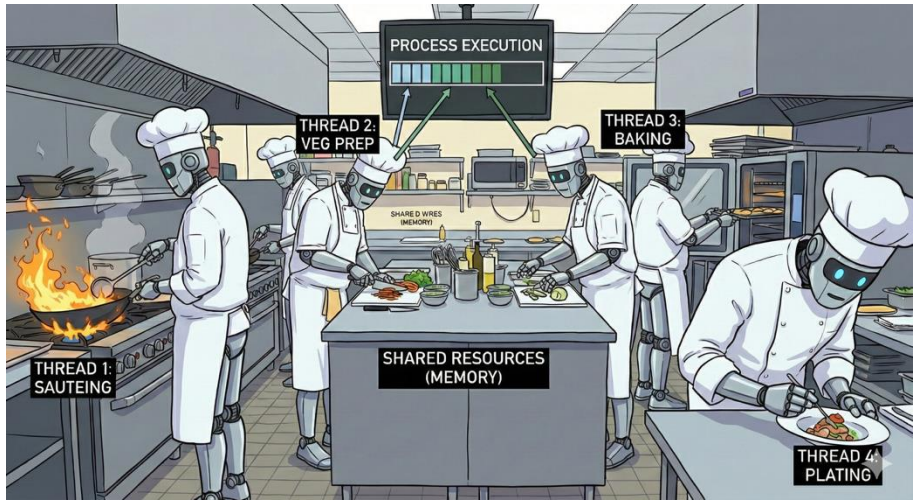
Multithreading

Problem Solving Lab for CS111

Stanford CS111ACE, Spring 2026



Attendance: tinyurl.com/cs111ace-section2-spr26

Today’s Agenda

- A quick recap of processes and multithreading
- Race conditions
- Deadlocks
- Condition Variables

Recap

- A **process** is an instance of your program that is running
  - Processes have isolated memory spaces, which make them safe.
  - Are a powerful OS principle that allow us to run multiple programs at the same time
  - We've explored piping and inter-process communication (IPC) schemes which allow multiple processes to share data

Processes

- Isolated virtual addresses
- Can run external programs (fork-exec)
- Hard to coordinate multiple tasks within the same program

Threads

- Share virtual address space (threads can refer to the same memory)
- Can’t run external programs as easily
- Can coordinate multiple tasks within one program

What is Multithreading?

How do we run code?

Source Code

```
#include <iostream>

void task(const std::string& name) {
    for (int i = 0; i < 3; ++i)
        std::cout << name << " says " << i << std::endl;
}

int main() {
    task("A");
    task("B");
    return 0;
}
```

Runs on

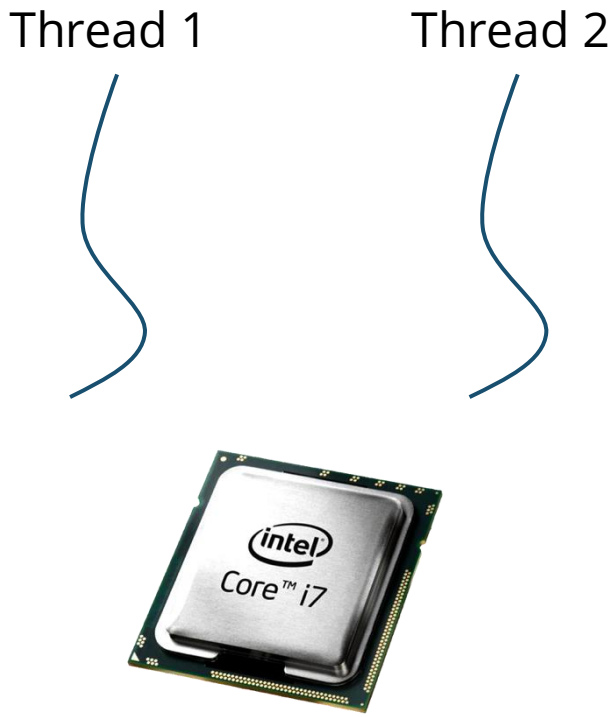
A multithreaded approach

```cpp
#include <iostream>
#include <thread>

void task(const std::string& name) {
    for (int i = 0; i < 3; ++i)
        std::cout << name << " says " << i << std::endl;
}
int main() {
    std::thread t1(task, "A");
    std::thread t2(task, "B");

    t1.join();
    t2.join();

    return 0;
}
```



