CS162
Operating Systems and Systems Programming
Lecture 7

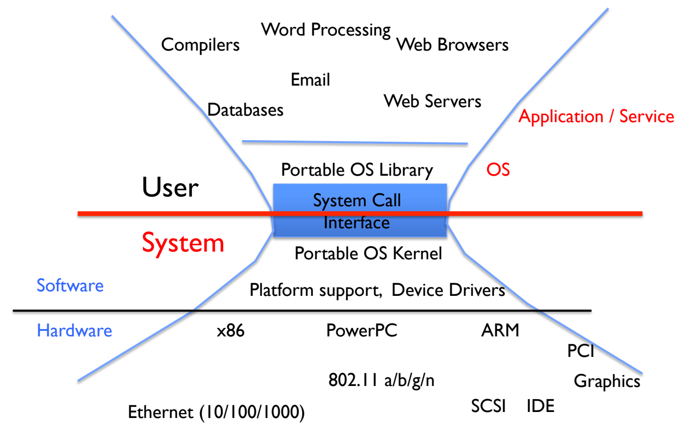
Concurrency (Continued), Synchronization

September 18th, 2017

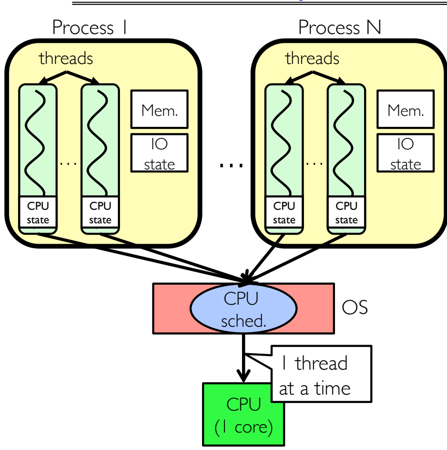
Ion Stoica

http://cs162.eecs.Berkeley.edu

Operating System as Design

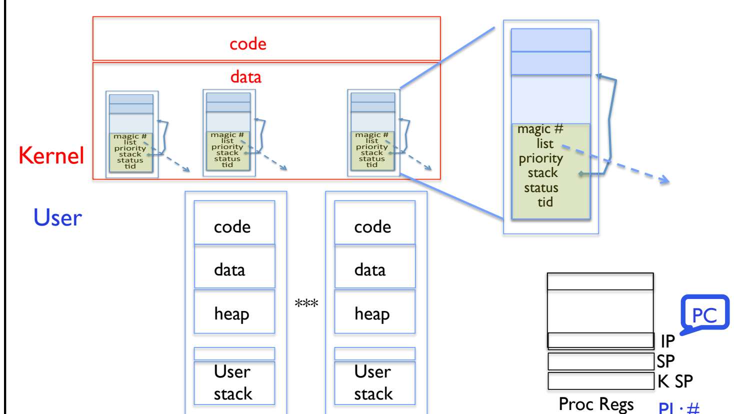


Conceptual Framework



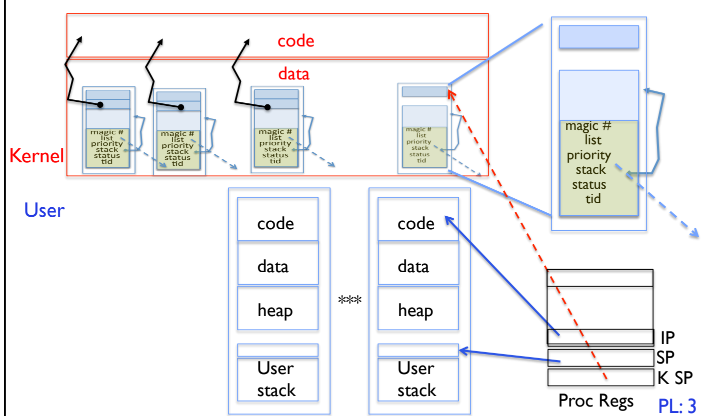
- Physical Addresses Shared
  - So: Processes and Address Translation
- Single CPU Must Be Shared
  - So: Threads
- Processes Aren't Trusted
  - So: Kernel/Userspace Split
- Threads Might Not Cooperate
  - So: Use timer interrupts to context switch ("preemption")

