CS162
Operating Systems and Systems Programming
Lecture 4

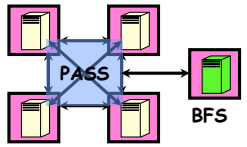
Synchronization, Atomic operations, Locks

January 30, 2012

Anthony D. Joseph and Ion Stoica
http://inst.eecs.berkeley.edu/~cs162

Space Shuttle Example

- Original Space Shuttle launch aborted 20 minutes before scheduled launch
- Shuttle has five computers:
  - Four run the "Primary Avionics Software System" (PASS)
    - » Asynchronous and real-time
    - » Runs all of the control systems
    - » Results synchronized and compared 440 times per second
  - The Fifth computer is the "Backup Flight System" (BFS)
    - » Stays synchronized in case it is needed
    - » Written by completely different team than PASS



- Countdown aborted because BFS disagreed with PASS
  - A 1/67 chance that PASS was out of sync one cycle
  - Bug due to modifications in initialization code of PASS
    - » A delayed init request placed into timer queue
    - » As a result, timer queue not empty at expected time to force use of hardware clock
  - Bug not found during extensive simulation

Concurrency

- Multiple computations (threads) executing in parallel to
  - share resources, and/or
  - share data
- **Share resources:** high utilization
- **Share data:** enable cooperation between apps, e.g.,
  - Browser sharing data with OS to send/receive packets
  - Web server: thread master sharing work & results with thread pool (see previous lecture)
  - Powerpoint sharing data with Excel and Word

Challenges

- Applications/programmers would like a system to behave as they were the only **one** using it (e.g., VM abstraction)
- Performance **isolation** and **predictability**
- Outputs should be **consistent** with application semantics
  - E.g., depositing $100 and then another $100 to your bank account should always increase your balance by $200

Sharing Granularity

- Fine grain sharing:
  - ⇑ increase concurrency → better performance
  - ⇓ more complex
- Coarse grain sharing:
  - ⇑ Simpler to implement
  - ⇓ Lower performance
- Examples:
  - Sharing CPU for 10ms vs. 1min
  - Sharing a database at the row vs. table granularity
    - A single query can access a row/table at a time
  - Allow one person vs. multiple persons in a supermarket!

Goals for Today

- Synchronization
- Hardware Support for Synchronization

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated by Kubiatowicz.

Motivation: "Too much milk"

- Great thing about OS's – analogy between problems in OS and problems in real life
  - Help you understand real life problems better
  - But, computers are much stupider than people
- Example: People need to coordinate:

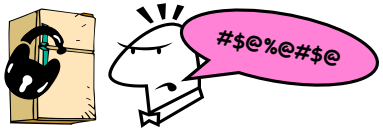
Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Definitions

- Synchronization: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic
  - We'll show that is hard to build anything useful with only reads and writes
- Mutual Exclusion: ensuring that only one thread does a particular thing at a time
  - One thread *excludes* the other while doing its task
- Critical Section: piece of code that only one thread can execute at once
  - Critical section and mutual exclusion are two ways of describing the same thing
  - Critical section defines sharing granularity

More Definitions

- Lock: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - » Important idea: all synchronization involves waiting
- Example: fix the milk problem by putting a lock on refrigerator
  - Lock it and take key if you are going to go buy milk
  - Fixes too much (coarse granularity): roommate angry if only wants orange juice



  - Of Course – We don't know how to make a lock yet

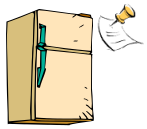
Too Much Milk: Correctness Properties

- Need to be careful about correctness of concurrent programs, since non-deterministic
  - Always write down **desired** behavior first
  - Impulse is to start coding first, then when it doesn't work, pull hair out
  - Instead, think first, then code
- What are the correctness properties for the "Too much milk" problem?
  - Never more than one person buys
  - Someone buys if needed
- Restrict ourselves to use only atomic load and store operations as building blocks

Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of "lock")
  - Remove note after buying (kind of "unlock")
  - Don't buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noMilk) {
   if (noNote) {
      leave Note;
      buy milk;
      remove note;
   }
}
```



- Result?

Too Much Milk: Solution #1

- Still too much milk but only occasionally!

