CS111, Lecture 12
Multithreading Introduction

Optional reading:

Operating Systems: Principles and Practice (2nd Edition): Chapter 4 and Chapter 5 up through Section 5.1

PollEV Notes

- Make sure to complete the location check-in prior to recording response – check-ins after a response is in aren't recorded.
- Avoid doing the location check-in in advance (could expire) – instead, complete the check-in right before you submit the response.
- If you encounter any issues during lecture, please raise your hand or come talk to me after lecture. If you notice any issues with your recorded score, please reach out to me (Nick) by the next lecture so that we can look into it further.

Topic 3: Multithreading - How can we have concurrency within a single process? How does the operating system support this?

CS111 Topic 3: Multithreading

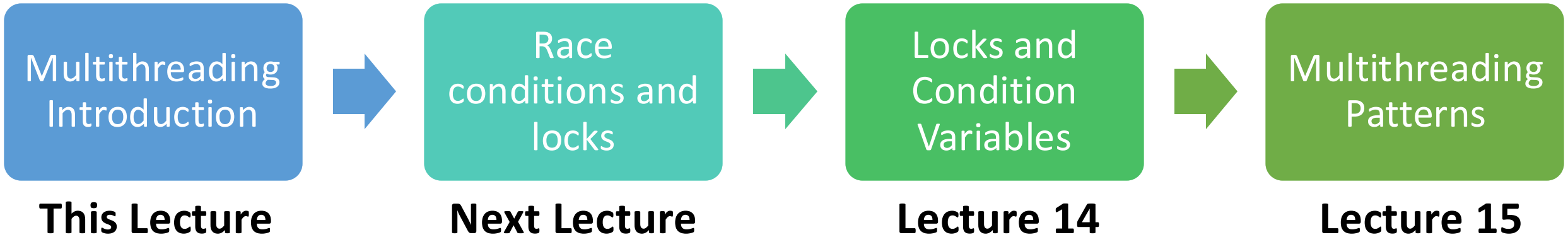
Multithreading - *How can we have concurrency within a single process? How does the operating system support this?*

Why is answering this question important?

- Helps us understand how a single process can do multiple things at the same time, a technique used in various software (today)
- Provides insight into *race conditions*, unpredictable orderings that can cause undesirable behavior, and how to fix them (next few lectures)
- Allows us to see how the OS schedules and switches between tasks (after midterm)

assign4: implement several multithreaded programs while eliminating race conditions

CS111 Topic 3: Multithreading, Part 1



assign4: implement several multithreaded programs while eliminating race conditions!

Learning Goals

- Learn about how threads allow for concurrency within a single process
- Understand the differences between threads and processes
- Discover some of the pitfalls of threads sharing the same virtual address space

Plan For Today

- Introducing multithreading
- **Example:** greeting friends
- Race conditions
- Threads share memory
- **Example:** selling tickets

```
cp -r /afs/ir/class/cs111/lecture-code/lect12 .
```

Plan For Today

- **Introducing multithreading**
- **Example:** greeting friends
- Race conditions
- Threads share memory
- **Example:** selling tickets

```
cp -r /afs/ir/class/cs111/lecture-code/lect12 .
```

From Processes to Threads

Multiprocessing has allowed us to spawn other processes to do tasks or run programs

- Powerful; can execute/wait on other programs, secure (separate memory space), communicate with pipes and signals
- But limited; interprocess communication is cumbersome, hard to share data/coordinate
- Is there another way we can have concurrency beyond multiprocessing that handles these tradeoffs differently?

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
- Multithreading is common to parallelize tasks, especially on multiple cores
- In C++: spawn a thread using **thread()** and the **thread** variable type and specify what function you want the thread to execute (optionally passing parameters!)
- Each thread operates within the same process, so they *share a virtual address space* (!) (globals, heap, pass by reference, etc.)
- The process's stack segment is divided into a "ministack" for each thread.
- In the OS, threads are actually the unit of concurrency, not processes (more on this later). Every process has at least 1 thread.
- Many similarities between threads and processes, but some key differences

Threads vs. Processes

Processes:

- isolate virtual address spaces (good: security and stability, bad: harder to share info)
- can run external programs easily (fork-exec) (good)
- harder to coordinate multiple tasks within the same program (bad)

Threads:

- share virtual address space (bad: security and stability, good: easier to share info)
- can't run external programs easily (bad)
- easier to coordinate multiple tasks within the same program (good)

Threads

Threads are a much more common approach to doing tasks in parallel on a system because of easier data sharing and lighter weight creation. Many modern software programs use multithreading:

- Mobile apps may spawn a background thread to download a web resource while allowing another thread to handle user input (e.g. taps, swipes)
- A web server may spawn threads to handle incoming requests in parallel
- Your computer's task manager can show you how many threads a process is using

