CS110 Lecture 14: Threads and Mutexes

CS110: Principles of Computer Systems

Winter 2021-2022

Stanford University

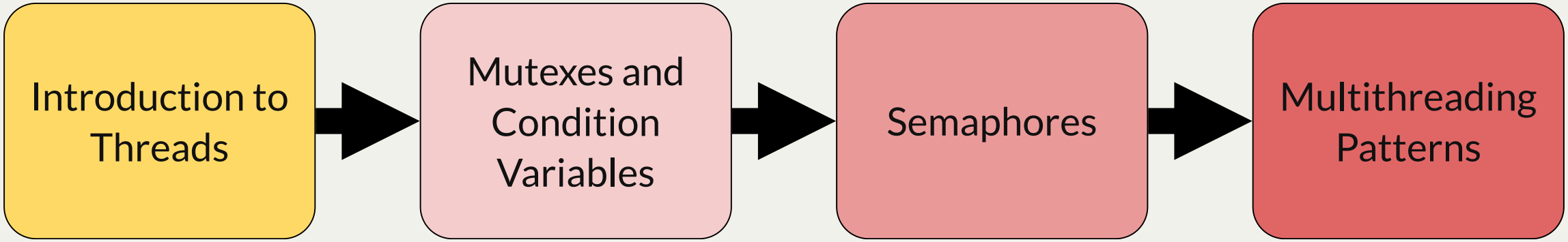
Instructors: Nick Troccoli and Jerry Cain

https://comic.browserling.com/53

PDF of this presentation

CS110 Topic 3: How can we have concurrency within a single process?

Learning About Multithreading



Lecture 13 | **This/next lecture** | **Lecture 16** | **Lectures 17/18**

assign5: implement your own multithreaded news aggregator to quickly fetch news from the web!

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**: C++ Threads and overselling tickets
- Critical Sections
- Mutexes
- Deadlock
- The Race Condition Checklist

Plan For Today

- **Recap: C++ Threads and overselling tickets**
- Critical Sections
- Mutexes
- Deadlock
- The Race Condition Checklist

Multithreading

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 very 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!)
- Thread manager switches between executing threads like the OS scheduler switches between executing processes
- Each thread operates within the same process, so they *share a virtual address space* (!) (globals, text, data, and heap segments)
- The processes's stack segment is divided into a "ministack" for each thread.
- Many similarities between threads and processes; in fact, threads are often called **lightweight processes**.

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
- Once initialized with this constructor, the thread may execute at any time!
- Thread function's return value is ignored (can pass by reference instead)

To pass objects by reference to a thread, use the **ref()** function:

```
void myFunc(int& x, int& y) {...}

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

C++ thread

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

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

... // do some work

// 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 (size_t i = 0; i < 5; i++) {
        friends[i].join();
}
```

