CS110 Lecture 13: Introduction to Multithreading

CS110: Principles of Computer Systems

Winter 2021-2022

Stanford University

Instructors: Nick Troccoli and Jerry Cain

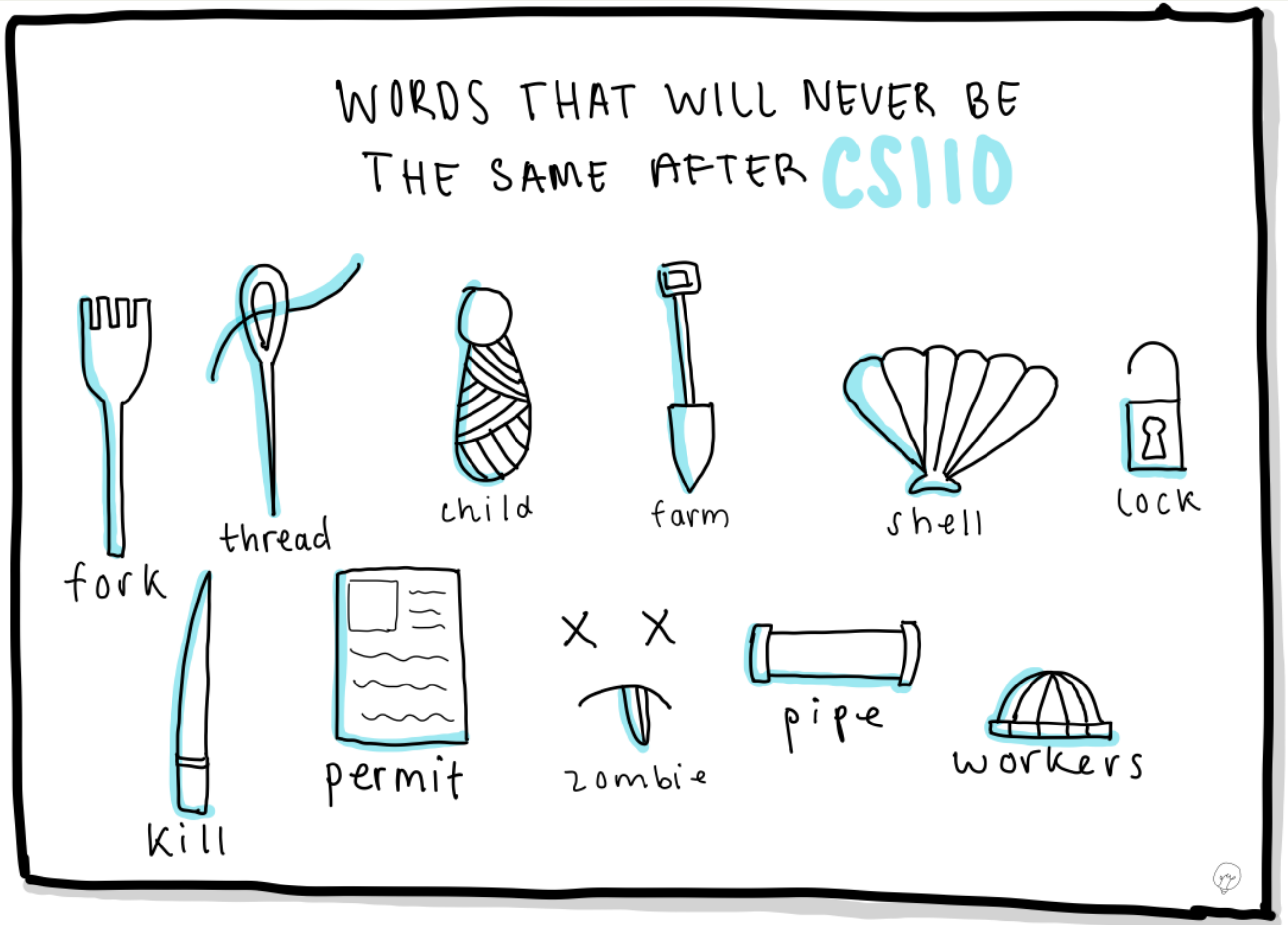
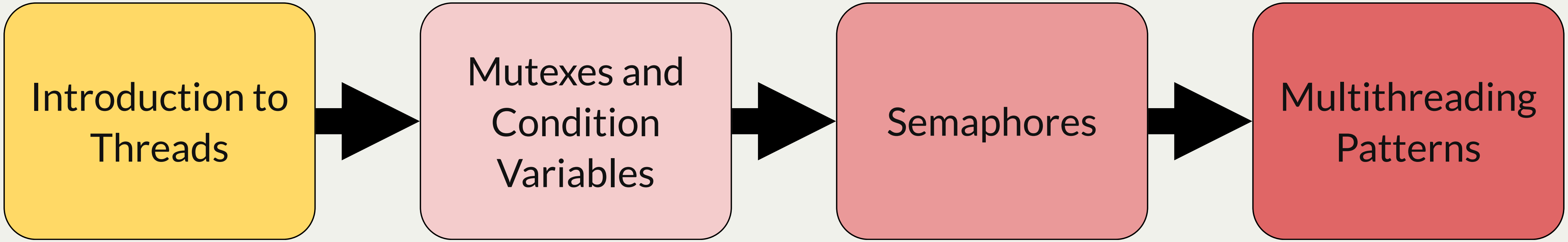