Recall: MT Kernel 1T Process ala Pintos/x86



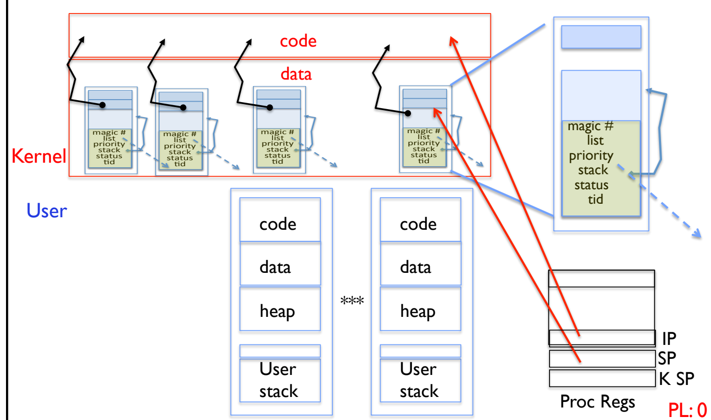
- Each user process/thread associated with a kernel thread, described by a 4KB page object containing TCB and kernel stack for the kernel thread

In User thread, w/ Kernel thread waiting



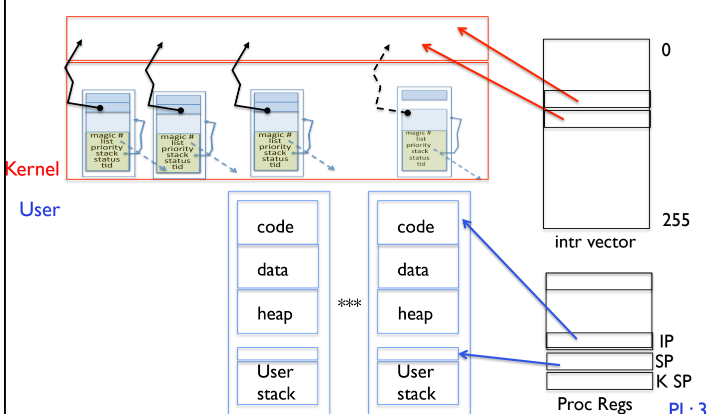
- x86 CPU holds interrupt SP in register
- During user thread execution, associated kernel thread is "standing by"

User → Kernel



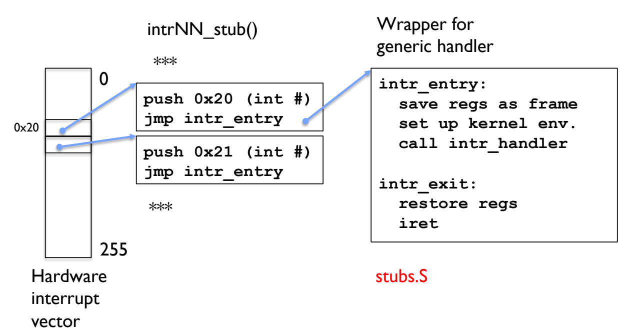
- Mechanism to resume k-thread goes through interrupt vector

User → Kernel via interrupt vector

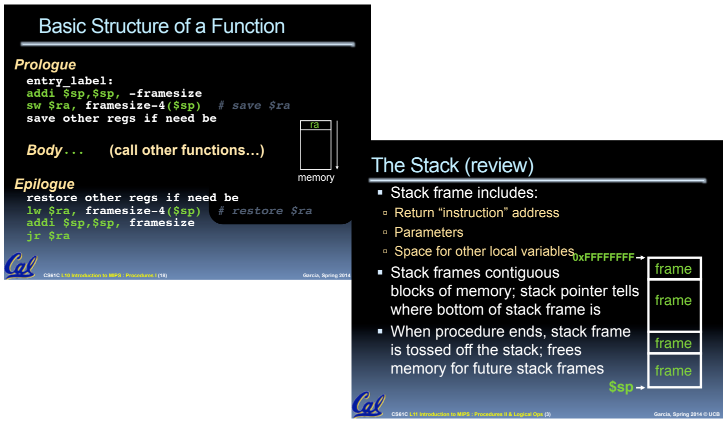


- Interrupt transfers control through the Interrupt Vector (IDT in x86)
- iret restores user stack and PL

