

**CS162**  
**Operating Systems and**  
**Systems Programming**  
**Lecture 6**

**Concurrency (Continued),**  
**Synchronization (Start)**

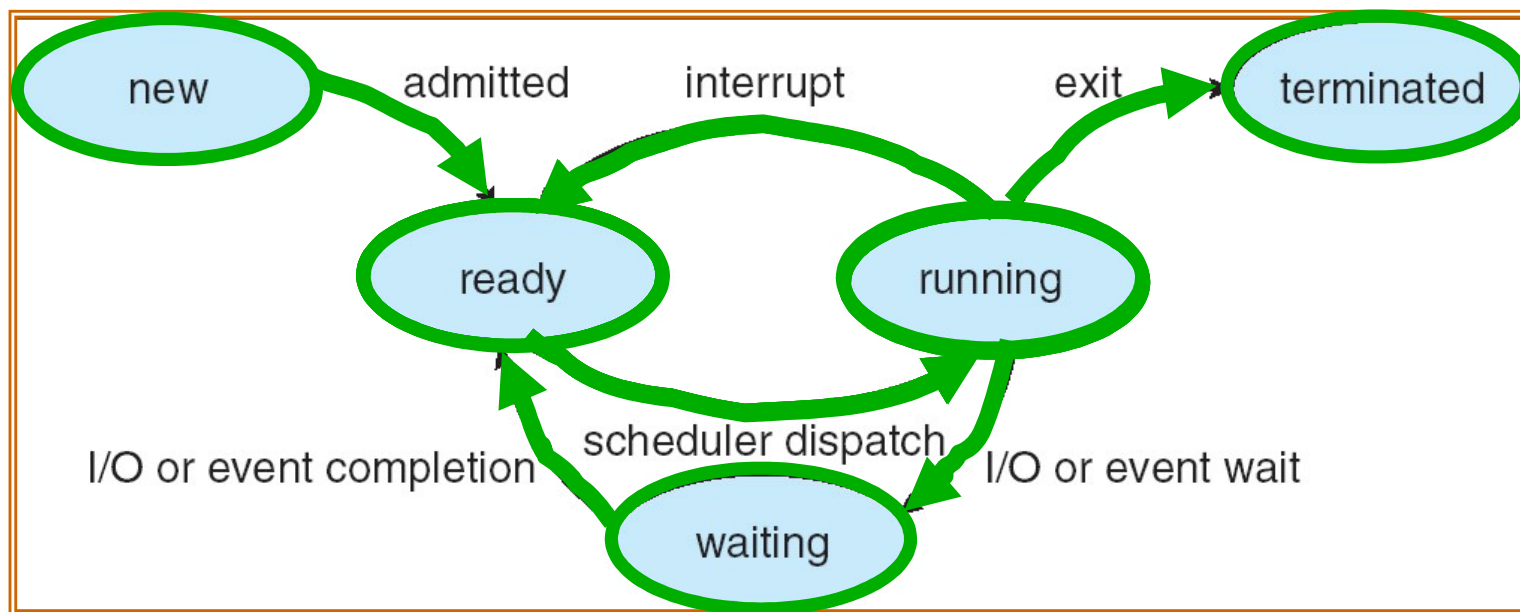
**February 8<sup>th</sup>, 2016**

**Prof. John Kubiatowicz**

**<http://cs162.eecs.Berkeley.edu>**

# Recall: Lifecycle of a Process

---



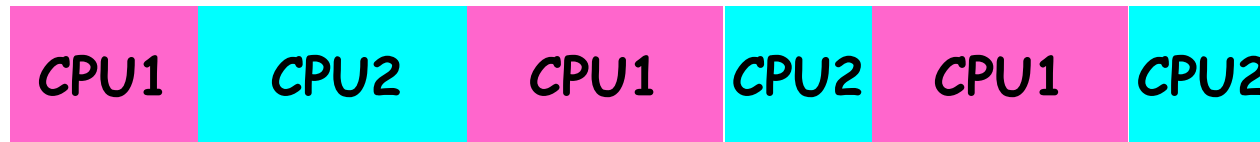
- As a process executes, it changes state:
  - **new**: The process is being created
  - **ready**: The process is waiting to run
  - **running**: Instructions are being executed
  - **waiting**: Process waiting for some event to occur
  - **terminated**: The process has finished execution

# Recall: Use of Threads

- Version of program with Threads (loose syntax):

```
main() {  
    ThreadFork (ComputePI ("pi.txt")) ;  
    ThreadFork (PrintClassList ("classlist.txt")) ;  
}
```

- What does “ThreadFork()” do?
  - Start independent thread running given procedure
- What is the behavior here?
  - Now, you would actually see the class list
  - This *should* behave as if there are two separate CPUs



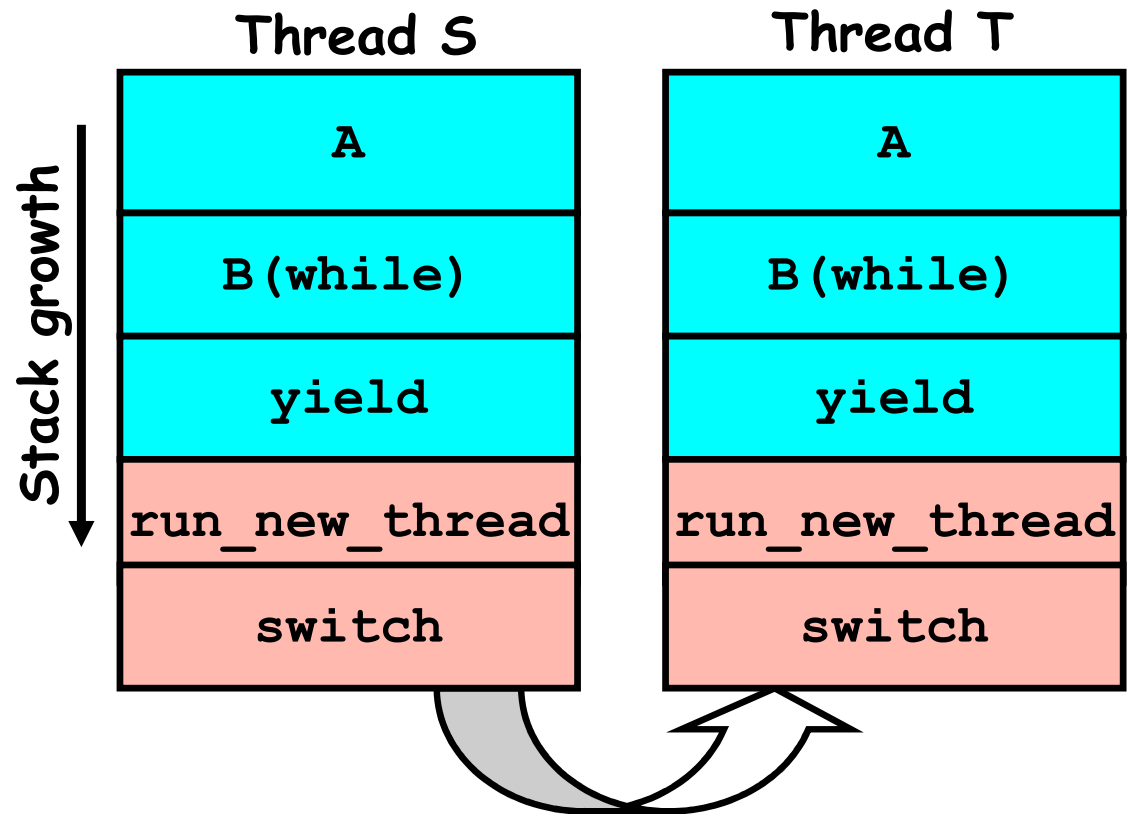
Time →

# Recall: Multithreaded stack switching

- Consider the following code blocks:

```
proc A() {  
    B();  
}  
  
proc B() {  
    while(TRUE) {  
        yield();  
    }  
}
```

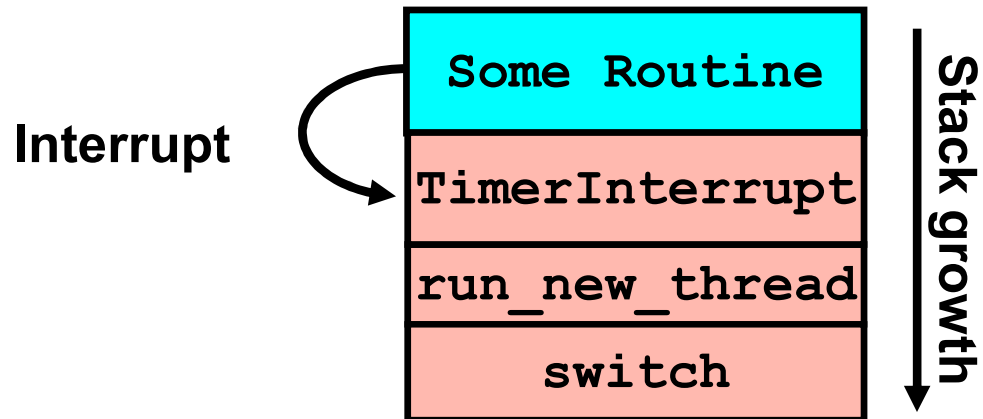
- Suppose we have 2 threads:
  - Threads S and T



# Recall: Use of Timer Interrupt to Return Control

---

- **Solution to our dispatcher problem**
  - Use the timer interrupt to force scheduling decisions



- **Timer Interrupt routine:**

```
TimerInterrupt() {  
    DoPeriodicHouseKeeping();  
    run_new_thread();  
}
```

# Per Thread Descriptor (Kernel Supported Threads)

---

- Each Thread has a **Thread Control Block (TCB)**
  - Execution State: CPU registers, program counter (PC), pointer to stack (SP)
  - Scheduling info: state, priority, CPU time
  - Various Pointers (for implementing scheduling queues)
  - Pointer to enclosing process (PCB) – user threads
  - Etc (add stuff as you find a need)
- OS Keeps track of TCBs in “kernel memory”
  - In Array, or Linked List, or ...
  - I/O state (file descriptors, network connections, etc)

# ThreadFork () : Create a New Thread

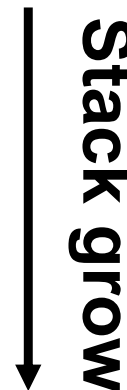
- **ThreadFork ()** is a user-level procedure that creates a new thread and places it on ready queue
  - We called this **CreateThread ()** earlier
- **Arguments to ThreadFork ()**
  - Pointer to application routine (**fcnPtr**)
  - Pointer to array of arguments (**fcnArgPtr**)
  - Size of stack to allocate
- **Implementation**
  - Sanity Check arguments
  - Enter Kernel-mode and Sanity Check arguments again
  - Allocate new Stack and TCB
  - Initialize TCB and place on ready list (Runnable)

# How do we initialize TCB and Stack?

---

- Initialize Register fields of TCB
  - Stack pointer made to point at stack
  - PC return address  $\Rightarrow$  OS (asm) routine ThreadRoot()
  - Two arg registers (a0 and a1) initialized to fcnPtr and fcnArgPtr, respectively
- Initialize stack data?
  - No. Important part of stack frame is in registers (ra)
  - Think of stack frame as just before body of ThreadRoot() really gets started

ThreadRoot stub



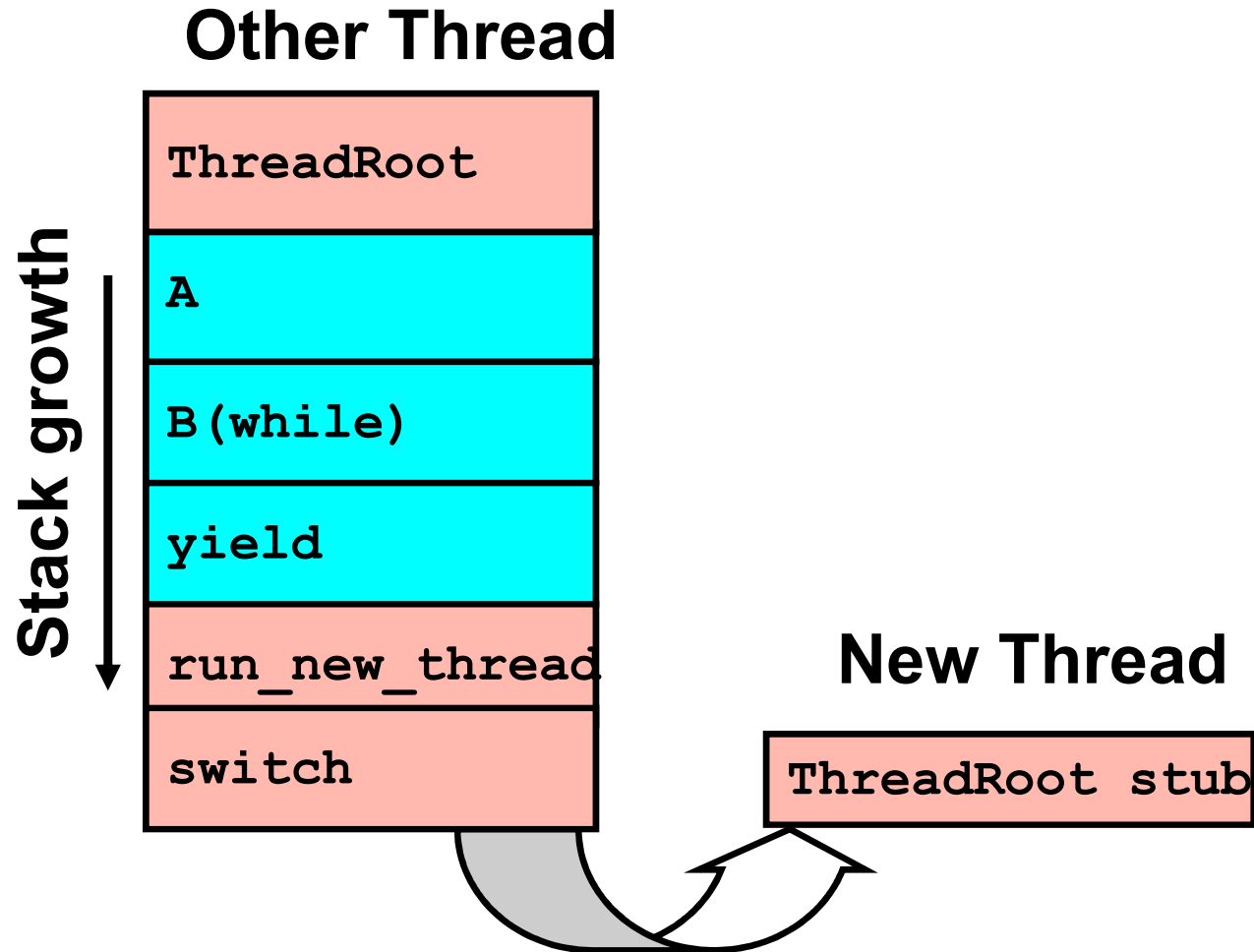
Stack growth

Initial Stack



# How does Thread get started?

---



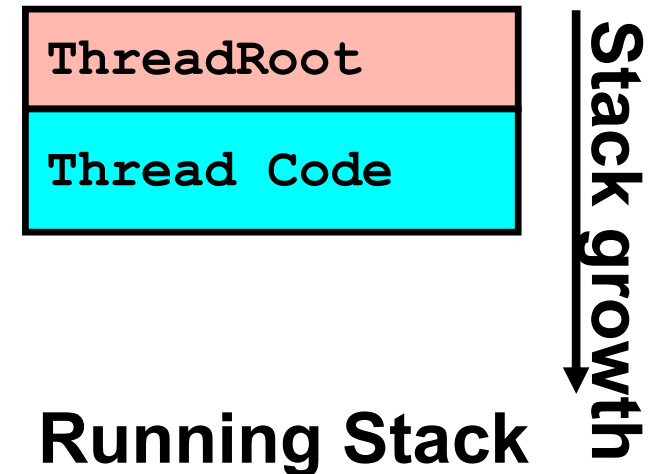
- Eventually, `run_new_thread()` will select this TCB and return into beginning of `ThreadRoot()`
  - This really starts the new thread

