Overview

Will talk about locks, semaphores, and monitors/condition variables

For each, will talk about:

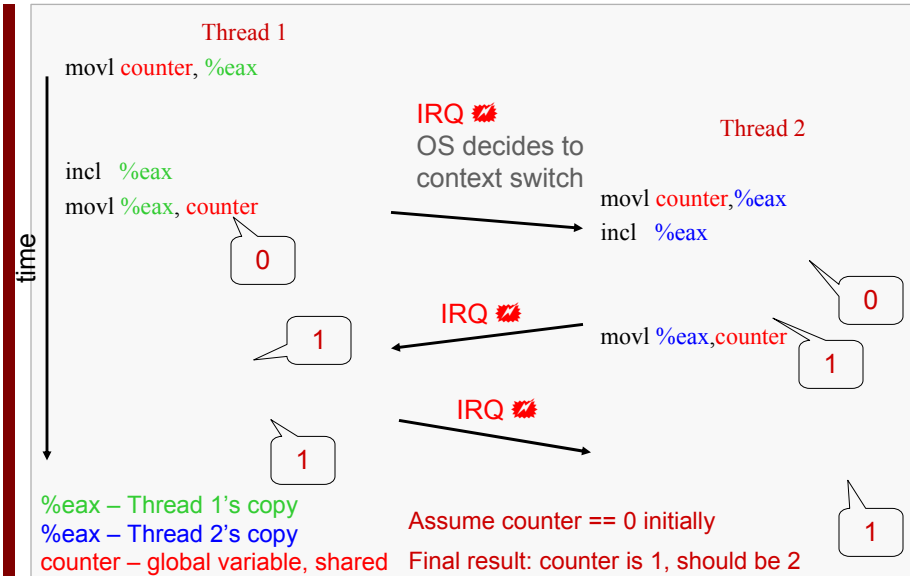
- What abstraction they represent
- How to implement them
- How and when to use them

Two major issues:

- Mutual exclusion
- Scheduling constraints

Project note: Pintos implements its locks on top of semaphores

A Race Condition



Race Conditions

Definition: *two or more threads read and write a shared variable, and final result depends on the order of the execution of those threads*

Usually timing-dependent and intermittent
- Hard to debug

Not a race condition if all execution orderings lead to same result
- Chances are high that you misjudge this

How to deal with race conditions:
- Ignore (!?)
  - Can be ok if final result does not need to be accurate
  - Never an option in CS 3204
- Don't share: duplicate or partition state
- Avoid "bad interleavings" that can lead to wrong result

Not Sharing: Duplication or Partitioning

Undisputedly best way to avoid race conditions
- Always consider it first
- Usually faster than alternative of sharing + protecting
- But duplicating has space cost; partitioning can have management cost
- Sometimes must share (B depends on A's result)

Examples:
- Each thread has its own counter (then sum counters up after join())
- Every CPU has its own ready queue
- Each thread has its own memory region from which to allocate objects

Truly ingenious solutions to concurrency involve a way to partition things people originally thought you couldn't

Aside: Thread-Local Storage

A concept that helps to avoid race conditions by giving each thread a copy of a certain piece of state

Recall:

- All local variables are already thread-local
  - But their extent is only one function invocation
- All function arguments are also thread-local
  - But must pass them along call-chain

TLS creates variables of which there's a separate value for each thread.

In PThreads/C (compiler or library-supported)

- Dynamic: pthread_create_key(), pthread_get_key(), pthread_set_key()
  - E.g. myvalue = keytable(key_a)→get(pthread_self());
- Static: using __thread storage class
  - E.g.: __thread int x;

Java: java.lang.ThreadLocal

In Pintos:
Add member to struct thread

Race Condition & Execution Order

Prevent race conditions by imposing constraints on execution order so the final result is the same regardless of actual execution order

- That is, exclude "bad" interleavings
- *Specifically*: disallow other threads to start updating shared variables while one thread is in the middle of doing so; make those updates *atomic* – threads either see old or new value, but none in between

Atomicity & Critical Sections

Atomic: indivisible

- Certain machine instructions are atomic
- But need to create larger atomic sections

Critical Section

- A synchronization technique to ensure atomic execution of a segment of code
  - Requires *entry()* and *exit()* operations

```
pthread_mutex_lock(&lock);      /* entry() */
counter++;
pthread_mutex_unlock(&lock);  /* exit() */
```


Critical Sections

Critical Section Problem also known as mutual exclusion problem

Only one thread can be inside critical section; others attempting to enter CS must wait until thread that's inside CS leaves it.