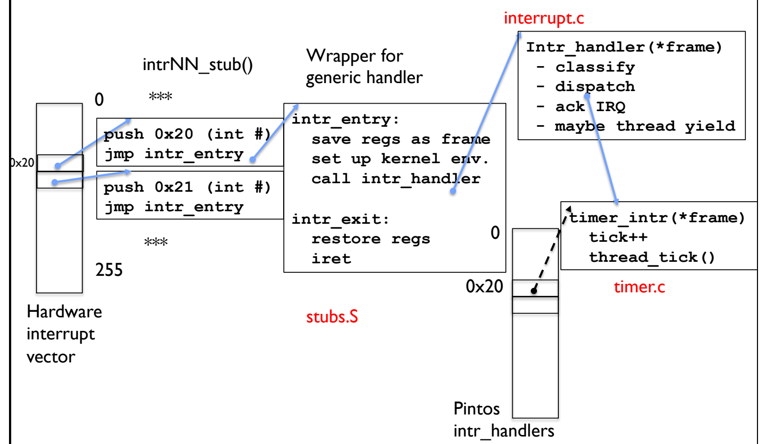
Pintos Interrupt Processing



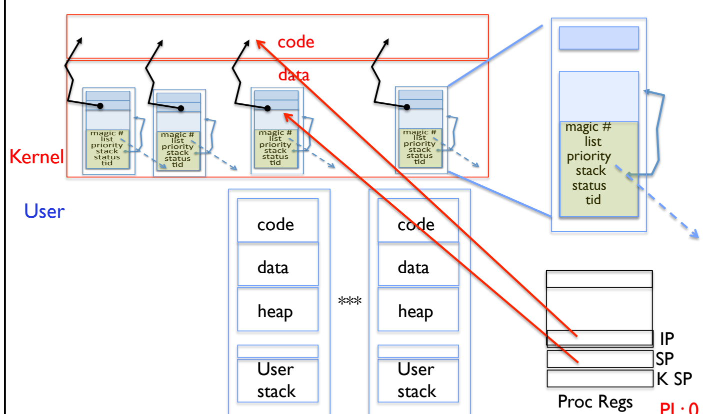
Recall: cs61C THE STACK FRAME



Pintos Interrupt Processing



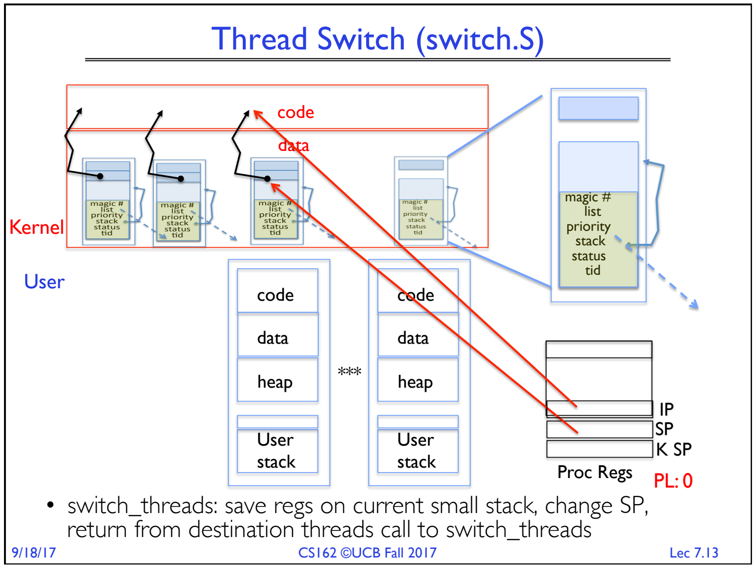