# What does ThreadRoot () look like?

- ThreadRoot () is the root for the thread routine:

```
ThreadRoot () {  
    DoStartupHousekeeping ();  
    UserModeSwitch (); /* enter user mode */  
    Call fcnPtr (fcnArgPtr);  
    ThreadFinish ();  
}
```

- Startup Housekeeping
  - Includes things like recording start time of thread
  - Other Statistics
- Stack will grow and shrink with execution of thread
- Final return from thread returns into ThreadRoot () which calls ThreadFinish ()
  - ThreadFinish () wake up sleeping threads



# Famous Quote wrt Scheduling: Dennis Richie

Dennis Richie,  
Unix V6, slp.c:

```
2230  /*
2231  * If the new process paused because it was
2232  * swapped out, set the stack level to the last call
2233  * to savu(u_ssav). This means that the return
2234  * which is executed immediately after the call to aretu
2235  * actually returns from the last routine which did
2236  * the savu.
2237  *
2238  * You are not expected to understand this.
2239  */
```

***“If the new process paused because it was swapped out, set the stack level to the last call to savu(u\_ssav). This means that the return which is executed immediately after the call to aretu actually returns from the last routine which did the savu.”***

***“You are not expected to understand this.”***

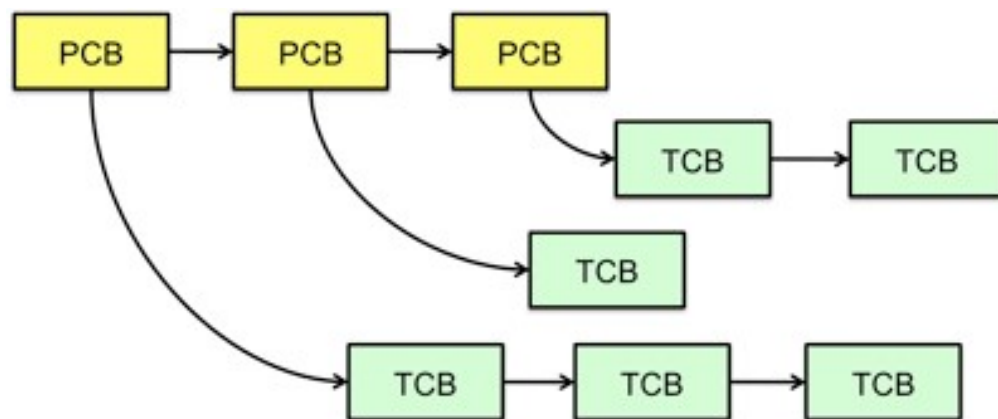
**Source: Dennis Ritchie, Unix V6 slp.c (context-switching code) as per The Unix Heritage Society(tuhs.org); gif by Eddie Koehler.**

**Included by Ali R. Butt in CS3204 from Virginia Tech**

# Multithreaded Processes

---

- **Process Control Block (PCBs) points to multiple Thread Control Blocks (TCBs):**



- **Switching threads within a block is a simple thread switch**
- **Switching threads across blocks requires changes to memory and I/O address tables**

# Administrivia

---

- **TA selection form**
  - Due tonight by midnight tonight !
  - We will try to do final section assignment tomorrow
- **Your section is your home for CS162**
  - The TA needs to get to know you to judge participation
  - All design reviews will be conducted by your TA
  - You can attend alternate section by same TA, but try to keep the amount of such cross-section movement to a minimum
- **Project #1: Starts today!**
- **HW1 due next Monday**
  - Submit via git

# Examples multithreaded programs

---

- **Embedded systems**
  - Elevators, Planes, Medical systems, Wristwatches
  - Single Program, concurrent operations
- **Most modern OS kernels**
  - Internally concurrent because have to deal with concurrent requests by multiple users
  - But no protection needed within kernel
- **Database Servers**
  - Access to shared data by many concurrent users
  - Also background utility processing must be done

# **Example multithreaded programs (con't)**

- **Network Servers**
  - Concurrent requests from network
  - Again, single program, multiple concurrent operations
  - File server, Web server, and airline reservation systems
- **Parallel Programming (More than one physical CPU)**
  - Split program into multiple threads for parallelism
  - This is called Multiprocessing
- **Some multiprocessors are actually uniprogrammed:**
  - Multiple threads in one address space but one program at a time

# A Typical Use Case

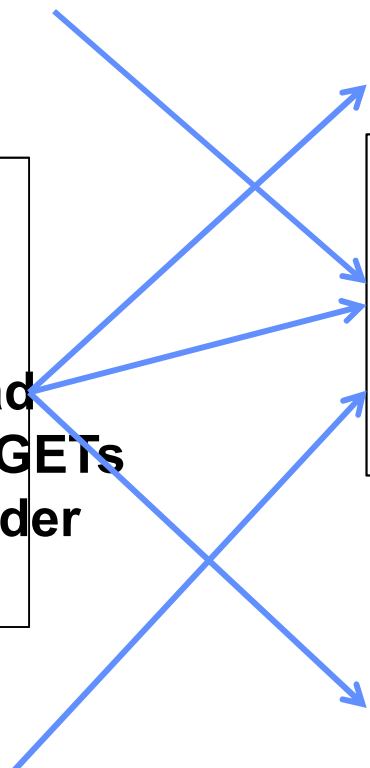
---

## Client Browser

- process for each tab
- thread to render page
- GET in separate thread
- multiple outstanding GETs
- as they complete, render portion

## Web Server

- fork process for each client connection
- thread to get request and issue response
- fork threads to read data, access DB, etc
- join and respond



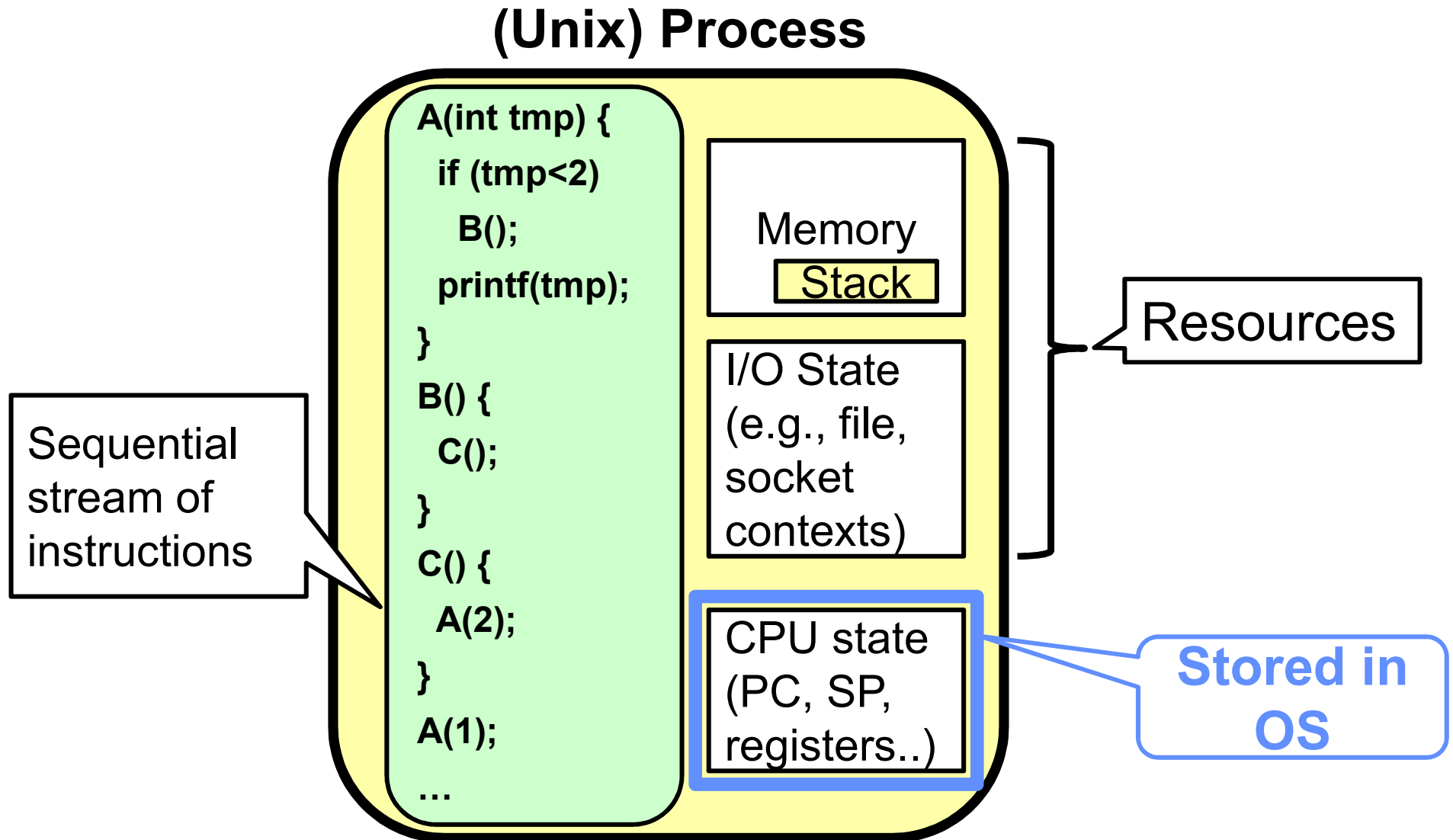


# Kernel Use Cases

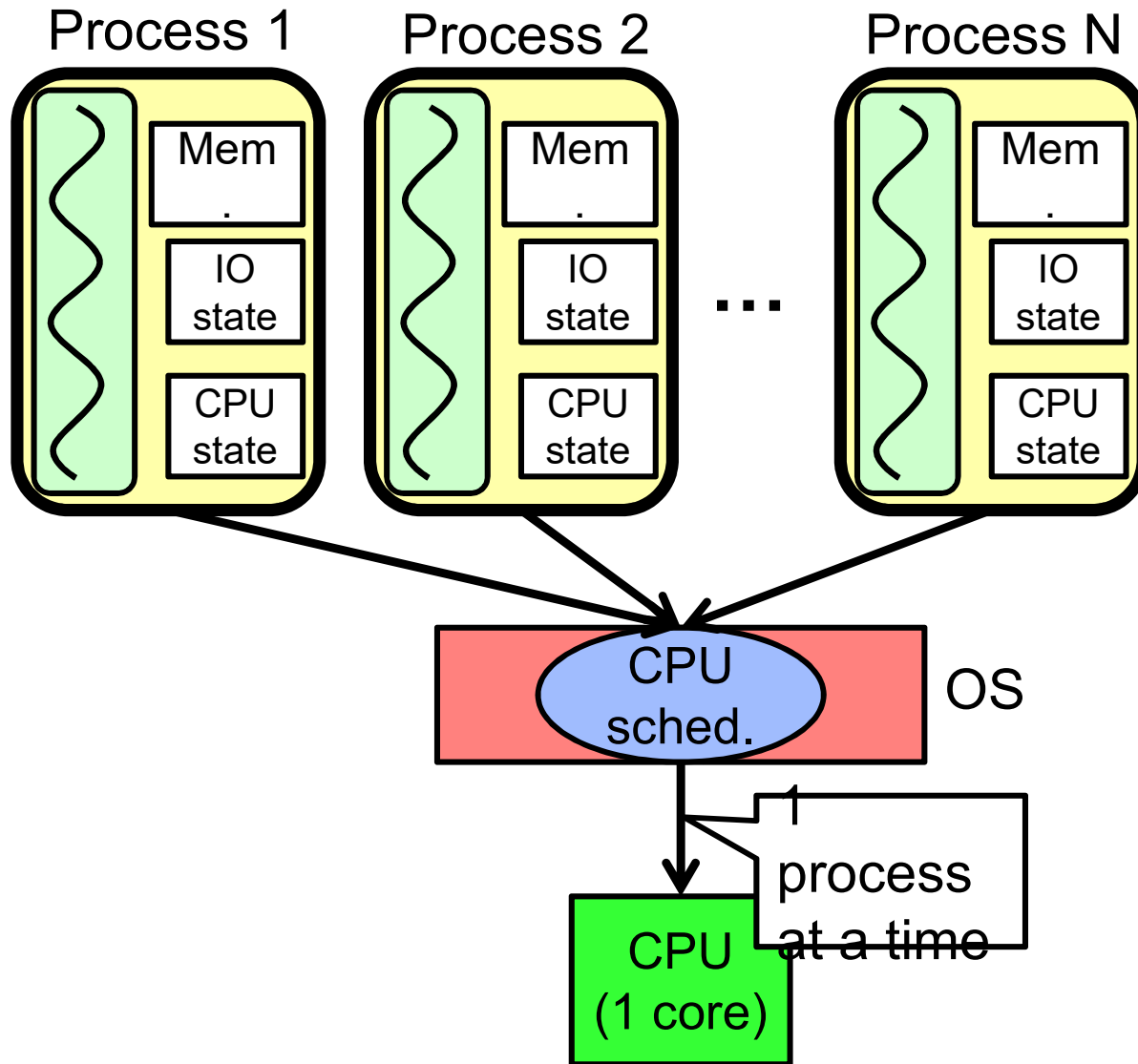
---

- Thread for each user process
- Thread for sequence of steps in processing I/O
- Threads for device drivers
- ...