Note: a critical section does not necessarily imply that thread executes section without interruption (i.e., preemption), or even that thread completes section – just that other threads can't enter this critical section while one thread is inside it/hasn't left it

Solutions can be entirely software, or entirely hardware

- Usually combined
- Different solutions for uniprocessor vs multiprocessor scenarios

Implementing Critical Sections

Will look at:

- Disabling interrupts approach
- Semaphores
- Locks

Disabling Interrupts

All asynchronous context switches start with interrupts

- So disable interrupts to avoid them!

```
intr_level old = intr_disable();
/* modify shared data */
intr_set_level(old);
```

```
void intr_set_level(intr_level to)
{
   if (to == INTR_ON)
     intr_enable();
   else
     intr_disable();
}
```

Handling CS by avoiding context switches: Variation (1)

Variation of "disabling-interrupts" technique

- That doesn't actually disable interrupts
- If IRQ happens, ignore it

Assumes writes to "taking_interrupts" are atomic and sequential wrt reads

```
taking_interrupts = false;
/* modify shared data */
taking_interrupts = true;
```

```
intr_entry()
{
   if (!taking_interrupts)
      iret
   intr_handle();
}
```


Handling CS by avoiding context switches: Variation (2)

Code on previous slide could lose interrupts

- Remember pending interrupts and check when leaving critical section

This technique can be used with Unix signal handlers (which are like "interrupts" sent to a Unix process)

- but tricky to get right

```
taking_interrupts = false;
/* modify shared data */
if (irq_pending)
    intr_handle();
taking_interrupts = true;
```

```
intr_entry()
{
   if (!taking_interrupts) {
      irq_pending = true;
      iret
   }
   intr_handle();
}
```

Avoiding context switches: Variation (3)

Instead of setting flag, have irq handler examine PC where thread was interrupted

See Bershad '92: Fast Mutual Exclusion on Uniprocessors

```
critical_section_start:
  /* modify shared data */
critical_section_end:
```

```
intr_entry()
{
  if (PC in (critical_section_start,
             critical_end_end)) {
    iret
  }
  intr_handle();
}
```


Disabling Interrupts: Summary

(this applies to all variations)
Sledgehammer solution

Infinite loop means machine locks up

Use this to protect data structures from concurrent access by interrupt handlers

- Keep sections of code where irqs are disabled minimal (nothing else can happen until irqs are reenabled – latency penalty!)
- If you block (give up CPU) mutual exclusion with other threads is not guaranteed
  - Any function that transitively calls thread_block() may block

Want something more fine-grained

- Key insight: don't exclude *everybody* else, only those contending for the same critical section

Critical Section Problem

A solution for the CS Problem must

1) Provide mutual exclusion: at most one thread can be inside CS
2) Guarantee Progress: (no deadlock)
   - if more than one threads attempt to enter, one will succeed
   - ability to enter should not depend on activity of other threads not currently in CS
3) Bounded Waiting: (no starvation)
   - A thread attempting to enter critical section eventually will (assuming no thread spends unbounded amount of time inside CS)

A solution for CS problem should be

- Fair (make sure waiting times are balanced)
- Efficient (not waste resources)
- Simple

Locks

Thread that enters CS locks it

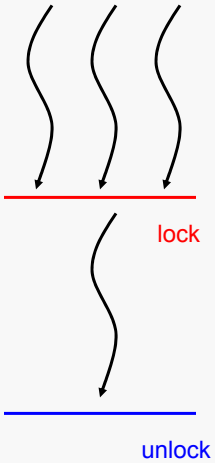
- Others can't get in and have to wait

Thread unlocks CS when leaving it

- Lets in next thread
- which one?
  - FIFO guarantees bounded waiting
  - Highest priority in Proj1

