Concurrency Control

Azza Abouzied

From Last Class: Synchronization Primitives

Race Conditions

Race Conditions

Order of threads affects outcome of the computation then we have race conditions. These create non-determinism!

Code paths that access/manipulate shared data are critical sections.

If a critical section executes atomically then we prevent concurrent access to shared data at critical sections.

Thread 1	Thread 2
get hc; increment hc; store hc;	
	get hc; increment hc; store hc;

Thread 1	Thread 2
	get hc; increment hc; store hc;
get hc; increment hc; store hc;	

Sources of Concurrency

- **Interrupts**
- **User-space preemption**: The scheduler decides when to preempt you and when to execute you
- **Kernel preemption**: The kernel itself is a multi-threaded beast sharing address space and is preemptive
- **Sleep, Block**
- **SMP**: two processors can be executing the same code at exactly the same time (kernel or user)

What about mutual exclusion? Locking

Thread 1	Thread 2
try to acquire lock	try to acquire lock
Success: lock acquired	Failed: wait …
get hc	wait
increment hc	wait
store hc	wait
unlock lock	wait
…	Success: lock acquired
	…

But you just pushed the problem to this lock thing … how do you make a lock?

What if another process ignores the locks: locks are advisory and voluntary

The Spin Lock

```
int hc_busy;
int hc;

void update_hit_counter(){
    while(1){       //Spin Lock
        if(test_and_set(hc_busy, 1)){
            hc++;   //Critical Section
            test_and_set(hc_busy, 0);
            return;
        }
    }
}
```

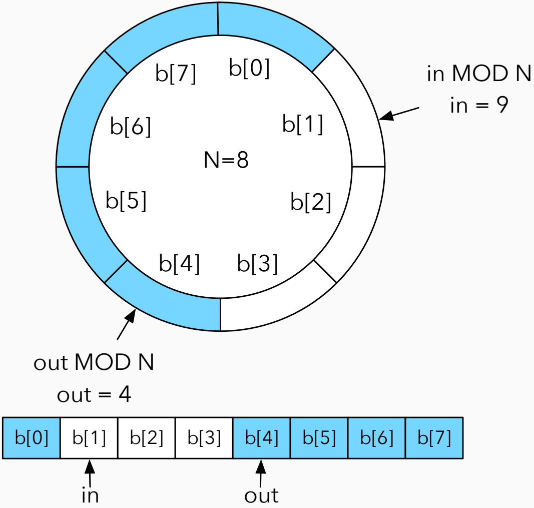
More Synchronization Primitives

Mutex — The sleeping version of a spin lock

```
acquire_lock(int* lock){
    while(1){
        if(test_and_set(lock, 1)) return;
        yield(); /* go to bed */
    }
}
```

```
release_lock(int* lock){
    test_and_set(lock, 0);
}
```

The bounded buffer



Synchronizing Producers and Consumers on the buffer: Sol 1

```
void produce(b, m){
 while(1){
  if(in - out < N){
   b[in % N] = m;
   in++;
   return;
  }
 }
}
```

```
msg consume(b){
 while(1){
  if(in > out){
   m = b[out % N]
   out++;
   return m;
  }
 }
}
```

What is wrong with sol 1?

1. Single Producer / Consumer

2. Spin Lock Solution

3. Tricky to implement: What happens if we swap increment & buffer write?

How to make this work with multiple producers?

```
void produce(b, m){
    while(1){
        acquire(write_lock);
        if(in - out < N){
            b[in % N] = m;
            in++;
            release(write_lock);
            return;
        }
    }
}
```

Will this work?

How to make this work with multiple producers?

```
void produce(b, m){
    while(1){
        acquire(write_lock);
        if(in - out < N){
            b[in % N] = m;
            in++;
            release(write_lock);
            return;
        }
        release(write_lock);
    }
}
```

Will this work?

What about multiple consumers?

Mutexes to the Rescue: Sol 2

```
void producer(){
    while(1){
        if(count == N)
            sleep();
        push(m, b);
        count++;
        if(count == 1)
            wakeup(consumer);
    }
}
```

```
void consumer(){
    while(1){
        if(count == 0)
            sleep();
        pull(m, b);
        count--;
        if(count == N - 1)
            wakeup(producer);
    }
}
```

The Nightmare Scenario

1. Buffer Empty: before consumer sleeps it is interrupted.
   if (count == 0) ... <INTERRUPT>

2. Buffer is empty, so producer puts an item, and wakes consumer up.

3. But consumer didn't really sleep, it will now go to sleep (and it will miss the wake up call) ... sleep();

4. Eventually producer fills up the buffer and they sleep in peace forever

The birth of Dijkstra's semaphore: the wake-up counter

```
void wait(Semaphore* s){
    while(1){
        acquire(s->lock);
        if(s->counter > 0){
            s->counter--;
            release(s->lock);
            return;
        }
        release(sem->lock);
        sleep(x ms);
    }
}
```

```
void signal(Semaphore* s){
    acquire(s->lock);
    s->counter++;
    release(s->lock);
}
```

Typically kernels use wait queues instead of sleep calls. Why?

Producer/Consumer with Semaphores

```
Semaphore empty = N;
Sempahore mutex = 1;
Sempahore full = 0;
void producer(){
    while(1){
        wait(empty);
        wait(mutex);
        push(m, b);
        signal(mutex);
        signal(full);
    }
}
```

```
void consumer(){
    while(1){
        wait(full);
        wait(mutex);
        pull(m, b);
        signal(mutex);
        signal(empty);
    }
}
```

The binary semaphore mutex does not have to be a semaphore!

What happens if we flip wait(mutex) and wait(full)?

Rules of thumb: Spin Lock or Mutex/Semaphore?

Spin Lock	Mutex/Semaphore
	
Short lock hold time Interrupt context locking Quick & Low overhead	Long lock hold time Process context locking Overhead of sleeping, maintaining wait queues, waking up threads can surpass lock time!

Questions?