CS162
Operating Systems and Systems Programming
Lecture 7

Concurrency (Continued), Synchronization

February 8th, 2017

Nathan Pemberton

http://cs162.eecs.Berkeley.edu

RoadMap

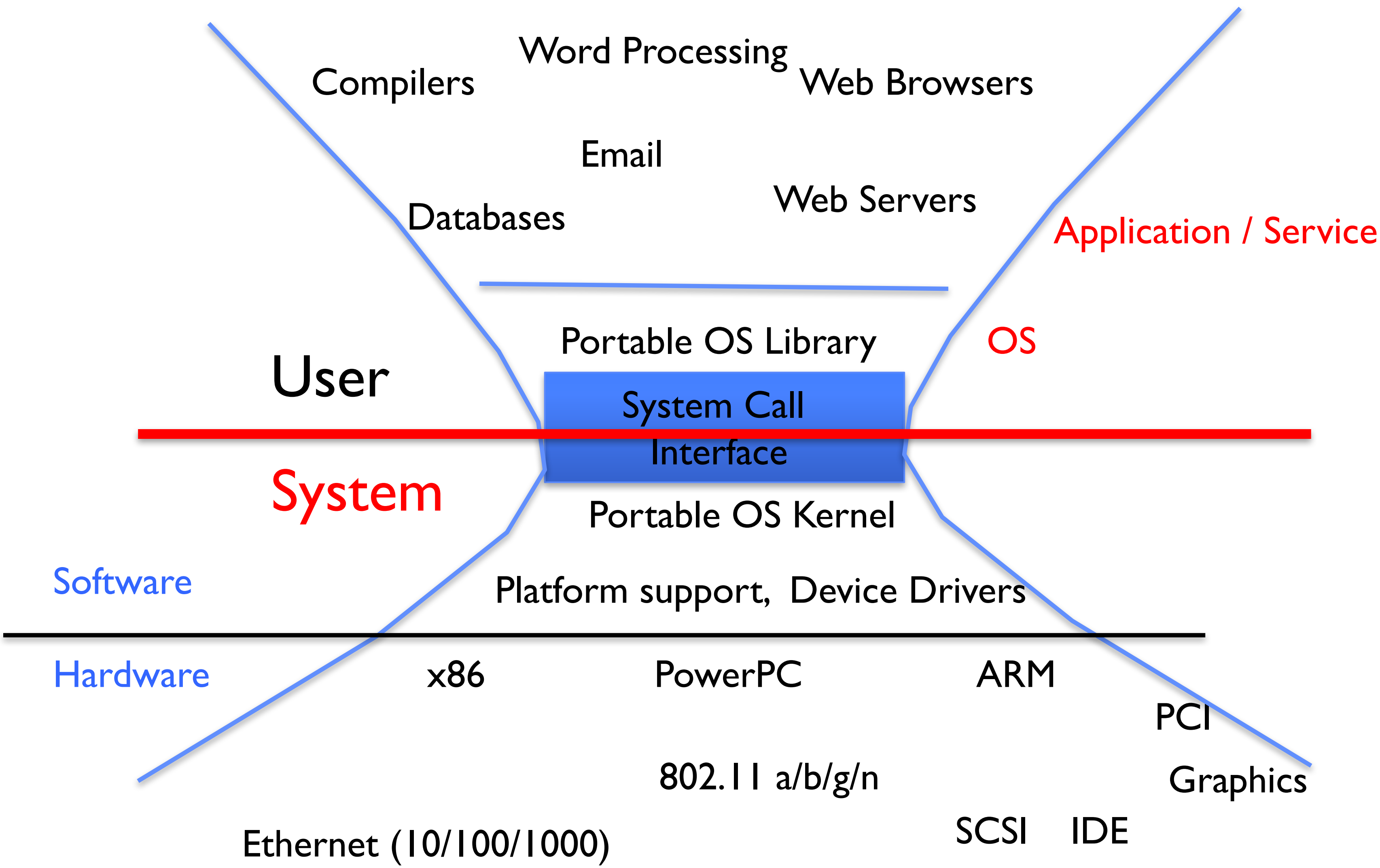
So Far…

- Processes
  - Thread(s) + address space
- Address Space
- Protection
- Dual Mode
- Interrupt handlers
  - Interrupts, exceptions, syscall
- Key Layers: OS Lib, Syscall, Subsystem, Driver
  - User handler on OS descriptors
- Process control
  - fork, wait, signal, exec
- Communication through sockets
  - Integrates processes, protection, file ops, concurrency
- Client-Server Protocol
- Concurrent Execution: Threads

To Come…

- Concurrency/Synchronization (today)
- Filesystems
- Scheduling
- Address Translation/Caching
- Transactions and Distributed Computing
- Security
- …

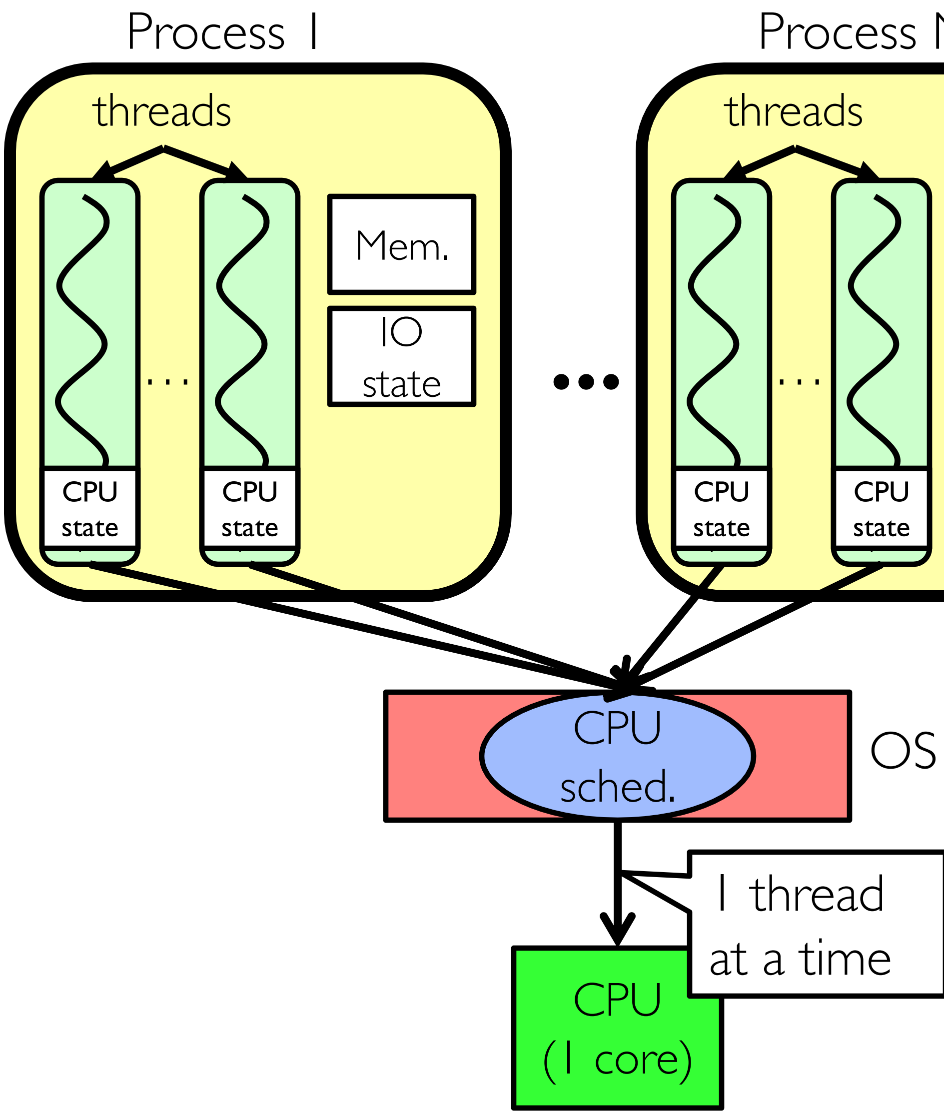
Operating System as Design



Perspective on 'groking' 162

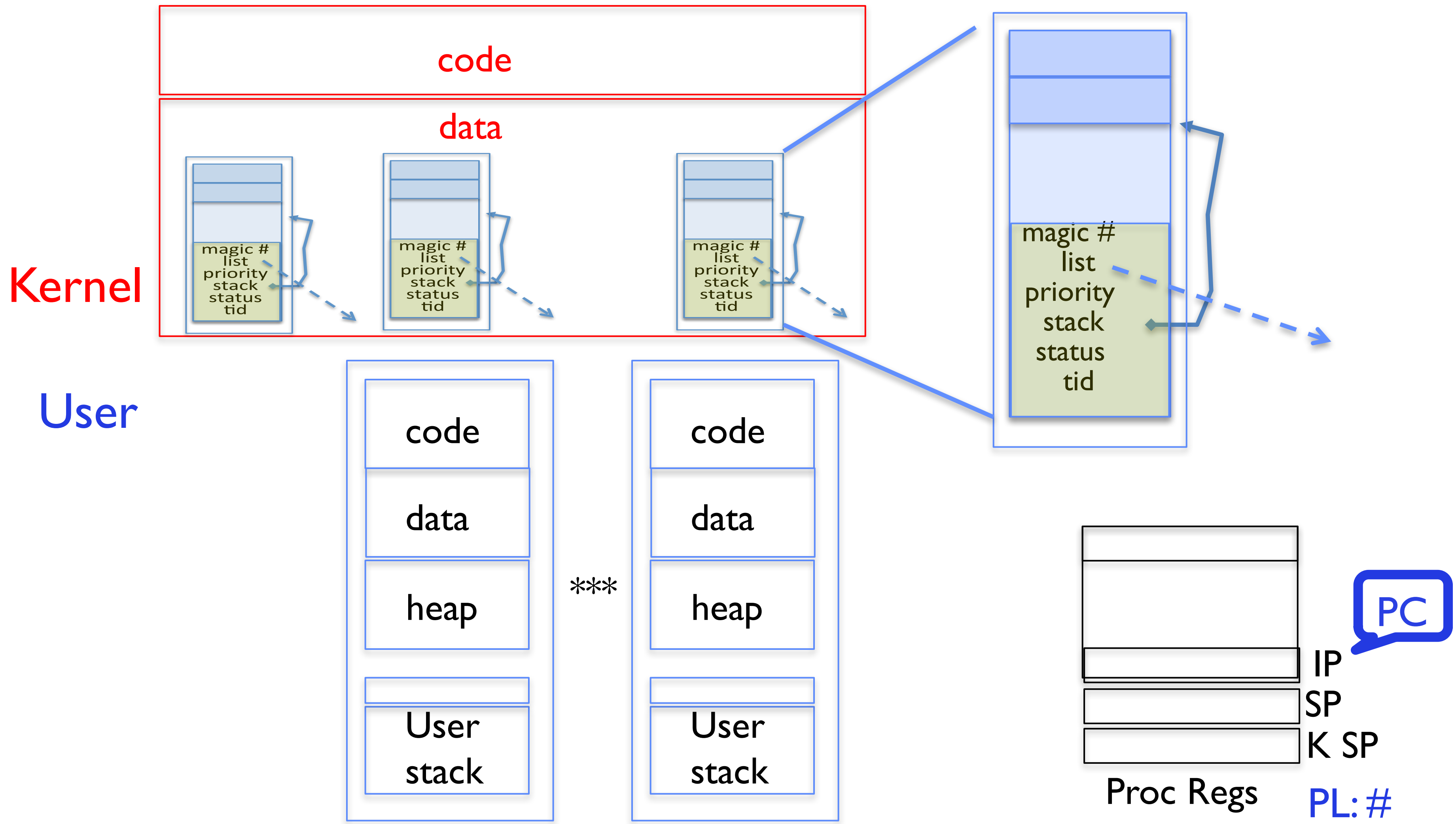
- Historically, OS was the most complex software
  - Concurrency, synchronization, processes, devices, communication, …
  - Core systems concepts developed there
- Today, many "applications" are complex software systems too
  - These concepts appear there
  - But they are realized out of the capabilities provided by the operating system
- Seek to understand how these capabilities are implemented upon the basic hardware
- See concepts multiple times from multiple perspectives
  - Lecture provides conceptual framework, integration, examples, …
  - Book provides a reference with some additional detail
  - Lots of other resources that you need to learn to use
    - » man pages, google, reference manuals, includes (.h)
- Section, Homework and Project provides detail down to the actual code AND direct hands-on experience