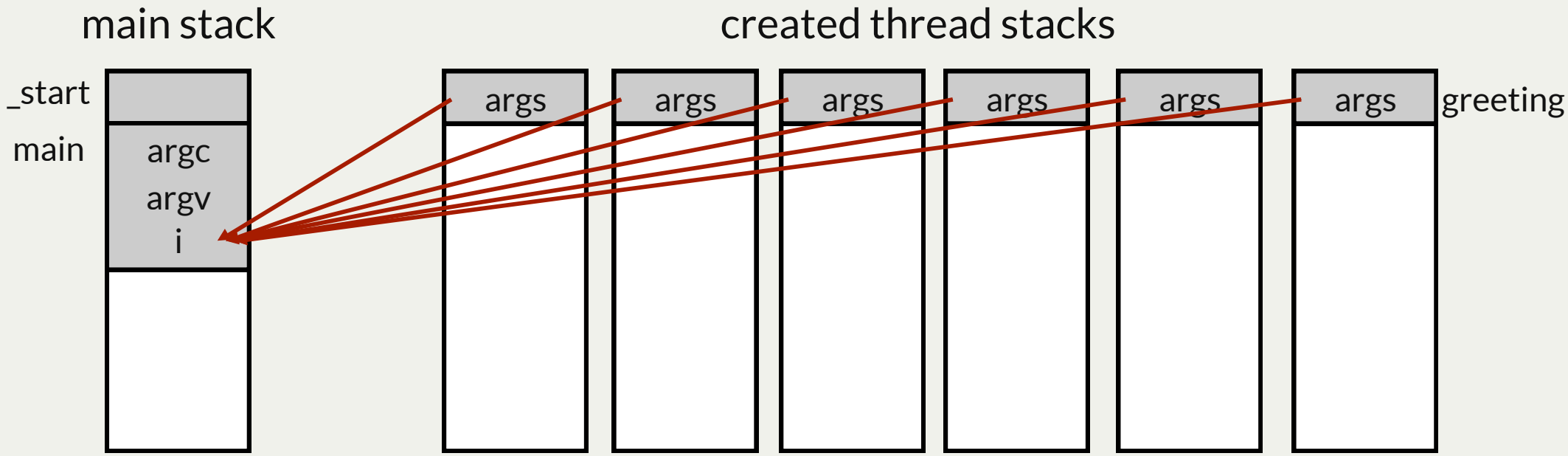
Race Conditions

- Like with processes, threads can execute in unpredictable orderings.
- A *thread-safe* function is one that will always execute correctly, even when called concurrently from multiple threads.
- **printf** is thread-safe, but **operator<<** is *not*. This means e.g. **cout** statements could get interleaved!
- To avoid this, use **oslock** and **osunlock** (custom CS110 functions - **#include "ostreamlock.h"**) around streams. They ensure at most one thread has permission to write into a stream at any one time.

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

Threads Share Memory

```
1 for (size_t i = 0; i < kNumFriends; i++) {
2     friends[i] = thread(greeting, ref(i));
3 }
```



Here, we can just pass by copy instead. But keep an eye out for consequences of shared memory!

Thread-Level Parallelism

- Threads allow a process to parallelize a problem 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

```
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 (thread& ticketAgent: ticketAgents) {
        ticketAgent.join();
    }

    cout << "Ticket selling done!" << endl;
    return 0;
}
```

Output

```
$ ./confused-ticket-agents
....
Thread #1 sold a ticket (7 remain).
Thread #5 sold a ticket (6 remain).
Thread #3 sold a ticket (4 remain).
Thread #4 sold a ticket (4 remain).
Thread #2 sold a ticket (3 remain).
Thread #8 sold a ticket (1 remain).
Thread #9 sold a ticket (0 remain).
Thread #0 sold a ticket (0 remain).
Thread #0 sees no remaining tickets to sell and exits.
Thread #9 sees no remaining tickets to sell and exits.
Thread #8 sees no remaining tickets to sell and exits.
Thread #6 sold a ticket (18446744073709551615 remain).
Thread #7 sold a ticket (18446744073709551613 remain).
Thread #1 sold a ticket (18446744073709551613 remain).
...
```

confused-ticket-agents.cc

Overselling Tickets

There is a *race condition* here!

- **Problem:** threads could interrupt each other in between checking tickets and selling them.

```
1 static void sellTickets(size_t id, size_t& remainingTickets)
2     while (remainingTickets > 0) {
3         sleep_for(500);  // simulate "selling a ticket"
4         remainingTickets--;
5         ...
6     }
```

- If a thread evaluates **remainingTickets > 0** to be **true** and commits to selling a ticket, another thread could come in and sell that same ticket before this thread does.
- This can happen because **remainingImages > 0** test and **remainingImages--** aren't atomic.
- 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 **without competition**.