# Putting it Together: Process

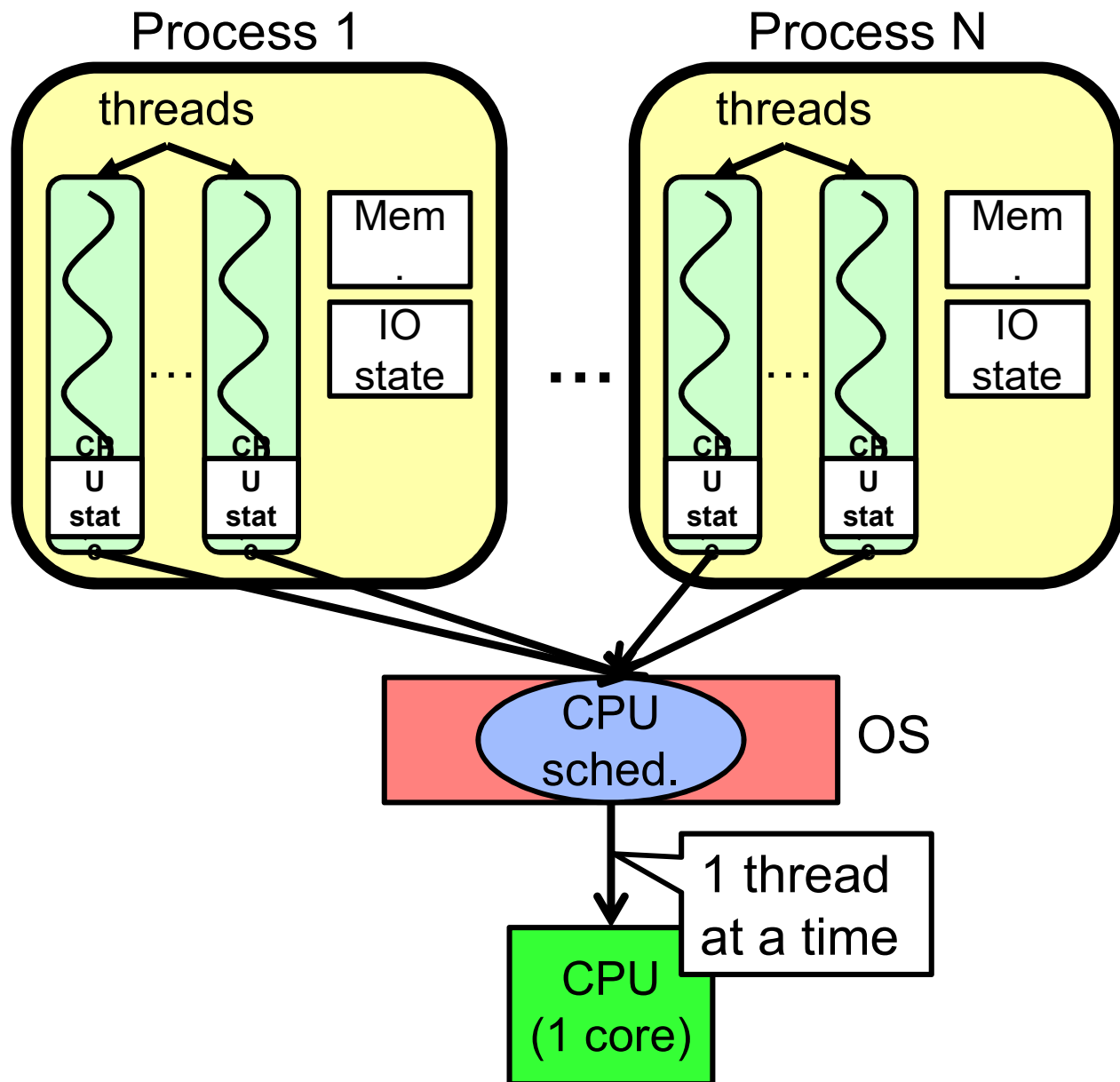


# Putting it Together: Processes



- **Switch overhead: high**
  - Kernel entry: **low (ish)**
  - CPU state: **low**
  - Memory/IO state: **high**
- **Process creation: high**
- **Protection**
  - CPU: **yes**
  - Memory/IO: **yes**
- **Sharing overhead: high** (involves at least a context switch)

# Putting it Together: Threads



- **Switch overhead: medium**
  - Kernel entry: **low(ish)**
  - CPU state: **low**
- **Thread creation: medium**
- **Protection**
  - CPU: **yes**
  - Memory/IO: **No**
- **Sharing overhead: low(ish)**  
(thread switch overhead low)

# Kernel versus User-Mode Threads

---

- **We have been talking about Kernel threads**
  - Native threads supported directly by the kernel
  - Every thread can run or block independently
  - One process may have several threads waiting on different things
- **Downside of kernel threads: a bit expensive**
  - Need to make a crossing into kernel mode to schedule
- **Lighter weight option: User Threads**

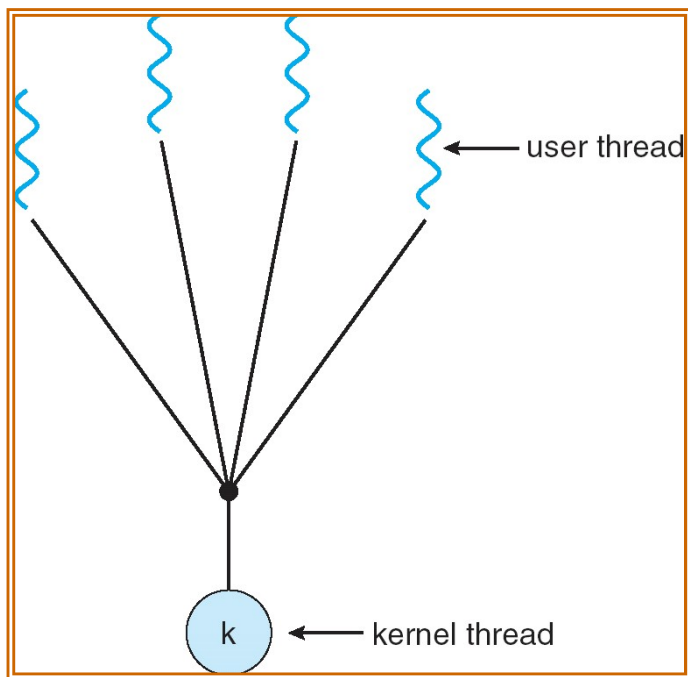
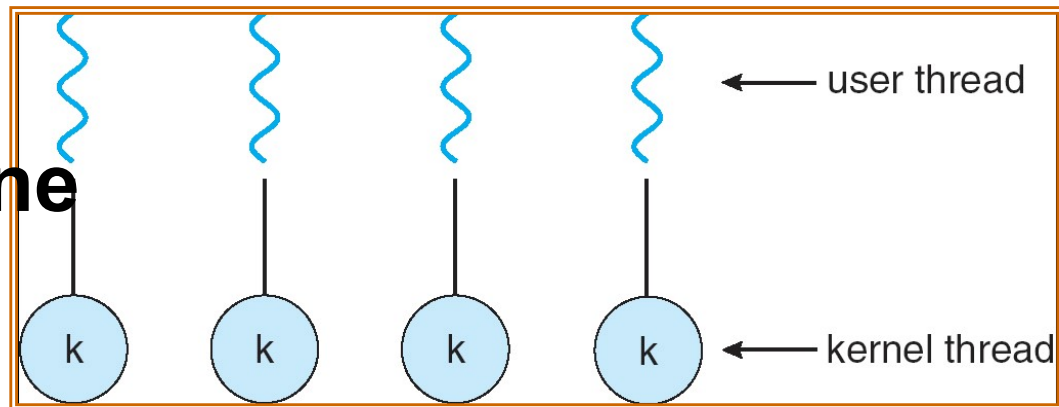
# User-Mode Threads

---

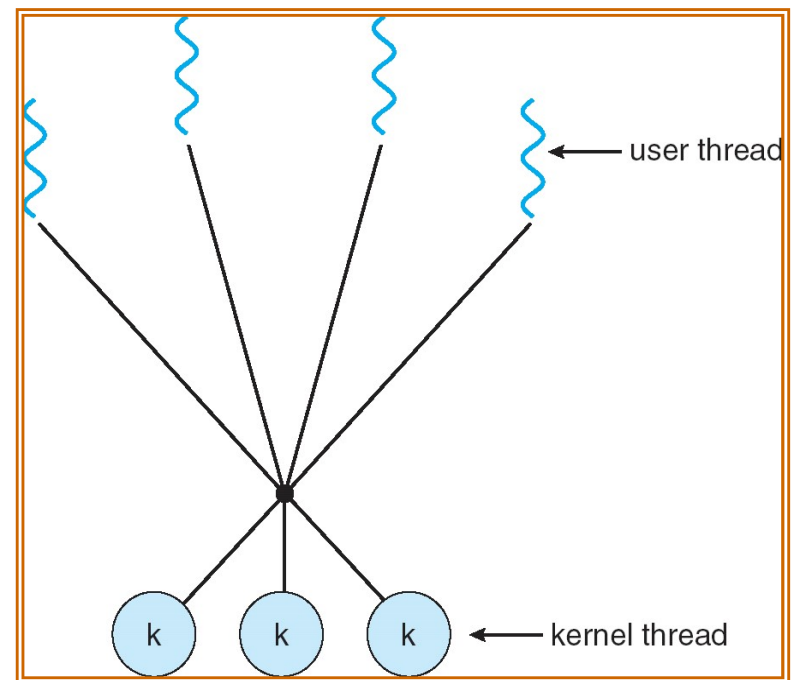
- **Lighter weight option:**
  - User program provides scheduler and thread package
  - May have several user threads per kernel thread
  - User threads may be scheduled non-preemptively relative to each other (only switch on `yield()`)
  - Cheap
- **Downside of user threads:**
  - When one thread blocks on I/O, all threads block
  - Kernel cannot adjust scheduling among all threads
  - Option: *Scheduler Activations*
    - » Have kernel inform user level when thread blocks...

# Some Threading Models

## Simple One-to-One Threading Model



## Many-to-One



## Many-to-Many

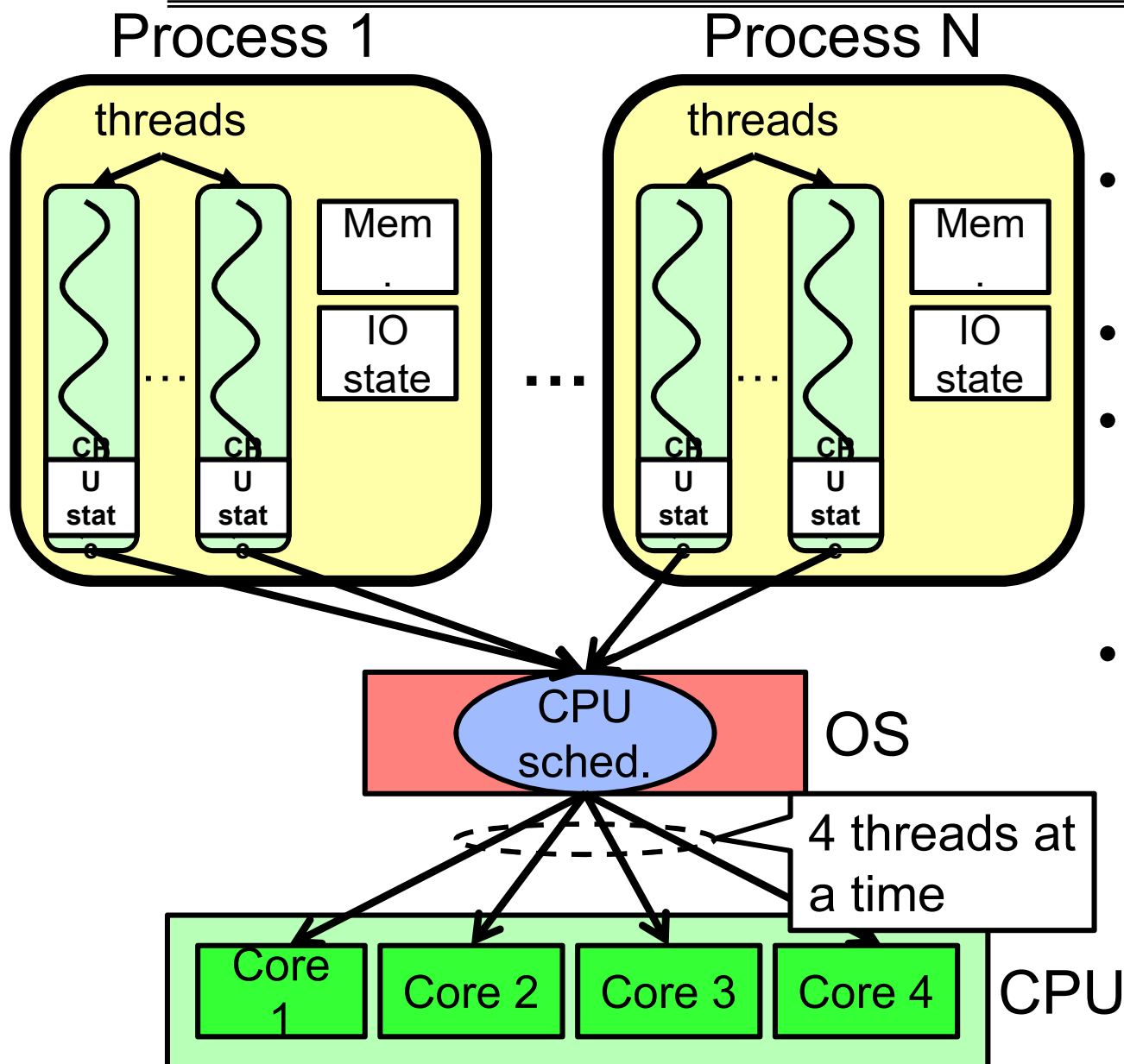
# Threads in a Process

---

- **Threads are useful at user-level: Parallelism, hide I/O latency, interactivity**
- **Option A (early Java): user-level library, within a single-threaded process**
  - Library does thread context switch
  - Kernel time slices between processes, e.g., on system call I/O
- **Option B (SunOS, Linux/Unix variants): green Threads**
  - User-level library does thread multiplexing
- **Option C (Windows): scheduler activations**
  - Kernel allocates processors to user-level library
  - Thread library implements context switch
  - System call I/O that blocks triggers upcall
- **Option D (Linux, MacOS, Windows): use kernel threads**
  - System calls for thread fork, join, exit (and lock, unlock,...)
  - Kernel does context switching
  - Simple, but a lot of transitions between user and kernel mode