Conceptual Framework



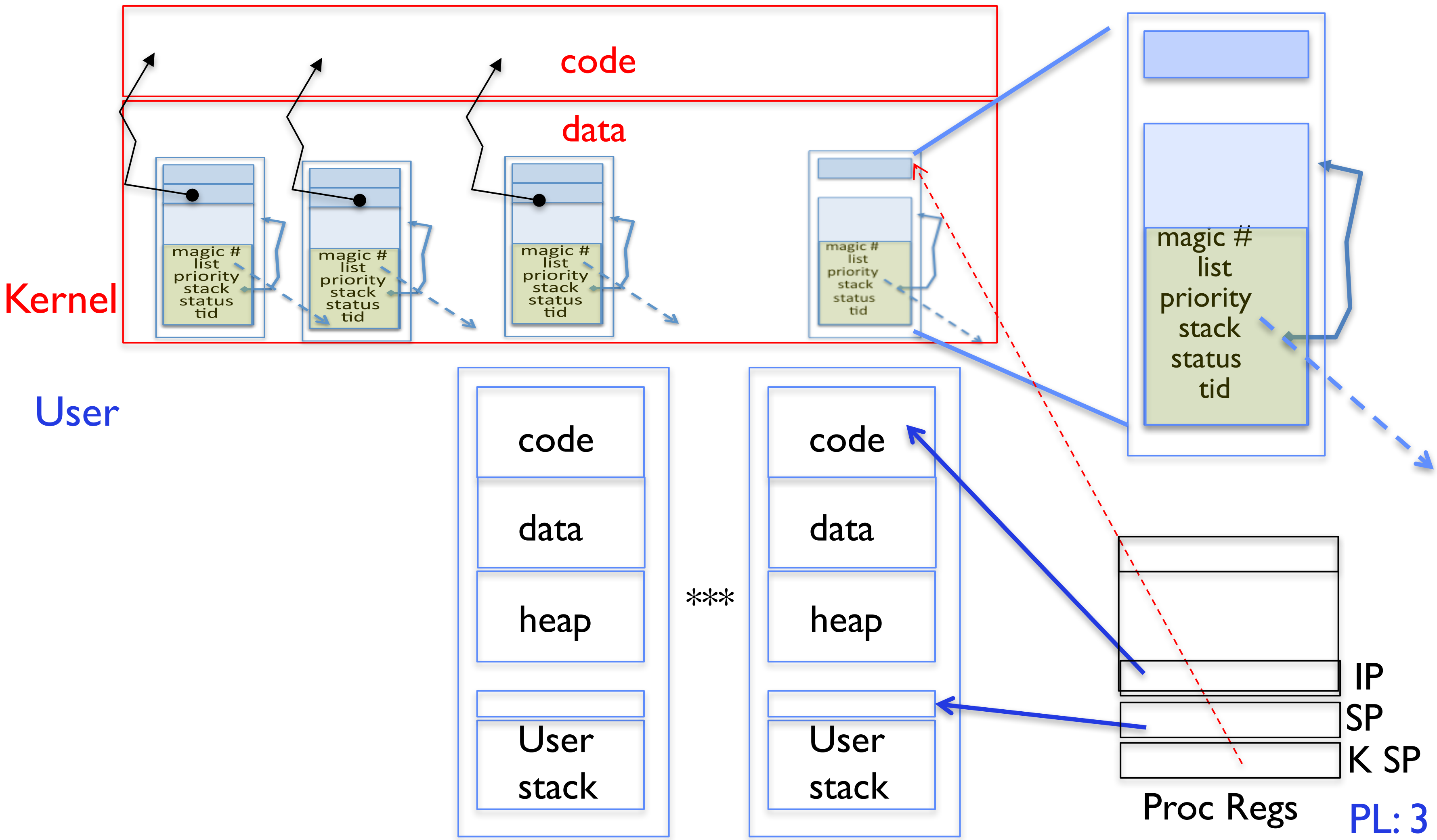
- Physical Addresses Shared
  - **So:** Processes and Address Translation
- Single CPU Must Be Shared
  - **So:** Threads
- Processes Aren't Trusted
  - **So:** Kernel/Userspace Split
- Threads Might Not Cooperate
  - **So:** Use timer interrupts to context switch ("preemption")

Recall: MT Kernel 1T Process ala Pintos/x86



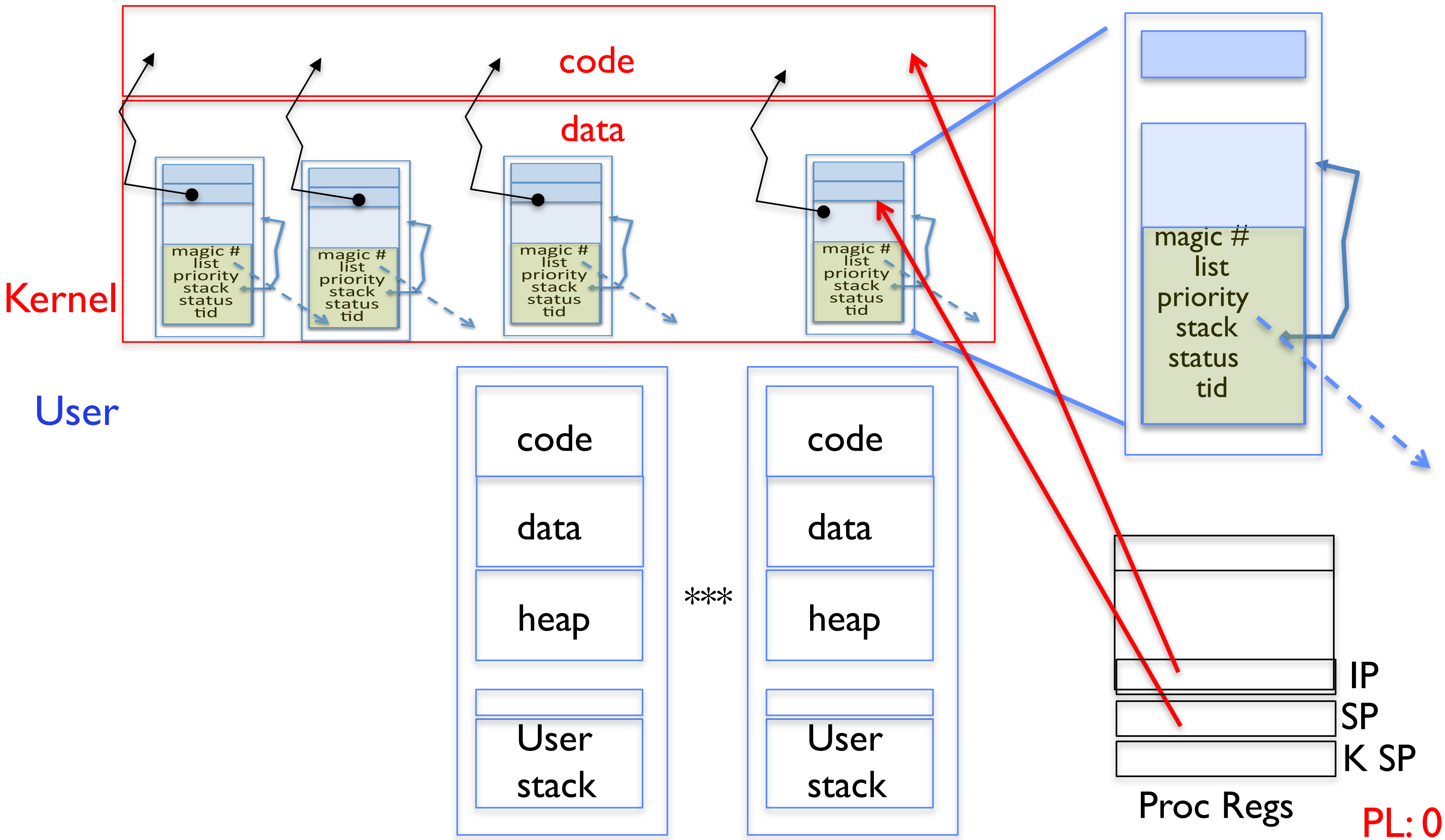
- Each user process/thread associated with a kernel thread, described by a 4KB page object containing TCB and kernel stack for the kernel thread

