CS162: Operating Systems and Systems Programming

Lecture 4: Concurrency: Processes and Threads

25 June 2015

Charles Reiss

https://cs162.eecs.berkeley.edu/

Recall: Big Concepts So Far (1)

Process = Address Space + Thread(s)

Protection with Address Translation

Dual Mode operation

- And interrupts/system calls/traps

Recall: Big Concepts So Far (2)

Layers: OS library, system call, drivers

POSIX API

- "descriptors" to identify OS resources
- File as unifying interface

Process control:

- **Fork**, **wait**, **sigaction/kill**, **exec**

Recall: POSIX IO: Everything is a file

Uniform interface for

- Devices (terminals, printers, etc.)
- Regular files on disk
- Networking (sockets)
- Local interprocess communication (pipes, sockets)

Based on open(), read(), write(), close()

Part of the process state

- accessed with file descriptors

Recall: ***Process***

Execution environment with restricted rights

- **Address space** + **Thread(s)**
- File descriptors, other resources, ...

Protected from each other

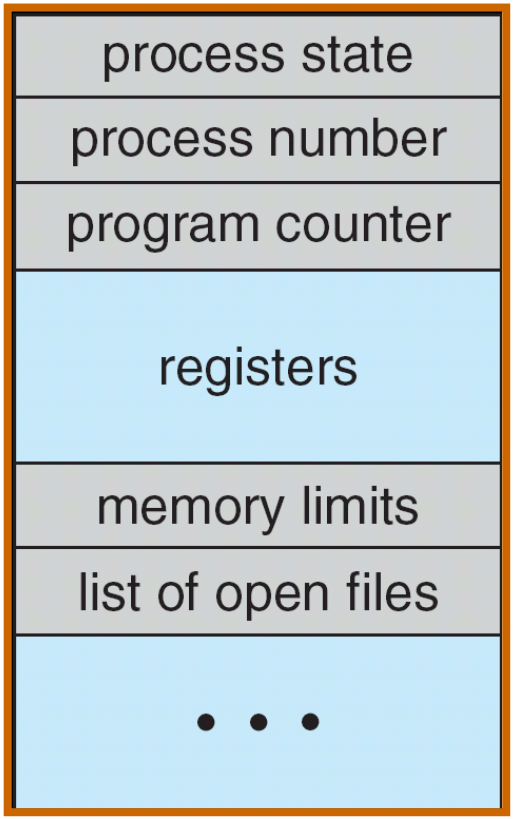
- Compromise: **sharing harder, less efficient**

OS protected from processes

Recall: Process Control Block (PCB)

Kernel representation of each process

- Status (running, **ready**, blocked)
- Register state (if not ready)
- Process ID
- Execution time
- Open files
- Memory translations, ...



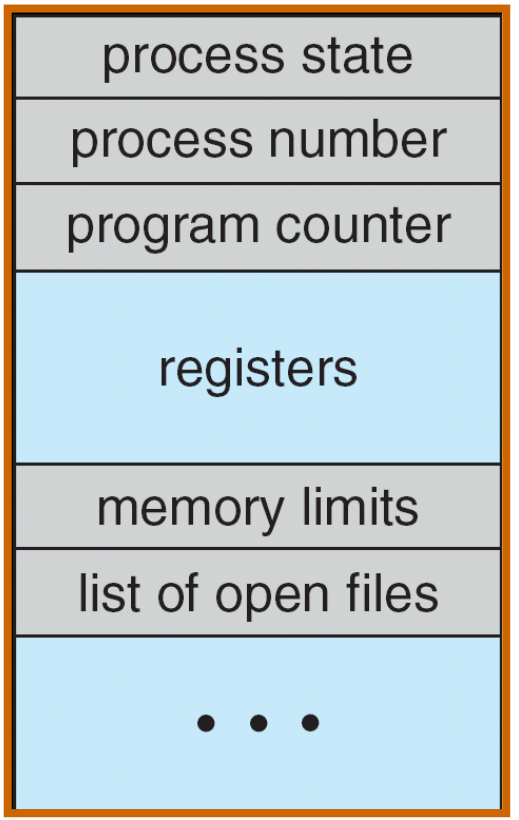
Process Control Block

Multiplexing processes

Snapshot of each process in its PCB
- Only one active at a time (for now)

