CS161: Operating Systems

Matt Welsh
mdw@eecs.harvard.edu

Lecture 5: Synchronization
February 15, 2007

© 2007 Matt Welsh – Harvard University

Synchronization

Threads cooperate in multithreaded programs in several ways:

Access to shared state
- e.g., multiple threads accessing a memory cache in a Web server

To coordinate their execution
- e.g., Pressing stop button on browser cancels download of current page
- "stop button thread" has to signal the "download thread"

For correctness, we have to control this cooperation

Must assume that threads can **interleave executions arbitrarily** and at **run at different rates**
- In some sense this is the "worst case" scenario.
- Our goal: to control thread cooperation using *synchronization*
  - *enables us to restrict the interleaving of executions*

© 2007 Matt Welsh – Harvard University

Shared Resources

We'll focus on coordinating access to shared resources

Basic problem:

- Two concurrent threads are accessing a shared variable
- If the variable is read/modified/written by both threads, then access to the variable must be controlled
  - *Otherwise, unexpected results may occur*

Over the next two lectures, we'll look at:

- Mechanisms to control access to shared resources
  - *Low-level mechanisms: locks*
  - *Higher level mechanisms: mutexes, semaphores, monitors, and condition variables*
- Patterns for coordinating access to shared resources
  - *bounded buffer, producer-consumer, …*

This stuff is complicated and rife with pitfalls

- Details are important for completing assignments
- Expect questions on the midterm/final!

© 2007 Matt Welsh – Harvard University

Shared Variable Example

Suppose we implement a function to withdraw money from a bank account:

```
int withdraw(account, amount) {
  balance = get_balance(account);
  balance = balance - amount;
  put_balance(account, balance);
  return balance;
}
```

Now suppose that you and your roommate share a bank account with a balance of $1500.00 (not that this is necessarily a good idea...)

- What happens if you both go to separate ATM machines, and simultaneously withdraw $100.00 from the account?

Example continued

We represent the situation by creating a *separate thread* for each ATM user doing a withdrawal

- Both threads run on the same bank server system

Thread 1

```
int withdraw(account, amount) {
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

Thread 2

```
int withdraw(account, amount) {
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

What's the problem with this?

- What are the possible balance values after each thread runs?

© 2007 Matt Welsh – Harvard University

Interleaved Execution

The execution of the two threads can be *interleaved*

- Assume preemptive scheduling
- Each thread can context switch after each instruction
- We need to worry about the worst-case scenario!

Execution sequence as seen by CPU

```
balance = get_balance(account);
balance -= amount;
```