In User thread, w/ Kernel thread waiting



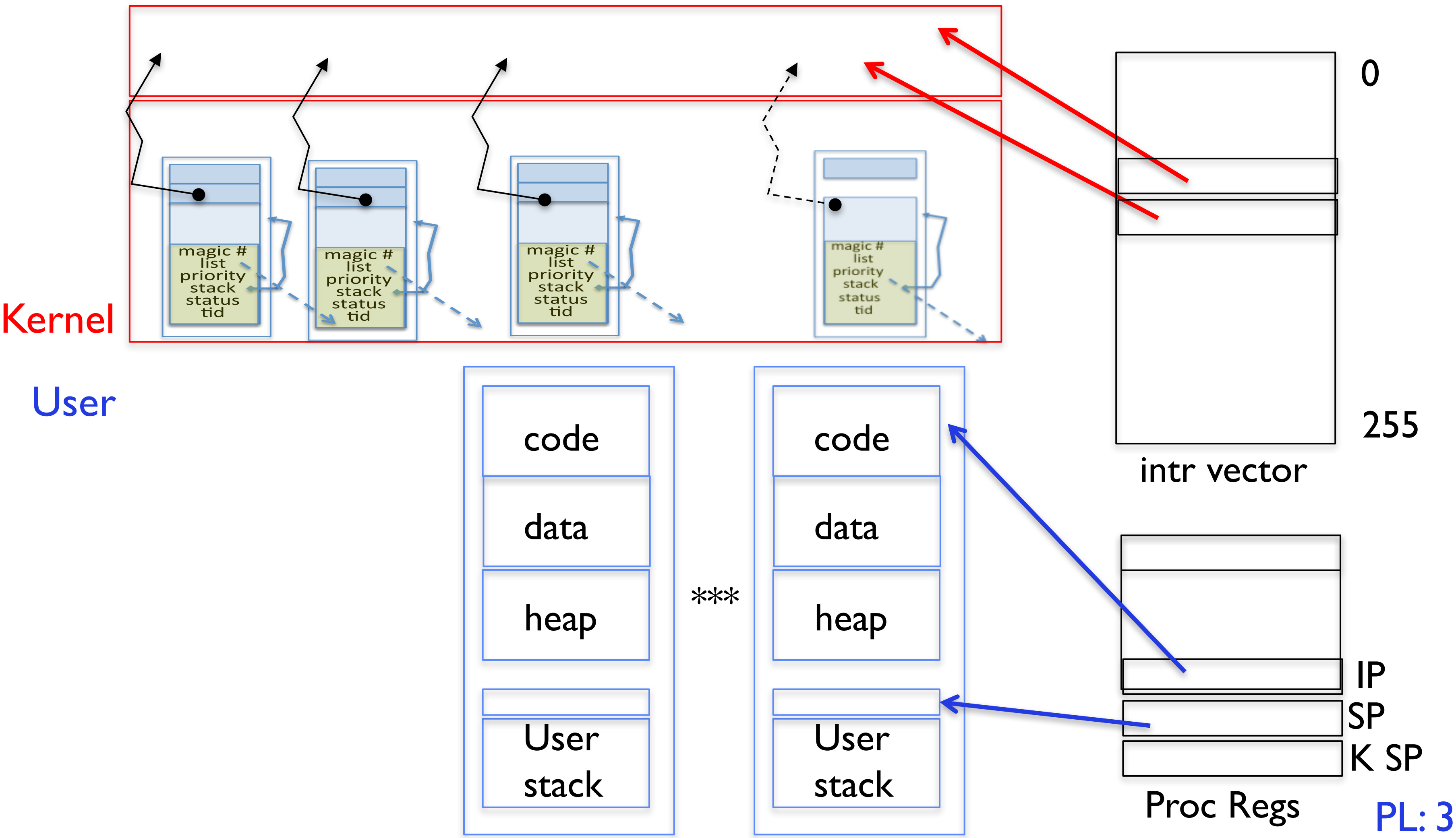
- x86 CPU holds interrupt SP in register
- During user thread execution, associated kernel thread is "standing by"

User → Kernel



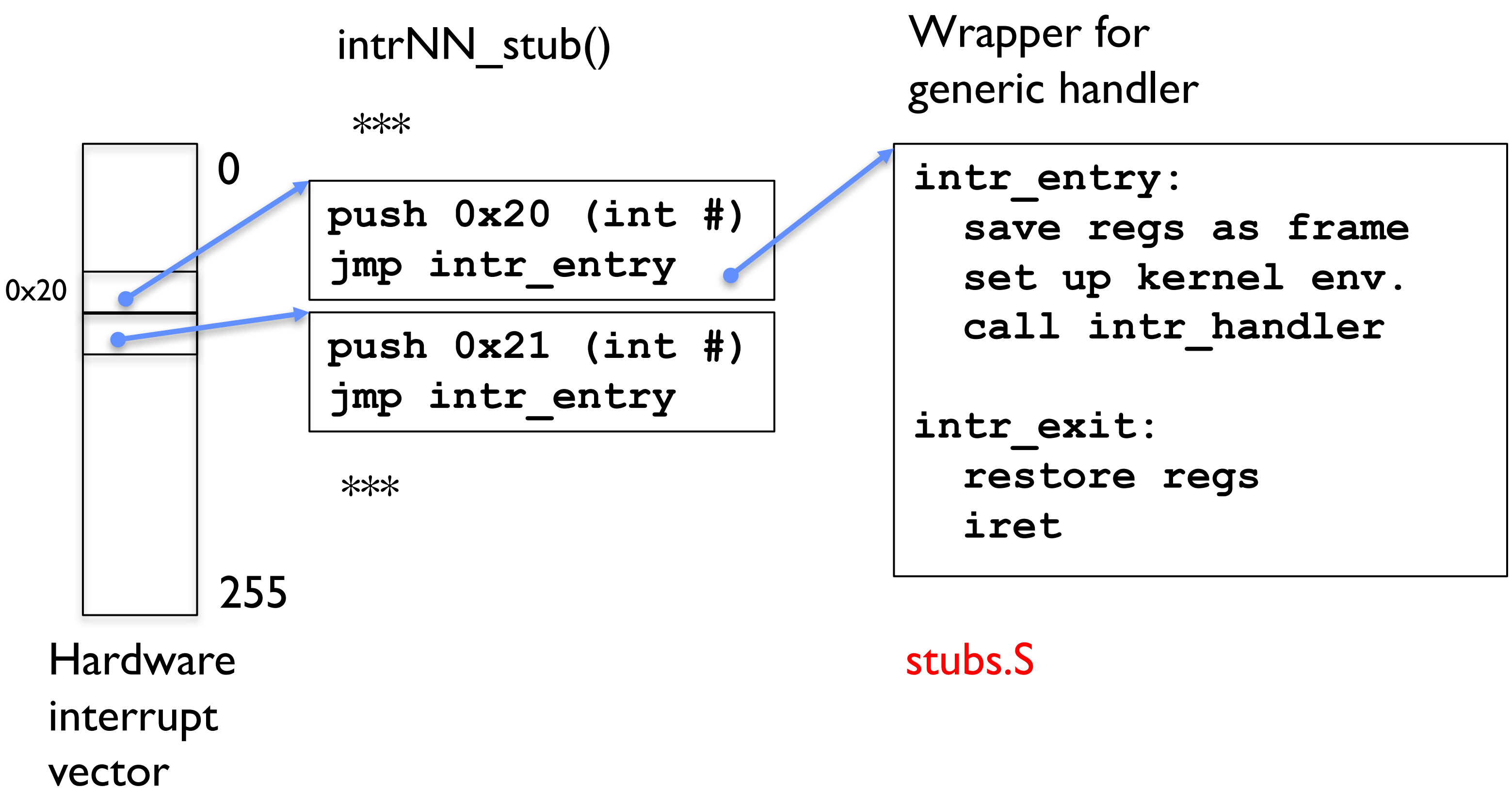
- Mechanism to resume k-thread goes through interrupt vector

User → Kernel via interrupt vector



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

Pintos Interrupt Processing



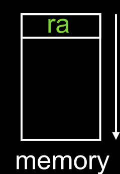
Recall: cs61C THE STACK FRAME

Basic Structure of a Function

Prologue

```
entry_label:
addi $sp,$sp, -framesize
sw $ra, framesize-4($sp)   # save $ra
save other regs if need be
```

Body . . . **(call other functions…)**



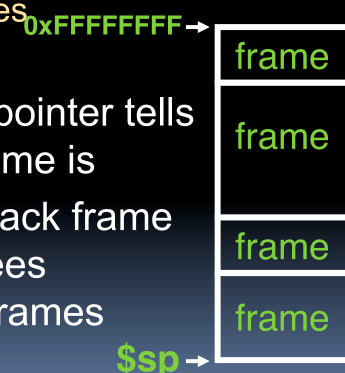
Epilogue

```
restore other regs if need be
lw $ra, framesize-4($sp)   # restore $ra
addi $sp,$sp, framesize
jr $ra
```