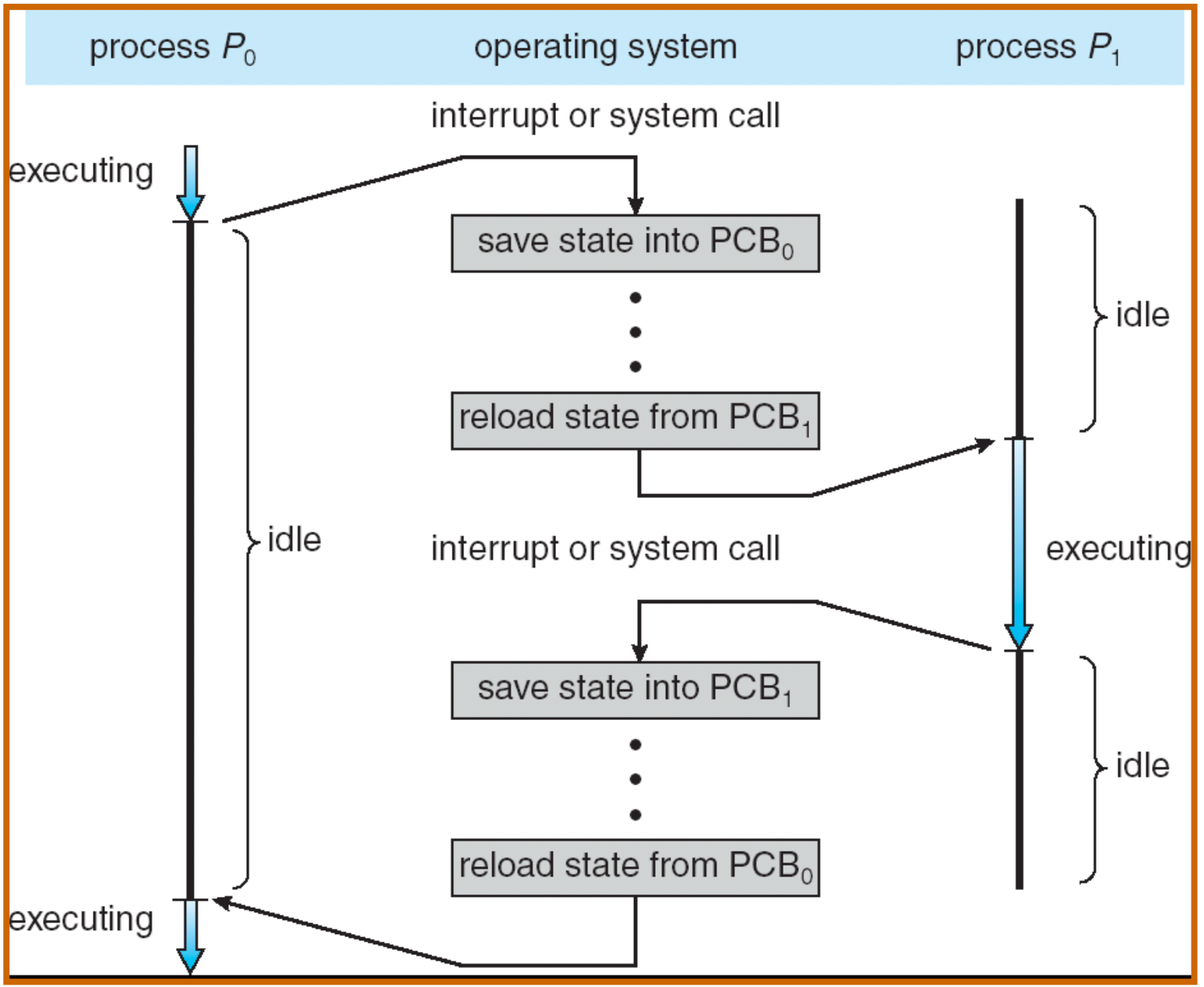
Give out CPU to different processes
- "***Scheduling***"
- One running at a time
- **Policy decision**