Multithreading

- Allows you to run multiple functions in *one* program concurrently

- Each thread runs within the same process, so they **share a virtual address space**.
  - A byproduct of this is that communication between threads is trivial
  - This also means that if two threads try to use the same resource (memory, file descriptors, streams, etc.), at the same time, it can have undefined behavior

- Each thread runs performs its function independently of the other

Revisiting our code

```cpp
#include <iostream>
#include <thread>

void task(const std::string& name) {
    for (int i = 0; i < 3; ++i)
        std::cout << name << " says " << i << std::endl;
}
int main() {
    std::thread t1(task, "A");
    std::thread t2(task, "B");

    t1.join();
    t2.join();

    return 0;
}
```

Possible Orderings

```
A says 0
B says 0
A says 1
B says 1
A says 2
B says 2
```

```
A says 0
A says 1
A says 2
B says 0
B says 1
B says 2
```

```
A says 0
B says 0
B says 1
A says 1
B says 2
A says 2
```

```
B says 0
B says 1
B says 2
A says 0
A says 1
A says 2
```

Revisiting our code

```cpp
#include <iostream>
#include <thread>

void task(const std::string& name) {
    for (int i = 0; i < 3; ++i)
        std::cout << name << " says " << i << std::endl;
}
int main() {
    std::thread t1(task, "A");
    std::thread t2(task, "B");

    t1.join();
    t2.join();

    return 0;
}
```

Another Possible Ordering

```
B says A says 0
B says 1
B says 2
0
A says 1
A says 2
```

What happened here?

If two threads try to use the same resource at the same time, it can have undefined behavior

What questions can I answer?

Race Conditions

What is a Race Condition

- An execution ordering that leads to undesired behavior in our programs

- The crux of thread safety. A thread safe function executes the same when called concurrently or sequentially.

```
B says A says 0
B says 1
B says 2
0
A says 1
A says 2
```

In our previous code, the `operator<<` in the `std::cout` stream is **not** thread safe

The threads are trying to access the same **shared** `std::cout` stream, which causes undesired behavior

In CS111

- You can add `oslock` add and `osunlock` to `std::cout` to make it thread safe.
- These facilities are not part of the C++ Standard Library, they are custom CS111 functions found in the file “ostreamlock.h”

```
std::cout << oslock << “Hello, World!” << std::endl << osunlock;
```

Concurrency with data races!

```cpp
#include <iostream>
#include <thread>
#include <mutex>

int counter = 0;

void increment() {
    for (int i = 0; i < 100; ++i) {
        ++counter;
    }
}
int main() {
    std::thread t1(increment);
    std::thread t2(increment);

    t1.join();
    t2.join();

    std::cout << "Final counter value: " << counter << std::endl;
    return 0;
}
```

The canonical example of a race condition is one with a counter!

Concurrency with data races!

```cpp
#include <iostream>
#include <thread>
#include <mutex>

int counter = 0;

void increment() {
    for (int i = 0; i < 100; ++i) {
        ++counter;
    }
}
int main() {
    std::thread t1(increment);
    std::thread t2(increment);

    t1.join();
    t2.join();

    std::cout << "Final counter value: " << counter << std::endl;
    return 0;
}
```

The canonical example of a race condition is one with a counter!

Question: What is the race condition here?

Concurrency with data races!

```cpp
#include <iostream>
#include <thread>
#include <mutex>

int counter = 0;

void increment() {
    for (int i = 0; i < 100; ++i) {
        ++counter;
    }
}
int main() {
    std::thread t1(increment);
    std::thread t2(increment);

    t1.join();
    t2.join();

    std::cout << "Final counter value: " << counter << std::endl;
    return 0;
}
```

The canonical example of a race condition is one with a counter!

Question: What is the race condition here?

1. `t1` reads `counter` as 27
1. `t2` reads `counter` as 27
1. `t1` increments `counter` to 28
1. `t2` increments `counter` as 28
1. `t1` saves `counter` as 28
1. `t1` saves `counter` as 28

Two writes occurred, but in the eyes of memory only one did!

What questions can I answer?

So, how do we fix this?

Mutexes (Mutual Exclusions)!

```cpp
#include <iostream>
#include <thread>
#include <mutex>

int counter = 0;
std::mutex counterLock;
void increment() {
    for (int i = 0; i < 100; ++i) {
        counterLock.lock();
        ++counter;
        counterLock.unlock();
    }
}
int main() {
    std::thread t1(increment);
    std::thread t2(increment);

    t1.join();
    t2.join();

    std::cout << "Final counter value: " << counter << std::endl;
    return 0;
}
```

The canonical example of a race condition is one with a counter!