Atomicity

- C++ statements aren't inherently atomic.
- We assume that assembly instructions are atomic; but even single C++ statements like **remainingTickets--** take multiple assembly instructions.

```
// gets remainingTickets
0x0000000000401a9b <+36>:    mov    -0x20(%rbp),%rax
0x0000000000401a9f <+40>:    mov    (%rax),%eax

// Decrements by 1
0x0000000000401aa1 <+42>:    lea    -0x1(%rax),%edx

// Saves updated value
0x0000000000401aa4 <+45>:    mov    -0x20(%rbp),%rax
0x0000000000401aa8 <+49>:    mov    %edx,(%rax)
```

- Even if we altered the code to be something like this, it still wouldn't fix the problem:

```
1 static void sellTickets(size_t id, size_t& remainingTickets) {
2     while (remainingTickets-- > 0) {
3         sleep_for(500);  // simulate "selling a ticket"
4         ...
5     }
```

Atomicity

```
// gets remainingImages
0x0000000000401a9b <+36>:    mov    -0x20(%rbp),%rax
0x0000000000401a9f <+40>:    mov    (%rax),%eax

// Decrements by 1
0x0000000000401aa1 <+42>:    lea    -0x1(%rax),%edx

// Saves updated value
0x0000000000401aa4 <+45>:    mov    -0x20(%rbp),%rax
0x0000000000401aa8 <+49>:    mov    %edx,(%rax)
```

- Each core has its own registers that it has to read from
- Each thread makes a local copy of the variable before operating on it
- **Problem:** What if multiple threads do this simultaneously? They all think there's only 128 tickets remaining and process #128 at the same time!

It would be nice if we could put the check-and-sell operation behind a ”locked door” and say ”only one thread may enter at a time to do this block of code”.

Plan For Today

- **Recap**: C++ Threads and overselling tickets
- **Critical Sections**
- Mutexes
- Deadlock
- The Race Condition Checklist

Critical Sections

```
1  static void sellTickets(size_t id, size_t& remainingTickets) {
2      while (true) {
3
4          if (remainingTickets == 0) break;
5          remainingTickets--;
6          sleep_for(500);  // simulate "selling a ticket"
7          cout << oslock << "Thread #" << id << " sold a ticket (" << remainingTickets << " remain)." << endl << osunlock;
8
9      }
10     cout << oslock << "Thread #" << id << " sees no remaining tickets to sell and exits." << endl << osunlock;
11 }
```

A **critical section** is a section of code that should be executed transactionally, without competition from other threads.

This means we want critical sections to be **atomic;** to other observers, it has either executed or not.

If we can fix this issue here, then **sellTickets** will be a **thread-safe** function; it will always execute correctly, even when called concurrently from multiple threads.

Critical Sections

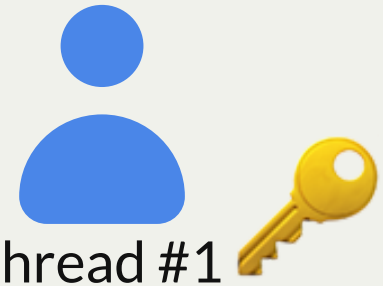
```
1  static void sellTickets(size_t id, size_t& remainingTickets) {
2      while (true) {
3
4          if (remainingTickets == 0) break;
5          remainingTickets--;
6          sleep_for(500);  // simulate "selling a ticket"
7          cout << oslock << "Thread #" << id << " sold a ticket (" << remainingTickets << " remain)." << endl << osunlock;
8
9      }
10     cout << oslock << "Thread #" << id << " sees no remaining tickets to sell and exits." << endl << osunlock;
11 }
```

We will put a lock here with only one key to unlock it. We will make it such that the lock must be unlocked to proceed.

Critical Sections

```
1  static void sellTickets(size_t id, size_t& remainingTickets) {
2      while (true) {
3
4          if (remainingTickets == 0) break;
5          remainingTickets--;
6          sleep_for(500);  // simulate "selling a ticket"
7          cout << oslock << "Thread #" << id << " sold a ticket (" << remainingTickets << " remain)." << endl << osunlock;
8
9      }
10     cout << oslock << "Thread #" << id << " sees no remaining tickets to sell and exits." << endl << osunlock;
11 }
```

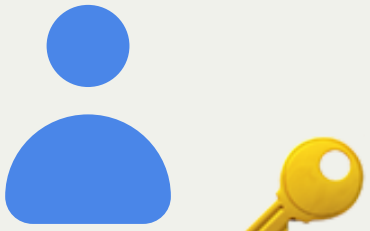
If a thread gets here and the key is available, the thread takes the key, locks the lock, and runs the code while holding onto the key.