Give out non-CPU resources
- Memory, I/O
- **Policy decision**

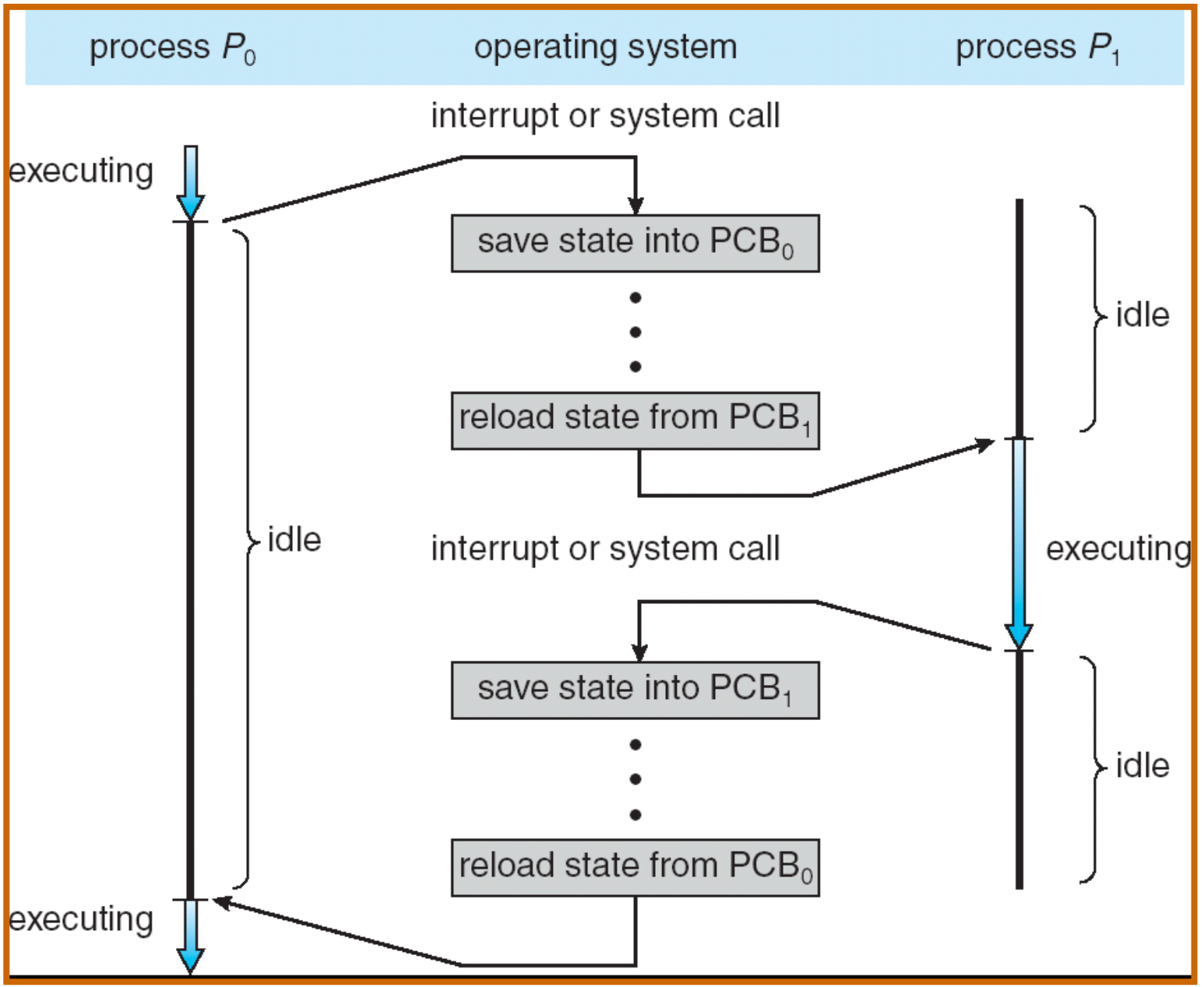


Process Control Block