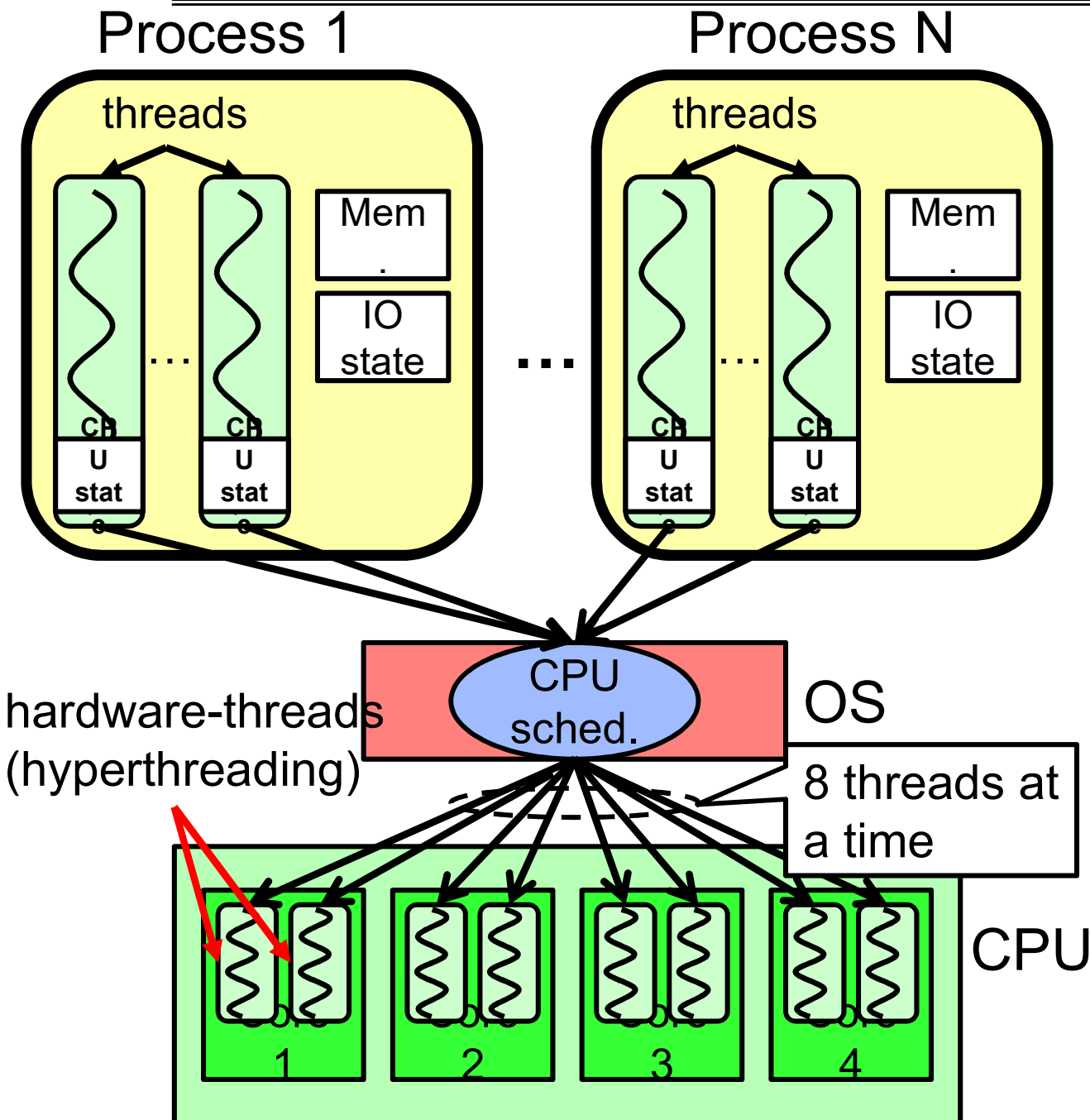


# Putting it Together: Multi-Cores



- **Switch overhead: low** (only CPU state)
- **Thread creation: low**
- **Protection**
  - CPU: **yes**
  - Memory/IO: **No**
- **Sharing overhead: low** (thread switch overhead low, may not need to switch at all!)

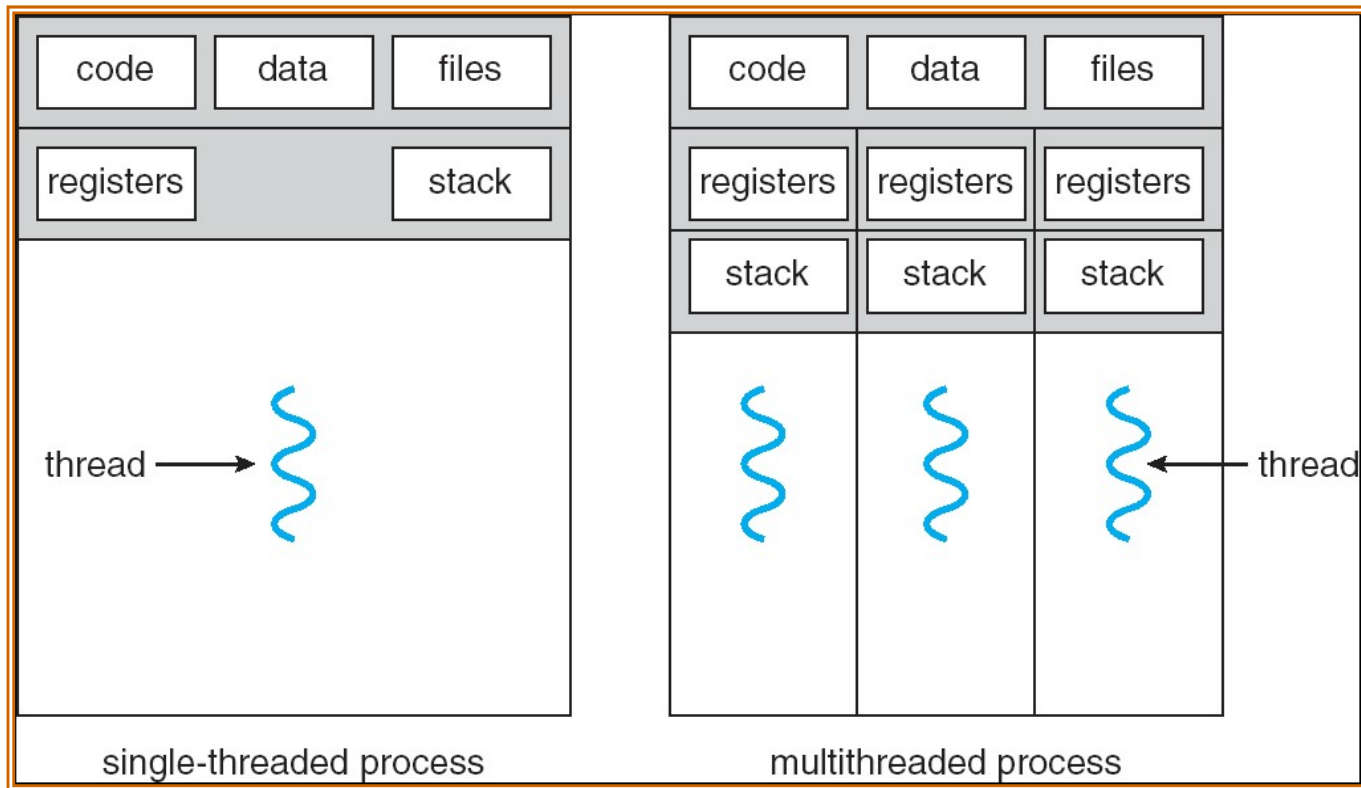
# Putting it Together: Hyper-Threading



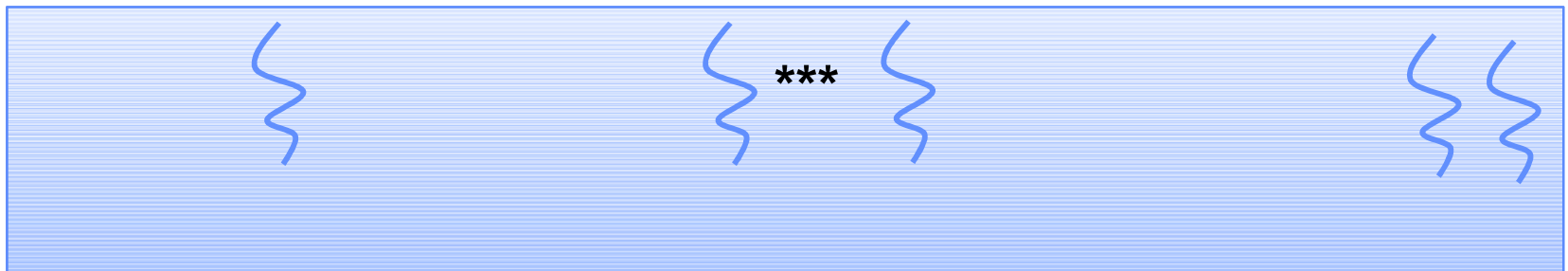
- Switch overhead between hardware-threads: **very-low** (done in hardware)
- Contention for **ALUs/FPU**s may hurt performance

# Supporting 1T and MT Processes

User

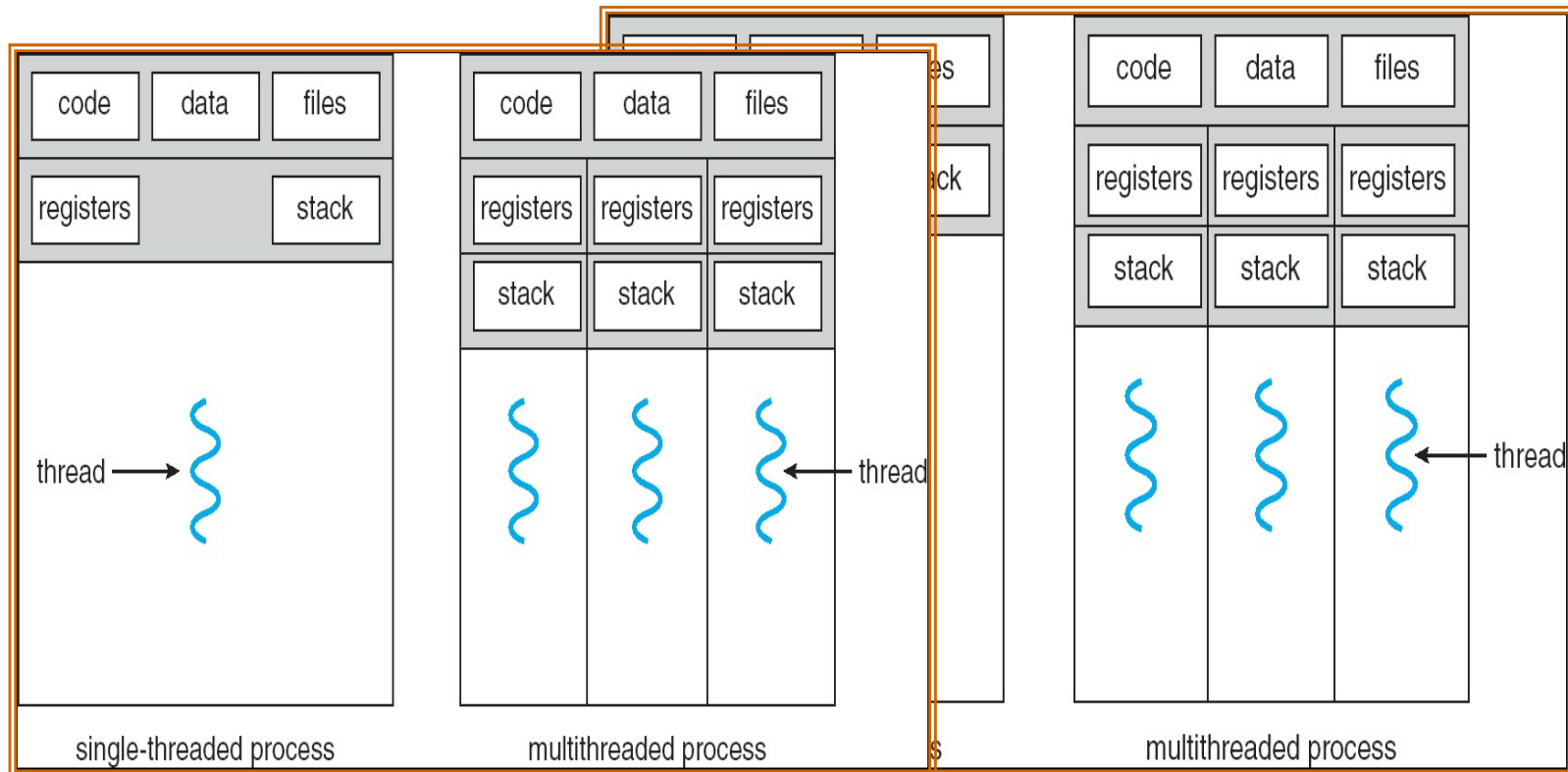


System



# Supporting 1T and MT Processes

User



System



# Classification

| # threads<br># of<br>addr<br>spaces: | One  | Many  |
|--------------------------------------|--|---|
| Per AS:<br>One                       | MS/DOS, early<br>Macintosh   | Traditional UNIX  |
| Many                                 | Embedded systems<br>(Geoworks, VxWorks,<br>JavaOS, etc)<br>JavaOS, Pilot(PC) | Mach, OS/2, Linux<br>Windows 9x???<br>Win NT to XP, Solaris,<br>HP-UX, OS X |

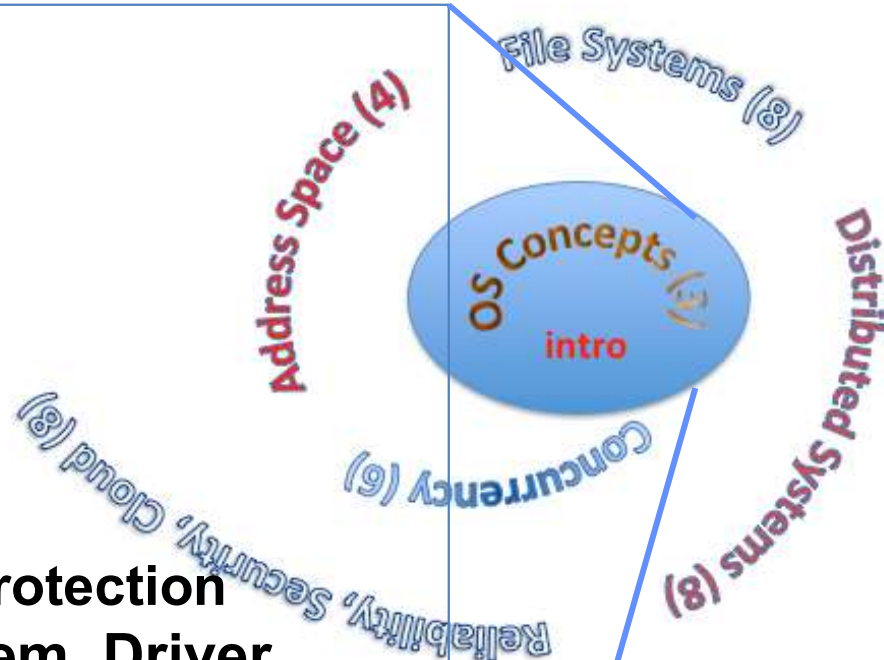
- Real operating systems have either
  - One or many address spaces
  - One or many threads per address space
- Did Windows 95/98/ME have real memory protection?
  - No: Users could overwrite process tables/System DLLs