context switch

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
```

context switch

```
put_balance(account, balance);
```

What's the account balance after this sequence?

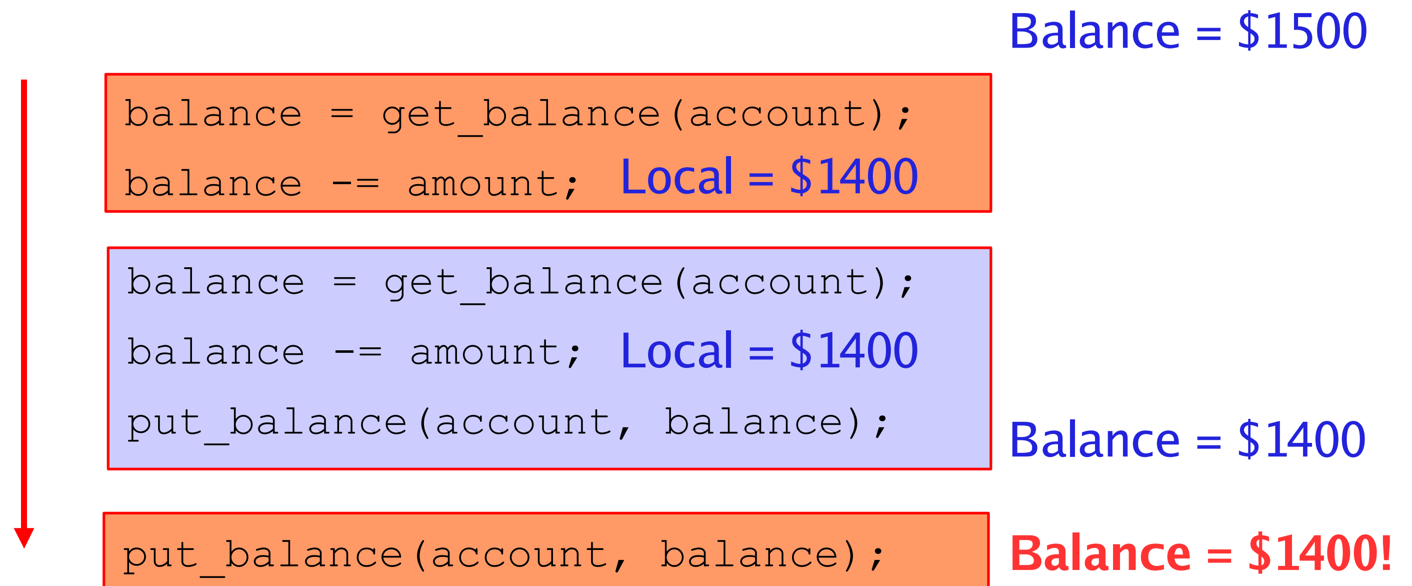
- And who's happier, the bank or you???

© 2007 Matt Welsh – Harvard University

Interleaved Execution

The execution of the two threads can be *interleaved*

- Assume preemptive scheduling
- Each thread can context switch after each instruction
- We need to worry about the worst-case scenario!



What's the account balance after this sequence?

- And who's happier, the bank or you???

© 2007 Matt Welsh – Harvard University

Race Conditions

The problem is that two concurrent threads access a shared resource without any synchronization

- This is called a ***race condition***
- The result of the concurrent access is non-deterministic
- Result depends on:
  - *Timing*
  - *When context switches occurred*
  - *Which thread ran at at context switch*
  - *What the threads were doing*

We need mechanisms for controlling access to shared resources in the face of concurrency

- This allows us to reason about the operation of programs
- Essentially, we want to re-introduce determinism into the thread's execution

Synchronization is necessary for any shared data structure

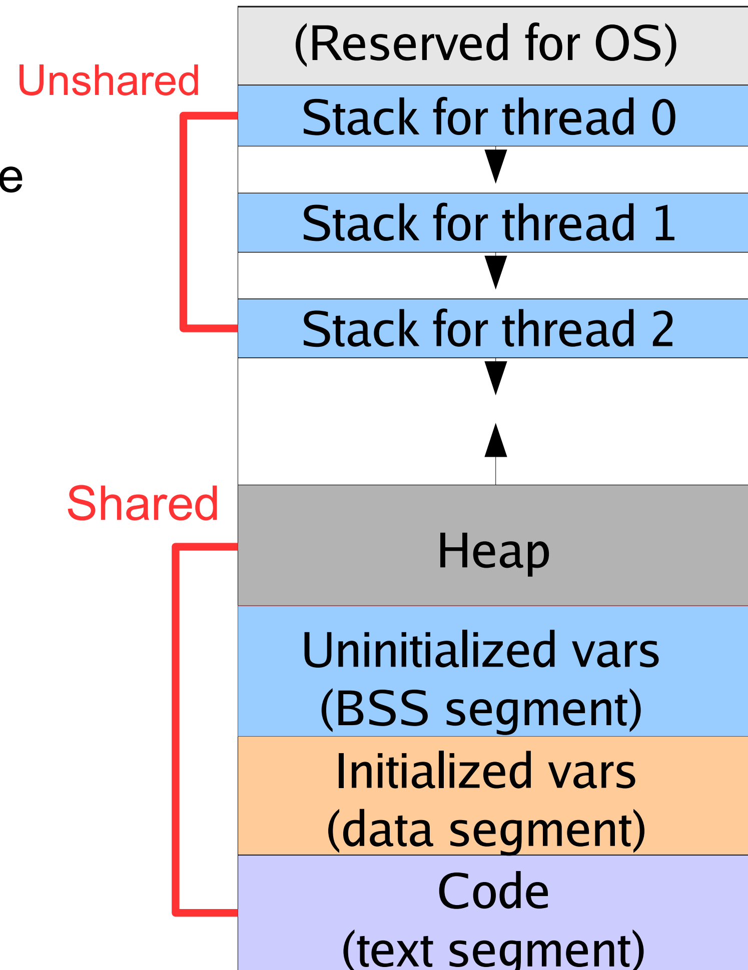
- buffers, queues, lists, hash tables, …

© 2007 Matt Welsh – Harvard University

Which resources are shared?

Local variables in a function are **not** shared

- They exist on the stack, and each thread has its own stack
- You can't safely pass a pointer from a local variable to another thread
  - *Why?*



