CSE150
Operating Systems
Lecture 7

Mutual Exclusion

Synchronization problem with Threads (Review)

- One thread per transaction, each running:

```
Deposit(acctId, amount) {
  acct = GetAccount(actId); /* May use disk I/O */
  acct->balance += amount;
  StoreAccount(acct);       /* Involves disk I/O */
}
```

- Unfortunately, shared state can get corrupted:

Thread 1	Thread 2
`load r1, acct->balance`	
	`load r1, acct->balance`
	`add r1, amount2`
	`store r1, acct->balance`
`add r1, amount1`	
`store r1, acct->balance`	

- Atomic Operation: an operation that always runs to completion or not at all
  – It is *indivisible:* it cannot be stopped in the middle and state cannot be modified by someone else in the middle

Too Much Milk Solution #2

- How about labeled notes?
  - Now we can leave note before checking
- Algorithm looks like this:

<u>Thread A</u>

```
leave note A;
if (noNote B) {
   if (noMilk) {
      buy Milk;
   }
}
remove note A;
```

<u>Thread B</u>

```
leave note B;
if (noNote A) {
   if (noMilk) {
      buy Milk;
   }
}
remove note B;
```

- Does this work?

Too Much Milk Solution #2.5

- Algorithm looks like this:

Thread A

```
if (noNote B) {
    leave note A;
    if (noMilk) {
        buy Milk;
    }
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {
    if (noMilk) {
        buy Milk;
    }
}
remover note B;
```

- Does this work?

Too Much Milk Solution #3 (Review)

- Here is a possible two-note solution:

Thread A

```
leave note A;
while (note B) {\\X
   do nothing;
}
if (noMilk) {
   buy milk;
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {\\Y
   if (noMilk) {
      buy milk;
   }
}
remove note B;
```

- Does this work? Yes. Both can guarantee that:
  – It is safe to buy, or
  – Other will buy, ok to quit
- At X:
  – if no note B, safe for A to buy,
  – otherwise wait to find out what will happen
- At Y:
  – if no note A, safe for B to buy
  – Otherwise, A is either buying or waiting for B to quit

Solution #3 discussion (Review)

- Our solution protects a single "Critical-Section" piece of code for each thread:

```
if (noMilk) {
   buy milk;
}
```

- Solution #3 works, but it's really unsatisfactory
  - Really complex – even for this simple an example
    - » Hard to convince yourself that this really works
  - A's code is different from B's – what if lots of threads?
    - » Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    - » This is called "busy-waiting"
- There's a better way

Too Much Milk: Solution #4

- Suppose we have some sort of implementation of a lock (more in a moment).
  - `Lock.Acquire()` – wait until lock is free, then grab
  - `Lock.Release()` – Unlock, waking up anyone waiting
  - These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock

- Then, our milk problem is easy:

```
milklock.Acquire();
if (noMilk)
   buy milk;
milklock.Release();
```

- Once again, section of code between `Acquire()` and `Release()` called a "Critical Section"

Why a thread joins another thread, but not does the job by its own?

- Multithreading
  - Share resources among multiple tasks
  - Modularity
  - Parallel on multiple CPU cores

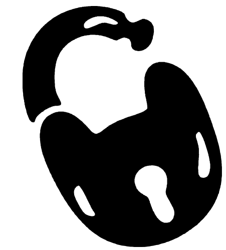
- One thread may wait for the result of another thread
- One thread may wait for the result of two other threads that run in parallel.

Today

- Locks
- Higher-level Synchronization Abstractions
  - Semaphores
- Programming paradigms for concurrent programs

How to implement Locks?

- Lock: prevents someone from accessing/doing something
  - Lock before entering critical section (e.g., before accessing shared data)
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - » Important idea: all synchronization involves waiting
    - » Should *sleep* if waiting for a long time



- Hardware Lock instruction
  - Is this a good idea?
  - What about putting a task to sleep?
    - » How do you handle the interface between the hardware and scheduler?
  - Complexity?
    - » Each feature makes hardware more complex and slow

Naïve use of Interrupt Enable/Disable

- How can we build multi-instruction atomic operations?
  - Recall: dispatcher gets control in two ways.
    - » Internal: Thread does something to relinquish the CPU
    - » External: Interrupts cause dispatcher to take CPU
  - On a uniprocessor, can avoid context-switching by:
    - » Avoiding internal events
    - » Preventing external events by disabling interrupts
- Consequently, naïve implementation of locks:

```
LockAcquire { disable ints; }
LockRelease { enable ints; }
```

Naïve use of Interrupt Enable/Disable

- Can't let user do this! Consider following:

```
LockAcquire();
While(TRUE) {;}
```

- Real-Time system—no guarantees on timing!
  - Critical Sections might be arbitrarily long
- What happens with I/O or other important events?
  - "Reactor about to meltdown. Help?"

Better Implementation of Locks by Disabling Interrupts

- Key idea: maintain a lock variable (in memory) and impose mutual exclusion only during operations (access/update) on that variable

```
int value = FREE;
```

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue
and Go to sleep();
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

```
Release() {
   disable interrupts;
   if (anyone on wait queue) {
      Take thread off wait queue
      Put at front of ready queue
   } else {
      value = FREE;
   }
   enable interrupts;
}
```

New Lock Implementation: Discussion

- Disable interrupts: avoid interrupting between checking and setting lock value
  - Otherwise two threads could think that they both have lock

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue
      and Go to sleep();
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

Critical Section

- Note: unlike previous solution, critical section very short
  - User of lock can take as long as they like in their own critical section
  - Critical interrupts taken in time

Interrupt re-enable in going to sleep

- What about re-enabling `ints` when going to sleep?