In Kernel thread

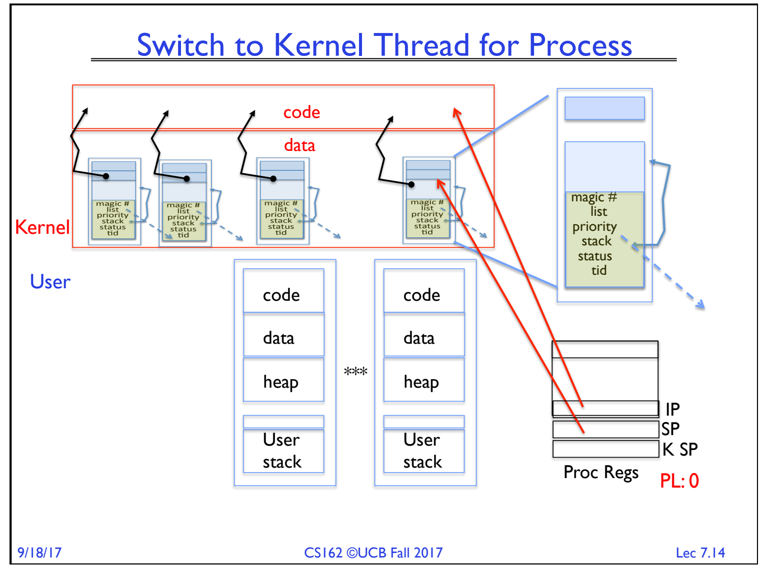


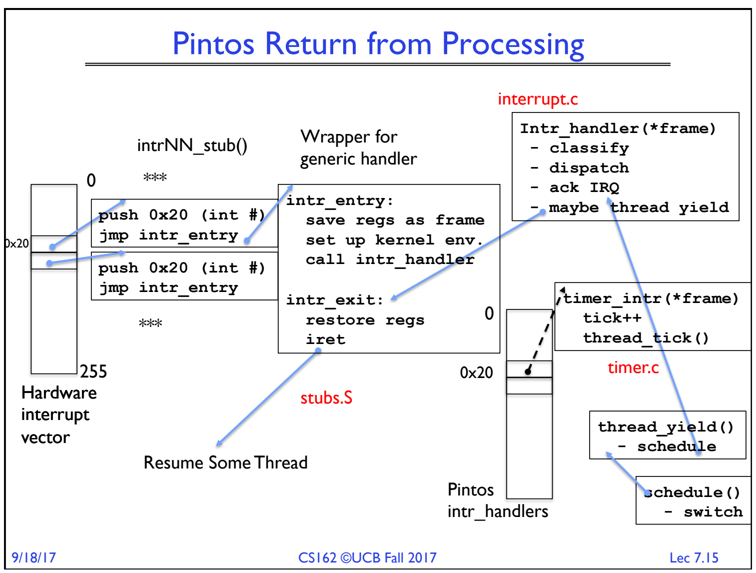
- Kernel threads execute with small stack in thread structure
- Scheduler selects among ready kernel and user threads

