Implementing Synchronization

Dr. Daniel Andresen

CIS520 – Operating Systems

Synchronization 101

Synchronization constrains the set of possible interleavings:

- Threads "agree" to stay out of each other's way and/or to coordinate their activities.
- Example: *mutual exclusion* primitives (*locks*)

  voluntary blocking or spin-waiting on entrance to critical sections

  notify blocked or spinning peers on exit from the critical section
- There are several ways to implement locks.

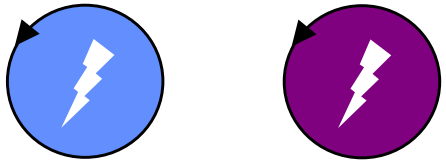
  spinning (*spinlock*) or blocking (*mutex*) or hybrid
- Correct synchronization primitives are "magic".

  requires hardware support and/or assistance from the scheduler

Using a Lock for the Counter/Sum Example

```
int counters[N];
int total;
Lock *lock;

/*
 * Increment a counter by a specified value, and keep a running sum.
 */
void
TouchCount(int tid, int value)
{
        lock->Acquire();
        counters[tid] += value;      /* critical section code is atomic...*/
        total += value;              /* …as long as the lock is held */
        lock->Release();
}
```



Implementing Spinlocks: First Cut

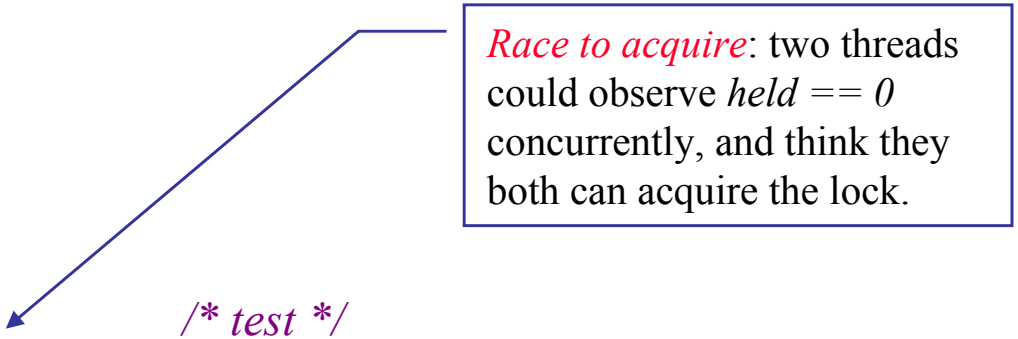
```
class Lock {
        int held;
}

void Lock::Acquire() {
        while (held);
        held = 1;
}

void Lock::Release() {
        held = 0;
}
```

"busy-wait" for lock holder to release

Spinlocks: What Went Wrong




```
void Lock::Acquire() {
        while (held);           /* test */
        held = 1;               /* set */
}

void Lock::Release() {
        held = 0;
}
```

What Are We Afraid Of?

Potential problems with the "rough" spinlock implementation:

(1) races that violate mutual exclusion

- involuntary context switch between **test** and **set**
- on a multiprocessor, race between **test** and **set** on two CPUs

(2) wasteful spinning

- lock holder calls **sleep** or **yield**
- interrupt handler acquires a busy lock
- involuntary context switch for lock holder

Which are implementation issues, and which are problems with spinlocks themselves?

The Need for 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.
- These primitives must somehow be made *atomic*.