Global variables ***are*** shared

- Stored in static data portion of the address space
- Accessible by any thread

Dynamically-allocated data **is** shared

- Stored in the heap, accessible by any thread

© 2007 Matt Welsh – Harvard University

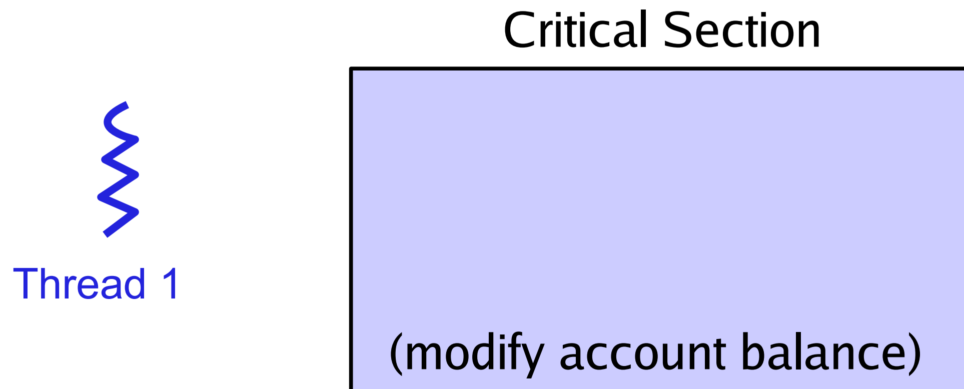
Mutual Exclusion

We want to use *mutual exclusion* to synchronize access to shared resources

- Meaning: When only one thread can access a shared resource at a time.

Code that uses mutual exclusion to synchronize its execution is called a *critical section*

- Only one thread at a time can execute code in the critical section
- All other threads are forced to wait on entry
- When one thread leaves the critical section, another can enter



© 2007 Matt Welsh – Harvard University

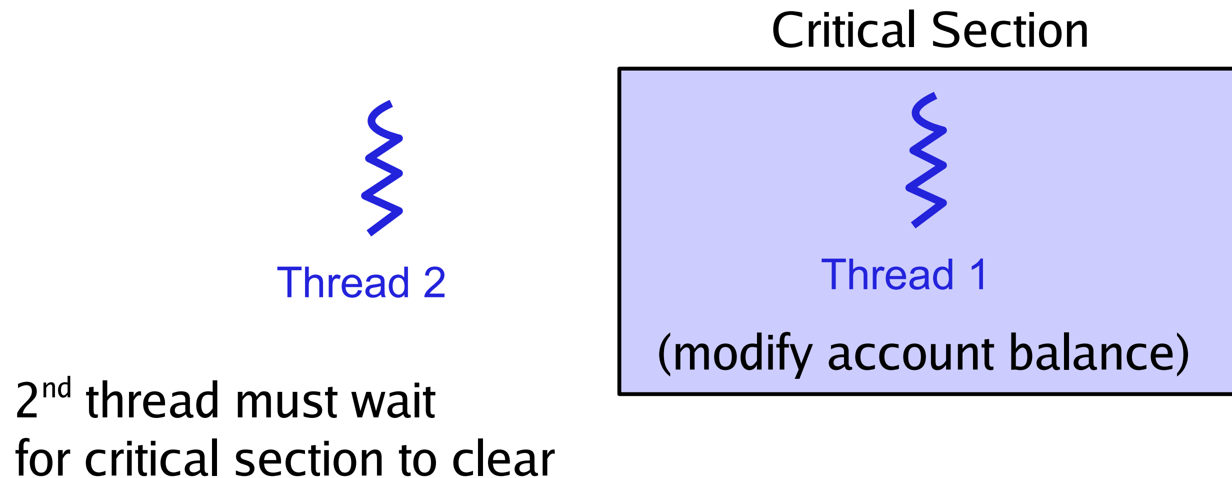
Mutual Exclusion

We want to use *mutual exclusion* to synchronize access to shared resources

- Meaning: When only one thread can access a shared resource at a time.

Code that uses mutual exclusion to synchronize its execution is called a *critical section*

- Only one thread at a time can execute code in the critical section
- All other threads are forced to wait on entry
- When one thread leaves the critical section, another can enter



© 2007 Matt Welsh – Harvard University

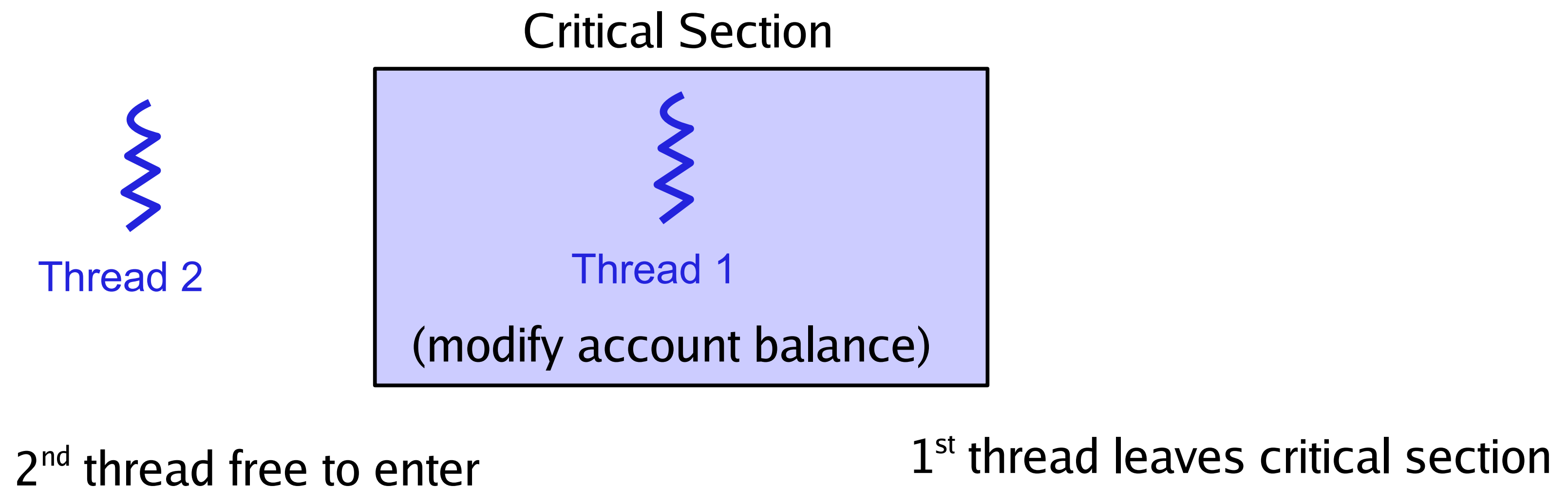
Mutual Exclusion

We want to use *mutual exclusion* to synchronize access to shared resources

- Meaning: When only one thread can access a shared resource at a time.

Code that uses mutual exclusion to synchronize its execution is called a *critical section*

- Only one thread at a time can execute code in the critical section