---

# BREAK

# You are here... why?

- **Processes**
  - Thread(s) + address space
- **Address Space**
- **Protection**
- **Dual Mode**
- **Interrupt handlers**
  - Interrupts, exceptions, syscall
- **File System**
  - Integrates processes, users, cwd, protection
- **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**
- **Scheduling**



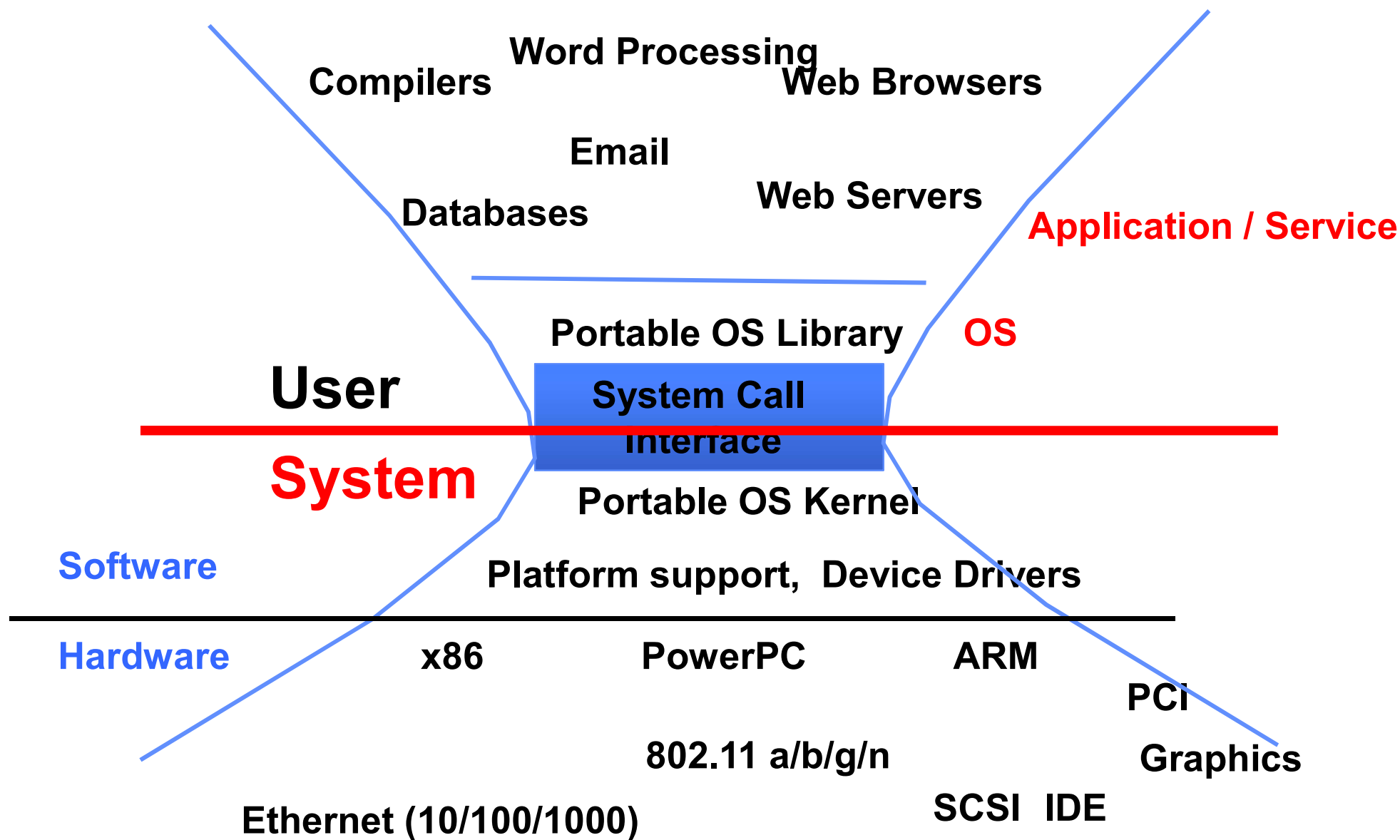
# Perspective on ‘groking’ 162

---

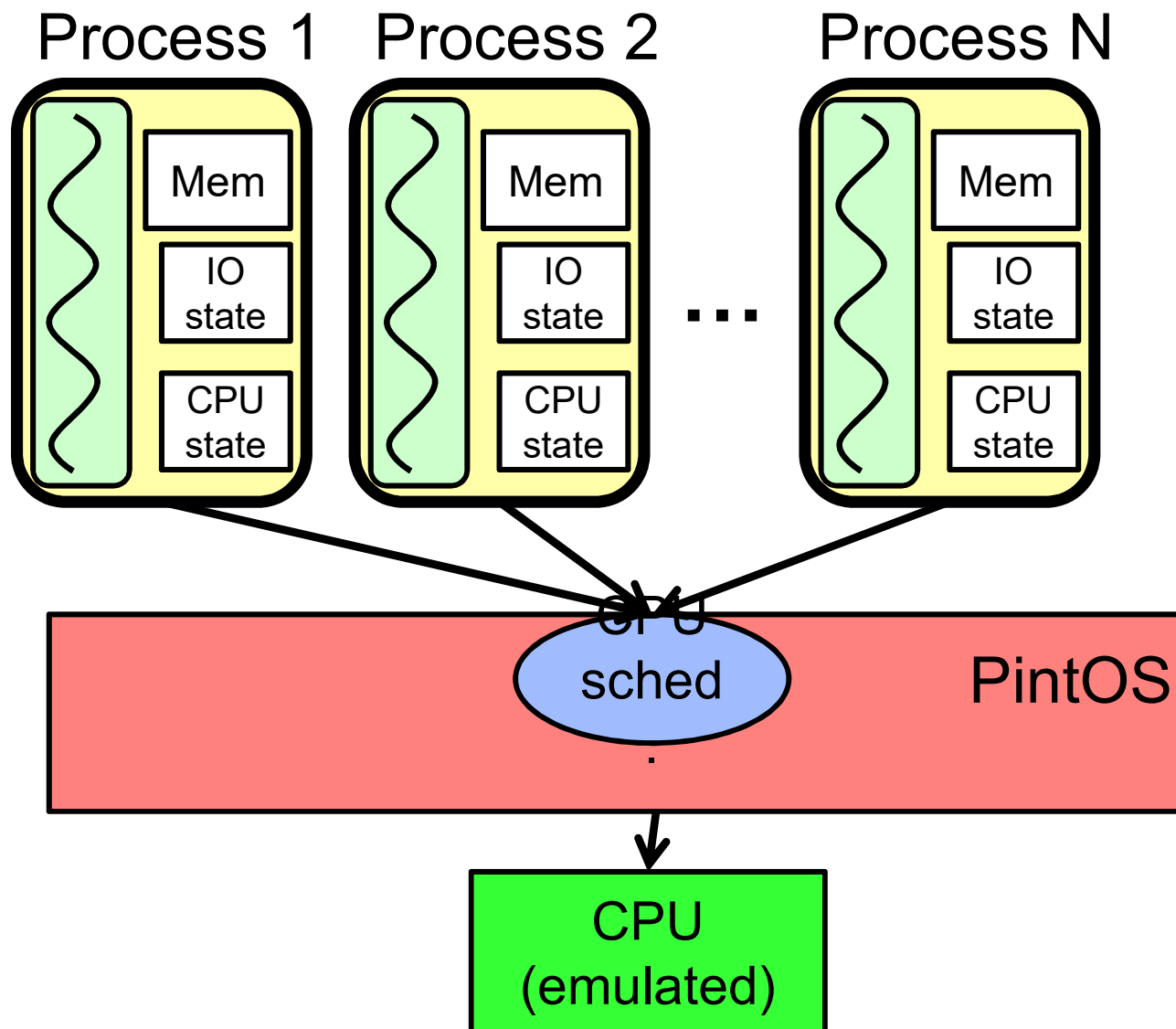
- **Historically, OS was the most complex software**
  - Concurrency, synchronization, processes, devices, communication, ...
  - Core systems concepts developed there
- **Today, many “applications” are complex software systems too**
  - These concepts appear there
  - But they are realized out of the 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**



# Operating System as Design

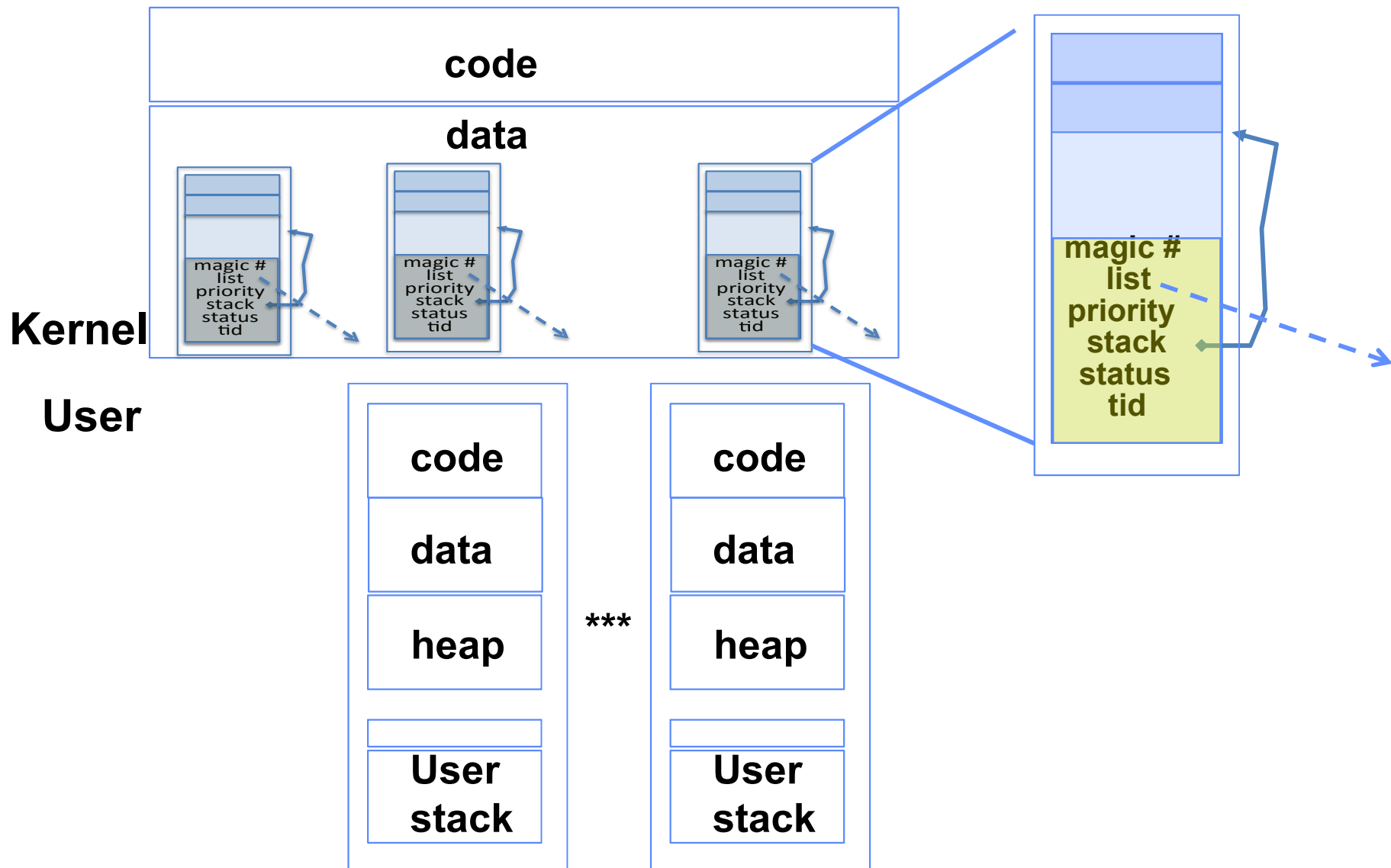


# Starting today: PintOS Projects



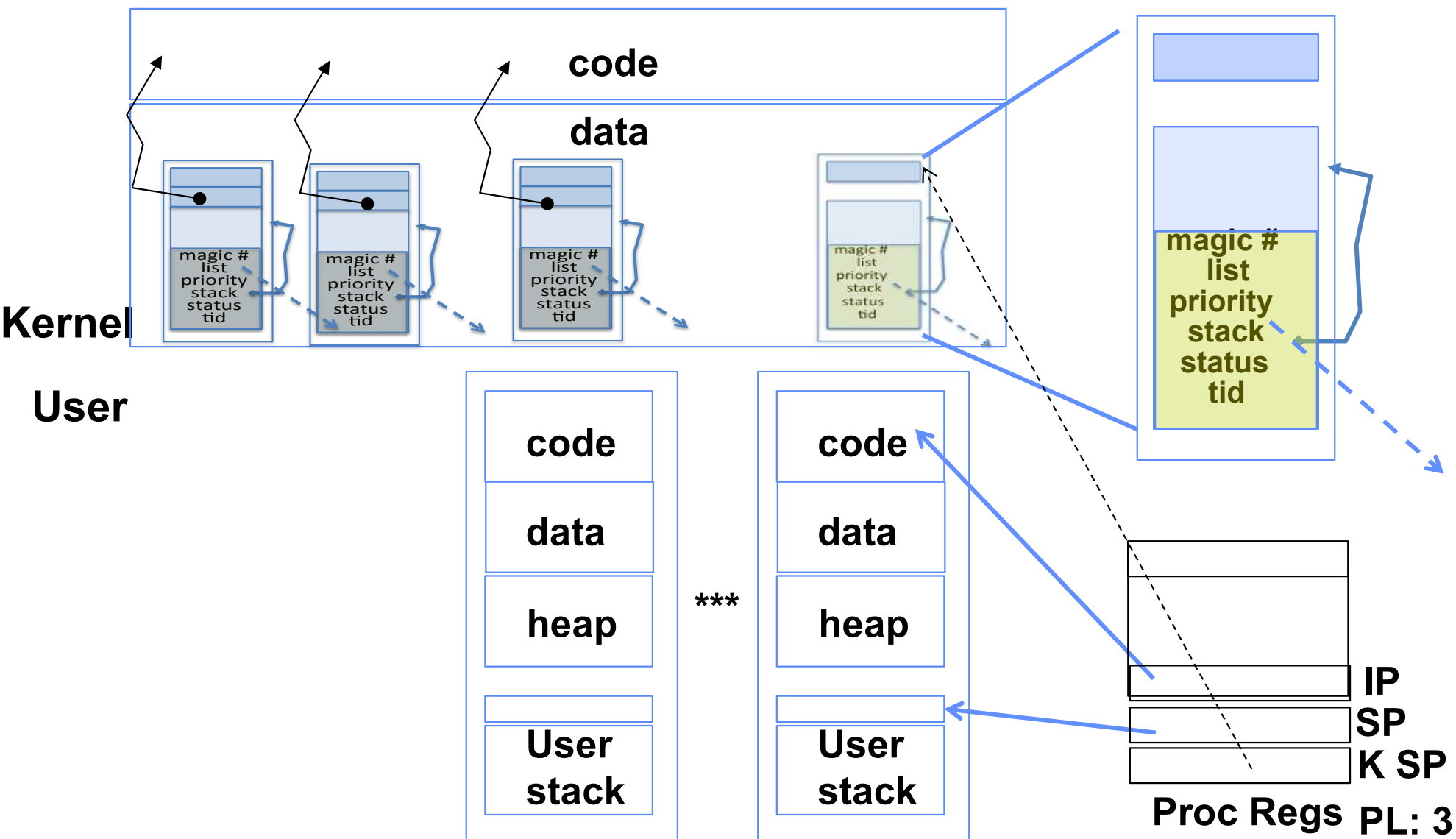
- **Groups almost all formed**
- **Work as one!**
- **10x homework**
- **P1: threads & scheduler**
- **P2: user process**
- **P3: file system**