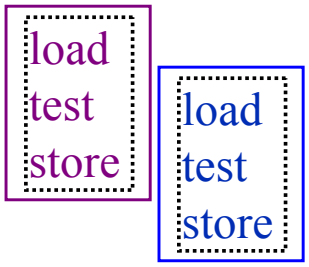
  uninterruptible

  a sequence of instructions that executes "all or nothing"
- Two solutions:

  (1) hardware support: *atomic instructions* (**test-and-set**)

  (2) scheduler control: *disable timeslicing* (**disable interrupts**)

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;
}

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
```

On Disabling Interrupts

Nachos has a primitive to *disable interrupts*, which we will use as a toehold for synchronization.

- Temporarily block notification of external events that could trigger a context switch.

  e.g., clock interrupts (ticks) or device interrupts

- In a "real" system, this is available *only to the kernel*.

  why?

- Disabling interrupts is *insufficient* on a multiprocessor.

  It is thus a dumb way to implement spinlocks.

- We will use it ONLY as a toehold to implement "proper" synchronization.

  a blunt instrument to use as a last resort

Implementing Locks: Another Try

```
class Lock {
}

void Lock::Acquire() {
        disable interrupts;
}

void Lock::Release() {
        enable interrupts;
}
```

Problems?

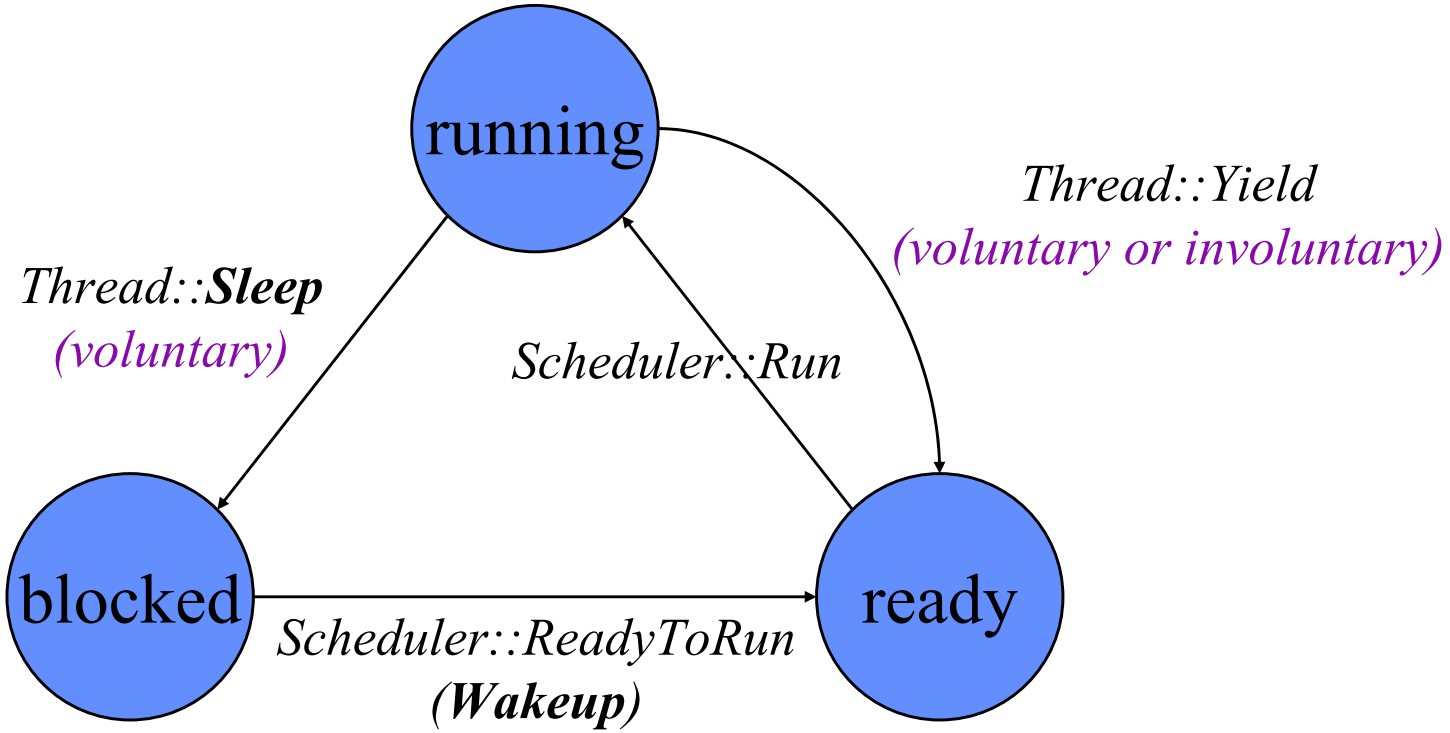
Implementing Mutexes: Rough Sketch

