9/1/10

Last Time

Defined: Atomic Operation + Critical Region

Mutual Exclusion

If one thread is in a Critical Region, no other thread, in the same process, is allowed to enter that Critical Region.

they must wait

4 Necessary Conditions for M.E.

1. Only 1 thread in a C.R. at a time

2. No thread should wait forever to enter a C.R.

3. No assumptions about CPU speed

4. No thread outside a C.R. can ~~to~~ block a thread in a C.R.

Example: Too Much Milk

2 UCLA football roommates

Problem: They like milk
Told: Buy 1 gallon of milk
@ a time

Solution #1

Leave a note to buy milk
Remove a note when have milk

Rule: If no milk &
you see a note
don't buy milk

Algorithm

```
1   if ( noMilk) {
2       if ( noNote ) {
3           leave a note
4           buy milk/ come home
5           remove note
        }
    }
```

Sequence of Events

A: 1, 2
B: 1, 2
A: 3,
B: 3,

Result: Too Much Milk

Solution #2

1. Use labeled notes
2. Leave your note first

	A	B
1	leave noteA	leave noteB
2	if(noNoteB)	if(noNoteA)
3	if(noMilk)	
4	buy milk;	
5	remove noteA	remove noteB

Sequence of Events

A: 1, → off to bathroom
B: 1, 2 → " " "
A: 2, 5 → off to class
B: 5 → " " "

Result: No milk

Solution #3 – One Waits

A	B

same as solution #2	leave noteB
	while (noteA);
	if (noMilk)
	    buy Milk
	remove noteB;

Sequence of Events

A: 1, →

B: 1, 2 (stuck in while)

A: 2, 5

B: 2, 3, 4, 5

This always works.

But... we have 2 problems

Problem #1: Code is different
for each student

Problem #2: Busy waiting
- waste of CPU

Goal: Solve these 2 problems,
BUT still be correct

Need: Higher-level atomic primitives
(in software)

We can only use hardware atomic operations

Hardware Atomic Operations

1. Assignment

$i = 10$;

print i;

2. Test & Set Lock (TSL)

a. Read a memory address & copies the value into a CPU register

* b. Sets the address to a non-zero value

c. Checks the original value (in register)

- if value == 0 ⇒ "thread may proceed"
- if value != 0 ⇒ "thread may not proceed"

3. Disable interrupts
We "own" the CPU until
① Restore interrupts
② Voluntarily give up CPU

Higher-Level Primitive #1 - Lock

Metaphor: Locking a door - 1 key

Once a door is locked, no one else can get in.

Only the entity that locked the ~~lock~~, is allowed to unlock the lock

2 Operations Needed

Acquire: "I locked an unlocked door"

Release: "I unlock the door I locked"

Both operations MUST be atomic

Solution #4: Use a lock
a shared resource

Lock milkLock;

Algorithm

```
    milkLock.Acquire();    ] atomic
        if (noMilk)
            buy Milk
    milkLock.Release();    ] atomic
```

application C.R.

+ it works
+ all threads have same code

Question: Busy waiting or not?

Lock Implementation

- Use interrupts
  - disable, restore

To make a lock operation atomic:

Disable interrupts
Perform the lock operation
Restore interrupts

Nachos Interrupt Class

Already exists
A global kernel variable called
'interrupt'

To disable:

```
int old= interrupt-> SetLevel
                        (IntOff);
```

To restore:

```
interrupt-> SetLevel( old);
```

Need Sleep/Wakeup mechanism

Sleep: A thread "state"
where it cannot gain
access to the CPU
- on its own
- by the CPU scheduler

Wakeup: The event, a sleeping
thread has been waiting
for, has occurred

Sleep Implementation

For a thread to gain access to the CPU

- it must be in the Ready Queue
- it must be in the Ready State

Goal #1: Remove a thread from Ready Queue

Nachos: currentThread is a Thread object pointer

Points to thread in CPU

Nachos Thread class has a Sleep() method

- removes a thread from CPU
- " " " " Ready Queue

currentThread -> Sleep();

Goal #2: Find sleeping threads

We use a separate queue to store sleeping threads

- Specific to each lock

Add a "wait" queue to the Nachos Lock class

BEFORE going to sleep, add myself to the Lock's wait queue

Wakeup Implementation

- Wakeup 1 sleeping thread
- Give this 1 thread ownership of the lock
- Put the thread in the Ready Queue in the Ready State

~~e~~. scheduler -> ReadyToRun(__);

thread pointer

Locks have 2 states
- FREE - available
- BUSY - not available

threads dir
synch.h
synch.cc

```
void Lock::Acquire(){
    //disable interrupts

    if( "I'm the lock owner"){
        // I already own it
        // Restore interrupts & return
    }
    if( "lock available" ){
        • make it BUSY
        • make myself the lock owner

    } else { // lock is busy
        • Add myself to lock waitQueue
        • Put myself to sleep ←
    }
    // restore interrupts
}
```

```
void Lock::Release(){
  // disable interrupts

  if( "this thread is not the lock owner"){
      //Print error msg, restore interrupts
                                  & return
  }

  if ( "a thread is waiting"){
     // wakeup 1 thread
        // Remove 1 thread from lock's
                             wait Q
        // Put them on Ready Q
        // Make them the lock owner
  } else {   //No thread waiting
      • Make lock available
      • Clear lock ownership
  }

  //restore interrupts
}
```