CS61C L10 Introduction to MIPS : Procedures I (18) Garcia, Spring 2014

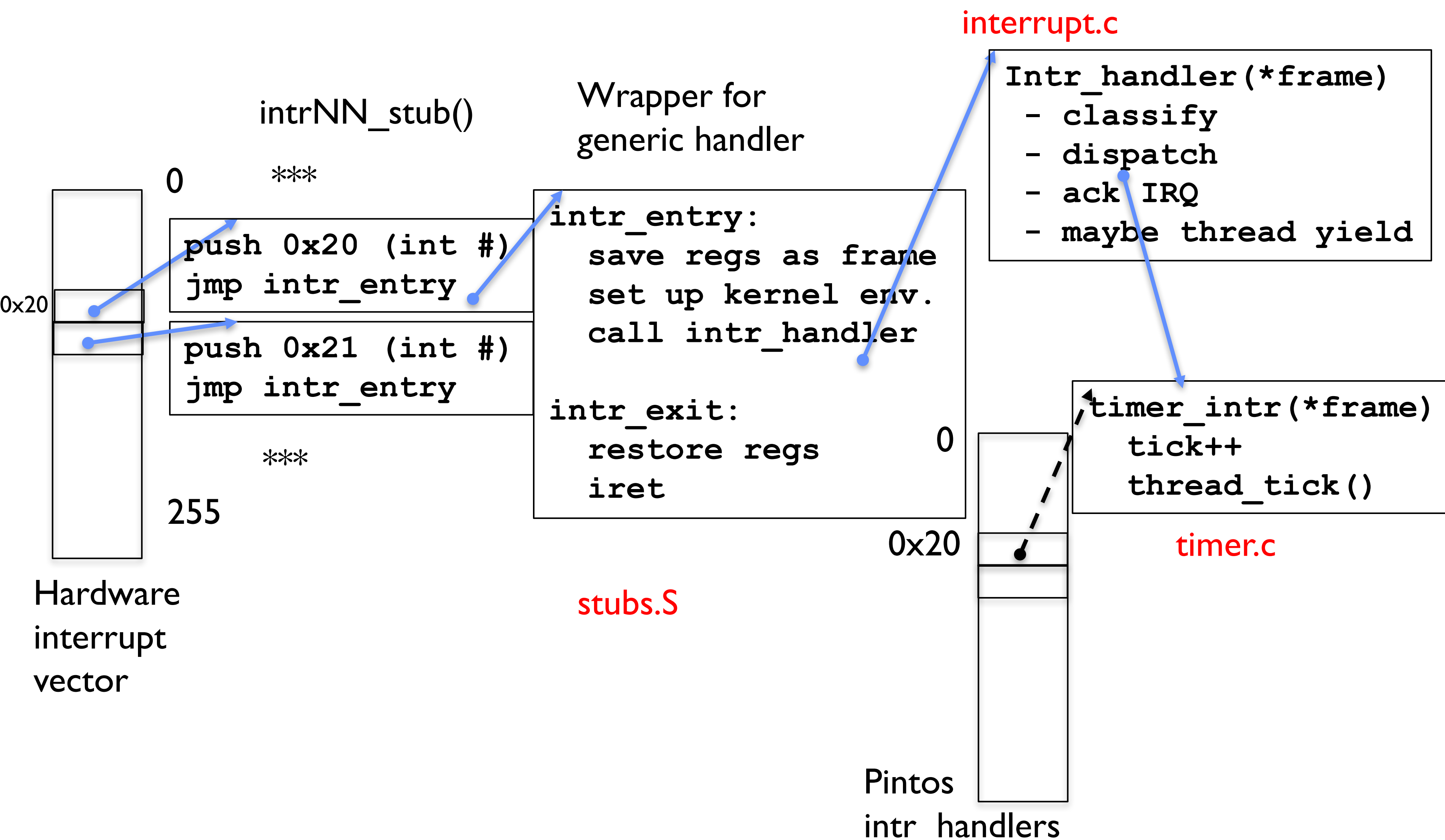
The Stack (review)

- Stack frame includes:
  - Return “instruction” address
  - Parameters
  - Space for other local variables
- Stack frames contiguous blocks of memory; stack pointer tells where bottom of stack frame is
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames

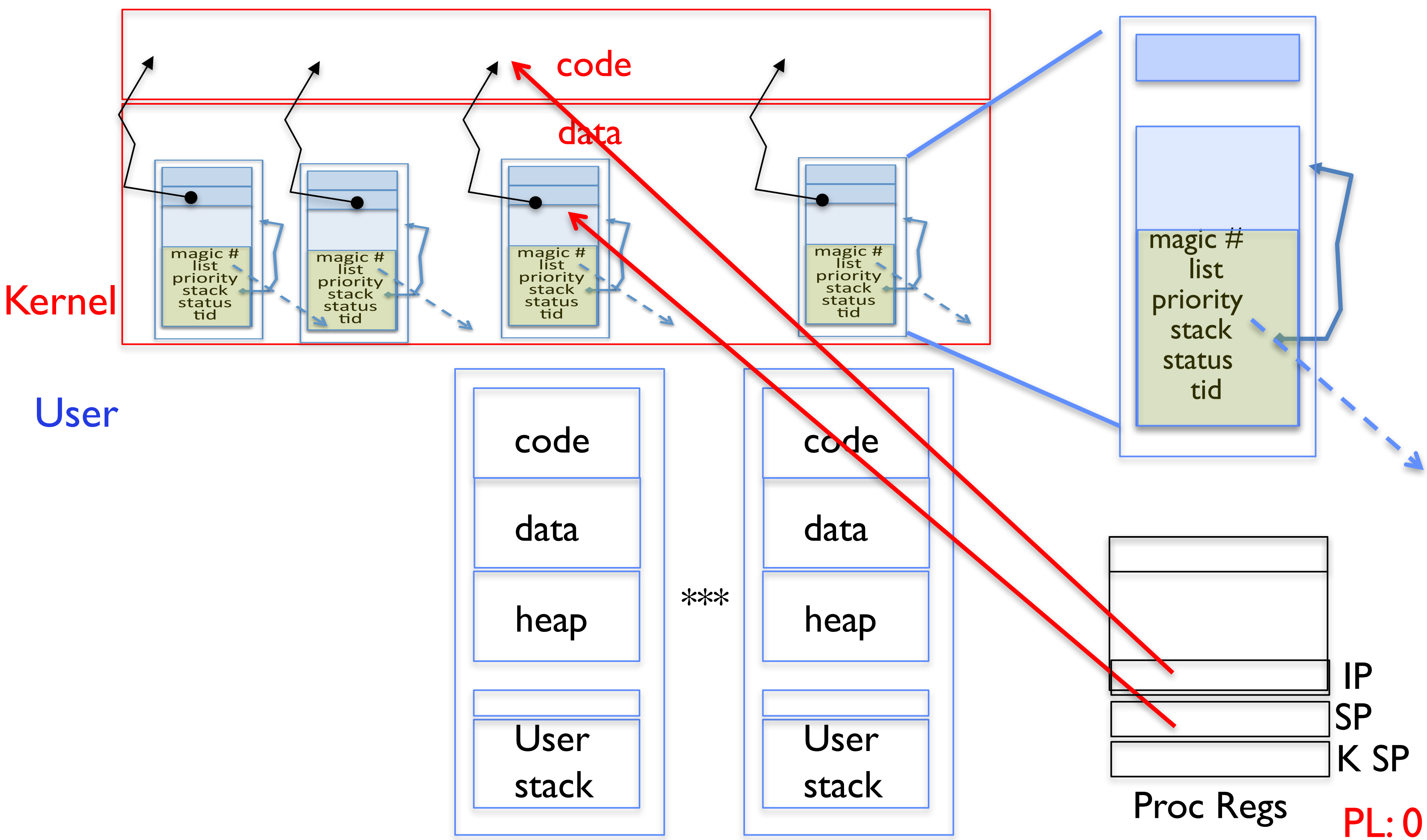


CS61C L11 Introduction to MIPS : Procedures II & Logical Ops (3) Garcia, Spring 2014 © UCB

Pintos Interrupt Processing



In Kernel thread

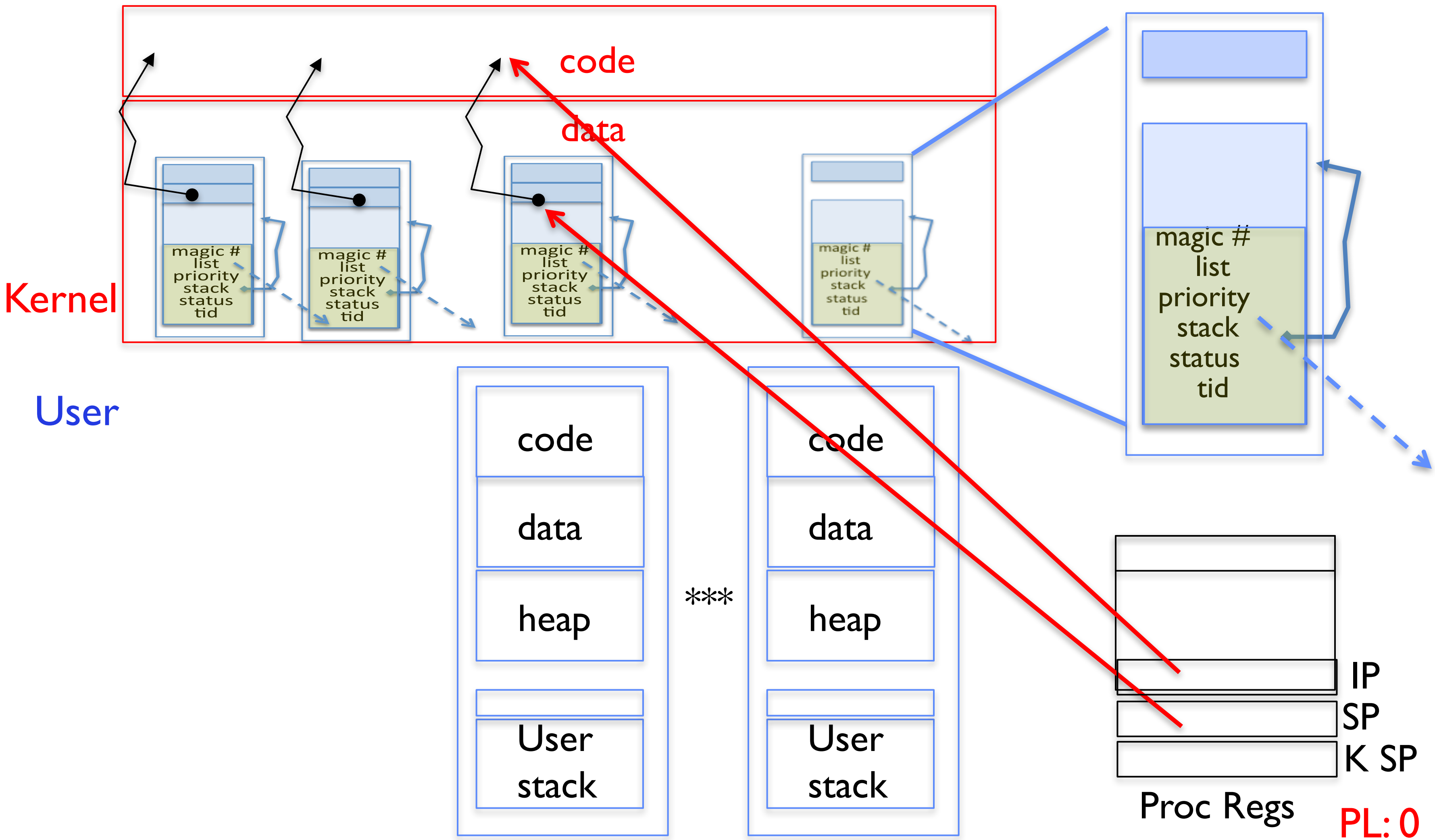


- Kernel threads execute with small stack in thread structure
- Scheduler selects among ready kernel and user threads