A mutex allows you to define some *critical section* such that only one thread can execute that section at a time.

Critical Section: An area where a thread would access a shared resource.

The critical section in this case is in between the highlighted segments of code.

How many mutexes should we create?

- The rule of thumb is one mutex for each single variable/critical section

```
void func1(int& counter1,
              mutex& counter1Lock) {
    counter1Lock.lock();
    counter1++;
    counter1Lock.unlock();
}
```

```
void func2(int& counter2,
              mutex& counter2Lock) {
    counter2Lock.lock();
    counter2--;
    counter2Lock.unlock();
}
```

```
int main() {
    int counter1 = 0;
    int counter2 = 0;
    mutex counter1Lock;
    mutex counter2Lock;
    thread t1(func1, ref(counter1), ref(counter1Lock));
    thread t2(func2, ref(counter2), ref(counter2Lock));
    ... // make more threads that also call these functions
```

What questions can I answer?

Condition Variables

What?

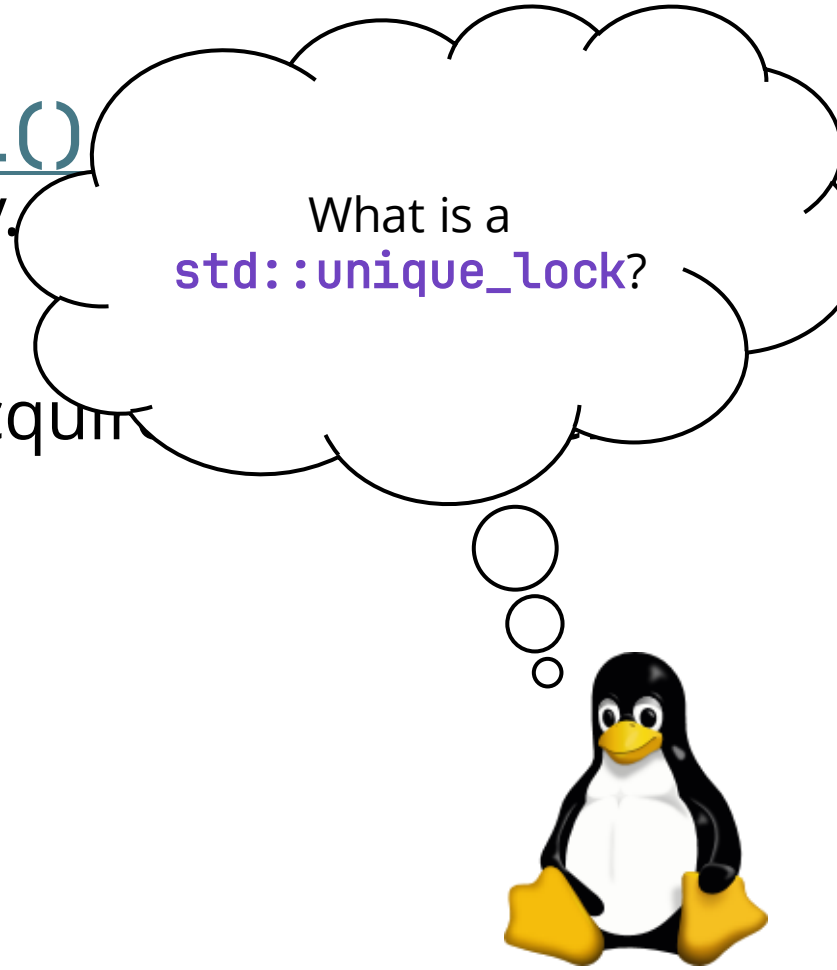
- A way to wait on an **event**
- **`cv.wait(lock)`** can be used to sleep until another thread sends a notification to the condition variable
- To send a notification to a condition variable you can use **`cv.notify_all()`**

```
void wait ( std::unique_lock<std::mutex>& lock );
```

- **Atomically** calls `lock.unlock()` and puts the caller to sleep
- The thread will be unblocked when `notify_all()` or `notify_one()` is executed. It may also be unblocked spuriously.
- When unblocked, calls **`lock.lock()`** to try to acquire the lock, then returns.

```
void wait ( std::unique_lock<std::mutex>& lock );
```