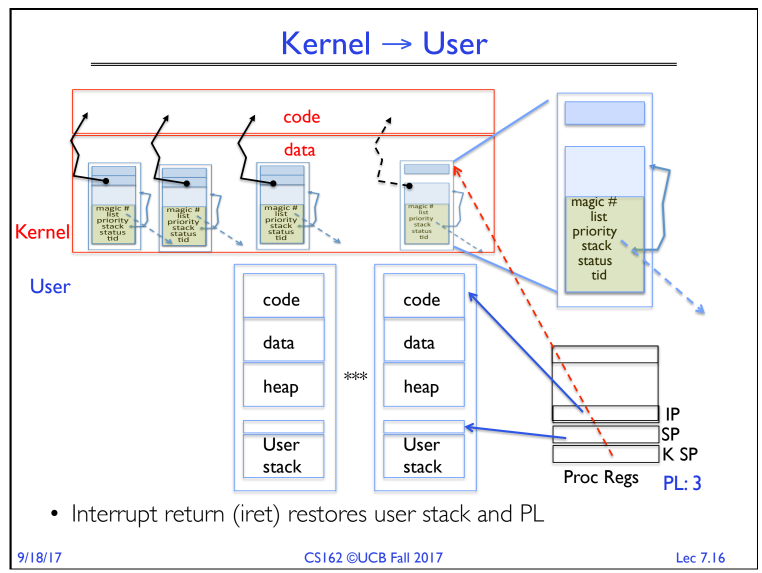
Timer may trigger thread switch

- thread_tick
  - Updates thread counters
  - If quanta exhausted, sets yield flag
- thread_yield
  - On path to rtn from interrupt
  - Sets current thread back to READY
  - Pushes it back on ready_list
  - Calls schedule to select next thread to run upon iret
- Schedule
  - Selects next thread to run
  - Calls switch_threads to change regs to point to stack for thread to resume
  - Sets its status to RUNNING
  - If user thread, activates the process
  - Returns back to intr_handler


Thread Switch (switch.S)
code
data
Kernel
User
magic #
list
priority
stack
status
tid
code
data
heap
User
stack
***
IP
SP
K SP
Proc Regs
PL: 0
• switch_threads: save regs on current small stack, change SP, return from destination threads call to switch_threads
9/18/17
CS162 ©UCB Fall 2017
Lec 7.13


Switch to Kernel Thread for Process
code
data
Kernel
User
magic #
list
priority
stack
status
tid
code
data
heap
User
stack
***
IP
SP
K SP
Proc Regs
PL: 0
9/18/17
CS162 ©UCB Fall 2017
Lec 7.14

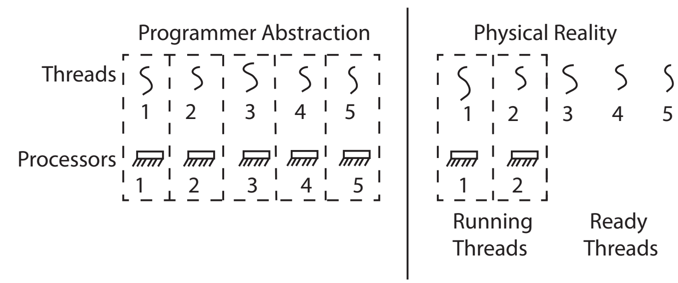

Pintos Return from Processing
interrupt.c
intrNN_stub()
Wrapper for
generic handler
0
***
0x20
push 0x20 (int #)
jmp intr_entry
push 0x20 (int #)
jmp intr_entry
***
255
Hardware
interrupt
vector
intr_entry:
save regs as frame
set up kernel env.
call intr_handler
intr_exit:
restore regs
iret
stubs.S
Resume Some Thread
Intr_handler(*frame)
- classify
- dispatch
- ack IRQ
- maybe thread yield
timer_intr(*frame)
tick++
thread_tick()
timer.c
0
0x20
thread_yield()
- schedule
schedule()
- switch
Pintos
intr_handlers
9/18/17
CS162 ©UCB Fall 2017
Lec 7.15


Kernel → User
code
data
Kernel
User
magic #
list
priority
stack
status
tid
code
data
heap
User
stack
***
IP
SP
K SP
Proc Regs
PL: 3
• Interrupt return (iret) restores user stack and PL
9/18/17
CS162 ©UCB Fall 2017
Lec 7.16

Rest of Today's Lecture

- The Concurrency Problem
- Synchronization Operations
- Higher-level Synchronization Abstractions
  - Semaphores, monitors, and condition variables

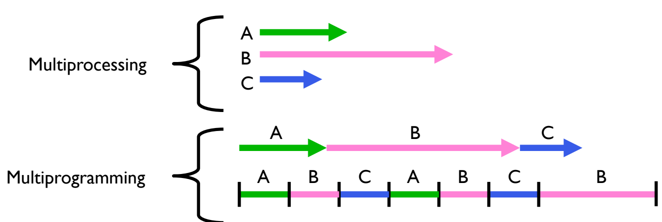
Recall: Thread Abstraction



- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule

Multiprocessing vs Multiprogramming

- Remember Definitions:
  - Multiprocessing ≡ Multiple CPUs or cores or hyperthreads (HW per-instruction interleaving)
  - Multiprogramming ≡ Multiple Jobs or Processes
  - Multithreading ≡ Multiple threads per Process
- What does it mean to run two threads "concurrently"?
  - Scheduler is free to run threads in any order and interleaving: FIFO, Random, …