Enable Position

Enable Position

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue
      and Go to sleep();
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

- Before putting thread on the wait queue?
  - Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
  - Misses wakeup
- Want to put it after `sleep()`. But, how?

How to Re-enable After `Sleep()`?

- Since `ints` are disabled when you call sleep:
  - Responsibility of the next thread to re-enable `ints`
  - When the sleeping thread wakes up, returns to acquire and re-enables interrupts

<u>Thread A</u>

```
.
.
disable ints
sleep
```

context switch

<u>Thread B</u>

```
yield return
enable ints
.
.
.
disable ints
yield
```

context switch

```
sleep return
enable ints
.
.
```

Interrupt disable and enable across context switches

- An important point about structuring code:
  - In Nachos code you will see lots of comments about assumptions made concerning when interrupts disabled
  - This is an example of where modifications to and assumptions about program state can't be localized within a small body of code
  - In these cases it is possible for your program to eventually "acquire" bugs as people modify code
- Other cases where this will be a concern?
  - What about exceptions that occur after lock is acquired? Who releases the lock?

```
mylock.acquire();
a = b / 0;
mylock.release()
```

Atomic Read-Modify-Write instructions

- Problems with previous solution:
  - Can’t give lock implementation to users
  - Doesn’t work well on multiprocessor
    - » Disabling interrupts on all processors requires messages and would be very time consuming
- Alternative: “richer” atomic instruction sequences
  - These instructions read a value from memory and write a new value atomically
  - Hardware is responsible for implementing this correctly
    - » on both uniprocessors (moderately hard, x86: yes, MIPS: partially)
    - » and multiprocessors (requires help from cache coherence protocol)
  - Unlike disabling interrupts, can be used on both uniprocessors and multiprocessors

Examples of Read-Modify-Write

- ```
  test&set (&address) {     /* most architectures */
      result = M[address];
      M[address] = 1;
      return result;
  }
  ```

- ```
  swap (&address, register) { /* x86 */
      temp = M[address];
      M[address] = register;
      register = temp;
  }
  ```

- ```
  compare&swap (&address, reg1, reg2) { /* 68000 */
      if (reg1 == M[address]) {
          M[address] = reg2;
          return success;
      } else {
          return failure;
      }
  }
  ```
  ```

Implementing Locks with test&set

- Simple solution:

```
int value = 0; // Free
Acquire() {
   while (test&set(value)); /
}
Release() {
   value = 0;
}
```

```
test&set (&address) {
  result = M[address];
  M[address] = 1;
  return result;
}
```

- Simple explanation:
  - If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits
  - If lock is busy, test&set reads 1 and sets value=1 (no change). It returns 1, so while loop continues
  - When we set value = 0, someone else can get lock

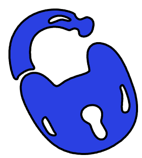
Problem: Busy-Waiting for Lock

- Positives for this solution
  - Machine can receive interrupts
  - User code can use this lock
  - Works on a multiprocessor
- Negatives
  - Inefficient: busy-waiting thread will consume cycles waiting
  - Waiting thread may take cycles away from thread holding lock (no one wins!)
  - Priority Inversion: If busy-waiting thread has higher priority than thread holding lock $\Rightarrow$ no progress!

Better Locks using test&set

- Can we build test&set locks without busy-waiting?
  - Can't entirely, but can minimize!
  - Idea: only busy-wait to atomically check lock value

```
int guard = 0;
int value = FREE;
```



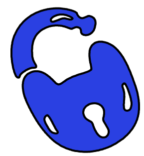
```
Acquire() {
   // Short busy-wait time
   while (test&set(guard));
   if (value == BUSY) {
      put thread on wait queue;
      go to sleep() & guard = 0;
   } else {
      value = BUSY;
      guard = 0;
   }
}
```

```
Release() {
   // Short busy-wait time
   while (test&set(guard));
   if anyone on wait queue {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   guard = 0;
```

- Note: sleep has to be sure to reset the guard variable
  - Why can't we do it just before or just after the sleep?

Locks using test&set vs. Interrupts

- Compare to “disable interrupt” solution



```
int value = FREE;

Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}

Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   enable interrupts;
}
```

- Basically replace
  - `disable interrupts` → `while (test&set(guard));`
  - `enable interrupts` → `guard = 0;`

Recap: Locks

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Acquire() {
  disable interrupts;
}
```

```
Release() {
  enable interrupts;
}
```

If one thread in critical section, no other activity (including OS) can run!

```
int value = 0;
Acquire() {
  // Short busy-wait time
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
    enable interrupts;
  }
}
```

```
Release() {
  // Short busy-wait time
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Recap: Locks

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
int value = 0;
Acquire() {
  while(test&set(value));
}
```

```
Release() {
  value = 0;
}
```

Threads waiting to enter critical section busy-wait

```
int guard = 0;
int value = 0;
Acquire() {
  // Short busy-wait time
  while(test&set(guard));
  if (value == 1) {
    put thread on wait-queue;
    go to sleep()& guard = 0;
  } else {
    value = 1;
    guard = 0;
  }
}
```

```
Release() {
  // Short busy-wait time
  while (test&set(guard));
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  guard = 0;
}
```

Summary

- Talked about hardware atomicity primitives:
  - Disabling of Interrupts, test&set, swap, comp&swap, load-linked/store conditional
- Showed several constructions of Locks
  - Must be very careful not to waste/tie up machine resources
    - » Shouldn't disable interrupts for long
    - » Shouldn't spin wait for long
  - Key idea: Separate lock variable, use hardware mechanisms to protect modifications of that variable