Critical Sections

```
1  static void sellTickets(size_t id, size_t& remainingTickets) {
2      while (true) {
3
4          if (remainingTickets == 0) break;
5          remainingTickets--;
6          sleep_for(500);  // simulate "selling a ticket"
7          cout << oslock << "Thread #" << id << " sold a ticket (" << remainingTickets << " remain)." << endl << osunlock;
8
9      }
10     cout << oslock << "Thread #" << id << " sees no remaining tickets to sell and exits." << endl << osunlock;
11 }
```

If another thread gets here and the lock is locked, it must wait its turn.



thread #1

waiting...

thread #2

Critical Sections

```
1  static void sellTickets(size_t id, size_t& remainingTickets) {
2      while (true) {
3
4          if (remainingTickets == 0) break;
5          remainingTickets--;
6          sleep_for(500);  // simulate "selling a ticket"
7          cout << oslock << "Thread #" << id << " sold a ticket (" << remainingTickets << " remain)." << endl << osunlock;
8
9      }
10     cout << oslock << "Thread #" << id << " sees no remaining tickets to sell and exits." << endl << osunlock;
11 }
```

When the executing thread gets here, it unlocks the lock and returns the key. Now another thread may use it.

thread #1

now it's my turn!

thread #2

Critical Sections

```
1  static void sellTickets(size_t id, size_t& remainingTickets) {
2      while (true) {
3
4          if (remainingTickets == 0) break;
5          remainingTickets--;
6          sleep_for(500);  // simulate "selling a ticket"
7          cout << oslock << "Thread #" << id << " sold a ticket (" << remainingTickets << " remain)." << endl << osunlock;
8
9      }
10     cout << oslock << "Thread #" << id << " sees no remaining tickets to sell and exits." << endl << osunlock;
11 }
```

When the executing thread gets here, it unlocks the lock and returns the key. Now another thread may use it.

thread #1

thread #2 

Plan For Today

- **Recap**: C++ Threads and overselling tickets
- Critical Sections
- **Mutexes**
- Deadlock
- The Race Condition Checklist

Mutexes

- We can create this lock-and-key combo by creating a variable of type **mutex**.
- A **mutex** is technically a type of lock; there are others, but we focus just on **mutexes**
- When you create a mutex, it is initially unlocked with the key available
- You call **lock()** on the mutex to attempt to lock it and take the key
- You call **unlock()** on a mutex *if you have ownership of it* and wish to unlock it and return the key. That thread continues normally; one waiting thread (if any) then takes the lock and is scheduled to run.

```
// Assume multiple threads share this same mutex
mutex myLock;

...

myLock.lock();
// only one thread can be executing here at a time
myLock.unlock()
```

Mutexes

When a thread calls **lock()**:

- **If the lock is unlocked:** the thread takes 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)

```
// Assume multiple threads share this same mutex
mutex myLock;

...

myLock.lock();
// only one thread can be executing here at a time
myLock.unlock()
```

A mutex is a way to add a *constraint* to your multithreaded program: ”only one thread may execute this code at a time.” We will learn how to add more ”constraints” in the coming lectures.

Mutex Usage

1. Identify a critical section; a section that only one thread should execute at a time.
2. Create a mutex and ***pass it by reference*** to all threads executing that critical section
3. Add a line to lock the mutex at the start of the critical section
4. Add a line to unlock the mutex at the end of the critical section

If you don't pass by reference, every thread will get its *own* mutex copy (its own lock-and-key); thus every thread will be able to acquire its own lock and run the code!

```
// Assume multiple threads share this same mutex
mutex myLock;

...

myLock.lock();
// only one thread can be executing here at a time
myLock.unlock()
```

Demo: adding a mutex to ticket agents

Plan For Today

- **Recap**: C++ Threads and overselling tickets
- Critical Sections
- Mutexes
- **Deadlock**
- The Race Condition Checklist

Deadlock

Deadlock is a situation where a thread or threads rely on mutually blocked-on resources that will never become available.