- **Atomically** calls `lock.unlock()` and puts the caller to sleep
- The thread will be unblocked when `notify_all()` is executed. It may also be unblocked spuriously.
- When unblocked, calls **`lock.lock()`** to try to acquire the lock before it returns.



```
void Bridge::leave_eastbound(size_t id) {
    bridge_lock.lock();
    n_crossing_eastbound--;
    if (n_crossing_eastbound == 0) {
        none_crossing_eastbound.notify_all();
    }
    print(id, "crossed", true);
    bridge_lock.unlock();
}
```

From CS111 W26 lecture 15, slide 24

```
void Bridge::leave_eastbound(size_t id) {
    unique_lock<mutex> lock(bridge_lock);
    n_crossing_eastbound--;
    if (n_crossing_eastbound == 0) {
        none_crossing_eastbound.notify_all();
    }
    print(id, "crossed", true);
}
```

```
void Bridge::leave_eastbound(size_t id) {
    unique_lock<mutex> lock(bridge_lock);
    n_crossing_eastbound--;
    if (n_crossing_eastbound == 0) {
        none_crossing_eastbound.notify_all();
    }
    print(id, "crossed", true);
}
```

Auto-unlocks lock here (goes out of scope)

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    while (permits == 0) {
        permitsCV.wait(permitsLock);
    }
    permits--;
    permitsLock.unlock();
}
```

From CS111 W26 lecture 15, slide 7

```
static void grantPermission(size_t& permits,
condition_variable_any& permitsCV, mutex& permitsLock) {
    permitsLock.lock();
    permits++;
    if (permits == 1) permitsCV.notify_all();
    permitsLock.unlock();
}
```

From CS111 W26 lecture 14, slide 33

What questions can I answer?

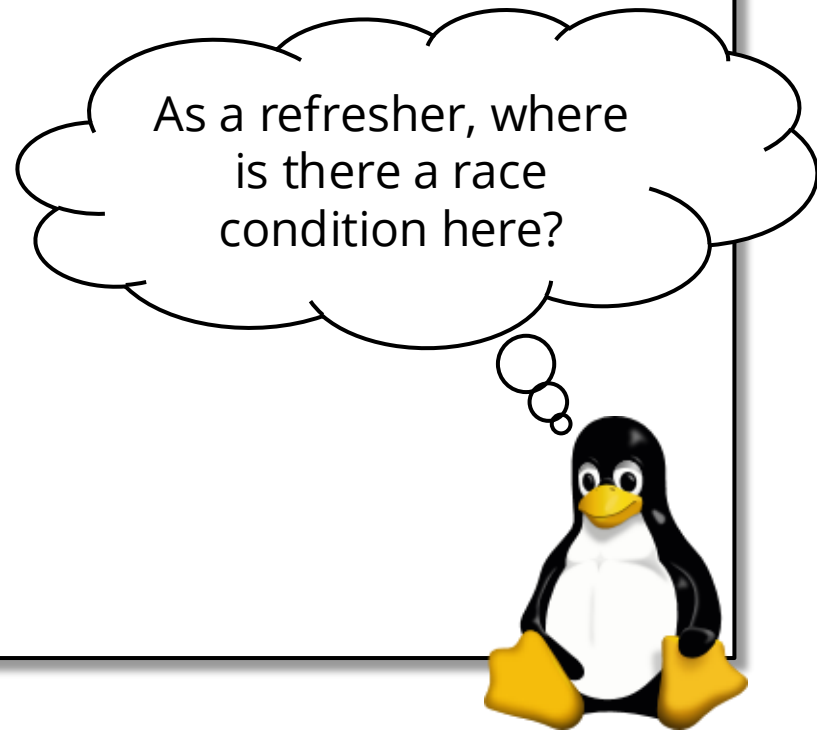
Implementing Locks

We can have a race condition in a lock

The example from lecture

```
Int locked = 0;
ThreadQueue = 0;

void Lock::lock() {
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```



From CS111 W26 lecture 19, slide 25

Thread 1 (Running)

```
Int locked = 0;
ThreadQueue = 0;

void Lock::lock() {
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Thread 2

```
Int locked = 0;
ThreadQueue = 0;

void Lock::lock() {
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Thread 1 (Running)

```
Int locked = 0;
ThreadQueue = 0;

void Lock::lock() {
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

After it checked the condition, but *before* it set `locked` equal to 1

Context switch

Thread 2

