Today

- Concurrency Problems
- Synchronization Mechanisms

Review

- What is a problem with multi-threaded programming?
- How do we solve that problem?

Review: Definitions

- ***Race condition***: output of a concurrent program depends on the order of operations between threads
- ***Mutual exclusion***: only one thread does a particular thing at a time
  - Critical section: piece of code that only one thread can execute at once
- ***Lock***: prevent someone from doing something
  - Lock before entering critical section, before accessing shared data
  - Unlock when leaving, after done accessing shared data
  - Wait if locked (all synchronization involves waiting!)
  - Also called ***mutex*** or ***mutex lock***

Review: Locks

- `Acquire`
  - wait until lock is free, then take it
- `Release`
  - release lock, waking up anyone waiting for it

1. At most one lock holder at a time (safety)
2. If no one holding, acquire gets lock (progress)
3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock (progress)

Discussion: Why only Acquire/Release?

- The Lock API seems a little too simple
- Suppose we add a method to a lock, to ask if the lock is free.
  - Suppose it returns true. Then what?

Will this code work?

```
if (p == null) {
     lock.acquire();
     p = newP();
     lock.release();
}
p.method();
```

```
newP() {
       p = new P(…);
       p.field1 = …
       p.field2 = …
       return p;
}
```

Will this code work?

```
if (p == null) {
     lock.acquire();
     p = newP();
     lock.release();
}
p.method();
```

```
newP() {
       p = new P(…);
       p.field1 = …
       p.field2 = …
       return p;
}
```

No!
p can be written before lock is acquired!

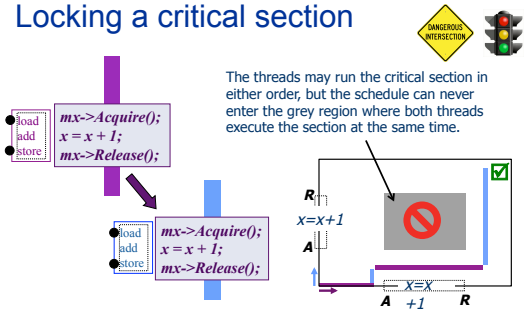
Will this code work?

```
lock.acquire();
if (p == null) {
     p = newP();
}
lock.release();
p.method();
```

```
newP() {
       p = new P(…);
       p.field1 = …
       p.field2 = …
       return p;
}
```

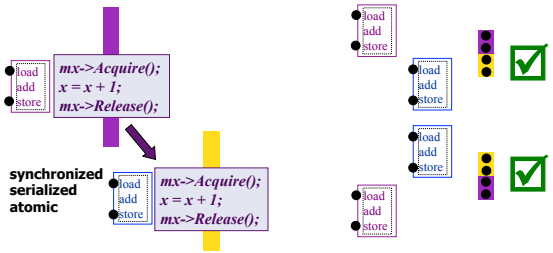
Assuming: method is thread safe.

Locking a critical section



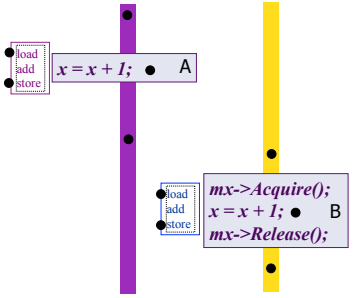
Holding a shared mutex prevents competing threads from entering a critical section protected by the shared mutex (monitor). At most one thread runs in the critical section at a time.

Locking a critical section

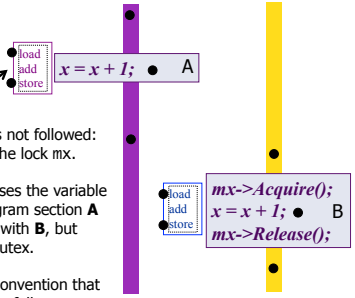


Holding a shared mutex prevents competing threads from entering a critical section. If the critical section code acquires the mutex, then its execution is serialized: only one thread runs it at a time.

How about this?



How about this?

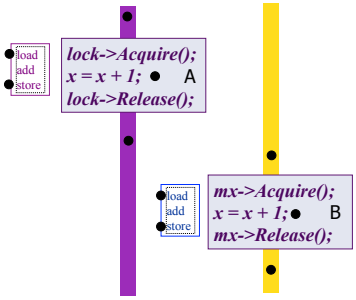


The locking discipline is not followed: purple fails to acquire the lock mx.

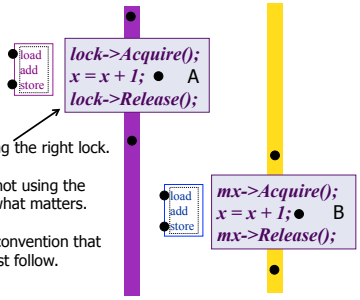
Or rather: purple accesses the variable **x** through another program section **A** that is mutually critical with **B**, but does not acquire the mutex.

A locking scheme is a convention that the entire program must follow.

How about this?



How about this?



This guy is not acquiring the right lock.

Or whatever. They're not using the same lock, and that's what matters.

A locking scheme is a convention that the entire program must follow.

Rules for Using Locks

- Lock is initially free
- Always acquire before accessing shared data structure
  - Beginning of procedure!
- Always release after finishing with shared data
  - End of procedure!
  - Only the lock holder can release
  - DO NOT throw lock for someone else to release
- Never access shared data without lock
  - Danger!

Revisiting a (Seemingly) Simple Program

```
l = new Lock();
x = 5;
```

```
l.acquire();
x=x+1;
print(x);
l.release();
```

```
l.acquire();
x=x+1;
print(x);
l.release();
```

What is the output?