# MT Kernel 1T Process ala Pintos/x86



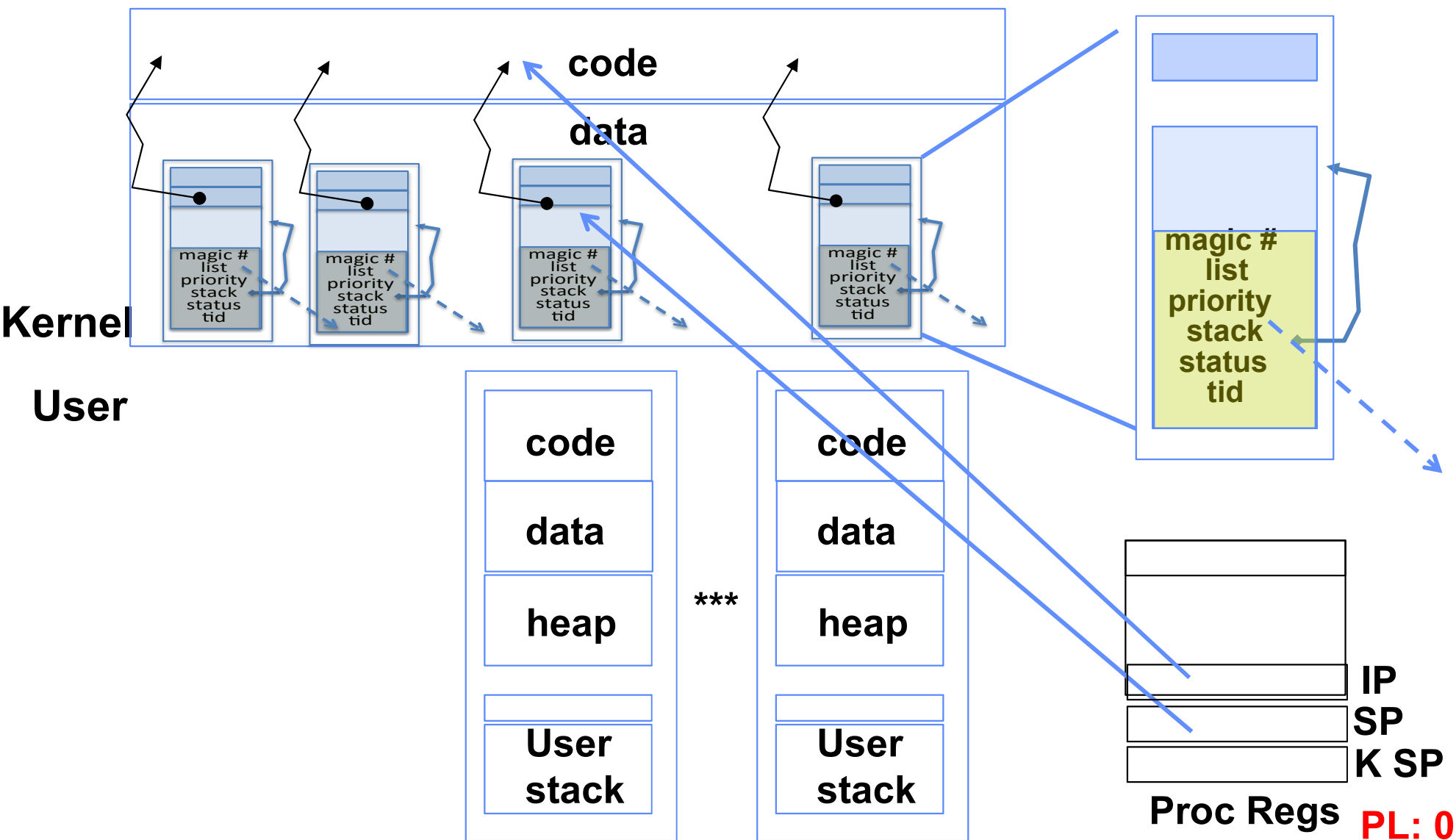
- Each user process/thread associated with a kernel thread, described by a 4kb Page object containing TCB and kernel stack for the kernel thread

# In User thread, w/ k-thread waiting



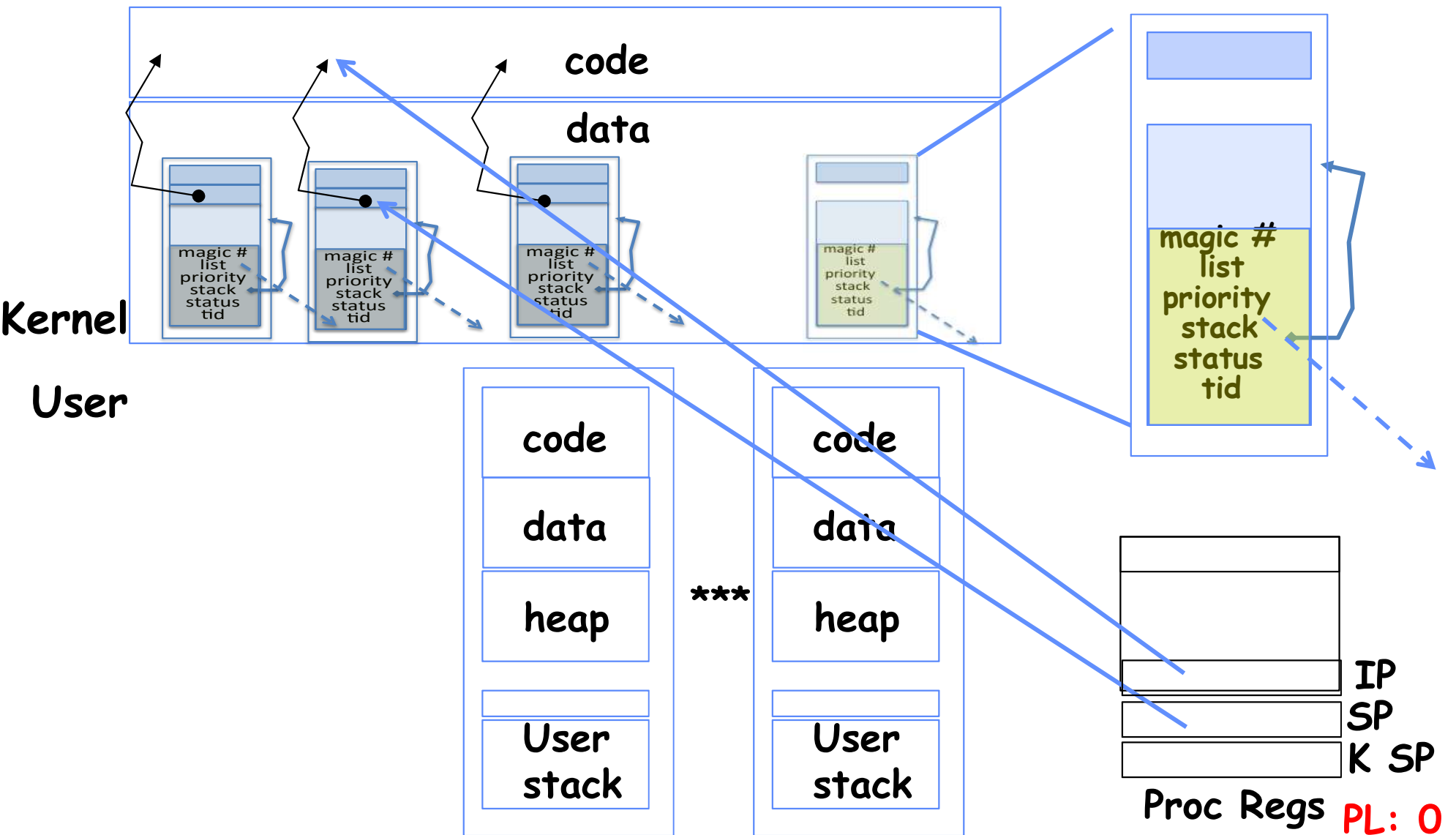
- x86 proc holds interrupt SP high system level
- During user thread exec, associate kernel thread is “standing by”

# In Kernel thread



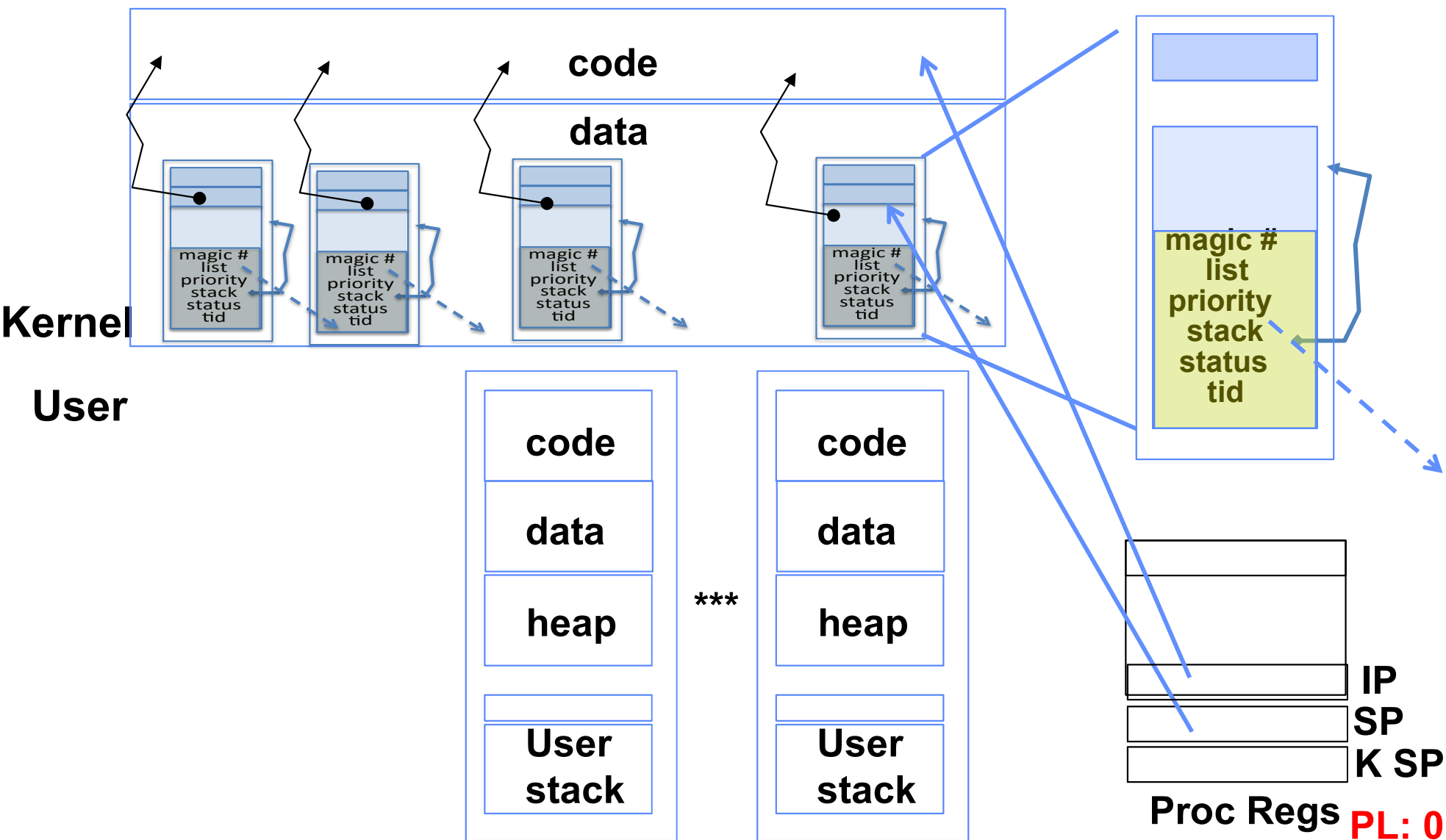
- Kernel threads execute with small stack in thread struct
- Scheduler selects among ready kernel and user threads

# Thread Switch (switch.S)

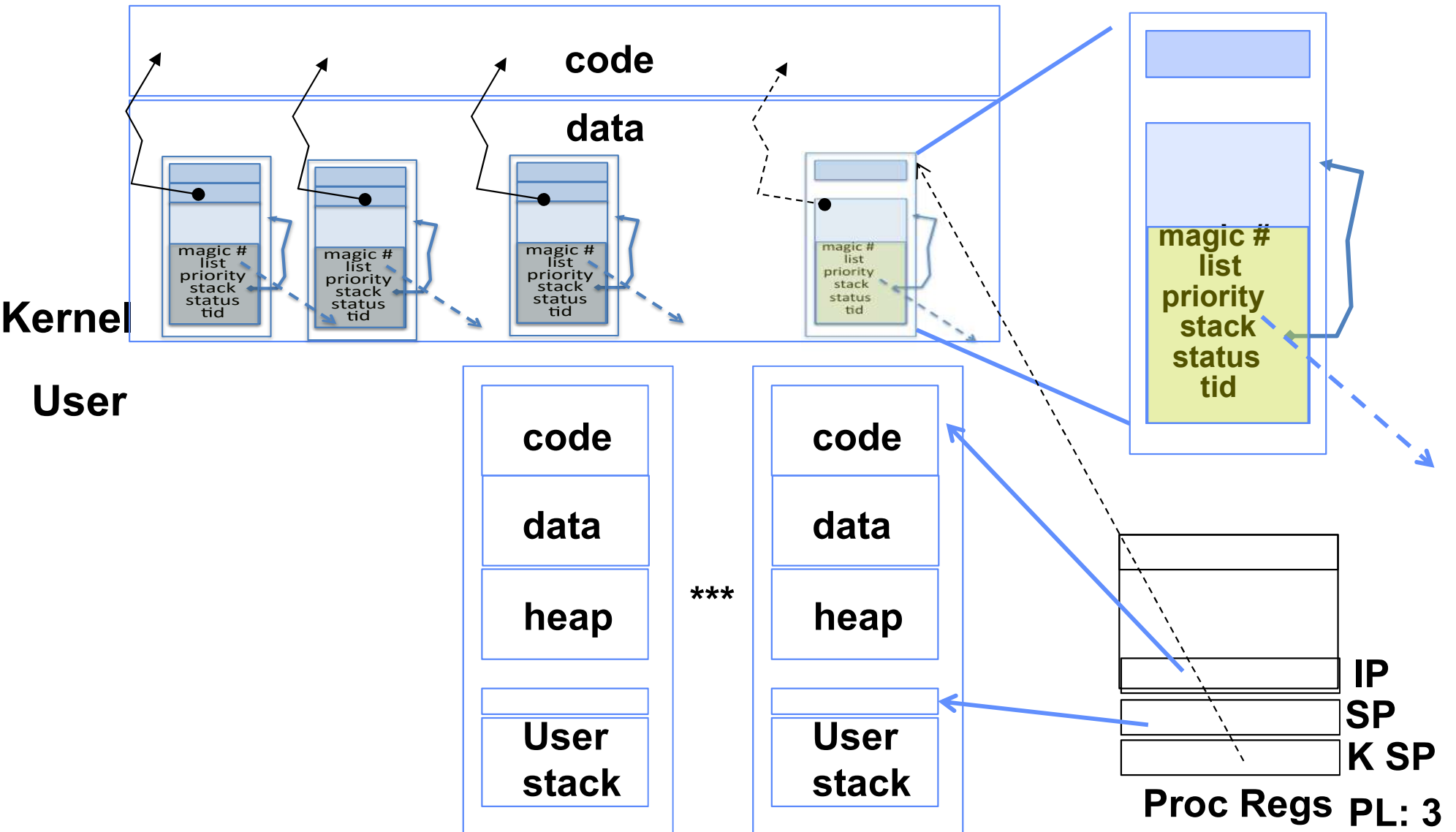


- **switch\_threads:** save regs on current small stack, change SP, return from destination threads call to switch\_threads