```
Int locked = 0;
ThreadQueue = 0;

void Lock::lock() {
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Thread 1

```
Int locked = 0;
ThreadQueue = 0;

void Lock::lock() {
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Thread 2 (Running)

```
Int locked = 0;
ThreadQueue = 0;

void Lock::lock() {
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Thread 2 now checks the condition and does not observe the Lock as locked

Thread 1

```
Int locked = 0;
ThreadQueue = 0;

void Lock::lock() {
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Thread 2 (Running)

```
Int locked = 0;
ThreadQueue = 0;

void Lock::lock() {
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Thread 2 now checks the condition and does not observe the Lock as locked

Thread 1

```
Int locked = 0;
ThreadQueue = 0;

void Lock::lock() {
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Thread 2 (Running)

```
Int locked = 0;
ThreadQueue = 0;

void Lock::lock() {
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Context switch

Thread 1 (Running)

```
Int locked = 0;
ThreadQueue = 0;

void Lock::lock() {
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Thread 2

```
Int locked = 0;
ThreadQueue = 0;

void Lock::lock() {
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Thread 1 (Running)

```
Int locked = 0;
ThreadQueue = 0;

void Lock::lock() {
    if (!locked) {
    ⇒ locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Thread 2

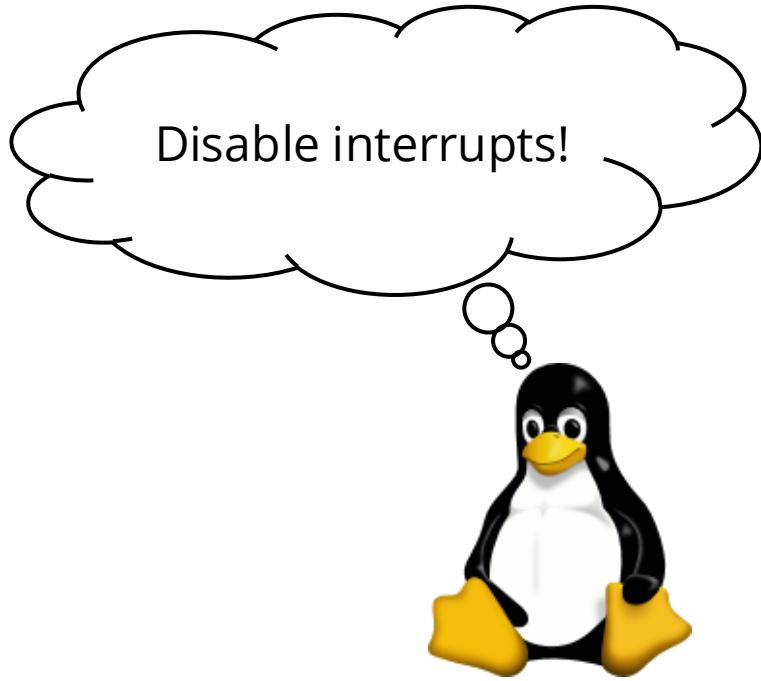
```
Int locked = 0;
ThreadQueue = 0;

void Lock::lock() {
    if (!locked) {
    ⇒ locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Context switch


NYPD

What is one sure way to get around this?



In a uniprocessor system all you have to do is disable interrupts

What about for a multi-core system? 🤔

Other threads running on *other* cores

- When you disable an interrupt, you only disable it for **one core**.
- If you have multiple cores, those other cores can still modify your state.
- We use the **`exchange`** method from **`std::atomic`**

std::atomic exchange

```cpp
#include <atomic>

int main() {
  std::atomic<int> var;
    int spinlock = 0;
  spinlock = var.exchange(1);
  return 0;
}
```

Let’s check it out!

lock

```
void Lock::lock() {
    while(spinlock.exchange(true)) {
        /* do nothing */
    }
    if (!locked) {
        locked = 1;
        spinlock = false;
    } else {
        q.add(currentThread);
        currentThread->state = BLOCKED;
        spinlock = false;
        redispatch();
    }
}
```

unlock

```
void Lock::Unlock() {
    while (spinlock.exchange(true)) {
        /* do nothing */
    }
    if (q.empty()) {
        locked = false;
    }
    else {
        unblockedThread(q.remove());
    }
    spinlock = false;
}
```