```
class Lock {
        int held;
        Thread* waiting;
}

void Lock::Acquire() {
        if (held) {
                waiting = currentThread;
                currentThread->Sleep();
        }
        held = 1;
}

void Lock::Release() {
        held = 0;
        if (waiting)            /* somebody's waiting: wake up */
                scheduler->ReadyToRun(waiting);
}
```

Nachos Thread States and Transitions



currentThread->Yield();

currentThread->Sleep();

Implementing Mutexes: A First Cut

```
class Lock {
        int held;
        List sleepers;
}

void Lock::Acquire() {
        while (held) {                          Why the while loop?
                sleepers.Append((void*)currentThread);
                currentThread->Sleep();
        }
        held = 1;                               Is this safe?
}

void Lock::Release() {
        held = 0;
        if (!sleepers->IsEmpty())           /* somebody's waiting: wake up */
                scheduler->ReadyToRun((Thread*)sleepers->Remove());
}
```

Mutexes: What Went Wrong

Potential *missed wakeup*: holder could *Release* before thread is on sleepers list.

Potential *corruption* of *sleepers* list in a race between two *Acquires* or an *Acquire* and a *Release*.

Potential *missed wakeup*: holder could call to wake up before we are "fully asleep".

```
void Lock::Acquire() {
        while (held) {
                sleepers.Append((void*)currentThread);
                currentThread->Sleep();
        }
        held = 1;
}

void Lock::Release() {
        held = 0;
        if (!sleepers->IsEmpty())          /* somebody's waiting: wake up */
                scheduler->ReadyToRun((Thread*)sleepers->Remove());
}
```

Race to acquire: two threads could observe *held == 0* concurrently, and think they both can acquire the lock.

The Trouble with Sleep/Wakeup

```
Thread* waiter = 0;

void await() {
        waiter = currentThread;                 /* "I'm sleeping" */
        currentThread->Sleep();                 /* sleep */
}

void awake() {
        if (waiter)
                scheduler->ReadyToRun(waiter); /* wakeup */
        waiter = (Thread*)0;
}
```

switch here for missed wakeup

any others?

A simple example of the use of *sleep/wakeup* in Nachos.

Using Sleep/Wakeup Safely

```
Thread* waiter = 0;

void await() {
        disable interrupts
        waiter = currentThread;                 /* "I'm sleeping" */
        currentThread->Sleep();                 /* sleep */
        enable interrupts
}

void awake() {
        disable interrupts
        if (waiter)                             /* wakeup */
                scheduler->ReadyToRun(waiter);
        waiter = (Thread*)0;                    /* "you're awake" */
        enable interrupts
}
```

Disabling interrupts prevents a context switch between "I'm sleeping" and "sleep".

Nachos *Thread::Sleep* *requires* disabling interrupts.

Disabling interrupts prevents a context switch between "wakeup" and "you're awake".
Will this work on a multiprocessor?

What to Know about Sleep/Wakeup

1. *Sleep/wakeup* primitives are the fundamental basis for *all* blocking synchronization.
2. All use of *sleep/wakeup* requires some additional low-level mechanism to avoid missed and double wakeups.

   disabling interrupts, and/or

   constraints on preemption, and/or *(Unix kernels use this instead of disabling interrupts)*

   spin-waiting *(on a multiprocessor)*

3. These low-level mechanisms are tricky and error-prone.
4. High-level synchronization primitives take care of the details of using *sleep/wakeup*, hiding them from the caller.

   semaphores, mutexes, condition variables