- All other threads are forced to wait on entry
- When one thread leaves the critical section, another can enter



© 2007 Matt Welsh – Harvard University

Critical Section Requirements

Mutual exclusion

- At most one thread is currently executing in the critical section

Progress

- If thread T1 is *outside* the critical section, then T1 cannot prevent T2 from entering the critical section

Bounded waiting (no starvation)

- If thread T1 is waiting on the critical section, then T1 will *eventually* enter the critical section
  - *Assumes threads eventually leave critical sections*

Performance

- The overhead of entering and exiting the critical section is small with respect to the work being done within it

© 2007 Matt Welsh – Harvard University

Locks

A *lock* is a object (in memory) that provides the following two operations:

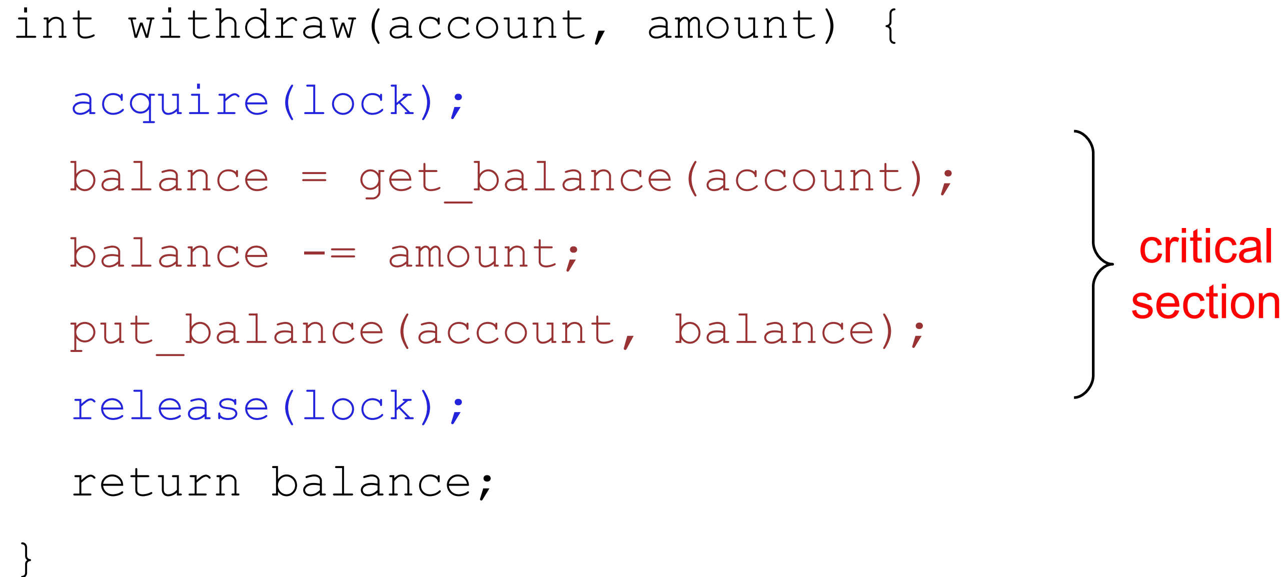
- acquire( ): a thread calls this before entering a critical section
    - *May require waiting to enter the critical section*
- release( ): a thread calls this after leaving a critical section
    - *Allows another thread to enter the critical section*

A call to acquire( ) must have a corresponding call to release( )

- Between acquire( ) and release( ), the thread *holds* the lock
- acquire( ) does not return until the caller holds the lock
    - *At most one thread can hold a lock at a time (usually!)*
    - *We'll talk about the exceptions later...*

What can happen if acquire( ) and release( ) calls are not paired?

Using Locks



Why is the "return" statement outside of the critical section?

© 2007 Matt Welsh – Harvard University

Execution with Locks