Timer may trigger thread switch

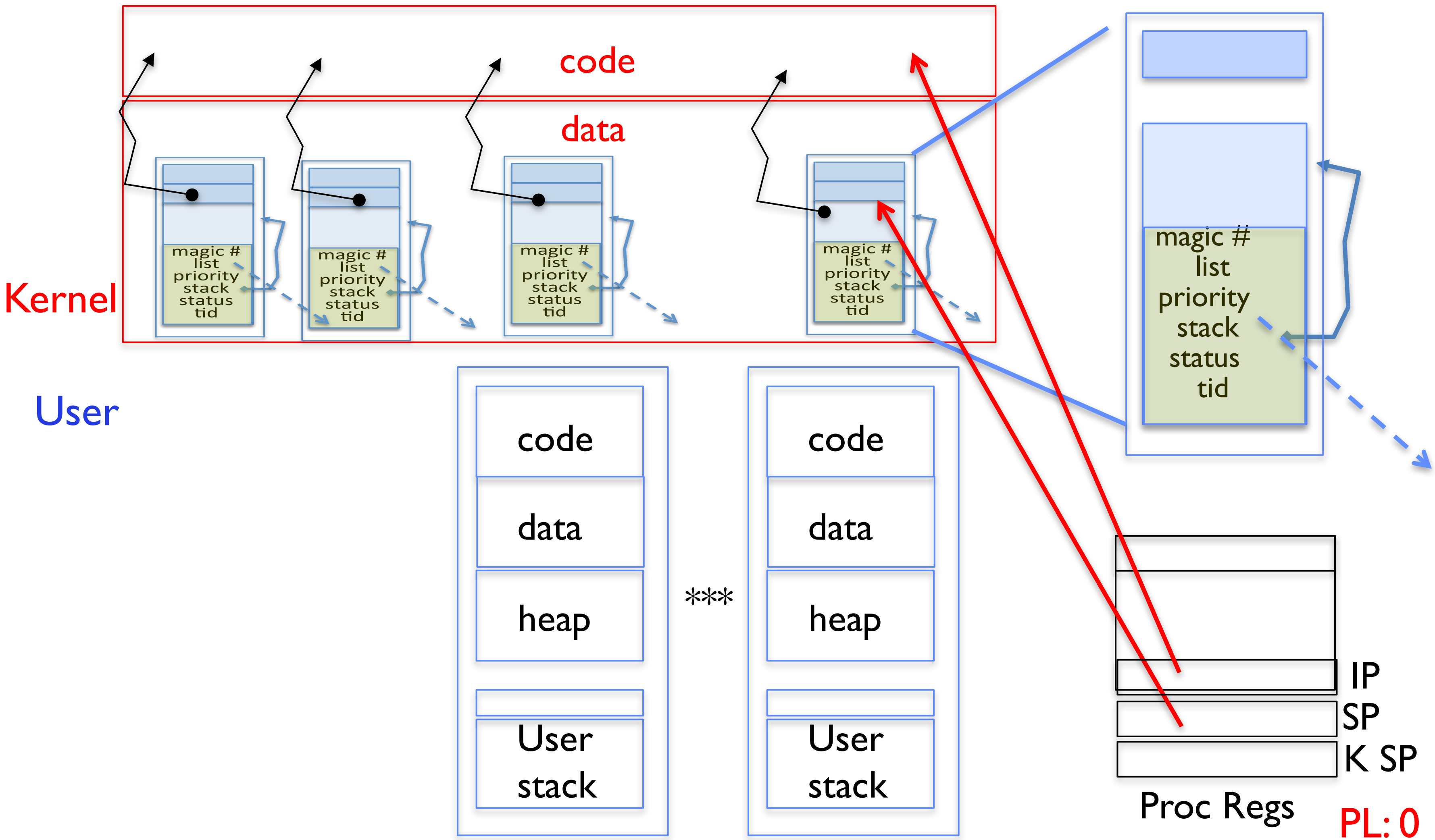
- thread_tick
  - Updates thread counters
  - If quanta exhausted, sets yield flag
- thread_yield
  - On path to rtn from interrupt
  - Sets current thread back to READY
  - Pushes it back on ready_list
  - Calls schedule to select next thread to run upon iret
- Schedule
  - Selects next thread to run
  - 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)

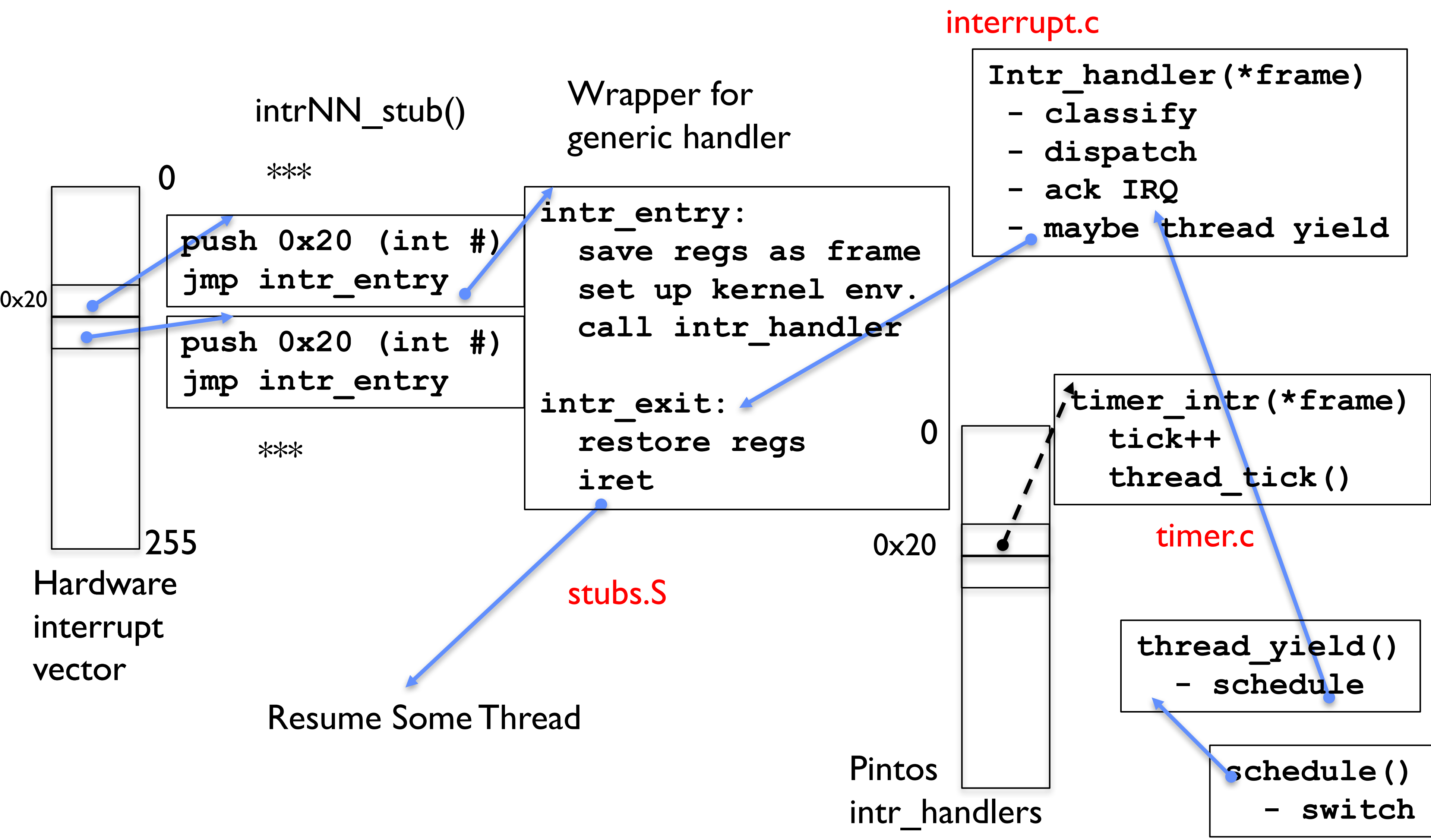


- switch_threads: save regs on current small stack, change SP, return from destination threads call to switch_threads