Can view Lock as an abstract data type

- Provides (at least) init, acquire, release



Implementing Locks

Locks can be implemented directly, or – among other options - on top of semaphores

- If implemented on top of semaphores, then semaphores must be implemented directly
- Will explain this layered approach first to help in understanding project code
- Issues in direct implementation of locks apply to direct implementation of semaphores as well

Semaphores

Invented by Edsger Dijkstra in 1960s

Counter S, initialized to some value, with two operations:

- P(S) or "down" or "wait" – if counter greater than zero, decrement. Else wait until greater than zero, then decrement
- V(S) or "up" or "signal" – increment counter, wake up any threads stuck in P.

Semaphores don't go negative:

- `#V + InitialValue - #P >= 0`

Note: direct access to counter value after initialization is not allowed

Counting vs Binary Semaphores

- Binary: counter can only be 0 or 1

Simple to implement, yet powerful

- Can be used for many synchronization problems

Semaphores as Locks

Semaphores can be used to build locks

- Pintos does just that

Must initialize semaphore with 1 to allow one thread to enter critical section

```
semaphore S(1); // allows initial down

lock_acquire()
{   // try to decrement, wait if 0
    sema_down(S);
}

lock_release()
{   // increment (wake up waiters if any)
    sema_up(S);
}
```

Easily generalized to allow at most N simultaneous threads: multiplex pattern (i.e., a resource can be accessed by at most N threads)

Implementing Locks Directly

NB: Same technique applies to implementing semaphores directly (as in done in Pintos)

- Will see two applications of the same technique

Different solutions exist to implement locks for uniprocessor and multiprocessors

Will talk about how to implement locks for uniprocessors first – next slides all assume uniprocessor

Implementing Locks, Take 1


```
lock_acquire(struct lock *l)
{
    while (l->state == LOCKED)
            continue;
    l->state = LOCKED;
}
```

```
lock_release(struct lock *l)
{
   l->state = UNLOCKED;
}
```

Does this work?

No – does not guarantee mutual exclusion property – more than one thread may see "state" in UNLOCKED state and break out of while loop. This implementation has itself a race condition.

Implementing Locks, Take 2


```
lock_acquire(struct lock *l)
{
    disable_preemption();
    while (l->state == LOCKED)
            continue;
    l->state = LOCKED;
    enable_preemption();
}
```

```
lock_release(struct lock *l)
{
   l->state = UNLOCKED;
}
```

Does this work?

No – does not guarantee progress property. If one thread enters the while loop, no other thread will ever be scheduled since preemption is disabled – in particular, no thread that would call lock_release will ever be scheduled.

Implementing Locks, Take 3


```
lock_acquire(struct lock *l)
{
    while (true) {
        disable_preemption();
        if (l->state == UNLOCKED) {
            l->state = LOCKED;
            enable_preemption();
            return;
        }
        enable_preemption();
    }
}
```

```
lock_release(struct lock *l)
{
    l->state = UNLOCKED;
}
```

Does this work?

Yes, this works – but is grossly inefficient. A thread that encounters the lock in the LOCKED state will busy wait until it is unlocked, needlessly using up CPU time.

Implementing Locks, Take 4


```
lock_acquire(struct lock *l)
{
    disable_preemption();
    while (l->state == LOCKED) {
        list_push_back(l->waiters,
                       &current->elem);
        thread_block(current);
    }
    l->state = LOCKED;
    enable_preemption();
}
```

```
lock_release(struct lock *l)
{
    disable_preemption();
    l->state = UNLOCKED;
    if (list_size(l->waiters) > 0)
        thread_unblock(
            list_entry(list_pop_front(l->waiters),
                       struct thread, elem));
    enable_preemption();
}
```

Correct & uses proper blocking.
Note that thread doing the unlock performs the work of unblocking the first waiting thread.