# Switch to Kernel Thread for Process



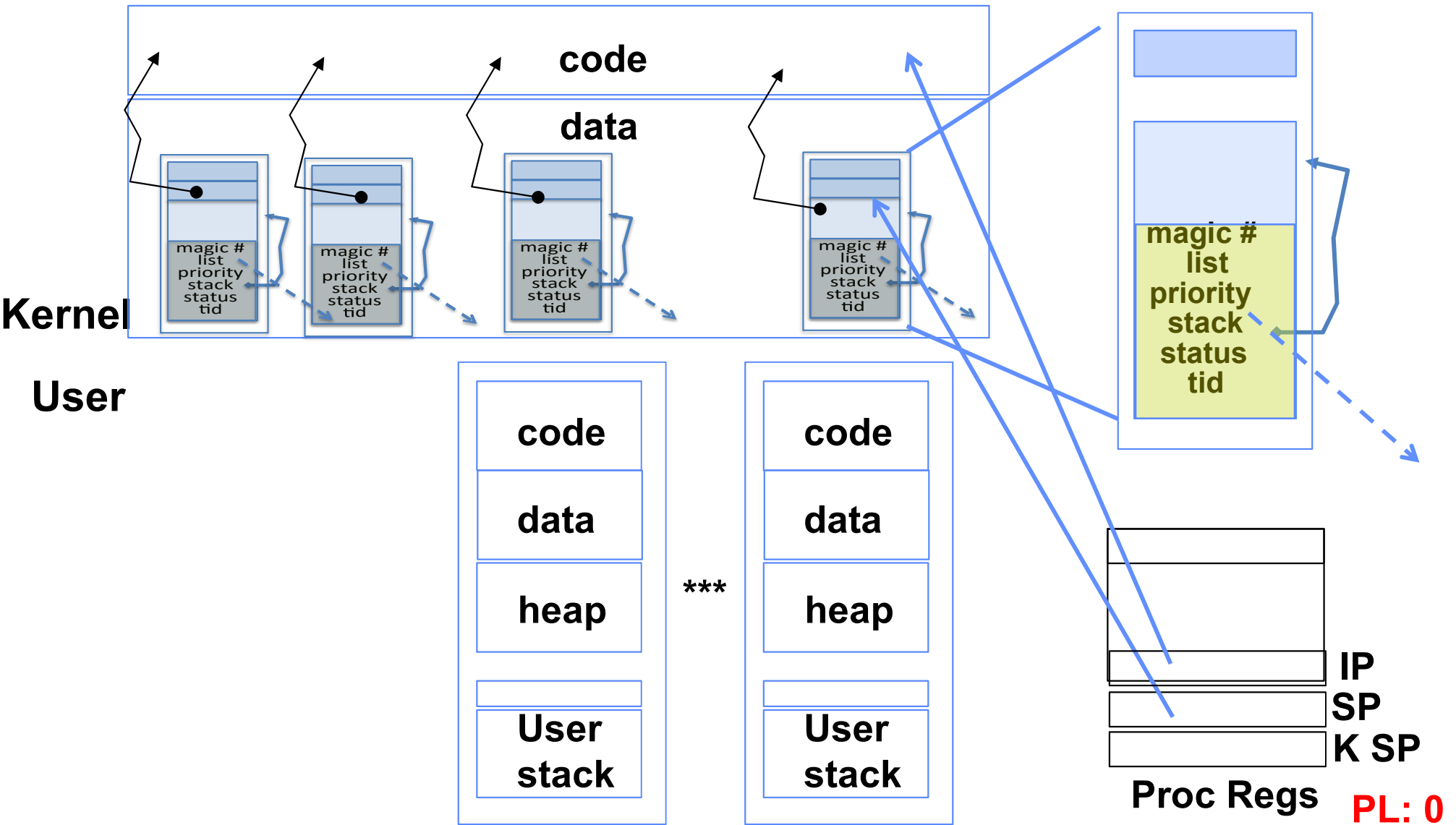
# Kernel->User



- **iret restores user stack and PL**

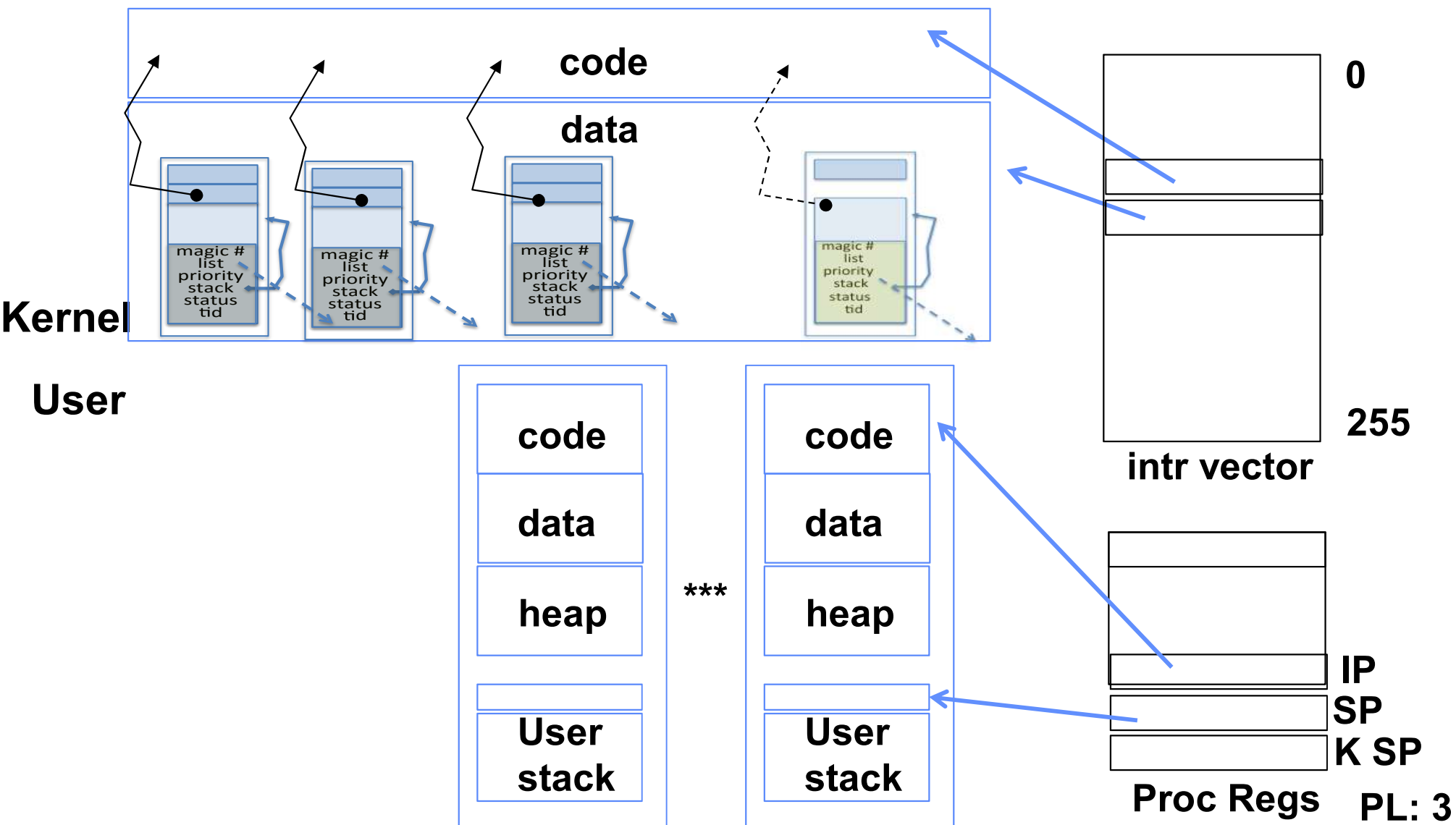


# User->Kernel



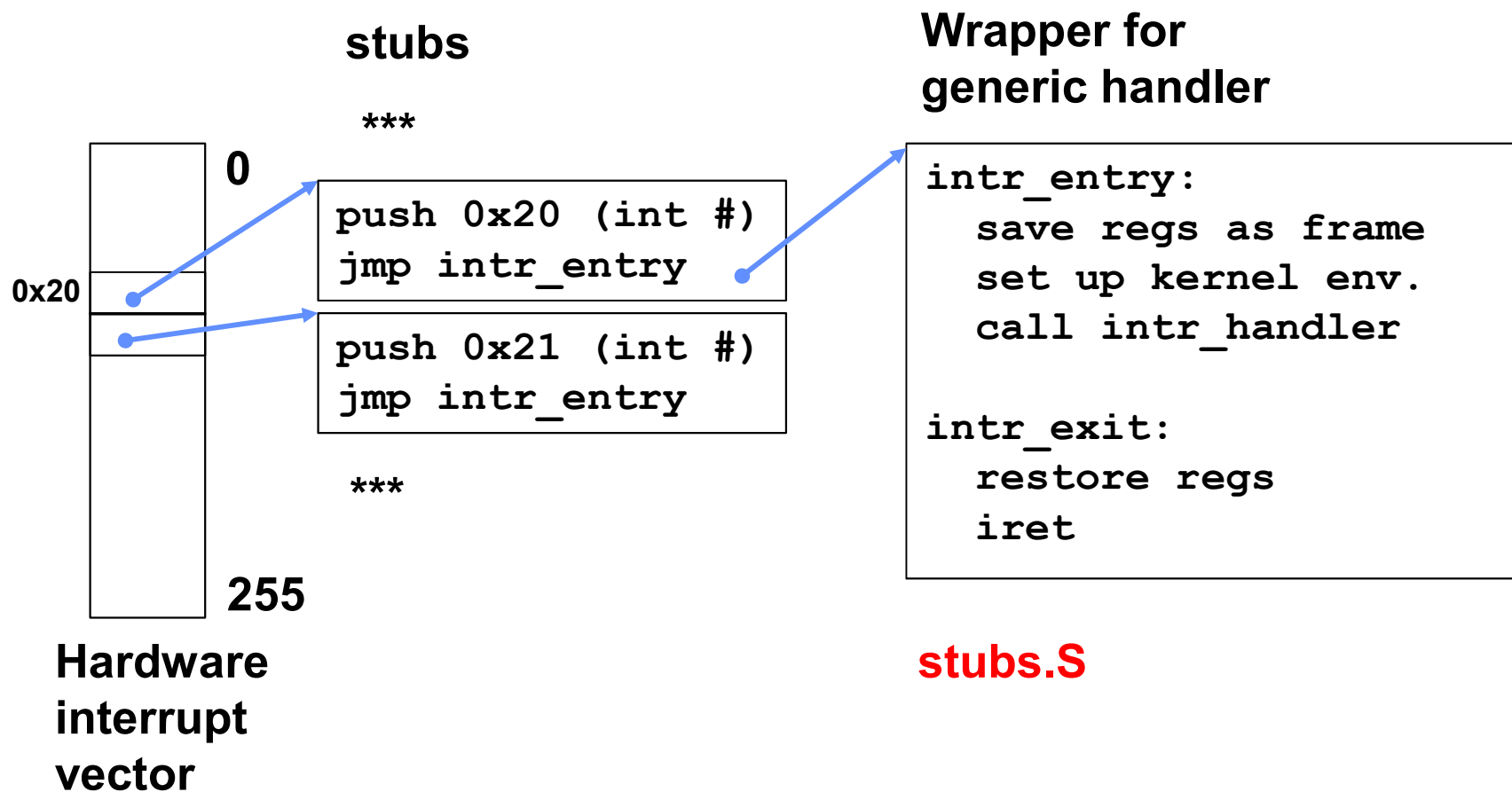
- Mechanism to resume k-thread goes through interrupt vector