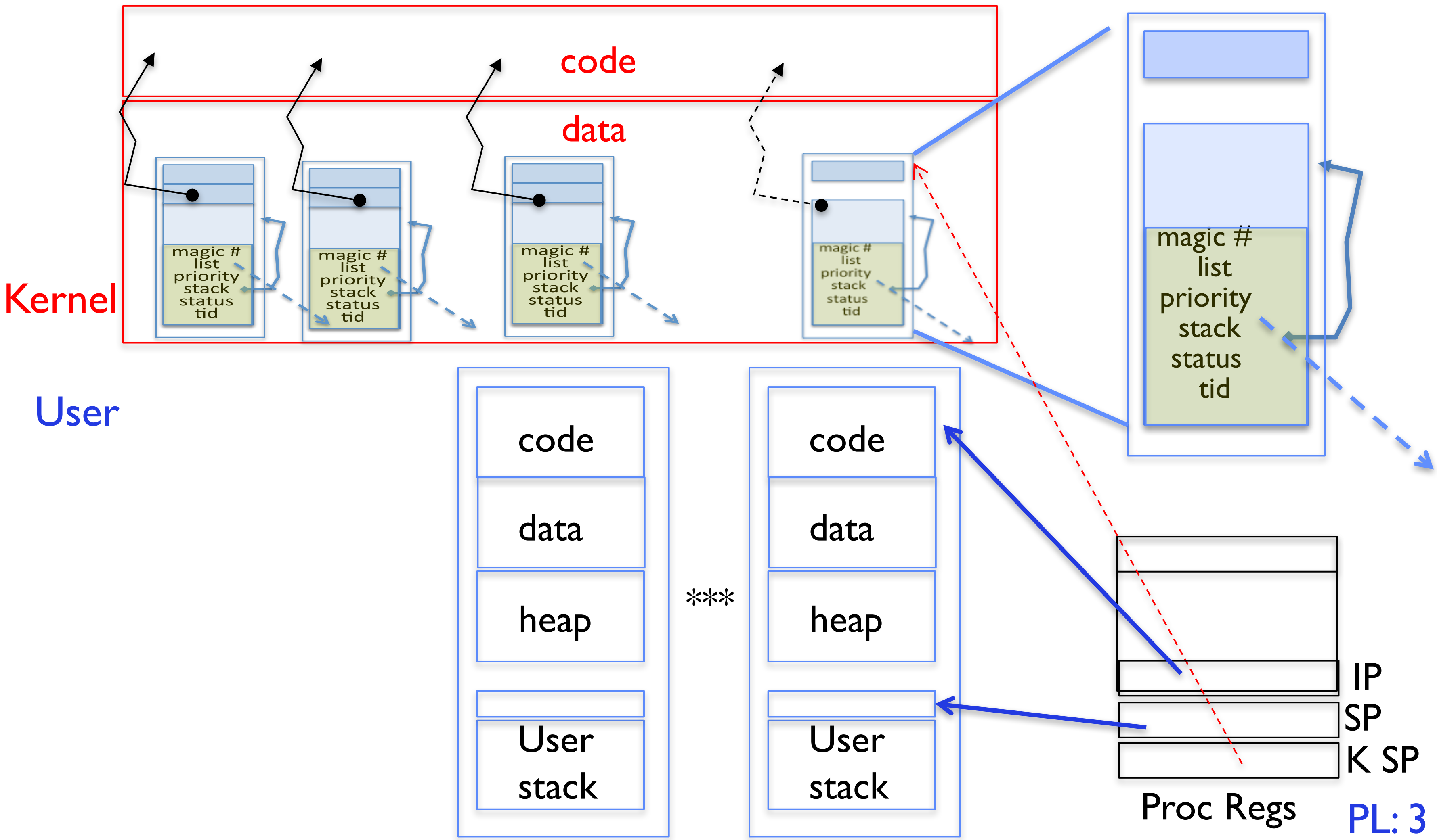
Switch to Kernel Thread for Process



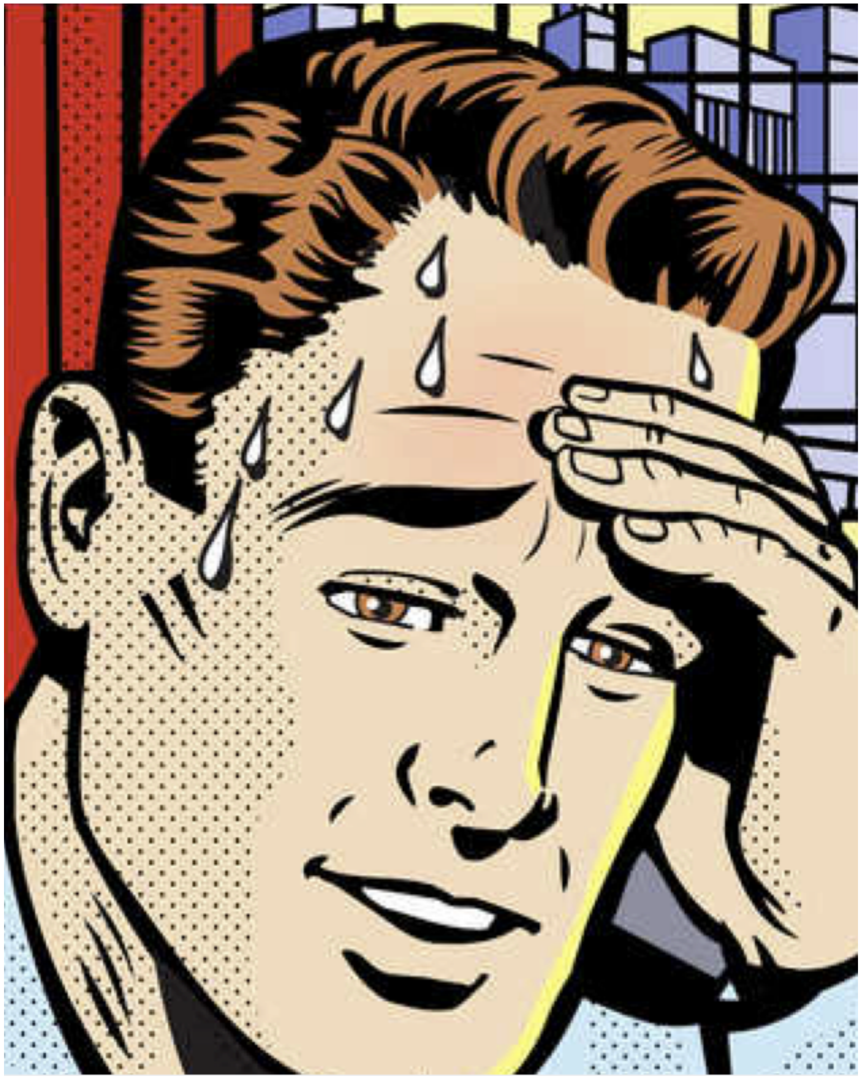
Pintos Return from Processing



Kernel → User



- Interrupt return (iret) restores user stack and PL

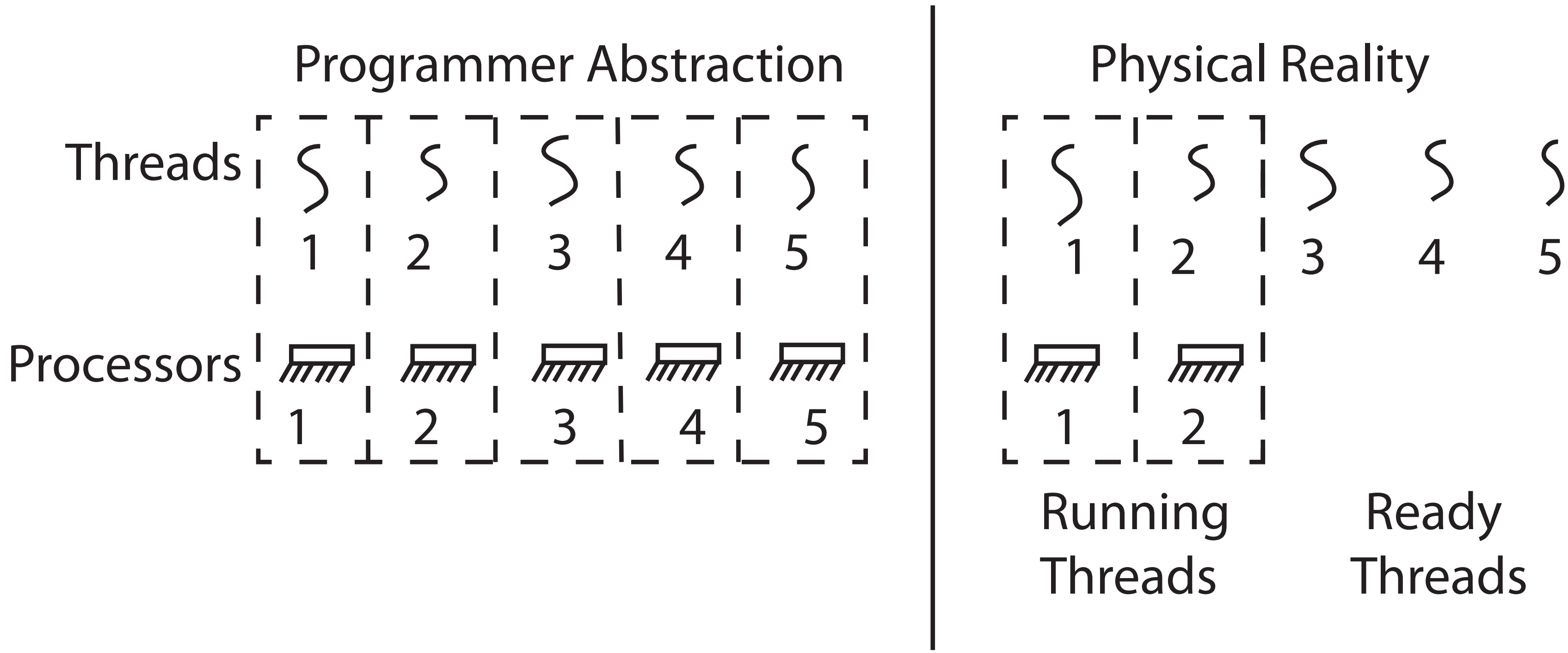


- Re-visit Slides and Lecture
- Read Book!
- Read Code!
  - e.g. thread.c and thread.S

Goals for Today

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

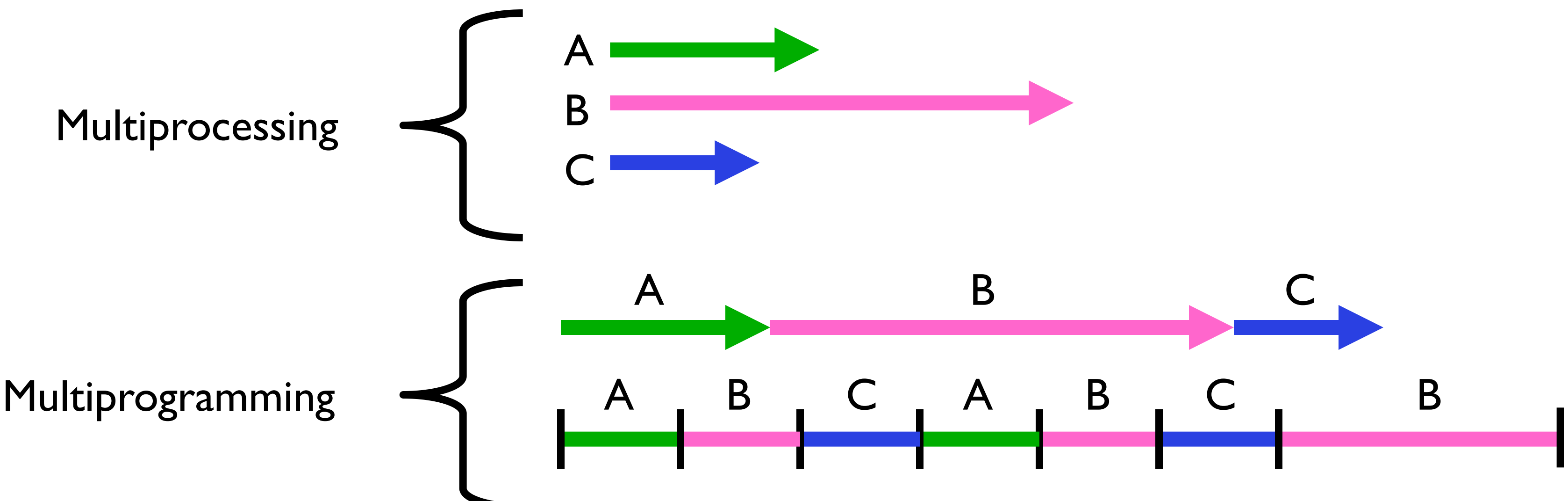
Recall: Thread Abstraction



- 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 $\Rightarrow$ Input state determines results
  - Reproducible $\Rightarrow$ 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