Correctness for systems with concurrent threads

- If dispatcher can schedule threads in any way, programs must work under all circumstances
  - Can you test for this?
  - How can you know if your program works?
- Independent Threads:
  - No state shared with other threads
  - Deterministic ⇒ Input state determines results
  - Reproducible ⇒ Can recreate Starting Conditions, I/O
  - Scheduling order doesn't matter (if `switch()` works!!!)
- Cooperating Threads:
  - Shared State between multiple threads
  - Non-deterministic
  - Non-reproducible
- Non-deterministic and Non-reproducible means that bugs can be intermittent
  - Sometimes called "Heisenbugs"

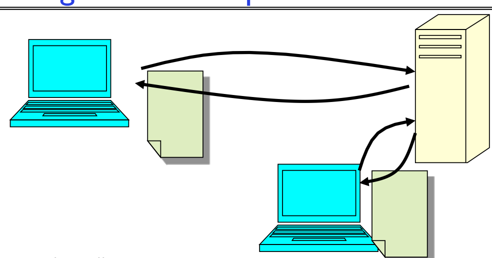
Interactions Complicate Debugging

- Is any program truly independent?
  - Every process shares the file system, OS resources, network, etc.
  - Extreme example: buggy device driver causes thread A to crash "independent thread" B
- Non-deterministic errors are really difficult to find
  - Example: Memory layout of kernel+user programs
    - » depends on scheduling, which depends on timer/other things
    - » Original UNIX had a bunch of non-deterministic errors
  - Example: Something which does interesting I/O
    - » User typing of letters used to help generate secure keys

Why allow cooperating threads?

- Advantage 1: Share resources
  - One computer, many users
  - One bank balance, many ATMs
    - » What if ATMs were only updated at night?
  - Embedded systems (robot control: coordinate arm & hand)
- Advantage 2: Speedup
  - Overlap I/O and computation
    - » Many different file systems do read-ahead
  - Multiprocessors – chop up program into parallel pieces
- Advantage 3: Modularity
  - More important than you might think
  - Chop large problem up into simpler pieces
    - » To compile, for instance, `gcc` calls `cpp | cc1 | cc2 | as | ld`
    - » Makes system easier to extend

High-level Example: Web Server



- Server must handle many requests
- Non-cooperating version:

```
serverLoop() {
    connection = AcceptCon();
    ProcessFork(ServiceWebPage(), connection);
}
```

- What are some disadvantages of this technique?

Threaded Web Server

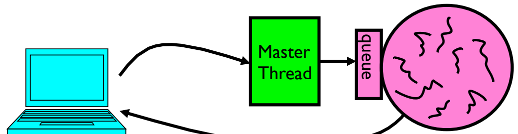
- Instead, use a single process
- Multithreaded (cooperating) version:

```
serverLoop() {
        connection = AcceptCon();
        ThreadFork(ServiceWebPage(), connection);
}
```

- Looks almost the same, but has many advantages:
  - Can share file caches kept in memory, results of CGI scripts, other things
  - Threads are *much* cheaper to create than processes, so this has a lower per-request overhead
- What about Denial of Service attacks or digg / Slashdot effects?

Thread Pools

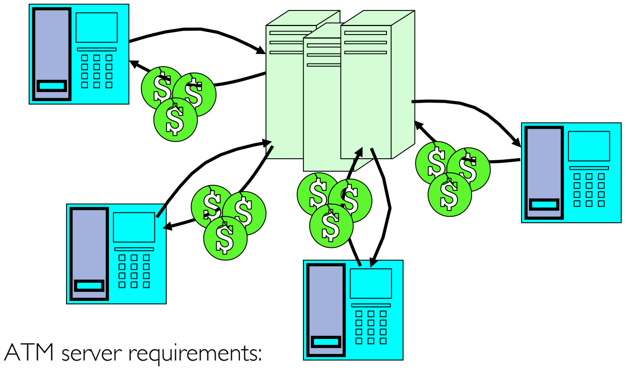
- Problem with previous version: Unbounded Threads
  - When web-site becomes too popular – throughput sinks
- Instead, allocate a bounded "pool" of worker threads, representing the maximum level of multiprogramming




```
master() {
   allocThreads(worker,queue);
   while(TRUE) {
      con=AcceptCon();
      Enqueue(queue,con);
      wakeUp(queue);
   }
}
```

```
worker(queue) {
   while(TRUE) {
      con=Dequeue(queue);
      if (con==null)
         sleepOn(queue);
      else
         ServiceWebPage(con);
   }
}
```