```
acquire(lock);
balance = get_balance(account);
balance -= amount;
```

Thread 1 runs

```
acquire(lock);
```

Thread 2 waits on lock

```
put_balance(account, balance);
release(lock);
```

Thread 1 completes

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
release(lock);
```

Thread 2 resumes

What happens when the blue thread tries to acquire the lock?

© 2007 Matt Welsh – Harvard University

Spinlocks

Very simple way to implement a lock:

```
struct lock {
  int held = 0;
}
void acquire(lock) {
   while (lock->held);
   lock->held = 1;
}
void release(lock) {
  lock->held = 0;
}
```

Why doesn't this work?

- Where is the race condition?

© 2007 Matt Welsh – Harvard University

Implementing Spinlocks

Problem is that the internals of the lock acquire/release have critical sections too!

- The acquire( ) and release( ) actions must be *atomic*
- *Atomic* means that the code cannot be interrupted during execution
  - *"All or nothing" execution*

```
struct lock {
  int held = 0;
}
void acquire(lock) {
   while (lock->held);
   lock->held = 1;
}
void release(lock) {
  lock->held = 0;
}
```

What can happen if there is a context switch here?

© 2007 Matt Welsh – Harvard University

Implementing Spinlocks

Problem is that the internals of the lock acquire/release have critical sections too!

- The acquire( ) and release( ) actions must be *atomic*
- *Atomic* means that the code cannot be interrupted during execution
  - *"All or nothing" execution*

```
struct lock {
  int held = 0;
}
void acquire(lock) {
   while (lock->held);
   lock->held = 1;
}
void release(lock) {
  lock->held = 0;
}
```

This sequence needs to be **atomic**

Implementing Spinlocks

Problem is that the internals of the lock acquire/release have critical sections too!

- The acquire( ) and release( ) actions must be *atomic*
- *Atomic* means that the code cannot be interrupted during execution
  - *"All or nothing" execution*

Doing this requires help from hardware!