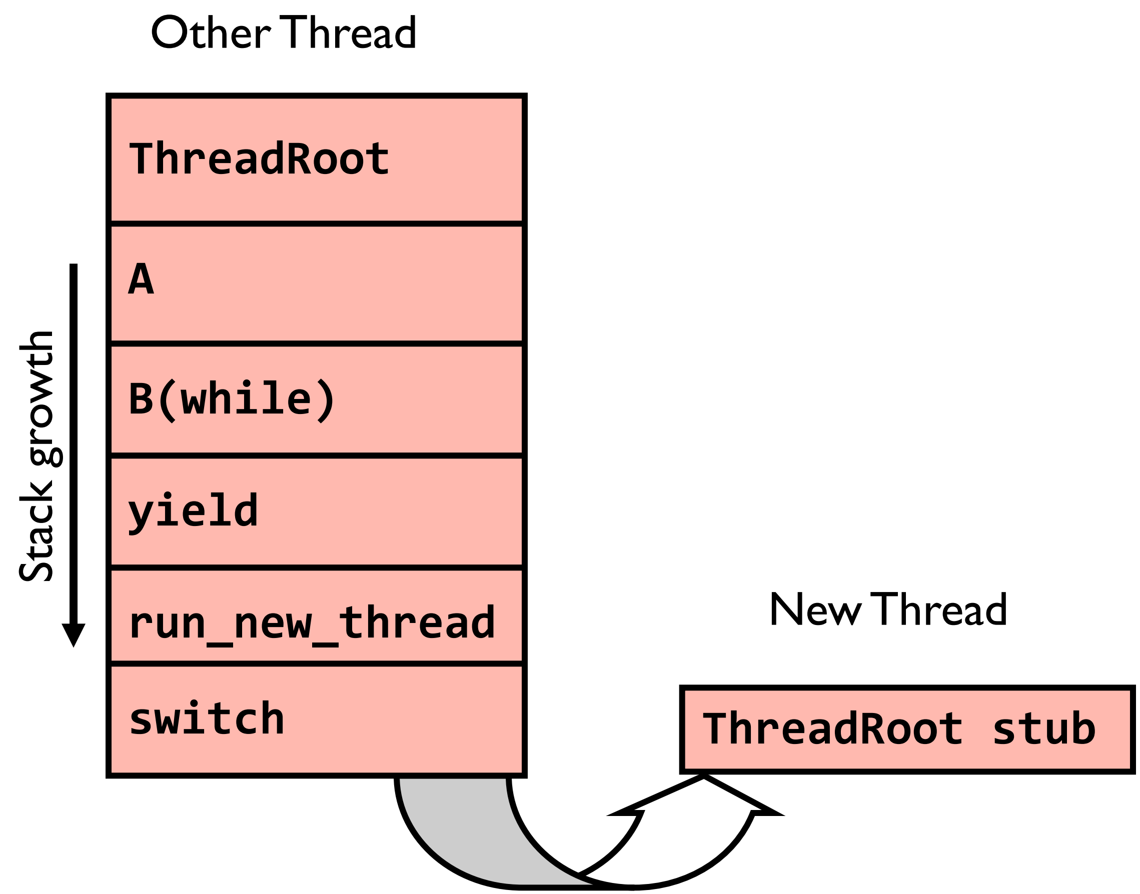
Why allow cooperating threads?

- 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

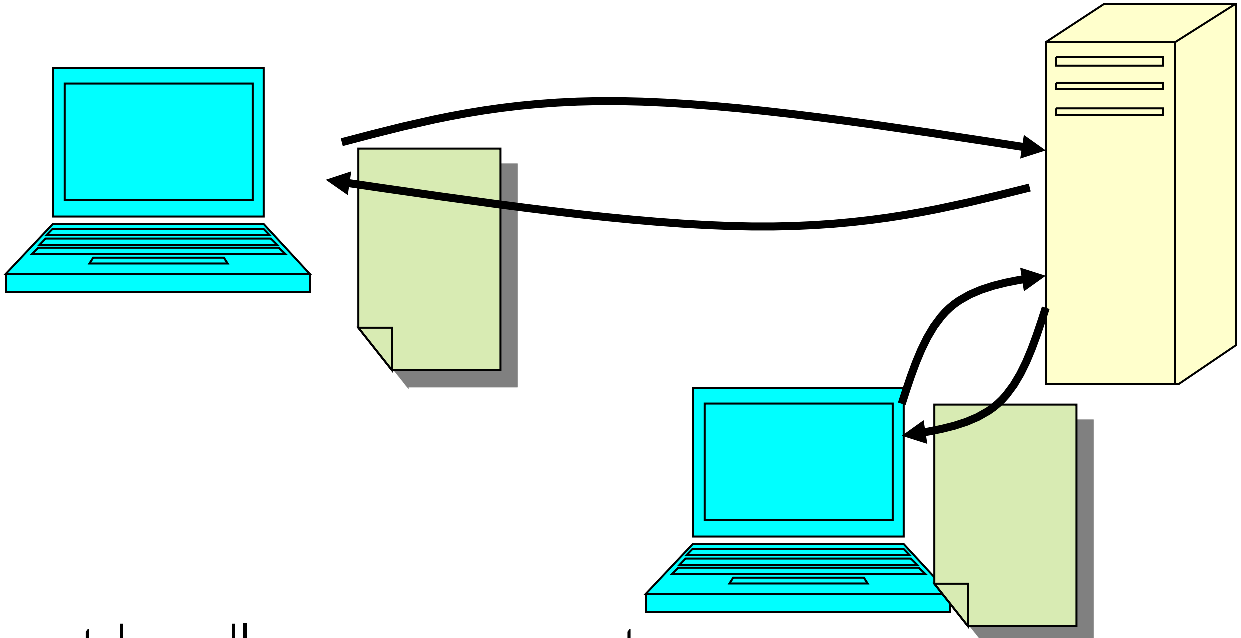
- Advantage 3: Modularity
  - More important than you might think
  - Chop large problem up into simpler pieces
    - To compile, for instance, `gcc` calls `cpp | cc1 | cc2 | as | ld`
    - Makes system easier to extend

Recall: How does Thread get started?



- Eventually, `run_new_thread()` will select this TCB and return into beginning of `ThreadRoot()`
  - This really starts the new thread

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:

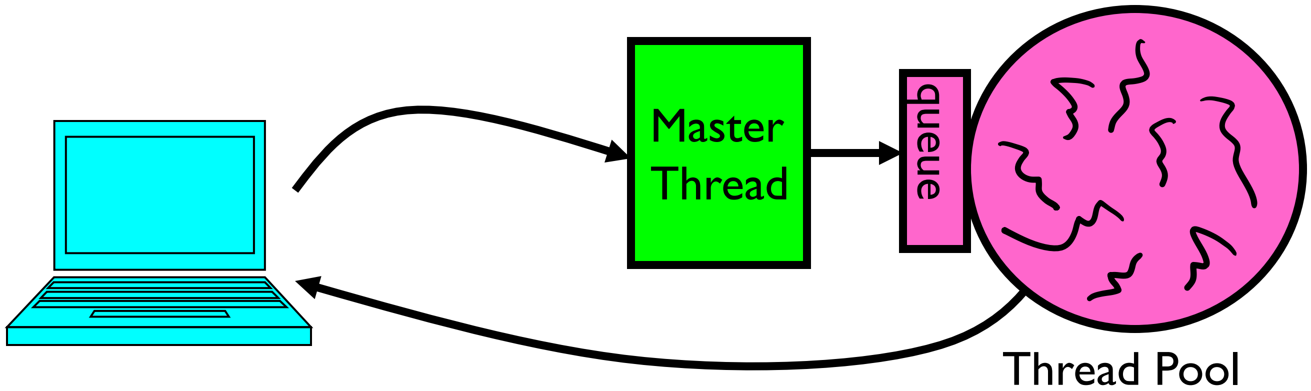
```
serverLoop() {
        connection = AcceptCon();
        ThreadFork(ServiceWebPage(), connection);
    }
```