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 upon execution
- **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();
}
```

Plan For Today

- Introducing multithreading
- **Example: greeting friends**
- Race conditions
- Threads share memory
- **Example:** selling tickets

```
cp -r /afs/ir/class/cs111/lecture-code/lect12 .
```

Our First Threads Program

```
static void greeting(size_t i) {
    cout << "Hello, world! I am thread " << i << endl;
}

...
```

friends.cc

Our First Threads Program

```
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;
}
```

Threads vs. Helper Functions

What's the difference between spawning threads to run **greeting** vs. just calling **greeting** in a loop without threads?

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

---- *vs.* ----

```
for (size_t i = 0; i < kNumFriends; i++) {
    greeting(i);
}
```

Without threads, the calls are sequential; threads allow parallel execution.

C++ Thread

We can make an array of threads as follows:

```
// declare array of empty thread handles
thread friends[5];

// Spawn threads
for (size_t i = 0; i < 5; i++) {
    friends[i] = thread(myFunc, arg1, arg2);
}
```

Equivalent alternative: (note the loop by reference):

```
thread friends[5];
for (thread& currFriend : friends) {
    currFriend = thread(myFunc, arg1, arg2);
}
```

Plan For Today

- Introducing multithreading
- **Example:** greeting friends
- **Race conditions**
- Threads share memory
- **Example:** selling tickets

```
cp -r /afs/ir/class/cs111/lecture-code/lect12 .
```

Race Conditions

- Like with processes, threads can execute in unpredictable orderings.
- A **race condition** is an ordering of events that causes undesired behavior.
- A *thread-safe* function is one that will always execute correctly, even when called concurrently from multiple threads.
- **operator<<** is *not* thread safe. This means 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;
```

Our First Threads Program

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

...
```

friends.cc

Plan For Today

- Introducing multithreading
- **Example:** greeting friends
- Race conditions
- **Threads share memory**
- **Example:** selling tickets

```
cp -r /afs/ir/class/cs111/lecture-code/lect12 .
```

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));
}
```

friends-ref.cc

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.

Plan For Today

- Introducing multithreading
- **Example:** greeting friends
- Race conditions
- Threads share memory
- **Example: selling tickets**

```
cp -r /afs/ir/class/cs111/lecture-code/lect12 .
```

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.

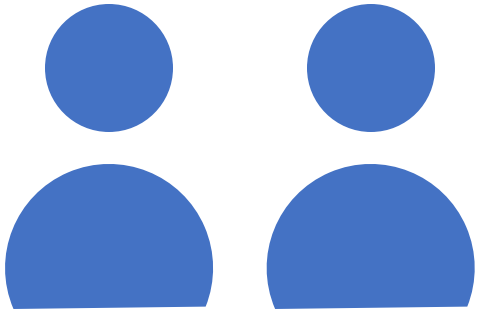
```
const size_t kNumTicketAgents = 10;
int main(int argc, const char *argv[]) {
    thread ticketAgents[kNumTicketAgents];
    size_t remainingTickets = 250;

    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`

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

Key Idea: threads can run in any order.

If we have 2 threads, how could their execution of the code above be interleaved such that they *both sell/decrement* the final ticket and cause an overflow?

Respond on PollEv:
pollev.com/cs111

How could two threads interleave their execution such that they both sell the final ticket?

Nobody has responded yet.

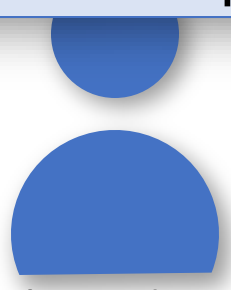
Hang tight! Responses are coming in.

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

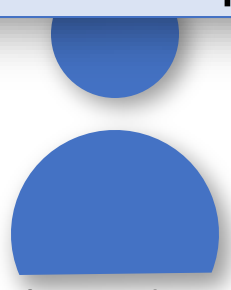
Thread #2

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

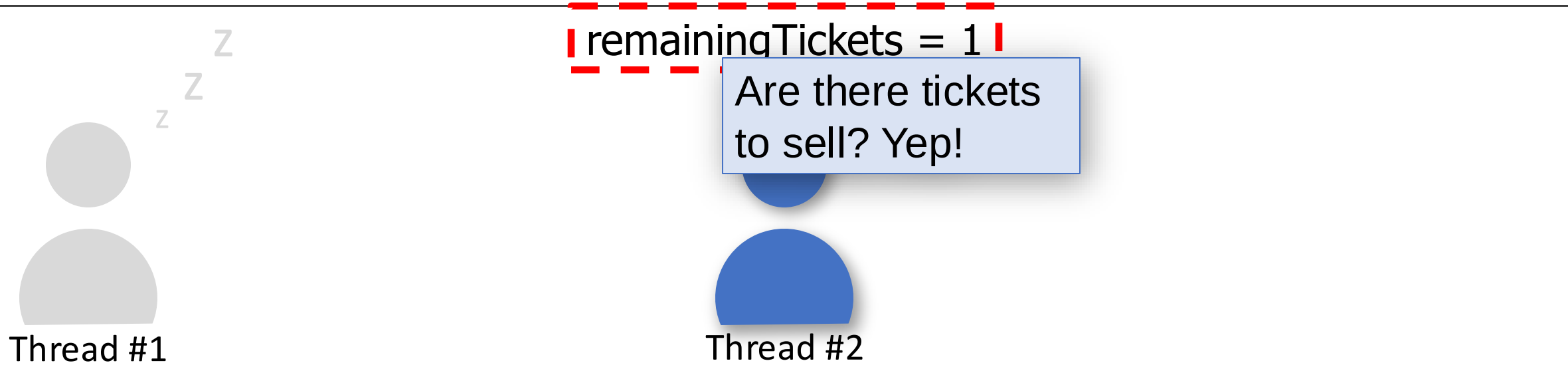
Thread #2

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;
}
```