Illustration courtesy of Ecy King, CS110 Champion, Spring 2021

PDF of this presentation

<u>**CS110 Topic 3:**</u> How can we have concurrency within a single process?

Learning About Multithreading



Today

Lectures 14/15

Lecture 16

Lectures 17/18

Today's 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
- Completing tasks in parallel
- **Example:** selling tickets

Plan For Today

- **Introducing multithreading**
- **Example**: greeting friends
- Race conditions
- Threads share memory
- Completing tasks in parallel
- **Example:** selling tickets

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?

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**.

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)

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)

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();
}
```

Plan For Today

- Introducing multithreading
- **Example: greeting friends**
- Race conditions
- Threads share memory
- Completing tasks in parallel
- **Example:** selling tickets

Our First Threads Program

```
static const size_t kNumFriends = 6;

static void greeting() {
    cout << "Hello, world!" << endl;
}

int main(int argc, char *argv[]) {
  cout << "Let's hear from " << kNumFriends << " threads." << endl;

   // declare array of empty thread handles
  thread friends[kNumFriends];

  // Spawn threads
  for (size_t i = 0; i < kNumFriends; i++) {
      friends[i] = thread(greeting);
  }

  // Wait for threads
  for (size_t i = 0; i < kNumFriends; i++) {
     friends[i].join();
  }

  cout << "Everyone's said hello!" << endl;
  return 0;
}
```

Our First Threads Program

https://cplayground.com/?p=whale-okapi-phil

Our First Threads Program

```
static const size_t kNumFriends = 6;

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

int main(int argc, char *argv[]) {
  cout << "Let's hear from " << kNumFriends << " threads." << endl;

   // declare array of empty thread handles
  thread friends[kNumFriends];

  // Spawn threads
  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;
}
```

Our First Threads Program

https://cplayground.com/?p=dunlin-coyote-pika

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);
}
```

We can also initialize an array of threads as follows (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
- Completing tasks in parallel
- **Example:** selling tickets

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.
- 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.

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

Our First Threads Program

```
static const size_t kNumFriends = 6;

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

int main(int argc, char *argv[]) {
  cout << "Let's hear from " << kNumFriends << " threads." << endl;

   // declare array of empty thread handles
  thread friends[kNumFriends];

  // Spawn threads
  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;
}
```

friends.cc

Plan For Today

- Introducing multithreading
- **Example**: greeting friends
- Race conditions
- **Threads share memory**
- Completing tasks in parallel
- **Example:** selling tickets

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:

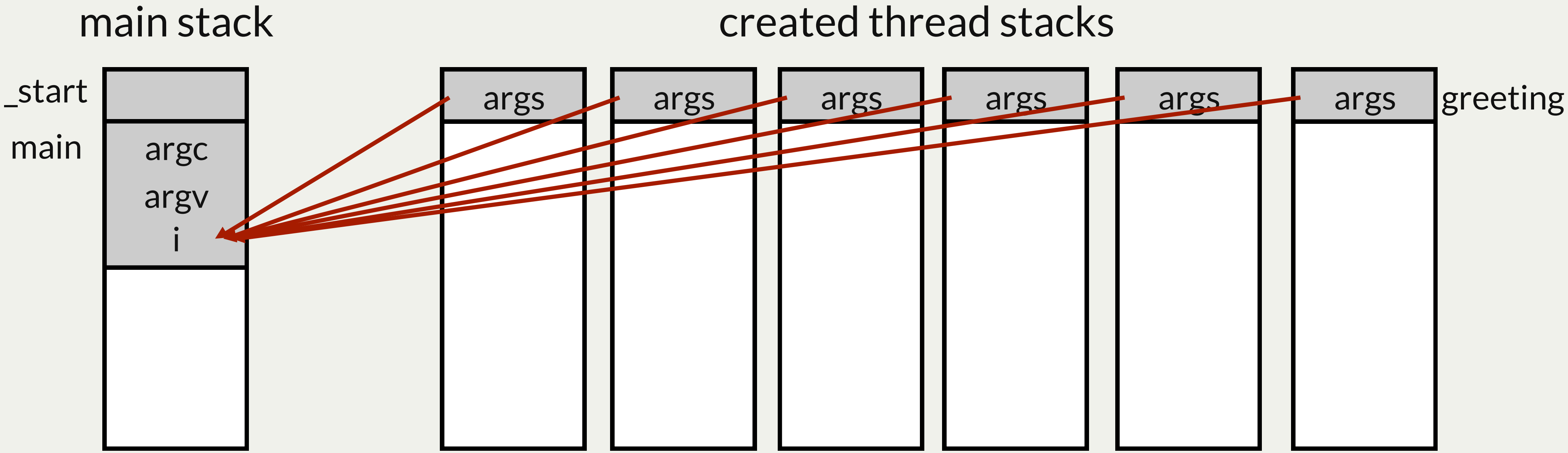
```
static void greeting(size_t& i) {
        ...
}

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

Threads Share Memory

https://cplayground.com/?p=crocodile-emu-cod

Threads Share Memory

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



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

Plan For Today

- Introducing multithreading
- **Example**: greeting friends
- Race conditions
- Threads share memory
- **Completing tasks in parallel**
- **Example:** selling tickets

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;
}
```

confused-ticket-agents.cc

Demo: confused-ticket-agents.cc

Overselling Tickets

- There is a **race condition** in this code caused by multiple threads accessing **remainingTickets**.

```
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

Overselling Tickets

- There is a **race condition** in this code caused by multiple threads accessing **remainingTickets**.

```
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          cout << oslock << "Thread #" << id << " sold a ticket (" << remainingTickets
6               << " remain)." << endl << osunlock;
7      }
8      cout << oslock << "Thread #" << id << " sees no remaining tickets to sell and exits."
9           << endl << osunlock;
10 }
```

remainingTickets = 1

Line 2: checking if there are tickets left. Yep!

thread #1

thread #2

thread #3

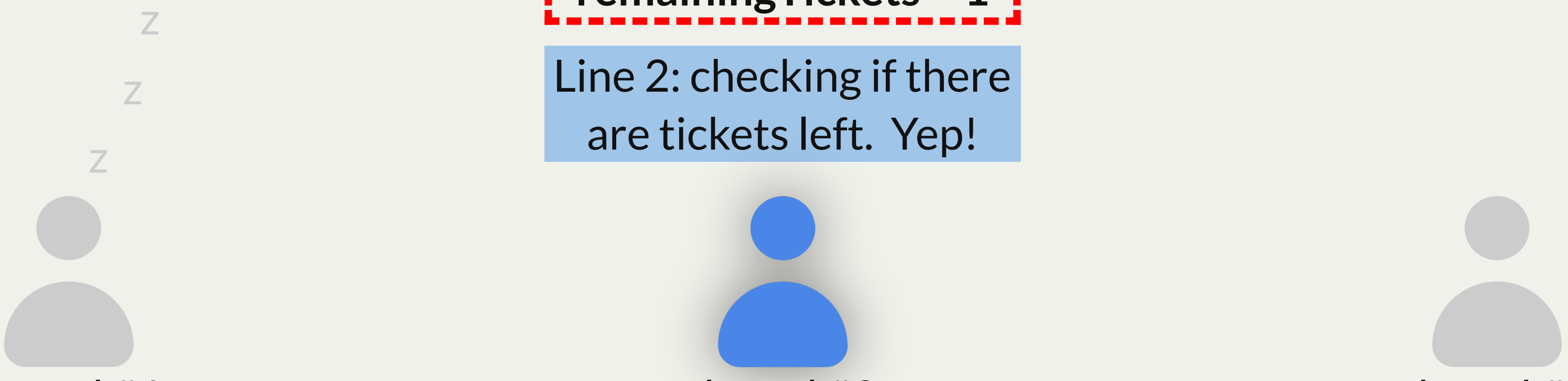
Overselling Tickets

- There is a **race condition** in this code caused by multiple threads accessing **remainingTickets**.