- **Example**: a thread must acquire a lock before proceeding. We forget to call unlock somewhere, so one thread keeps the lock forever while others are stuck waiting for the lock forever.

```
1  static void sellTickets(size_t id, size_t& remainingTickets, mutex& counterLock) {
2      while (true) {
3          size_t myTicket;
4
5          counterLock.lock();
6          if (remainingTickets == 0) {
7              break;
8          } else {
9              myTicket = remainingTickets;
10             remainingTickets--;
11             counterLock.unlock();
12         }
13         ...
14     }
15     ...
16 }
```

Deadlock

Deadlock is a situation where a thread or threads rely on mutually blocked-on resources that will never become available.

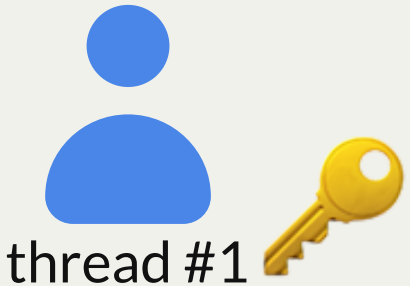
```
1  static void sellTickets(size_t id, size_t& remainingTickets, mutex& counterLock) {
2      while (true) {
3          size_t myTicket;
4
5          counterLock.lock();
6          if (remainingTickets == 0) {
7              break;
8          } else {
9              myTicket = remainingTickets;
10             remainingTickets--;
11             counterLock.unlock();
12         }
13         ...
14     }
15     ...
16 }
```

thread #1

Deadlock

Deadlock is a situation where a thread or threads rely on mutually blocked-on resources that will never become available.

```
1  static void sellTickets(size_t id, size_t& remainingTickets, mutex& counterLock) {
2      while (true) {
3          size_t myTicket;
4
5          counterLock.lock();
6          if (remainingTickets == 0) {
7              break;
8          } else {
9              myTicket = remainingTickets;
10             remainingTickets--;
11             counterLock.unlock();
12         }
13         ...
14     }
15     ...
16 }
```



Deadlock

Deadlock is a situation where a thread or threads rely on mutually blocked-on resources that will never become available.

```
1  static void sellTickets(size_t id, size_t& remainingTickets, mutex& counterLock) {
2      while (true) {
3          size_t myTicket;
4
5          counterLock.lock();
6          if (remainingTickets == 0) {
7              break;
8          } else {
9              myTicket = remainingTickets;
10             remainingTickets--;
11             counterLock.unlock();
12         }
13         ...
14     }
15     ...
16 }
```

Deadlock

Deadlock is a situation where a thread or threads rely on mutually blocked-on resources that will never become available.

```
1  static void sellTickets(size_t id, size_t& remainingTickets, mutex& counterLock) {
2      while (true) {
3          size_t myTicket;
4
5          counterLock.lock();
6          if (remainingTickets == 0) {
7              break;
8          } else {
9              myTicket = remainingTickets;
10             remainingTickets--;
11             counterLock.unlock();
12         }
13         ...
14     }
15     ...
16 }
```

Deadlock

Deadlock is a situation where a thread or threads rely on mutually blocked-on resources that will never become available.

```
1  static void sellTickets(size_t id, size_t& remainingTickets, mutex& counterLock) {
2      while (true) {
3          size_t myTicket;
4
5          counterLock.lock();
6          if (remainingTickets == 0) {
7              break;
8          } else {
9              myTicket = remainingTickets;
10             remainingTickets--;
11             counterLock.unlock();
12         }
13         ...
14     }
15     ...
16 }
```

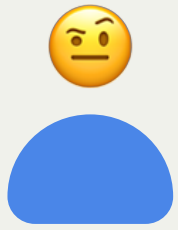
thread #2

Huh. Guess I gotta wait for the key.