- 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

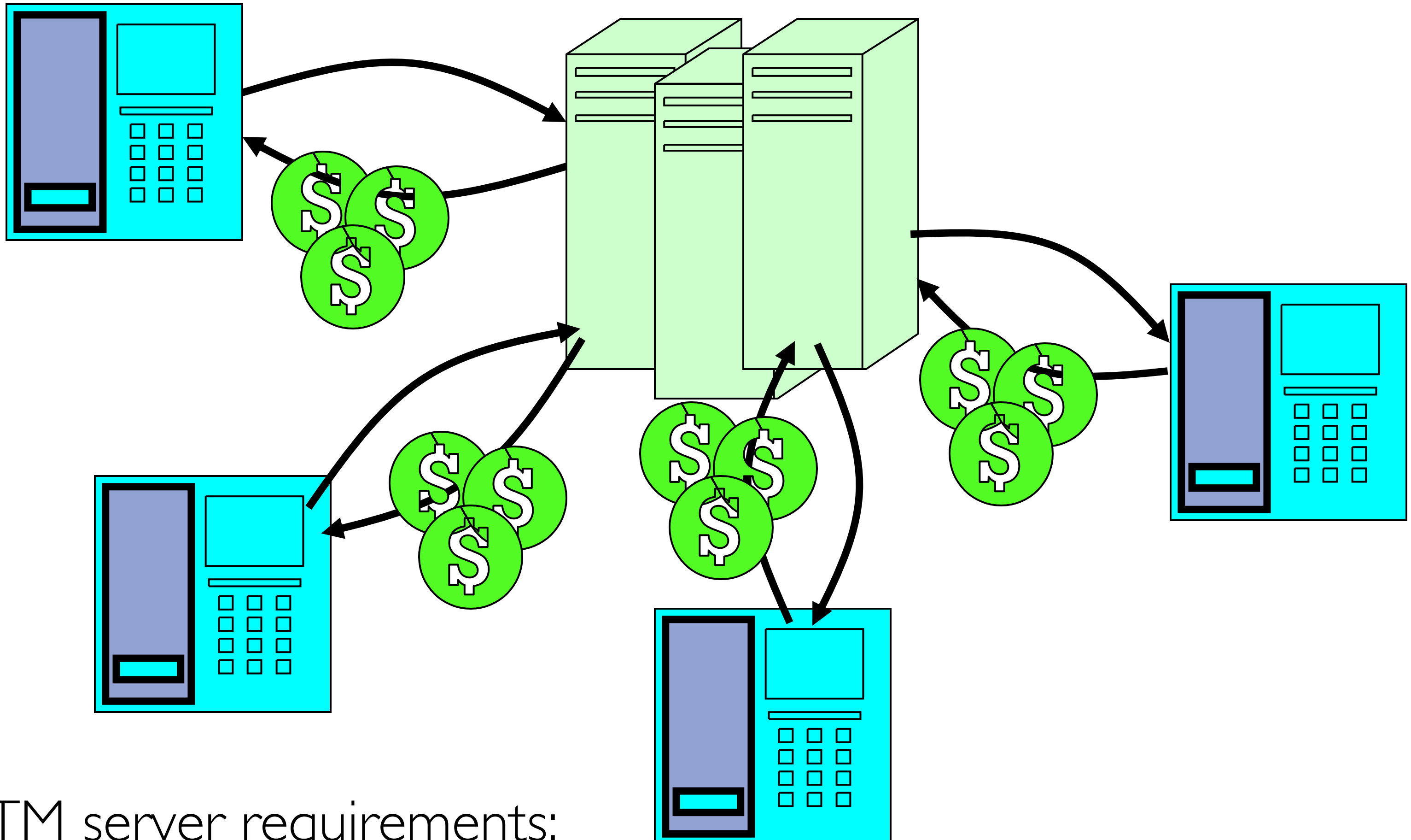
- Problem with previous version: Unbounded Threads
  – When web-site becomes too popular – throughput sinks
- Instead, allocate a bounded "pool" of worker threads, representing the maximum level of multiprogramming




```
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:

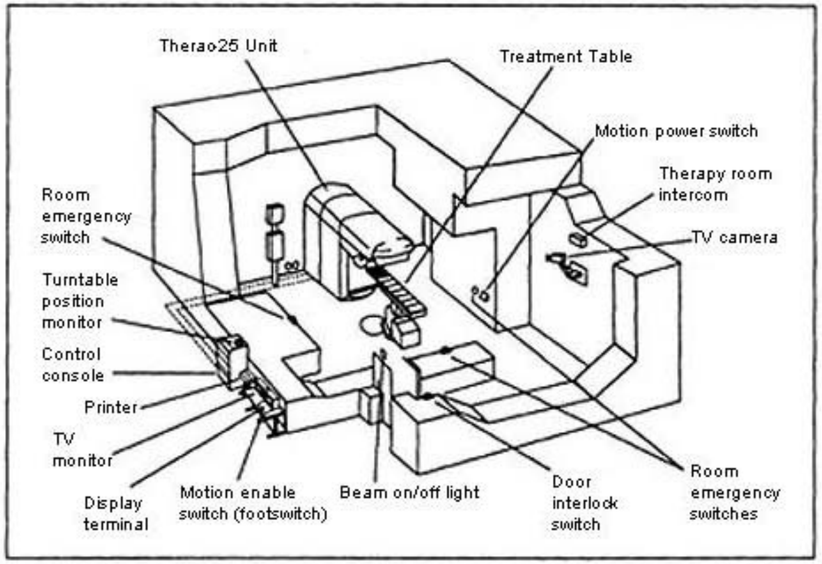


Figure 1. Typical Therac-25 facility

Another Concurrent Program Example

- Two threads, A and B, compete with each other
  - One tries to increment a shared counter
  - The other tries to decrement the counter

Thread A

```
i = 0;
while (i < 10)
   i = i + 1;
printf("A wins!");
```

Thread B

```
i = 0;
while (i > -10)
   i = i - 1;
printf("B wins!");
```

- Assume that memory loads and stores are atomic, but incrementing and decrementing are *not* atomic
- Who wins? Could be either
- Is it guaranteed that someone wins? Why or why not?
- What if both threads have their own CPU running at same speed? Is it guaranteed that it goes on forever?

Hand Simulation Multiprocessor Example

- Inner loop looks like this:

Thread A		Thread B	
`r1=0`	`load r1, M[i]`		
		`r1=0`	`load r1, M[i]`
`r1=1`	`add  r1, r1, 1`		
		`r1=-1`	`sub r1, r1, 1`
`M[i]=1`	`store r1, M[i]`		
		`M[i]=-1`	`store r1, M[i]`

- Hand Simulation:
  - And we're off. A gets off to an early start
  - B says "hmph, better go fast" and tries really hard
  - A goes ahead and writes "1"
  - B goes and writes "-1"
  - A says "HUH??? I could have sworn I put a 1 there"
- Could this happen on a uniprocessor?
  - Yes! Unlikely, but if you are depending on it not happening, it will and your system will break…

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 5% 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:

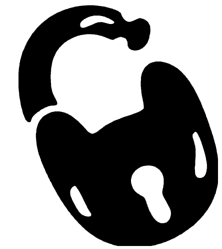
Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Definitions

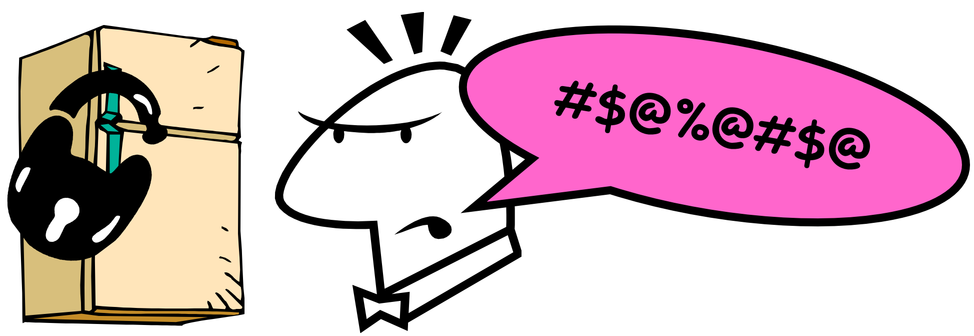
- Synchronization: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic
  - We are going to show that its hard to build anything useful with only reads and writes
- Mutual Exclusion: ensuring that only one thread does a particular thing at a time
  - 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

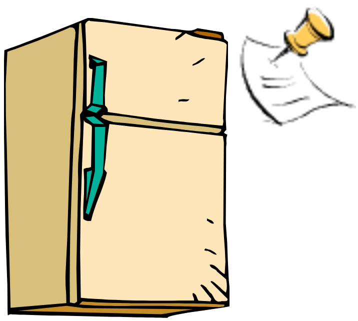
Too Much Milk: Correctness Properties

- 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;
   }
}
```



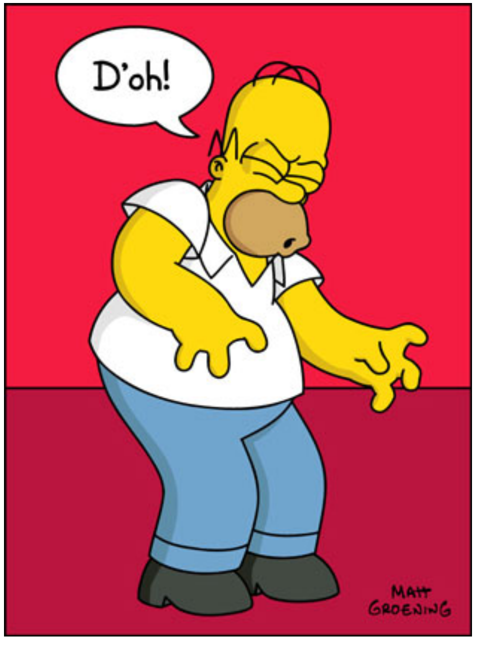
- Result?
  - Still too much milk but only occasionally!
  - Thread can get context switched after checking milk and note but before buying milk!
- Solution makes problem worse since fails intermittently
  - Makes it really hard to debug…
  - Must work despite what the dispatcher does!

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:

Thread A

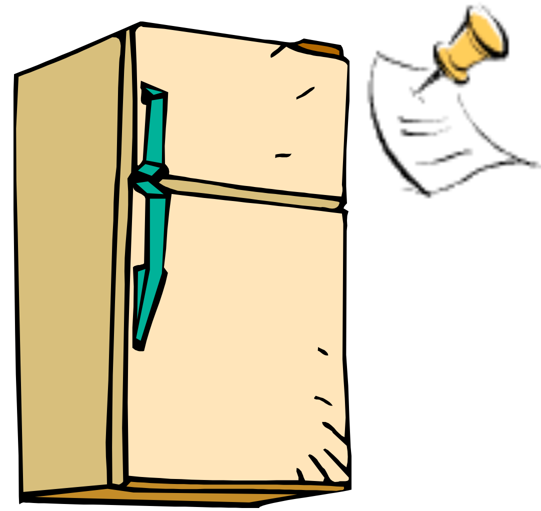
```
leave note A;
if (noNote B) {
   if (noMilk) {
      buy Milk;
   }
}
remove note A;
```

Thread B

```
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!"

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;
```

- 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
- Does this work? Yes. Both can guarantee that:
  – It is safe to buy, or
  – Other will buy, ok to quit

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

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"
- 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 (1 of 2)

- 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

Next Time

- How to build a lock
- Other synchronization primitives and techniques
- Parallel Programming Paradigms