ATM Bank Server



- ATM server requirements:
  - Service a set of requests
  - Do so without corrupting database
  - Don't hand out too much money

ATM bank server example

- Suppose we wanted to implement a server process to handle requests from an ATM network:

```
BankServer() {
   while (TRUE) {
      ReceiveRequest(&op, &acctId, &amount);
      ProcessRequest(op, acctId, amount);
   }
}
ProcessRequest(op, acctId, amount) {
   if (op == deposit) Deposit(acctId, amount);
   else if …
}
Deposit(acctId, amount) {
   acct = GetAccount(acctId); /* may use disk I/O */
   acct->balance += amount;
   StoreAccount(acct); /* Involves disk I/O */
}
```

- How could we speed this up?
  - More than one request being processed at once
  - Event driven (overlap computation and I/O)
  - Multiple threads (multi-proc, or overlap comp and I/O)

Event Driven Version of ATM server

- Suppose we only had one CPU
  - Still like to overlap I/O with computation
  - Without threads, we would have to rewrite in event-driven style
- Example

```
BankServer() {
   while(TRUE) {
      event = WaitForNextEvent();
      if (event == ATMRequest)
         StartOnRequest();
      else if (event == AcctAvail)
         ContinueRequest();
      else if (event == AcctStored)
         FinishRequest();
   }
}
```

  - What if we missed a blocking I/O step?
  - What if we have to split code into hundreds of pieces which could be blocking?
  - This technique is used for programming GPUs (Graphics Processing Unit)

Can Threads Make This Easier?

- Threads yield overlapped I/O and computation without having to "deconstruct" code into non-blocking fragments
  - One thread per request
- Requests proceeds to completion, blocking as required:

```
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
```


Problem is at the Lowest Level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

- However, what about (Initially, y = 12):

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

  - What are the possible values of x?
- Or, what are the possible values of x below?

Thread A	Thread B
x = 1;	x = 2;

  - X could be 1 or 2 (non-deterministic!)
  - Could even be 3 for serial processors:
    - » Thread A writes 0001, B writes 0010 → scheduling order ABABABBA yields 3!

Atomic Operations

- To understand a concurrent program, we need to know what the underlying indivisible operations are!
- 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
  - Fundamental building block – if no atomic operations, then have no way for threads to work together
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
  - Consequently – weird example that produces "3" on previous slide can't happen
- Many instructions are not atomic
  - Double-precision floating point store often not atomic
  - VAX and IBM 360 had an instruction to copy a whole array

Correctness Requirements

- Threaded programs must work for all interleavings of thread instruction sequences
  - Cooperating threads inherently non-deterministic and non-reproducible
  - Really hard to debug unless carefully designed!
- Examples:

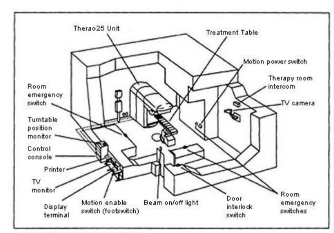


Figure 1. Typical Therac-25 facility

Administrivia

- Group/Section assignments finalized!
  - If you are not in group, talk to us immediately!
- Attend assigned sections
  - Need to know your TA!
    - » Participation is 8% of your grade
    - » Should attend section with your TA
- First design doc due next Wednesday
  - This means you should be well on your way with Project 1
  - Watch for notification from your TA to sign up for design review
- Basic semaphores work in PintOS!
  - However, you will need to implement priority scheduling behavior both in semaphore and ready queue

- BREAK

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:

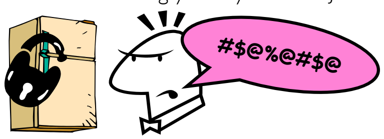
Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Definitions

- Synchronization: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic
  - We are going to show that its 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. Only one thread at a time will get into this section of code
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing the same thing

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
- For example: fix the milk problem by putting a key on the refrigerator
  - Lock it and take key if you are going to go buy milk
  - Fixes too much: roommate angry if only wants OJ



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

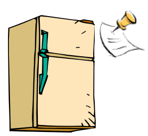
Too Much Milk: Correctness Properties

- Need to be careful about correctness of concurrent programs, since non-deterministic
  - Impulse is to start coding first, then when it doesn't work, pull hair out
  - Instead, think first, then code
  - Always write down behavior first
- 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;
   }
}
```



Too Much Milk: Solution #1

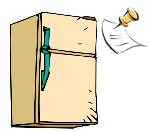
- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of "lock")
  - Remove note after buying (kind of "unlock")
  - Don't buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

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

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?
  - Still too much milk but only occasionally!
  - 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 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) {
        leave Note;
        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:

<u>Thread A</u>

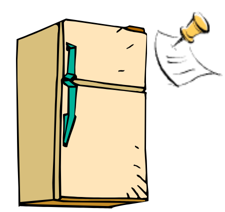
```
leave note A;
if (noNote B) {
    if (noMilk) {
        buy Milk;
    }
}
remove note A;
```

<u>Thread B</u>

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

- Does this work?
- Possible for neither thread to buy milk
  - Context switches at exactly the wrong times can lead each to think that the other is going to buy
- Really insidious:
  - Extremely unlikely that this would happen, but will at worse possible time
  - Probably something like this in UNIX

Too Much Milk Solution #2: problem!



- I thought *you* had the milk! But I thought *you* had the 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

Case 1

- "`leave note A`" happens before "`if (noNote A)`"

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

if (noMilk) {
    buy milk;}
}
remove note A;
```


Case 1

- "`leave note A`" happens before "`if (noNote A)`"

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

if (noMilk) {
    buy milk;}
}
remove note A;
```


Case 1

- "`leave note A`" happens before "`if (noNote A)`"

```
leave note A;                  happened    leave note B;
while (note B) {\\X            before      if (noNote A) {\\Y
    do nothing;                                if (noMilk) {
};                                                 buy milk;
                                               }
        Wait for                           }
        note B to                          remove note B;
        be remove
if (noMilk) {
    buy milk;}
}
remove note A;
```

Case 2

- "`if (noNote A)`" happens before "`leave note A`"

```
leave note A;
while (note B) {\\X
    do nothing;
};

if (noMilk) {
    buy milk;}
}
remove note A;
```

happened before

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


Case 2

- "`if (noNote A)`" happens before "`leave note A`"

```
leave note A;
while (note B) {\\X
    do nothing;
};

if (noMilk) {
    buy milk;}
}
remove note A;
```

happened before

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


Case 2

- "`if (noNote A)`" happens before "`leave note A`"

```
leave note A;
while (note B) {\\X
    do nothing;
};
```

Wait for note B to be remove

```
if (noMilk) {
    buy milk;}
}
remove note A;
```

happened before

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


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 higher-level primitives than atomic load & store
  - Build even higher-level programming abstractions on this hardware support

Too Much Milk: Solution #4

- Suppose we have some sort of implementation of a lock
  - `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"
- Of course, you can make this even simpler: suppose you are out of ice cream instead of milk
  - Skip the test since you always need more ice cream ;-)

Where are we going with synchronization?

Programs	Shared Programs			
Higher-level API	Locks	Semaphores	Monitors	Send/Receive
Hardware	Load/Store	Disable Ints	Test&Set	Compare&Swap

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

Summary

- Concurrent threads are a very useful abstraction
  - Allow transparent overlapping of computation and I/O
  - Allow use of parallel processing when available
- Concurrent threads introduce problems when accessing shared data
  - Programs must be insensitive to arbitrary interleavings
  - Without careful design, shared variables can become completely inconsistent
- 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