- Disabling interrupts
  - *Why does this prevent a context switch from occurring?*
- Atomic instructions – CPU guarantees entire action will execute atomically
  - *Test-and-set*
  - *Compare-and-swap*

© 2007 Matt Welsh – Harvard University

Spinlocks using test-and-set

CPU provides the following as one atomic instruction:

```
bool test_and_set(bool *flag) {
  bool old = *flag;
  *flag = True;
  return old;
}
```

So to fix our broken spinlocks, we do this:

```
struct lock {
  int held = 0;
}
void acquire(lock) {
   while(test_and_set(&lock->held));
}
void release(lock) {
  lock->held = 0;
}
```

© 2007 Matt Welsh – Harvard University

What's wrong with spinlocks?

OK, so spinlocks work (if you implement them correctly), and they are simple. So what's the catch?

```
struct lock {
  int held = 0;
}
void acquire(lock) {
   while(test_and_set(&lock->held));
}
void release(lock) {
  lock->held = 0;
}
```

© 2007 Matt Welsh – Harvard University

Problems with spinlocks

Horribly wasteful!

- Threads waiting to acquire locks spin on the CPU
- Eats up lots of cycles, slows down progress of other threads
  - *Note that other threads can still run ... how?*
- What happens if you have a lot of threads trying to acquire the lock?

Only want spinlocks as *primitives* to build higher-level synchronization constructs

© 2007 Matt Welsh – Harvard University

Disabling Interrupts

An alternative to spinlocks:

```
struct lock {
  // Note - no state!
}
void acquire(lock) {
   cli();   // disable interrupts
}
void release(lock) {
   sti();   // reenable interupts
}
```

- Can two threads disable/reenable interrupts at the same time?

What's wrong with this approach?

Disabling Interrupts

An alternative to spinlocks:

```
struct lock {
  // Note - no state!
}
void acquire(lock) {
   cli();   // disable interrupts
}
void release(lock) {
   sti();   // reenable interupts
}
```

- Can two threads disable/reenable interrupts at the same time?

What's wrong with this approach?

- Can only be implemented at kernel level (why?)
- Inefficient on a multiprocessor system (why?)
- All locks in the system are mutually exclusive
  - *No separation between different locks for different bank accounts*

© 2007 Matt Welsh – Harvard University

Mutexes – Blocking Locks

Really want a thread waiting to enter a critical section to *block*

- Put the thread to sleep until it can enter the critical section
- Frees up the CPU for other threads to run

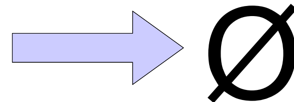
Straightforward to implement using our TCB queues!

Thread 1

1) Check lock state

Lock state

Lock wait queue



Ø

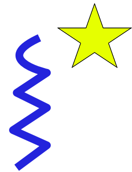
© 2007 Matt Welsh – Harvard University

Mutexes – Blocking Locks

Really want a thread waiting to enter a critical section to *block*

- Put the thread to sleep until it can enter the critical section
- Frees up the CPU for other threads to run

Straightforward to implement using our TCB queues!



Thread 1

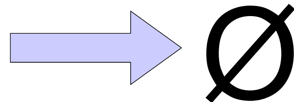
1) Check lock state

2) Set state to locked

3) Enter critical section

Lock state

Lock wait queue



Ø

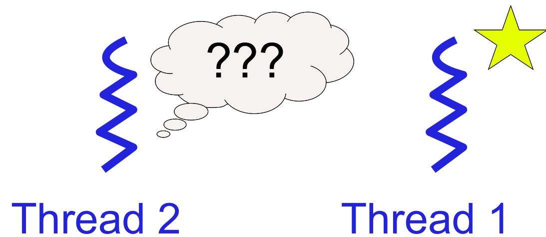
© 2007 Matt Welsh – Harvard University

Mutexes – Blocking Locks

Really want a thread waiting to enter a critical section to *block*

- Put the thread to sleep until it can enter the critical section
- Frees up the CPU for other threads to run

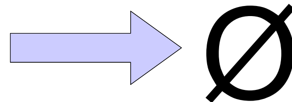
Straightforward to implement using our TCB queues!