Deadlock

Deadlock is a situation where a thread or threads rely on mutually blocked-on resources that will never become available.

```
1  static void sellTickets(size_t id, size_t& remainingTickets, mutex& counterLock) {
2      while (true) {
3          size_t myTicket;
4
5          counterLock.lock();
6          if (remainingTickets == 0) {
7              break;
8          } else {
9              myTicket = remainingTickets;
10             remainingTickets--;
11             counterLock.unlock();
12         }
13         ...
14     }
15     ...
16 }
```



thread #2

100 years later

Deadlock

Deadlock is a situation where a thread or threads rely on mutually blocked-on resources that will never become available.

```
1  static void sellTickets(size_t id, size_t& remainingTickets, mutex& counterLock) {
2      while (true) {
3          size_t myTicket;
4
5          counterLock.lock();
6          if (remainingTickets == 0) {
7              counterLock.unlock();
8              break;
9          } else {
10             myTicket = remainingTickets;
11             remainingTickets--;
12             counterLock.unlock();
13         }
14         ...
15     }
16     ...
17 }
```

We can fix the issue here by making sure to unlock in all scenarios where a thread no longer needs the lock, including when we exit the loop.

Deadlock is a common issue in multiprocessing. Make sure to trace each thread's possible paths of execution to ensure they always return shared resources like locks.

Mutex Usage

1. Identify a **critical section**; a section that only one thread should execute at a time.
2. Create a mutex and ***pass it by reference*** to all threads executing that critical section
3. Add a line to lock the mutex at the start of the critical section
4. Add a line to unlock the mutex at the end of the critical section

Mutex Usage

1. **Identify a critical section; a section that only one thread should execute at a time.**
2. Create a mutex and ***pass it by reference*** to all threads executing that critical section
3. Add a line to lock the mutex at the start of the critical section
4. Add a line to unlock the mutex at the end of the critical section

```
1  static void sellTickets(size_t id, size_t& remainingTickets) {
2      while (true) {
3
4          if (remainingTickets == 0) break;
5          remainingTickets--;
6          sleep_for(500);  // simulate "selling a ticket"
7          cout << oslock << "Thread #" << id << " sold a ticket (" << remainingTickets << " remain)." << endl << osunlock;
8
9      }
10     cout << oslock << "Thread #" << id << " sees no remaining tickets to sell and exits." << endl << osunlock;
11 }
```

Mutex Usage

1. Identify a critical section; a section that only one thread should execute at a time.
2. **Create a mutex and *pass it by reference* to all threads executing that critical section**
3. Add a line to lock the mutex at the start of the critical section
4. Add a line to unlock the mutex at the end of the critical section

```
1  int main(int argc, const char *argv[]) {
2      thread ticketAgents[kNumTicketAgents];
3      size_t remainingTickets = 250;
4      mutex counterLock;
5
6      for (size_t i = 0; i < kNumTicketAgents; i++) {
7          ticketAgents[i] = thread(sellTickets, i, ref(remainingTickets), ref(counterLock));
8      }
9      ...
10 }
```

Mutex Usage

1. Identify a critical section; a section that only one thread should execute at a time.
2. Create a mutex and ***pass it by reference*** to all threads executing that critical section
3. **Add a line to lock the mutex at the start of the critical section**
4. Add a line to unlock the mutex at the end of the critical section

```
1  static void sellTickets(size_t id, size_t& remainingTickets, mutex& counterLock) {
2      while (true) {
3          size_t myTicket;
4
5          counterLock.lock();
6          if (remainingTickets == 0) {
7              counterLock.unlock();
8              break;
9          } else {
10             myTicket = remainingTickets;
11             remainingTickets--;
12             counterLock.unlock();
13         }
14
15         sleep_for(500);  // simulate "selling a ticket"
16         cout << oslock << "Thread #" << id << " sold a ticket (" << myTicket << " remain)." << endl << osunlock;
17     }
18     ...
19 }
```