Debugging non-determinism

- Requires worst-case reasoning
  - Eliminate all ways for program to break
- Debugging is hard
  - Can't test all possible interleavings
  - Bugs may only happen sometimes
- Heisenbug
  - Re-running program may make the bug disappear
  - Doesn't mean it isn't still there!

Spinlock: a first try

```
int avail = 0;

acquire() {
      while (avail == 1)
            {;}
      ASSERT (avail == 0);

      avail = 1;
}

release();
      ASSERT(avail == 1)
      avail = 0;
}
```

Global spinlock variable

Busy-wait until lock is free.

Spinlocks provide mutual exclusion among cores without blocking
→ don't need to context switch

Spinlocks are useful for lightly contended critical sections
where there is no risk that a thread is preempted while it is holding the lock, i.e., in the lowest levels of the kernel.

Spinlock: what went wrong

```
int avail = 0;

acquire() {
     while (avail == 1)
          {;}
     avail = 1;
}

release();
     avail = 0;
}
```

Race to acquire.
Two (or more) cores see s == 0.

We need an atomic "toehold"

- To implement safe mutual exclusion, we need support for some sort of "magic toehold" for synchronization.
  - The lock primitives themselves have critical sections to test and/or set the lock flags.
- Safe mutual exclusion on multicore systems requires some hardware support: ***atomic instructions***
  - Examples: test-and-set, compare-and-swap, fetch-and-add.
  - These instructions perform an atomic read-modify-write of a memory location. We use them to implement locks.
  - If we have any of those, we can build higher-level synchronization objects.
  - Note: we also must be careful of interrupt handlers….
  - They are expensive, but necessary.

Takeaway: Mutexes are often implemented using hardware

Atomic instructions: Test-and-Set

Problem: interleaved load/ test/store.

Solution: TSL atomically sets the flag and leaves the old value in a register.

```
Spinlock::Acquire () {
     while(held);
     held = 1;
}
```

One example: tsl test-and-set-lock (from an old machine)

Wrong

```
     load 4(SP), R2        ; load "this"
busywait:
     load  4(R2), R3       ; load "held" flag
     bnz R3, busywait      ; spin if held wasn't zero
     store #1, 4(R2)       ; held = 1
```

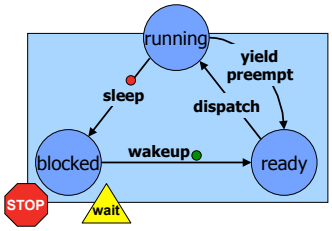
Right

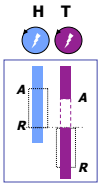
```
     load 4(SP), R2        ; load "this"
busywait:
     tsl 4(R2), R3         ; test-and-set this->held
     bnz R3, busywait      ; spin if held wasn't zero
```

bnz means "branch if not zero"

Locking and blocking

If thread **T** attempts to acquire a lock that is busy (held), **T** must spin **and/or block** (sleep) until the lock is free. By sleeping, **T** frees up the core for some other use. Just sitting and spinning is wasteful!



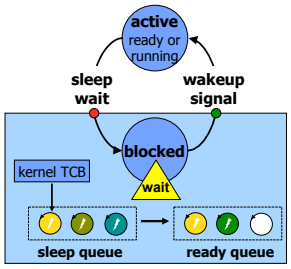


Note: **H** is the lock holder when **T** attempts to acquire the lock.

Blocking

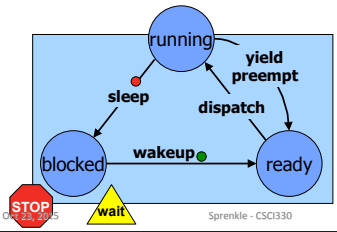
This slide applies to the process abstraction too, or, more precisely, to the main thread of a process.

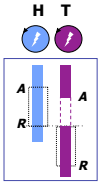
When a thread is **blocked** on a **synchronization object,** its TCB is placed on a ***sleep queue*** of threads waiting for an event on **that object**.



Locking and blocking

T enters the kernel (via syscall) to block.
Suppose **T** is sleeping in the kernel to wait for a contended lock (mutex).
When the lock holder **H** releases, **H** enters the kernel (via syscall) to wakeup a waiting thread (e.g., **T**).





H can block too, perhaps for some other resource!
H doesn't implicitly release the lock just because it blocks.

New Problem: Ping-Pong

Alternate threads working, in pseudocode:

```
void
PingPong() {
   while(not done) {
      …
      if (blue)
            switch to purple;
      if (purple)
            switch to blue;
   }
}
```

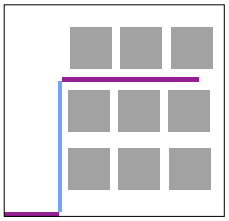
How would we implement using locks?

Ping-Pong with Mutexes?

```
void
PingPong() {
   while(not done) {
      Mx->Acquire();
      …
      Mx->Release();
   }
}
```

This solution doesn't work. Why?

Mutexes don't work for ping-pong



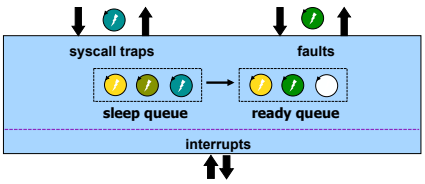
Mutexes can't ensure alternating between the threads.
Ex: Blue could take two turns before Purple gets a turn.

Looking Ahead

- Project 2 due in two Wednesdays
- Extra slides next, to help your understanding

Sleeping in the kernel

Any trap or fault handler may suspend (**sleep**) the current thread, leaving its state (call frames) on its kernel stack and a saved context in its TCB.



A later event/action (such as an interrupt or code running on some other thread) may **wakeup** the sleeping thread.