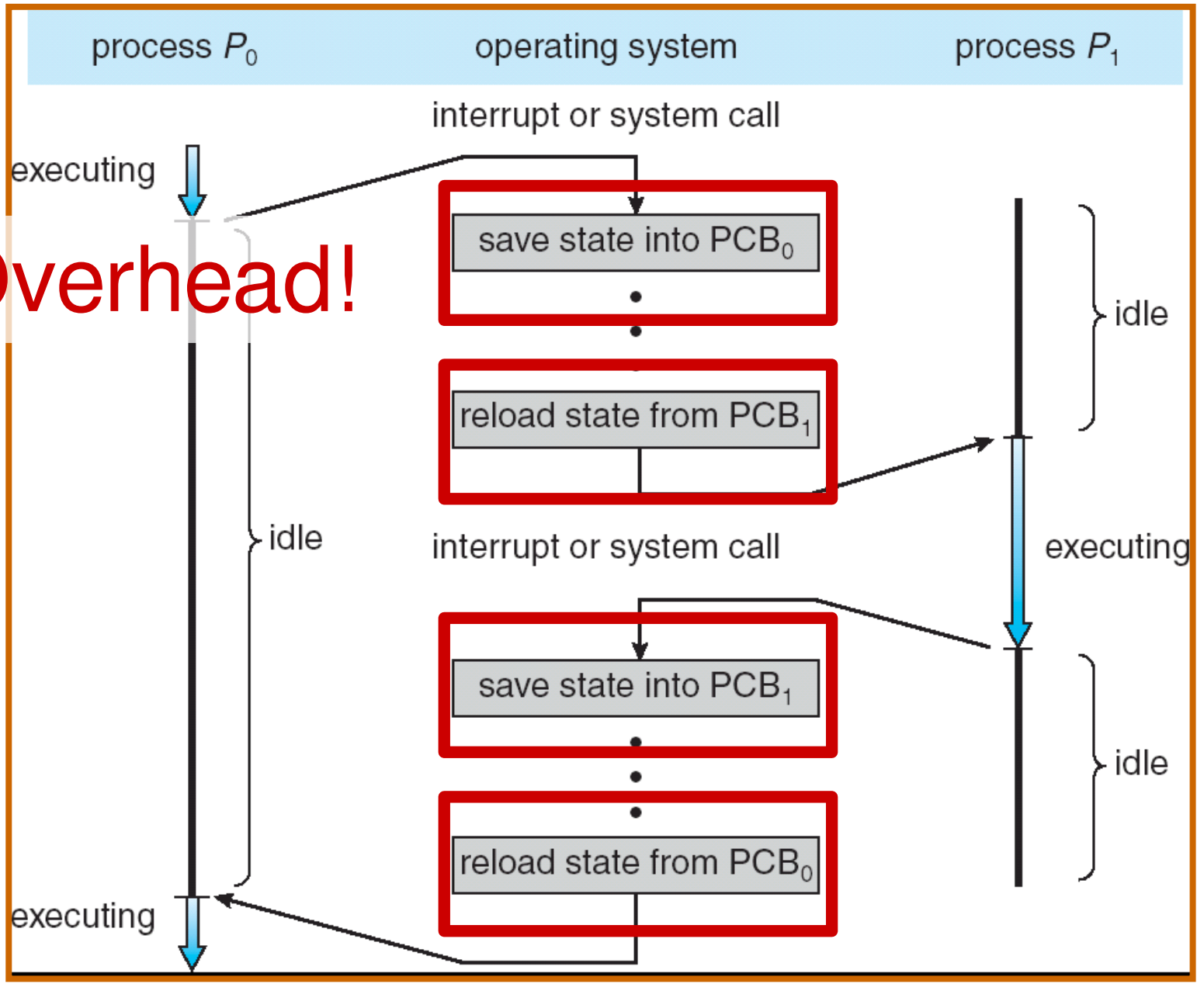
The Context Switch



The Context Switch

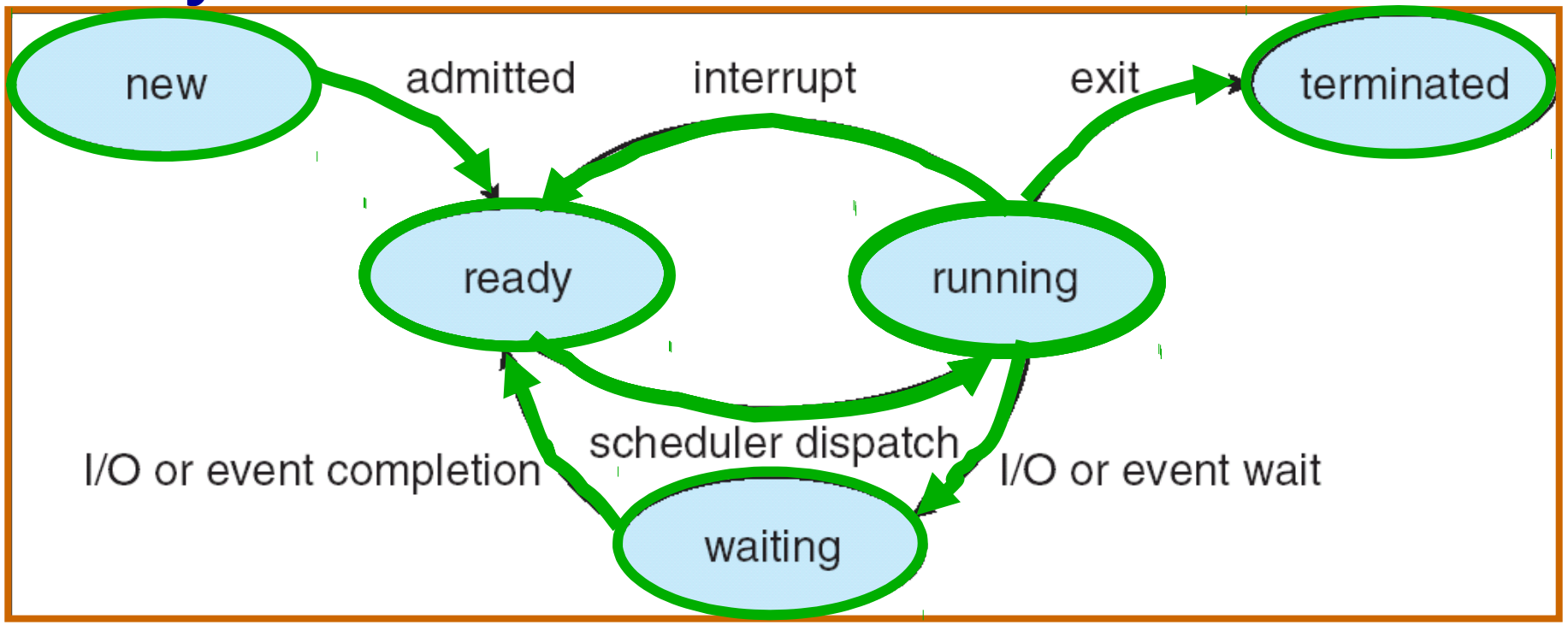


The Context Switch