```
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          cout << oslock << "Thread #" << id << " sold a ticket (" << remainingTickets
6              << " remain)." << endl << osunlock;
7      }
8      cout << oslock << "Thread #" << id << " sees no remaining tickets to sell and exits."
9       << endl << osunlock;
10 }
```

remainingTickets = 1

Line 2: checking if there are tickets left. Yep!



thread #1 thread #2 thread #3

Overselling Tickets

- There is a **race condition** in this code caused by multiple threads accessing **remainingTickets**.

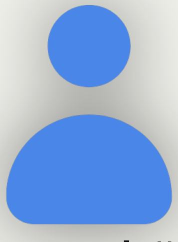
```
1  static void sellTickets(size_t id, size_t& remainingTickets) {
2      while (remainingTickets > 0) {
3          sleep_for(500);  // simulate "selling a ticket"
4          remainingTickets--;
5          cout << oslock << "Thread #" << id << " sold a ticket (" << remainingTickets
6             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

remainingTickets = 1

Line 2: checking if there are tickets left. Yep!

thread #1

thread #2



thread #3

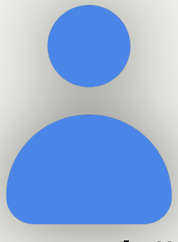
Overselling Tickets

- There is a **race condition** in this code caused by multiple threads accessing **remainingTickets**.

```
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          cout << oslock << "Thread #" << id << " sold a ticket (" << remainingTickets
6               << " remain)." << endl << osunlock;
7      }
8      cout << oslock << "Thread #" << id << " sees no remaining tickets to sell and exits."
9           << endl << osunlock;
10 }
```

remainingTickets = 0

Line 4: Selling ticket!



thread #1

thread #2

thread #3

Overselling Tickets

- There is a **race condition** in this code caused by multiple threads accessing **remainingTickets**.

```
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          cout << oslock << "Thread #" << id << " sold a ticket (" << remainingTickets
6               << " remain)." << endl << osunlock;
7      }
8      cout << oslock << "Thread #" << id << " sees no remaining tickets to sell and exits."
9           << endl << osunlock;
10 }
```

remainingTickets = <really large number>

Line 4: Selling ticket!

z
z
z

thread #1

thread #2

thread #3

Overselling Tickets

- There is a **race condition** in this code caused by multiple threads accessing **remainingTickets**.

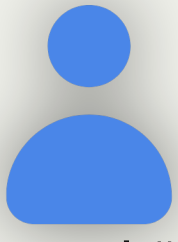
```
1  static void sellTickets(size_t id, size_t& remainingTickets) {
2      while (remainingTickets > 0) {
3          sleep_for(500);  // simulate "selling a ticket"
4          remainingTickets--;
5          cout << oslock << "Thread #" << id << " sold a ticket (" << remainingTickets
6             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

remainingTickets = <really large number - 1>

Line 4: Selling ticket!

thread #1

thread #2


thread #3

Overselling Tickets

There is a *race condition* here!

- **Problem:** threads could interrupt each other in between checking 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 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 <u>atomic.</u>
- 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 **<u>uninterrupted</u>**.

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:

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

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”.

Recap

- Introducing multithreading
- **Example**: greeting friends
- Race conditions
- Threads share memory
- Completing tasks in parallel
- **Example:** selling tickets

Next time: introducing **mutexes**