CS111, Lecture 13
Race Conditions and Locks

Optional reading:

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

😷 masks recommended

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

CS111 Topic 3: Multithreading

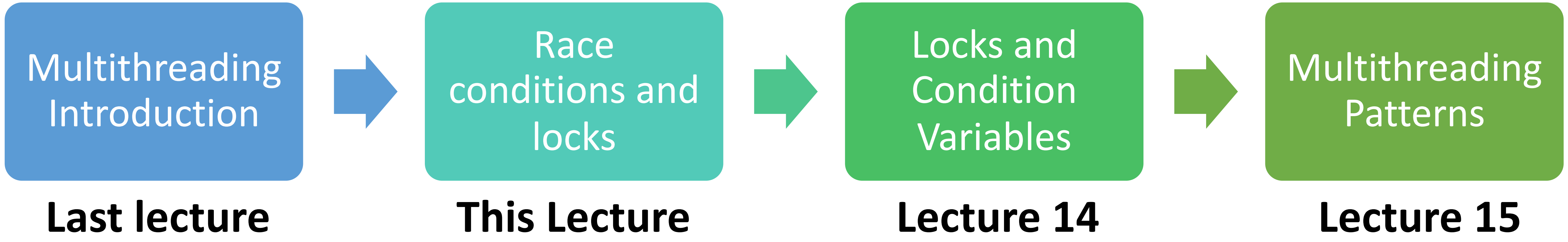
Multithreading - *How can we have concurrency within a single process? How does the operating system support this?*

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 (today)
- 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


Multithreading Introduction
Last lecture
Race conditions and locks
This Lecture
Locks and Condition Variables
Lecture 14
Multithreading Patterns
Lecture 15

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 and overselling tickets
- Critical Sections
- Mutexes
- Deadlock
- Dining Philosophers

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

Plan For Today

- **Recap: threads and overselling tickets**
- Critical Sections
- Mutexes
- Deadlock
- Dining Philosophers

```
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!

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);
...
// 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.

Race Condition: Overselling Tickets

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;
}
```

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.
- We want a thread to do the entire check-and-sell operation **uninterrupted** by other threads. We want this portion of code to be **atomic**.

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 and overselling tickets
- **Critical Sections**
- Mutexes
- Deadlock
- Dining Philosophers

```
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 << " 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 << " remain)." << endl << osunlock;
    }
    cout << oslock << "Thread #" << id
    << " sees no remaining tickets to sell and exits." << endl << osunlock;
}
```

Plan For Today

- **Recap:** threads and overselling tickets
- Critical Sections
- **Mutexes**
- Deadlock
- Dining Philosophers

```
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 << " 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 << " 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) 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 << " remain)." << endl << osunlock;
    }
    ...
```

Demo: `stalled-ticket-agents.cc`

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) 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 << " remain)." << endl << osunlock;
    }
    ...
```

What might have caused some ticket agents to stall?

Respond with your thoughts on PollEv:
pollev.com/cs111 or text CS111 to 22333 once to join.

What might have caused some ticket agents to stall?

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)?

Plan For Today

- **Recap:** threads and overselling tickets
- Critical Sections
- Mutexes
- **Deadlock**
- Dining Philosophers

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

Deadlock

Deadlock occurs when multiple threads are all blocked, waiting on a resource owned by one of the other threads. None can make progress! Example:

```
Thread A              Thread B
mutex1.lock();        mutex2.lock();
mutex2.lock();        mutex1.lock();
...                   ...
```

E.g. if thread A executes 1 line, then thread B executes 1 line, deadlock!

One prevention technique - prevent circularities: all threads request resources in the same order (e.g., always lock l1 before l2.)

Another – limit number of threads competing for a shared resource

Plan For Today

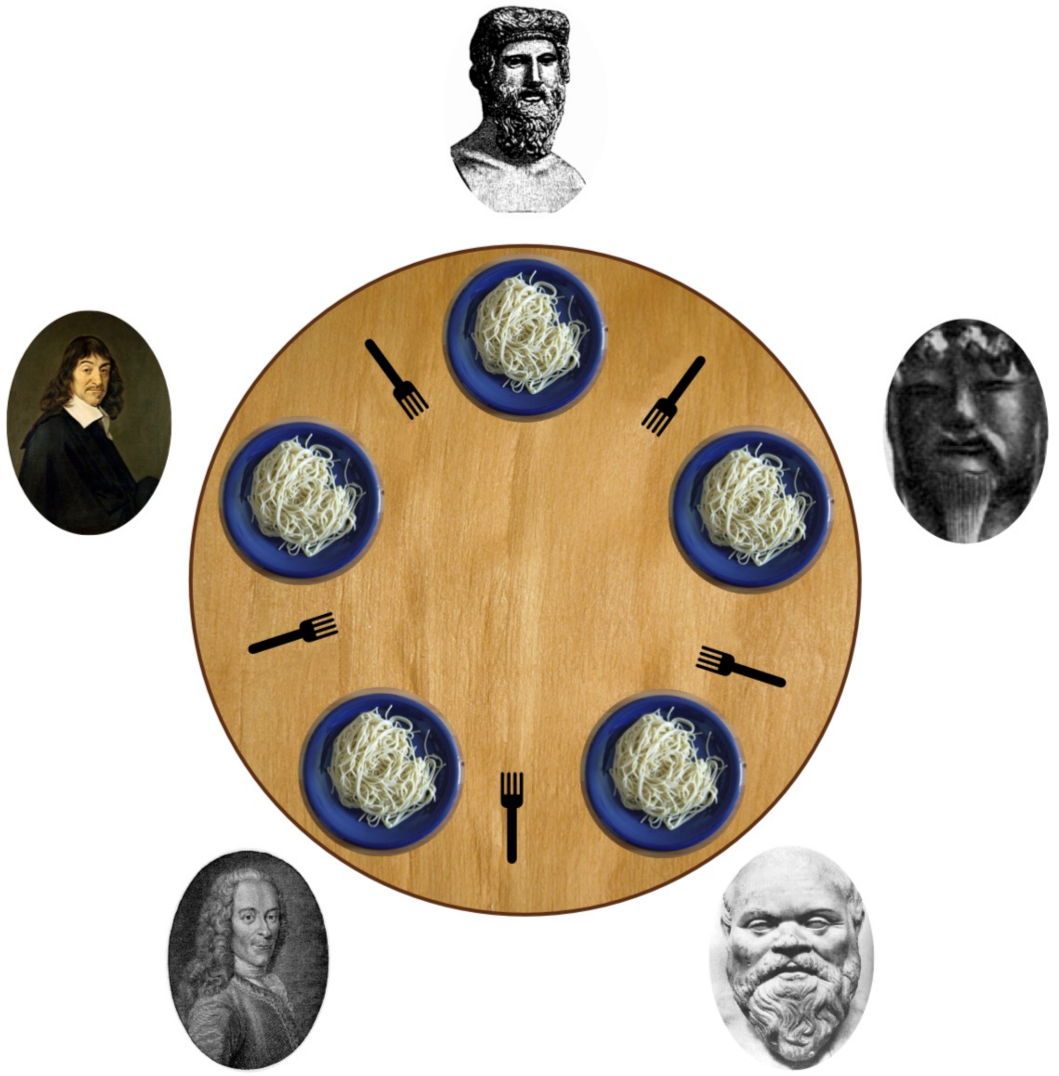
- **Recap:** threads and overselling tickets
- Critical Sections
- Mutexes
- Deadlock
- **Dining Philosophers**

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

Deadlock Example: Dining Philosophers Simulation

- Five philosophers sit around a **circular table**, eating spaghetti
- There is **one fork** for each of them
- Each philosopher **thinks, then eats**, and repeats this **three times** for their three daily meals.
- **To eat**, a philosopher must grab the fork on their left *and* the fork on their right. Then they chow on spaghetti to nourish their big, philosophizing brain.
- When they're full, they put down the forks in the same order they picked them up and return to thinking for a while.
- **To think**, a philosopher keeps to themselves for some amount of time. Sometimes they think for a long time, and sometimes they barely think at all.

Dining Philosophers



https://commons.wikimedia.org/wiki/File:An_illustration_of_the_dining_philosophers_problem.png

Dining Philosophers

Goal: we must encode resource constraints into our program.

Example: for a given fork, how many philosophers can use it at a time? **One**.

How can we encode this into our program? Make a mutex for each fork.

Dining Philosophers

```
static void philosopher(size_t id, mutex& left, mutex&
right) { ... }

int main(int argc, const char *argv[]) {
    mutex forks[kNumForks];
    thread philosophers[kNumPhilosophers];
    for (size_t i = 0; i < kNumPhilosophers; i++) {
        philosophers[i] = thread(philosopher, i,
                    ref(forks[i]),
                    ref(forks[(i + 1) % kNumPhilosophers]));
    }
    for (thread& p: philosophers) p.join();
    return 0;
}
```

Dining Philosophers

A philosopher thinks and eats, and repeats this 3 times.

```
static void philosopher(size_t id, mutex& left, mutex&
right) {
    for (size_t i = 0; i < kNumMeals; i++) {
        think(id);
        eat(id, left, right);
    }
}
```

Dining Philosophers

think is modeled as sleeping the thread for some amount of time.

```
static void think(size_t id) {
    cout << oslock << id << " starts thinking."
         << endl << osunlock;
    sleep_for(getThinkTime());
    cout << oslock << id << " all done thinking. "
         << endl << osunlock;
}
```

Dining Philosophers

eat is modeled as grabbing the two forks, sleeping for some amount of time, and putting the forks down.

```
static void eat(size_t id, mutex& left, mutex& right) {
    left.lock();
    right.lock();
    cout << oslock << id << " starts eating om nom nom 
nom." << endl << osunlock;
    sleep_for(getEatTime());
    cout << oslock << id << " all done eating." << endl
         << osunlock;
    left.unlock();
    right.unlock();
}
```

Spoiler: there is a race condition here that leads to deadlock. What is it?

Food For Thought

What if: all philosophers grab their left fork and then go off the CPU?

- Deadlock! All philosophers will wait on their right fork, which will never become available
- **Testing our hypothesis**: insert a **sleep_for** call in between grabbing the two forks
- We should be able to insert a **sleep_for** call anywhere in a thread routine and have no concurrency issues. Let's try it!

dining-philosophers-with-deadlock.cc

Food For Thought

We are (incorrectly) assuming that at least one philosopher is always able to pick up both of their forks.

We can prevent **deadlock** here by limiting the number of threads competing for a shared resource.

Idea: have a counter of “permits”. Initially 4. A philosopher must have a permit to eat. Once done eating, a philosopher returns its permit. (More next time…)

Recap

- **Recap:** threads and overselling tickets
- Critical Sections
- Mutexes
- Deadlock
- Dining Philosophers

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 and deadlock!

Next time: condition variables

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