t& remainingTickets, mutex&
counterLock) {
    while (true) {
        counterLock.lock(); // only 1 thread can proceed at a time
        if (remainingTickets == 0) {
            counterLock.unlock(); // must unlock in all cases!
            break;
        }
        size_t myTicket = remainingTickets;
        remainingTickets--;
        counterLock.unlock(); // once thread passes here, another can go
        sleep_for(500); // simulate "selling a ticket"
        cout << oslock << "Thread #" << id << " sold a ticket ("
            << myTicket - 1 << " remain)." << endl << osunlock;
```

# Stalled Ticket Agents

```
static void sellTickets(size_t id, size_t& remainingTickets, mutex&
counterLock) {
    while (true) {
        counterLock.lock(); // only 1 thread can proceed at a time
        if (remainingTickets == 0) {
            counterLock.unlock(); // must give up lock before exiting
            break;
        }
        size_t myTicket = remainingTickets;
        remainingTickets--;
        counterLock.unlock(); // once thread passes here, another can go
        sleep_for(500); // simulate "selling a ticket"
        ...
    }
}
```

Make sure to trace each thread's possible paths of execution to ensure they **always** give back shared resources like locks.

# Mutex Uses

Other times you need a mutex:

- When there are multiple threads **writing** to a variable
- When there is a thread **writing** and one or more threads **reading**

Why do you not need a mutex when there are no writers (only readers)?

# Recap

- **Recap:** Threads
- Threads share memory
- Critical Sections
- Mutexes

**Next time:** condition variables

**Lecture 13 takeaway:** A mutex (“lock”) can help us limit critical sections to 1 thread at a time. A thread can lock a mutex to take ownership of it, and unlock it to give it back. Locking a locked mutex will block the thread until the mutex is available. We must watch out for race conditions!