Synchronization Part 1

CSE 410, Spring 2004

Computer Systems

http://www.cs.washington.edu/education/courses/410/04sp/

Readings and References

- Reading
  » Chapter 7, *Operating System Concepts*, Silberschatz, Galvin, and Gagne. Read the following sections: 7.1, 7.2 (skim subsections), 7.3
- Other References
  » Chapter 6, *Multithreaded Programming with Pthreads*, First edition, Bil Lewis and Daniel J. Berg, Sun Microsystems Press
  » Sections 5.8.3, Atomicity and Atomic Changes, 5.8.4, Critical Regions with Interrupts Enabled, *See MIPS Run*, Dominic Sweetman

Too Much Milk

	You	**Your Roommate**
3:00	Look in fridge; no milk	
3:05	Leave for store	
3:10	Arrive at store	Look in fridge; no 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
		Oh no, Mr. Bill, too much milk!

Modeling the Problem

- Model you and your roommate as threads
- "Looking in the fridge" and "putting away milk" are reading/writing a variable

YOU:

```
// look in fridge
if( milkAmount == 0 ) {
  // buy milk
  milkAmount++;
}
```

YOUR ROOMMATE:

```
// look in fridge
if( milkAmount == 0 ) {
  // buy milk
  milkAmount++;
}
```

Correctness Properties

- Decomposed into safety and liveness
  » safety
    - the program never does anything bad

  » liveness
    - the program eventually does something good

- Although easy to state, these properties are not always easy to meet

Synchronization Definitions

- Synchronization
  » coordinated access by more than one thread to shared state variables
- Mutual Exclusion
  » only one thread does a particular thing at a time. One thread doing it excludes all others.
- Critical Section
  » only one thread executes in a critical section at once

Locks

- A lock provides mutual exclusion
  » Only one thread can hold the lock at a time
  » A lock is also called a mutex (for mutual exclusion)
- Thread must *acquire the lock* before entering a critical section of code
- Thread *releases the lock* after it leaves the critical section

Too Much Milk: A Solution

YOU:

```
MilkLock->Acquire();
if( milkAmount == 0 ){
    // buy milk
    milkAmount++;
  }
}
MilkLock->Release();
```

YOUR ROOMMATE:

```
MilkLock->Acquire();
   (delay)
if( milkAmount == 0 ){
    // buy milk
    milkAmount++;
  }
}
MilkLock->Release();
```

Lock Implementation Issue

- A context switch can happen *at any time*
  - » very simple acquire/release functions don’t work
  - » in this case, both threads think they set lockInUse

```
Lock::Release() {
  lockInUse = false;
}
Lock::Acquire() {
  while( lockInUse ) {}
  lockInUse = true;
}
```

```
Lock::Acquire() {
  while( lockInUse ) {}
  lockInUse = true;
}
```


Disable interrupts during critical section

- disable interrupts to prevent a context switch
  - » simple but imperfect solution

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

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

- Kernel can’t get control when interrupts disabled
- Critical sections may be long
  - » turning off interrupts for a long time is very bad
- Turning off interrupts is difficult and costly in multiprocessor systems

Disable Interrupts with flag

Only disable interrupts when updating a lock flag

```
initialize value = FREE;

Lock::Acquire() {
  disable interrupts;
  while(value != FREE){
    enable interrupts;
    disable interrupts;
  }
  value = BUSY;
  enable interrupts
}
```

```
Lock::Release() {
  disable interrupts;
  value = FREE;
  enable interrupts;
}
```


Atomic Operations

- An *atomic operation* is an operation that cannot be interrupted
- On a multiprocessor disabling interrupts doesn’t work well
- Modern processors provide **atomic read-modify-write** instruction or equivalent
- These instructions allow locks to be implemented on a multiprocessor

Examples of Atomic Instructions

- **Test and set** (many architectures)
  - » sets a memory location to 1 and returns the previous value
  - » if result is 1, lock was already taken, keep trying
  - » if result is 0, you are the one who set it so you've got the lock
- **Exchange** (x86)
  - » swaps value between register and memory
- **Compare & swap** (68000)

```
read location value
if location value equals comparison value
    store update value, set flag true
else
    set flag false
```


Quasi-atomic for load/store ISA

- **Remember our MIPS pipeline**
  - » only one memory stage per instruction
  - » thus, can't do atomic "read, modify, write" directly
- **Load linked and store conditional**
  - » read value in one instruction (LL—load linked) and remember where the value came from
  - » do some operation on the value
  - » when store occurs, check if value has been modified in the meantime (SC—store conditional)
  - » if not modified, store new value and return "success"
  - » if modified, return "failure"

Locks with Test and Set

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

Lock::Acquire() {
  while(TestAndSet(value)) {}
}
```

This works, but take a careful look at the while loop ... when does it exit?

Busy Waiting

- CPU cycles are consumed while the thread is waiting for value to become 0
- This is very inefficient
- Big problem if the thread that is waiting has a higher priority than the thread that holds the lock

Locks with Minimal Busy Waiting

- Use a queue for threads waiting on the lock
- A guard variable provides mutual exclusion

```
Lock::Acquire() {
  while(TestAndSet(guard)){}
  if( value != FREE ) {
    Put self on wait queue;
    guard = 0 and switch();
  } else {
    value = BUSY;
    guard = 0;
  }
}
```

```
Lock::Release() {
  while(TestAndSet(guard){}
  if(anyone on wait queue){
    move thread from wait
      queue to ready queue;
  } else {
    value = FREE;
  }
  guard = 0;
}
```


Synchronization Summary

- Threads often work independently
- But sometimes threads need to access shared data
- Access to shared data must be mutually exclusive to ensure **safety** and **liveness**
- **Locks** are a good way to provide *mutual exclusion*