Mutex Usage

1. Identify a critical section; a section that only one thread should execute at a time.
2. Create a mutex and ***pass it by reference*** to all threads executing that critical section
3. Add a line to lock the mutex at the start of the critical section
4. **Add a line to unlock the mutex at the end of the critical section**

```
1  static void sellTickets(size_t id, size_t& remainingTickets, mutex& counterLock) {
2      while (true) {
3          size_t myTicket;
4
5          counterLock.lock();
6          if (remainingTickets == 0) {
7              counterLock.unlock();
8              break;
9          } else {
10             myTicket = remainingTickets;
11             remainingTickets--;
12             counterLock.unlock();
13         }
14
15         sleep_for(500);  // simulate "selling a ticket"
16         cout << oslock << "Thread #" << id << " sold a ticket (" << myTicket << " remain)." << endl << osunlock;
17     }
18     ...
19 }
```

Critical sections can impact code performance. Keep critical sections as small as possible!

Mutex Usage

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)?

Mutex Analogies

A mutex is a variable type that is like a "locked door".

https://www.flickr.com/photos/ofsmallthings/8220574255

You can **lock** the door:

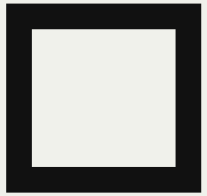
- if it's unlocked, you go through the door and lock it
- if it's locked, you *wait for it to unlock first*

If you most recently locked the door, you can **unlock** the door:

- door is now unlocked, another may go in now

Mutex Analogies

A mutex is a variable type that is like a "ball in a bucket".



To proceed, you must take the ball from the bucket and hold onto it.

If you find the bucket is empty, you must wait for the ball to be returned.

When you are done executing, you return the ball to the bucket.

Plan For Today

- **Recap**: C++ Threads and overselling tickets
- Critical Sections
- Mutexes
- Deadlock
- **The Race Condition Checklist**

The Race Condition Checklist

☐ **Identify shared data that may be modified concurrently.** What shared data is used across threads, passed by reference or globally?

☐ **Document and confirm an ordering of events that causes unexpected behavior**. What assumptions are made in the code that can be broken by certain orderings?

☐ **Use concurrency directives to force expected orderings and add constraints**. How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s)?

The Race Condition Checklist

☐ **Identify shared data that may be modified concurrently.** What shared data is used across threads, passed by reference or globally? ***the ticket count.***

☐ **Document and confirm an ordering of events that causes unexpected behavior.** What assumptions are made in the code that can be broken by certain orderings?

☐ **Use concurrency directives to force expected orderings and add constraints.** How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s)?

The Race Condition Checklist

☑ **Identify shared data that may be modified concurrently.** What shared data is used across threads, passed by reference or globally? ***the ticket count.***

☐ **Document and confirm an ordering of events that causes unexpected behavior.** What assumptions are made in the code that can be broken by certain orderings? ***one thread will check-and-sell a ticket at a time.***

☐ **Use concurrency directives to force expected orderings and add constraints.** How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s)?

The Race Condition Checklist

☑ **Identify shared data that may be modified concurrently.** What shared data is used across threads, passed by reference or globally? ***the ticket count.***

☑ **Document and confirm an ordering of events that causes unexpected behavior.** What assumptions are made in the code that can be broken by certain orderings? ***one thread will check-and-sell a ticket at a time.***

☐ **Use concurrency directives to force expected orderings and add constraints.** How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s)? ***add a mutex that must be acquired before checking-and-selling a ticket.***

The Race Condition Checklist

☑ **Identify shared data that may be modified concurrently.** What shared data is used across threads, passed by reference or globally? ***the ticket count.***

☑ **Document and confirm an ordering of events that causes unexpected behavior**. What assumptions are made in the code that can be broken by certain orderings? ***one thread will check-and-sell a ticket at a time.***

☑ **Use concurrency directives to force expected orderings and add constraints**. How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s)? ***add a mutex that must be acquired before checking-and-selling a ticket.***

Recap

- **Recap**: C++ Threads and overselling tickets
- Critical Sections
- Mutexes
- Deadlock
- The Race Condition Checklist

Next time: adding more constraints with **condition variables.**