1) Check lock state

Lock state

Lock wait queue



Ø

© 2007 Matt Welsh – Harvard University

Mutexes – Blocking Locks

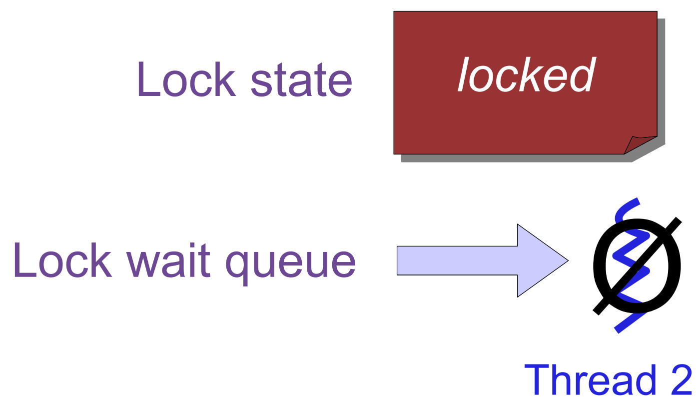
Really want a thread waiting to enter a critical section to *block*

- Put the thread to sleep until it can enter the critical section
- Frees up the CPU for other threads to run

Straightforward to implement using our TCB queues!

1) Check lock state

2) Add self to wait queue (sleep)



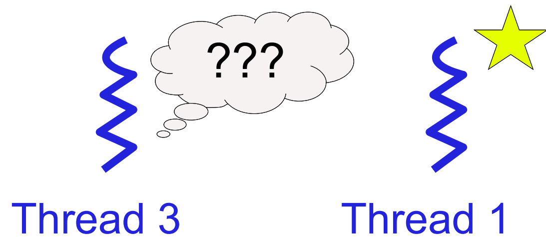
© 2007 Matt Welsh – Harvard University

Mutexes – Blocking Locks

Really want a thread waiting to enter a critical section to *block*

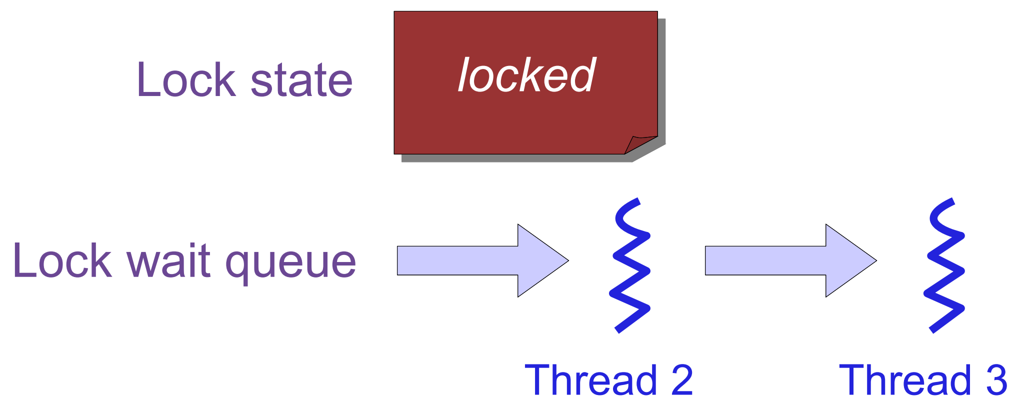
- Put the thread to sleep until it can enter the critical section
- Frees up the CPU for other threads to run

Straightforward to implement using our TCB queues!



1) Check lock state

2) Add self to wait queue (sleep)



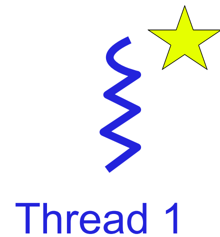
© 2007 Matt Welsh – Harvard University

Mutexes – Blocking Locks

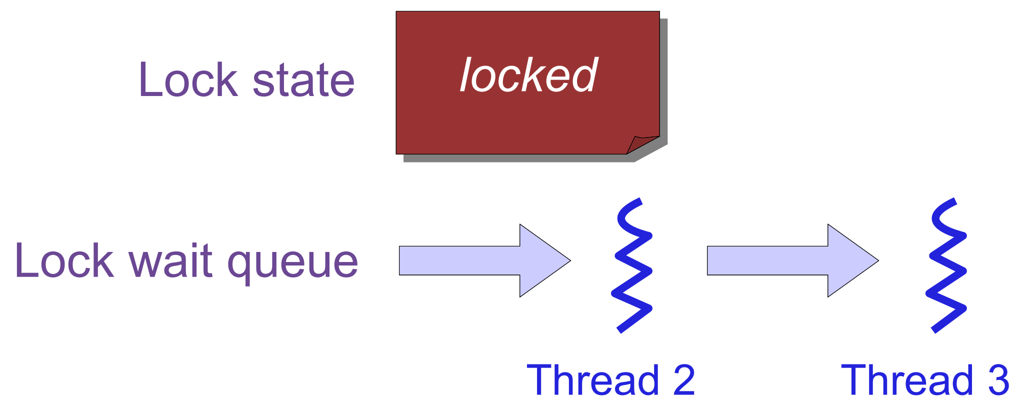
Really want a thread waiting to enter a critical section to *block*