```
Thread A                          Thread B
if (noMilk)
  if (noNote) {
                                  if (noMilk)
                                    if (noNote) {
    leave Note;
    buy milk;
    remove note;
  }
}
                                    leave Note;
                                    buy milk;
                                    …
```

- Thread can get context switched after checking milk and note but before buying milk!
- Solution makes problem worse since fails intermittently
  - Makes it really hard to debug…
  - Must work despite what the thread dispatcher does!

Too Much Milk: Solution #1½

- Clearly the Note is not quite blocking enough
  - Let's try to fix this by placing note first
- Another try at previous solution:

```
leave Note;
if (noMilk) {
   if (noNote) {
      buy milk;
   }
}
remove note;
```

- What happens here?
  - Well, with human, probably nothing bad
  - With computer: no one ever buys milk

Too Much Milk Solution #2

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

Thread A

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

Thread B

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

- Does this work?

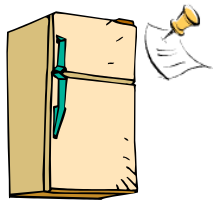
Too Much Milk Solution #2

- Possible for neither thread to buy milk!

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

- Really insidious:
  - Unlikely that this would happen, but will at worse possible time

Too Much Milk Solution #2: problem!



- *I'm* not getting milk, *You're* getting milk
- This kind of lockup is called "starvation!"

Review: Too Much Milk Solution #3

- 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

Review: Solution #3 discussion

- 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
  - Have hardware provide better (higher-level) primitives than atomic load and store
  - Build even higher-level programming abstractions on this new hardware support

5min Break

High-Level Picture

- The abstraction of threads is good:
  - Maintains sequential execution model
  - Allows simple parallelism to overlap I/O and computation
- Unfortunately, still too complicated to access state shared between threads
  - Consider "too much milk" example
  - Implementing a concurrent program with only loads and stores would be tricky and error-prone
- We'll implement higher-level operations on top of atomic operations provided by hardware
  - Develop a "synchronization toolbox"
  - Explore some common programming paradigms

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, 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"

How to Implement Lock?

- Lock: prevents someone from accessing 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 long time

- Hardware lock instructions
  - Is this a good idea?
  - What about putting a task to sleep?
    - » How do handle interface between hardware and scheduler?
  - Complexity?
    - » Each feature makes hardware more complex and slower

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 (although virtual memory tricky)
    - » Preventing external events by disabling interrupts
- Consequently, naïve Implementation of locks:

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


Naïve use of Interrupt Enable/Disable: Problems

- 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 and impose mutual exclusion only during operations on that variable

```
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;
}
```


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;
    Go to sleep();
    // Enable interrupts?
  } 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?

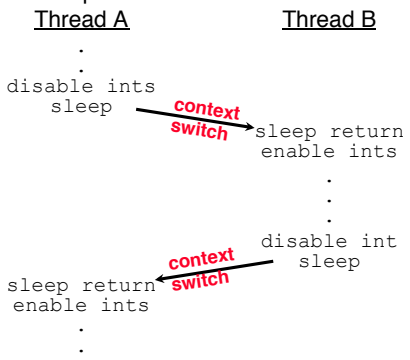
Enable Position
Enable Position
Enable Position

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

- Before putting thread on the wait queue?
  - Release can check the queue and not wake up thread
- After putting the 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 and still holds lock (deadlock!)
- 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



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: 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 (not too hard)
    - » 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!
  - Priority Inversion: If busy-waiting thread has higher priority than thread holding lock ⇒ no progress!
- Priority Inversion problem with original Martian rover
- For semaphores and monitors (see next lecture), waiting thread may wait for an arbitrary length of time!
  - Even if OK for locks, definitely not ok for other primitives
  - Homework/exam solutions should not have busy-waiting!

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;`

Summary

- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives
- Talked about hardware atomicity primitives:
  - Disabling of Interrupts, test&set
- 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