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;
}
```

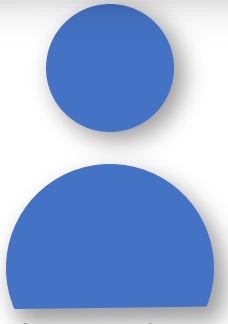


Race Condition: Overselling Tickets

```
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 = 0

Let's sell a ticket!



Thread #1

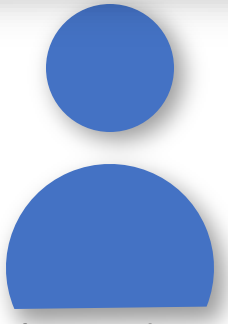
Thread #2

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 = 0

Let's sell a ticket!



Thread #1

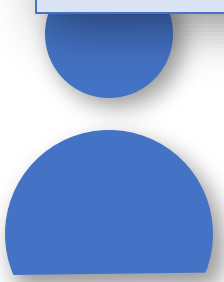
Thread #2

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!



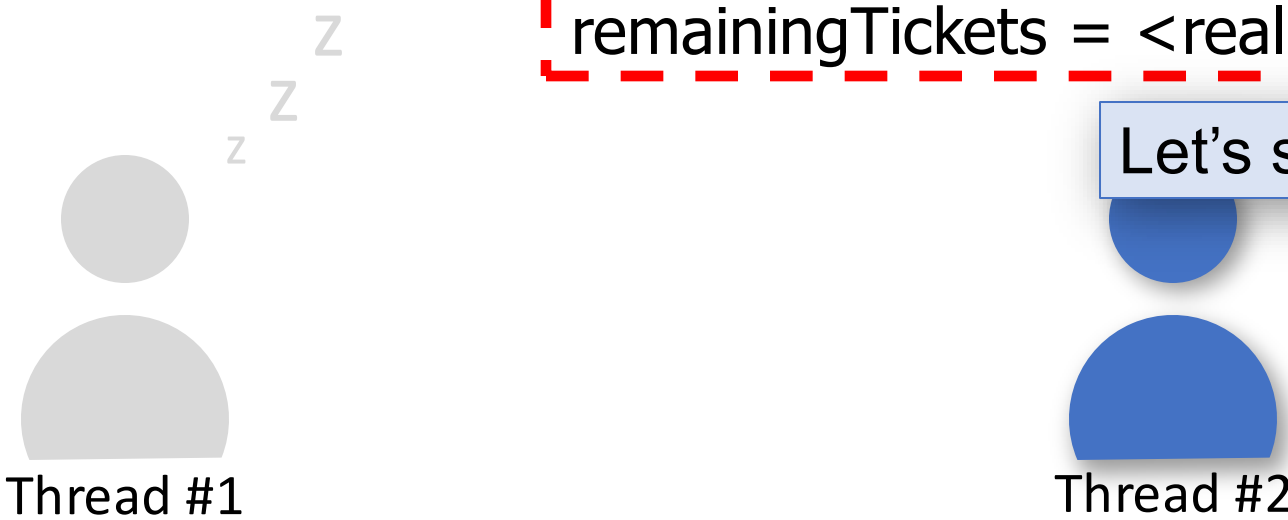
Thread #1

Thread #2

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>



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--;
        ...
    }
    ...
}
```

- **Atomic** means it happens in its entirety without interruption. Cannot be observed in the middle.
- We want a thread to do the entire check-and-sell operation **uninterrupted** by other threads executing this region.

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.

Recap

- Introducing multithreading
- **Example:** greeting friends
- Race conditions
- Threads share memory
- **Example:** selling tickets

Next time: introducing mutexes

Lecture 12 takeaway: A process can have multiple threads executing tasks simultaneously. Threads share the same virtual address space, and race conditions can cause unintended problems!

```
cp -r /afs/ir/class/cs111/lecture-code/lect12 .
```