- Put the thread to sleep until it can enter the critical section
- Frees up the CPU for other threads to run

Straightforward to implement using our TCB queues!



1) Thread 1 finishes critical section



© 2007 Matt Welsh – Harvard University

Mutexes – Blocking Locks

Really want a thread waiting to enter a critical section to *block*

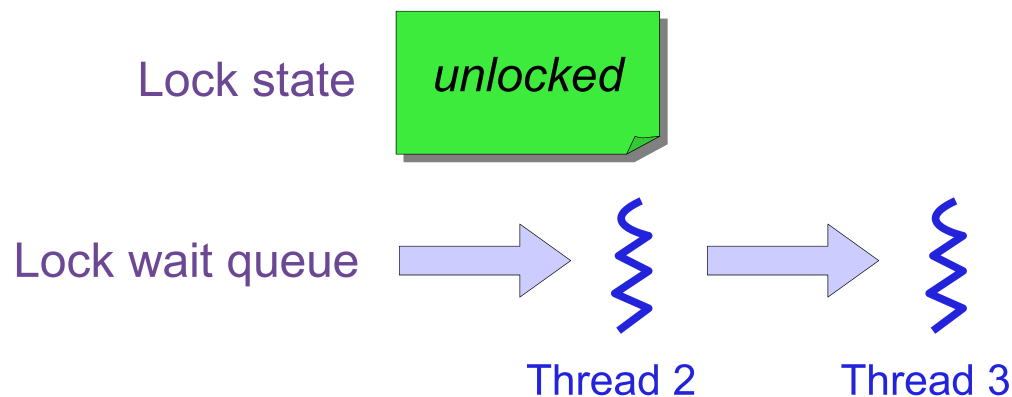
- Put the thread to sleep until it can enter the critical section
- Frees up the CPU for other threads to run

Straightforward to implement using our TCB queues!

1) Thread 1 finishes critical section

2) Reset lock state to unlocked

3) Wake one thread from wait queue



© 2007 Matt Welsh – Harvard University

Mutexes – Blocking Locks

Really want a thread waiting to enter a critical section to *block*

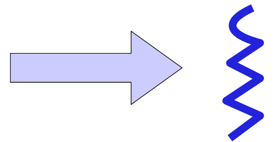
- Put the thread to sleep until it can enter the critical section
- Frees up the CPU for other threads to run

Straightforward to implement using our TCB queues!

Thread 3 can now grab lock and enter critical section

Lock state

Lock wait queue



Thread 2

© 2007 Matt Welsh – Harvard University

Limitations of locks

Locks are great, and simple. What can they not easily accomplish?

What if you have a data structure where it's OK for many threads to read the data, but only one thread to write the data?

- Bank account example.
- Locks only let one thread access the data structure at a time.

© 2007 Matt Welsh – Harvard University

Limitations of locks

Locks are great, and simple. What can they not easily accomplish?

What if you have a data structure where it's OK for many threads to read the data, but only one thread to write the data?

- Bank account example.
- Locks only let one thread access the data structure at a time.

What if you want to protect access to two (or more) data structures at a time?

- e.g., Transferring money from one bank account to another.
- Simple approach: Use a separate lock for each.
- What happens if you have transfer from account A -> account B, at the same time as transfer from account B -> account A?
  - *Hmmmmm ... tricky.*
  - *We will get into this next time.*

© 2007 Matt Welsh – Harvard University

Next Lecture

Higher level synchronization primitives:
  How do to fancier stuff than just locks

Semaphores, monitors, and condition variables

- Implemented using basic locks as a primitive

Allow applications to perform more complicated coordination schemes

© 2007 Matt Welsh – Harvard University