# User->Kernel via interrupt vector



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

# Pintos Interrupt Processing



# Recall: cs61C THE STACK FRAME

## Basic Structure of a Function

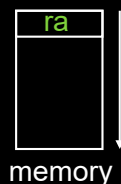
### Prologue

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

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

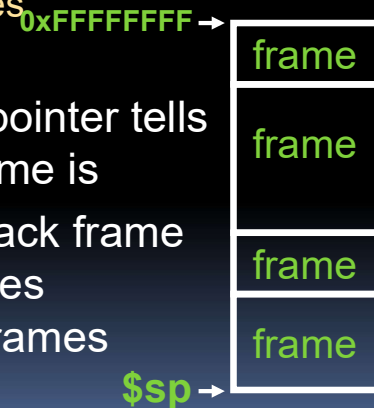
### Epilogue

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



## 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 L10 Introduction to MIPS : Procedures I (18)

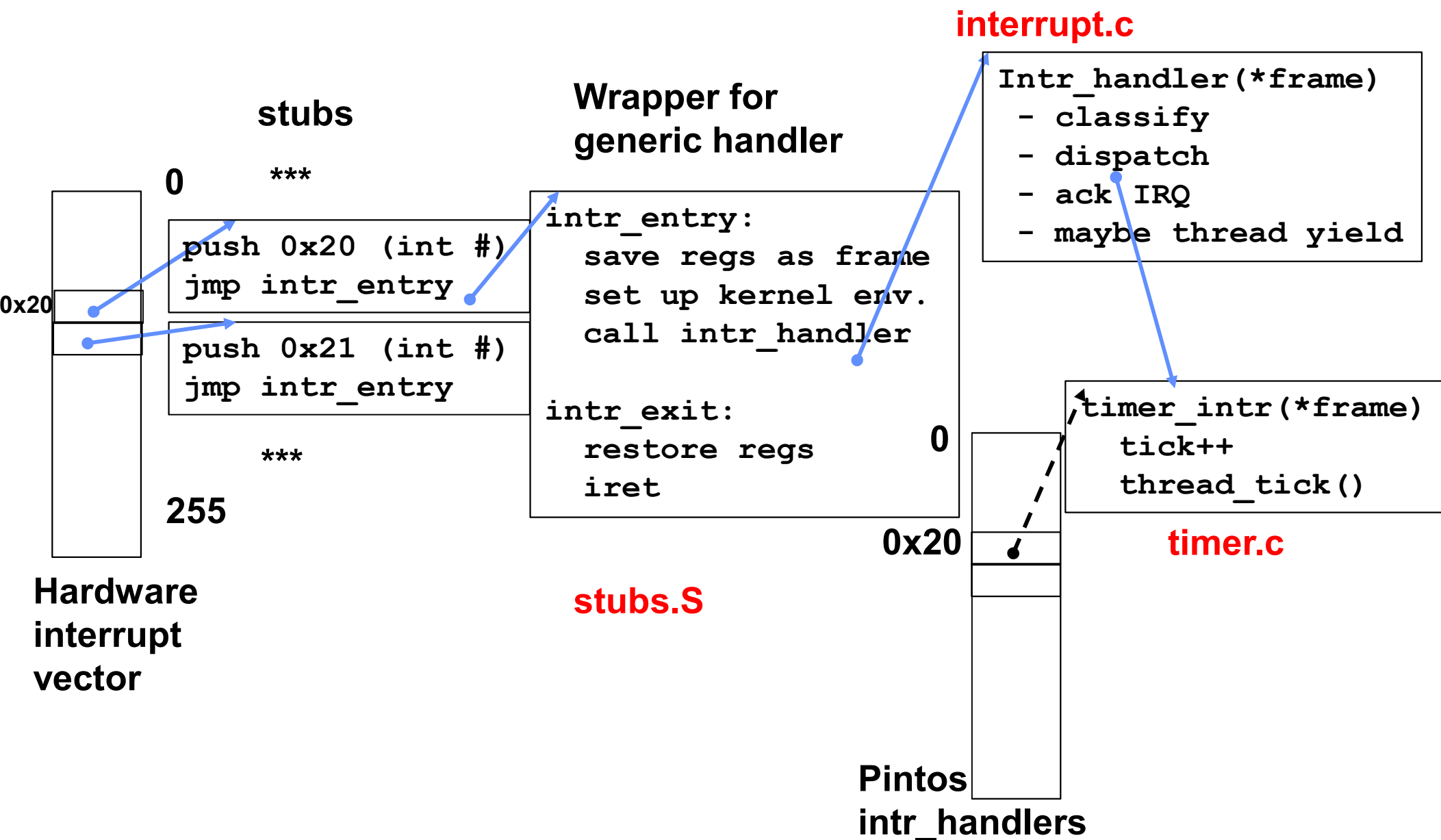
Garcia, Spring 2014



CS61C L11 Introduction to MIPS : Procedures II & Logical Ops (3)

Garcia, Spring 2014 © UCB

# Pintos Interrupt Processing

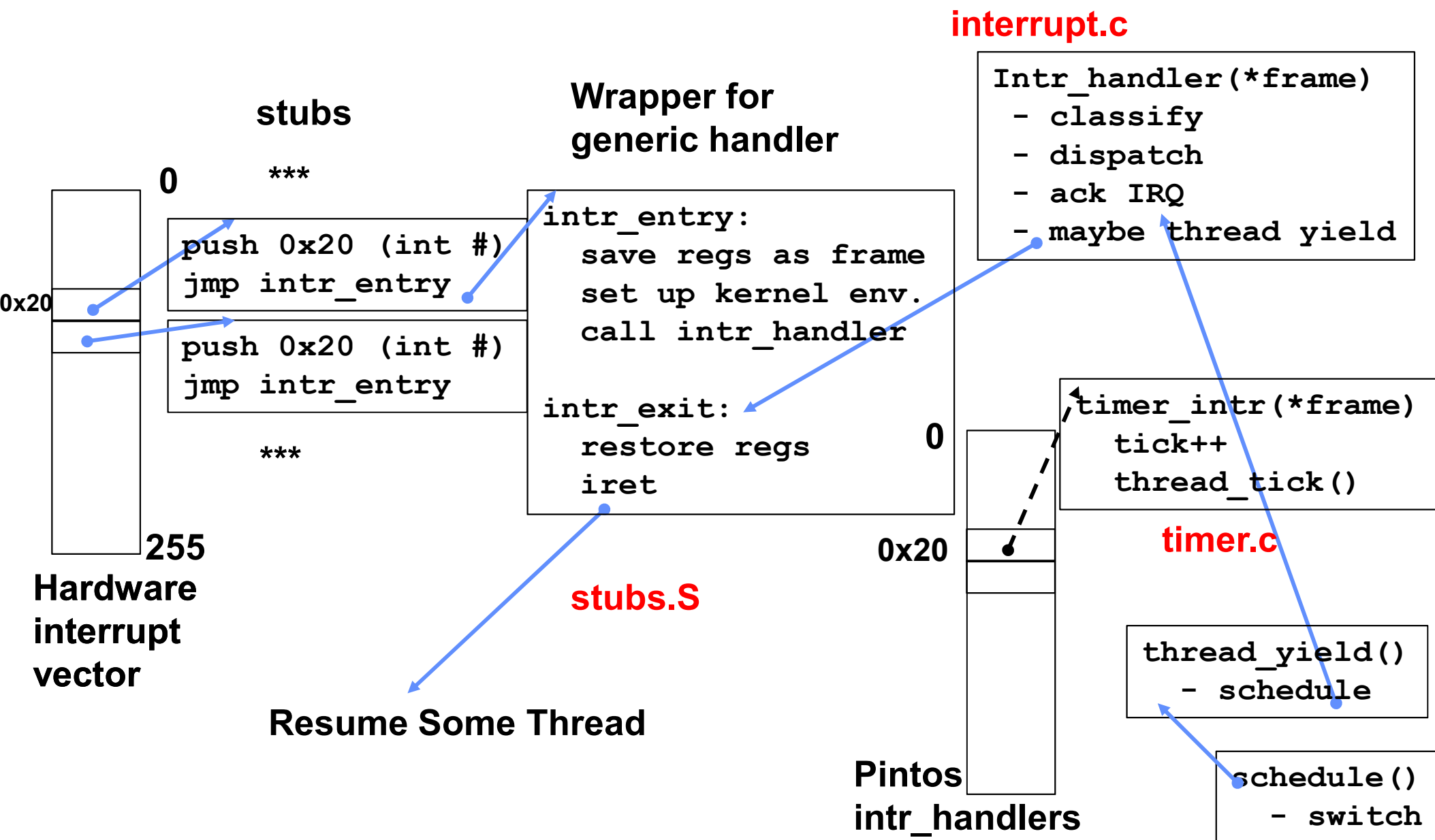


# Timer may trigger thread switch

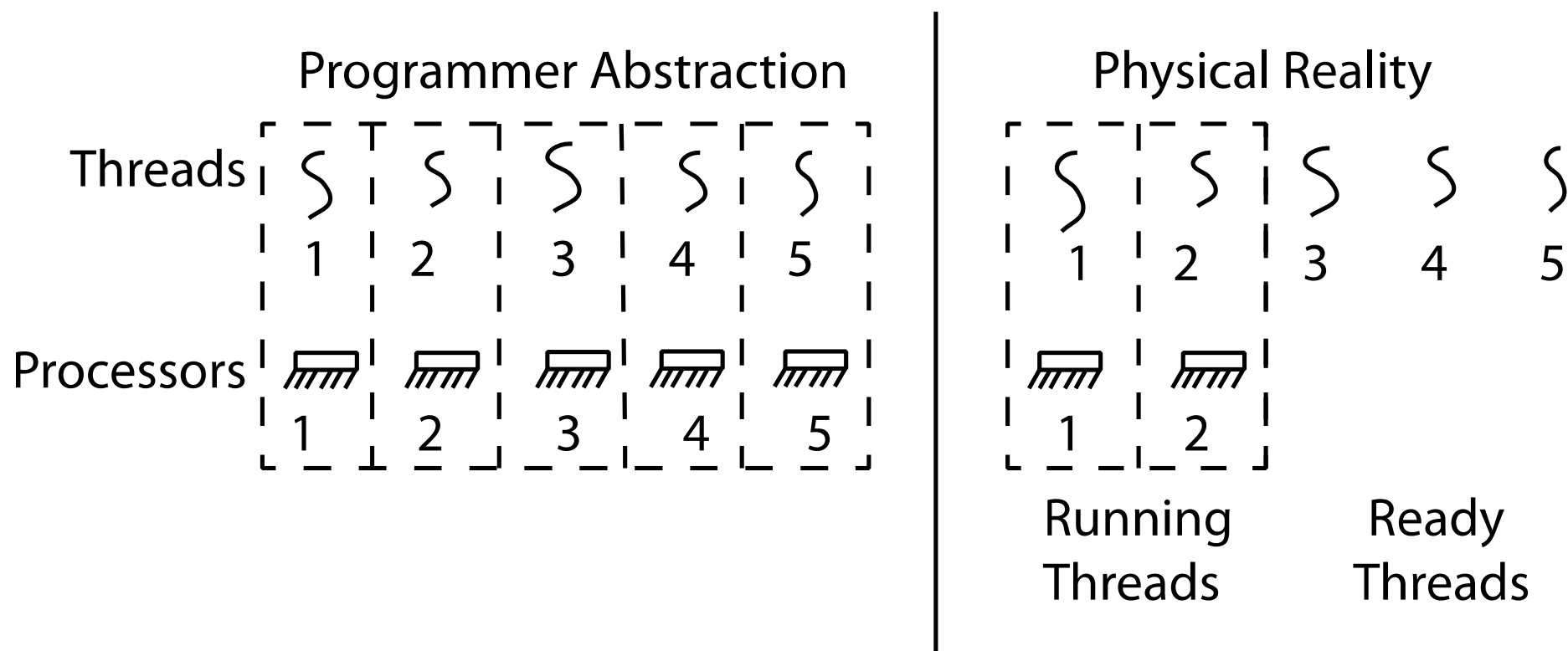
---

- **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

# Pintos Return from Processing



# Recall: Thread Abstraction

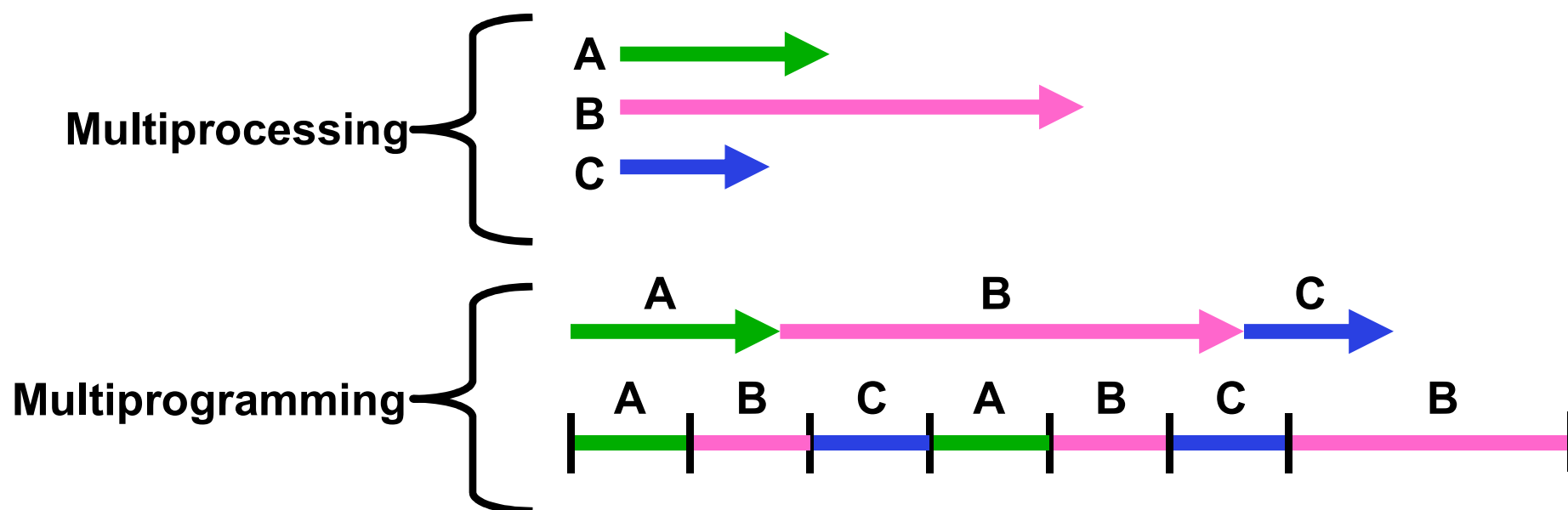


- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule



# Multiprocessing vs Multiprogramming

- **Remember Definitions:**
  - Multiprocessing  $\equiv$  Multiple CPUs
  - Multiprogramming  $\equiv$  Multiple Jobs or Processes
  - Multithreading  $\equiv$  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, ...
  - Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks



# 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
- **You probably don’t realize how much you depend on reproducibility:**
  - Example: Evil C compiler
    - » Modifies files behind your back by inserting errors into C program unless you insert debugging code
  - Example: Debugging statements can overrun stack
- **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?

---

- **People cooperate; computers help/enhance people's lives, so computers must cooperate**
  - By analogy, the non-reproducibility/non-determinism of people is a notable problem for “carefully laid plans”
- **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

# Summary (1 of 2)

---

- **Processes have two parts**
  - Threads (Concurrency)
  - Address Spaces (Protection)
- **Concurrency accomplished by multiplexing CPU Time:**
  - Unloading current thread (PC, registers)
  - Loading new thread (PC, registers)
  - Such context switching may be voluntary (`yield()`, I/O operations) or involuntary (timer, other interrupts)
- **Protection accomplished restricting access:**
  - Memory mapping isolates processes from each other
  - Dual-mode for isolating I/O, other resources
- **Various Textbooks talk about *processes***
  - When this concerns concurrency, really talking about thread portion of a process
  - When this concerns protection, talking about address space portion of a process

## Summary (2 or 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
- **Showed how to protect a critical section with only atomic load and store  $\Rightarrow$  pretty complex!**