Is this good enough?

```
void Lock::lock() {
    while(spinlock.exchange(true)) {
         /* do nothing */
    }
    if (!locked) {
        locked = 1;
        spinlock = false;
    } else {
        q.add(currentThread);
        currentThread->state = BLOCKED;
        spinlock = false;
        redispatch();
    }
}
```

```
void Lock::Unlock() {
    while (spinlock.exchange(true)) {
        /* do nothing */
    }
    if (q.empty()) {
        locked = false;
    }
    else {
        unblockedThread(q.remove());
    }
    spinlock = false;
}
```

Is this good enough?

```
void Lock::lock() {
    intrDisable();
    while(spinlock.exchange(true)) {
        /* do nothing */
    }
    if (!locked) {
        locked = 1;
        spinlock = false;
    } else {
        q.add(currentThread);
        currentThread->state = BLOCKED;
        spinlock = false;
        redispatch();
    }
    intrEnable();
}
```

```
void Lock::Unlock() {
    intrDisable();
    while (spinlock.exchange(true)) {
        /* do nothing */
    }
    if (q.empty()) {
        locked = false;
    }
    else {
        unblockedThread(q.remove());
    }
    spinlock = false;
    intrEnable();
}
```

What questions can I answer?

Deadlocks

Under what conditions do we deadlock?

- A **deadlock** is when all threads cannot make forward progress because they are waiting on another thread that has a shared resource

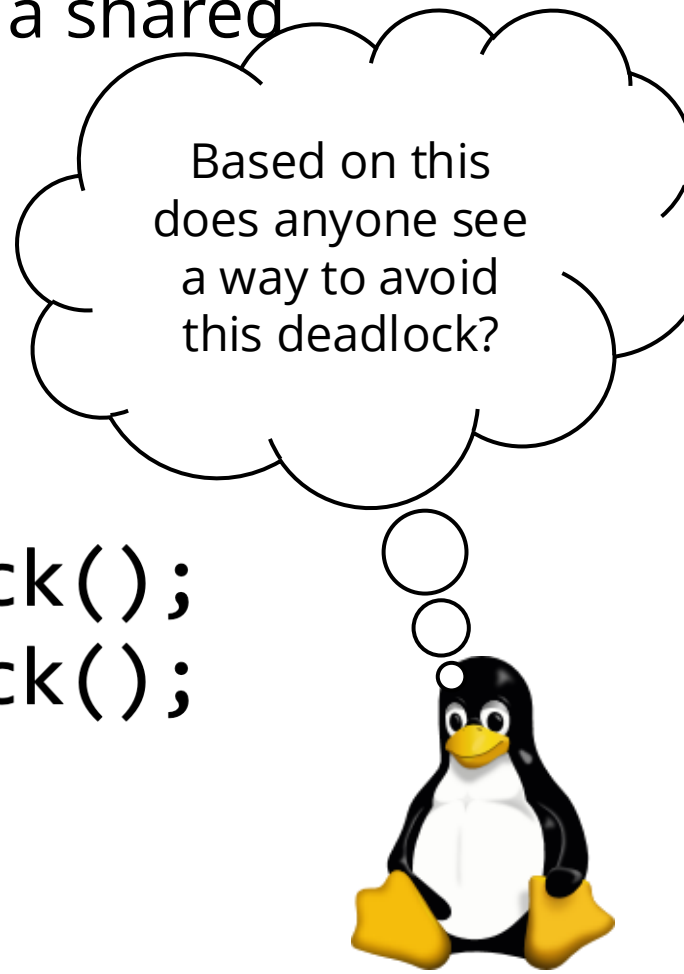
- Lecture example:

Thread A

```
mutex1.lock();
mutex2.lock();
...
```

Thread B

```
mutex2.lock();
mutex1.lock();
...
```



Prevent circularities, request a mutex in the same order

Bad, no matter what here we get a deadlock

Thread A

```
mutex1.lock();
mutex2.lock();
...
```

Thread B

```
mutex2.lock();
mutex1.lock();
...
```

What are the four Deadlock conditions?

1. Using locks in the first place – having mutual exclusion

1. No preemption, this means resources are not taken away once given to a thread

1. Multiple independent requests where threads don’t ask for resources all at once and they hold resources while waiting.

1. Circularity as we saw before!

Think-Pair-Share (3-min)

In what situations do we want to disable interrupts?

⭐ Any time that we don’t want to be interrupted by a timer interrupt! ⭐

When do we not want to be interrupted?

- When we are changing some state
  - i.e. when setting/changing the value of `locked`

- When we want to do something *atomically*, we want to make sure that we're not interrupted
  - Think about cases where that is the when thinking about how locks are implemented