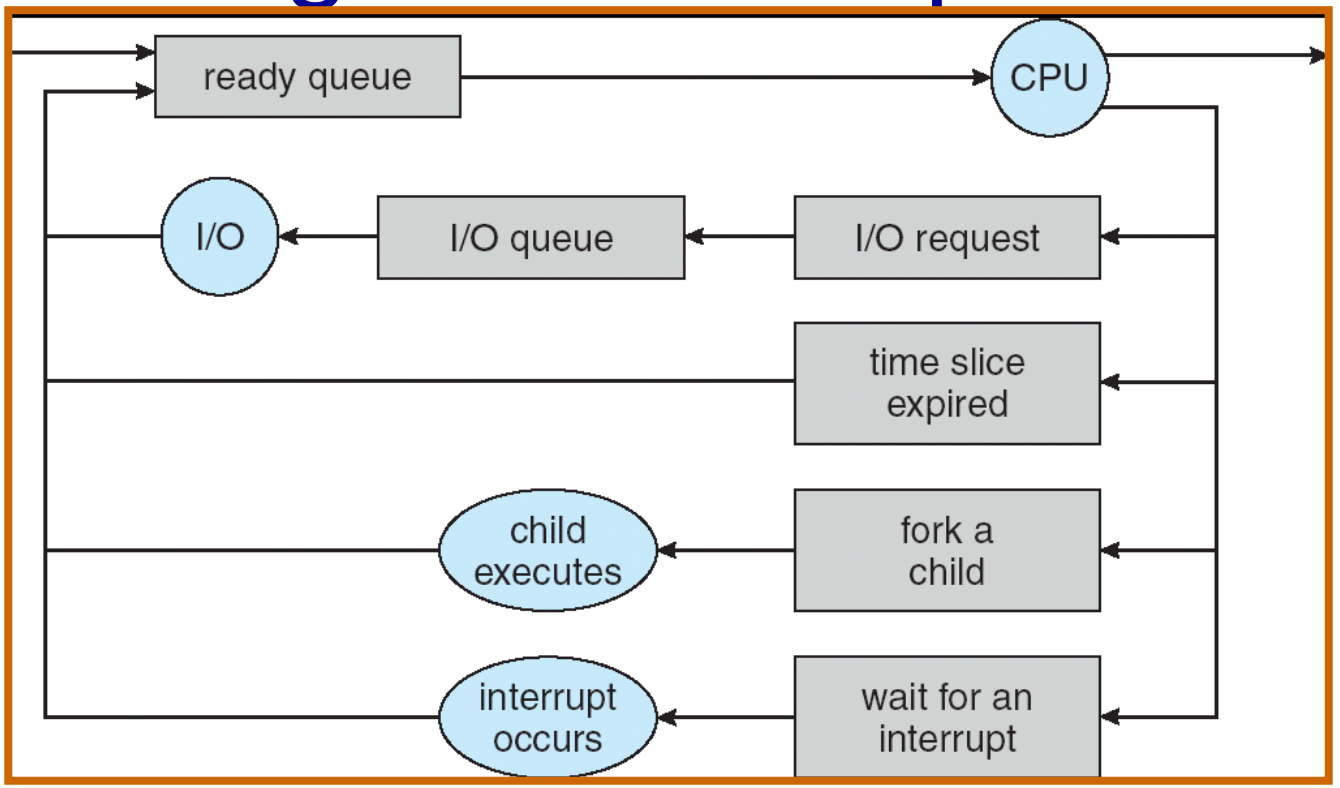
Overhead!

Lifecycle of a Process



- **New**: being created
- **Ready**: waiting to run
- **Running**: instructions executing on the CPU
- **Waiting**: waiting for some event to occur (e.g. keypress)
- **Terminated**: completed execution

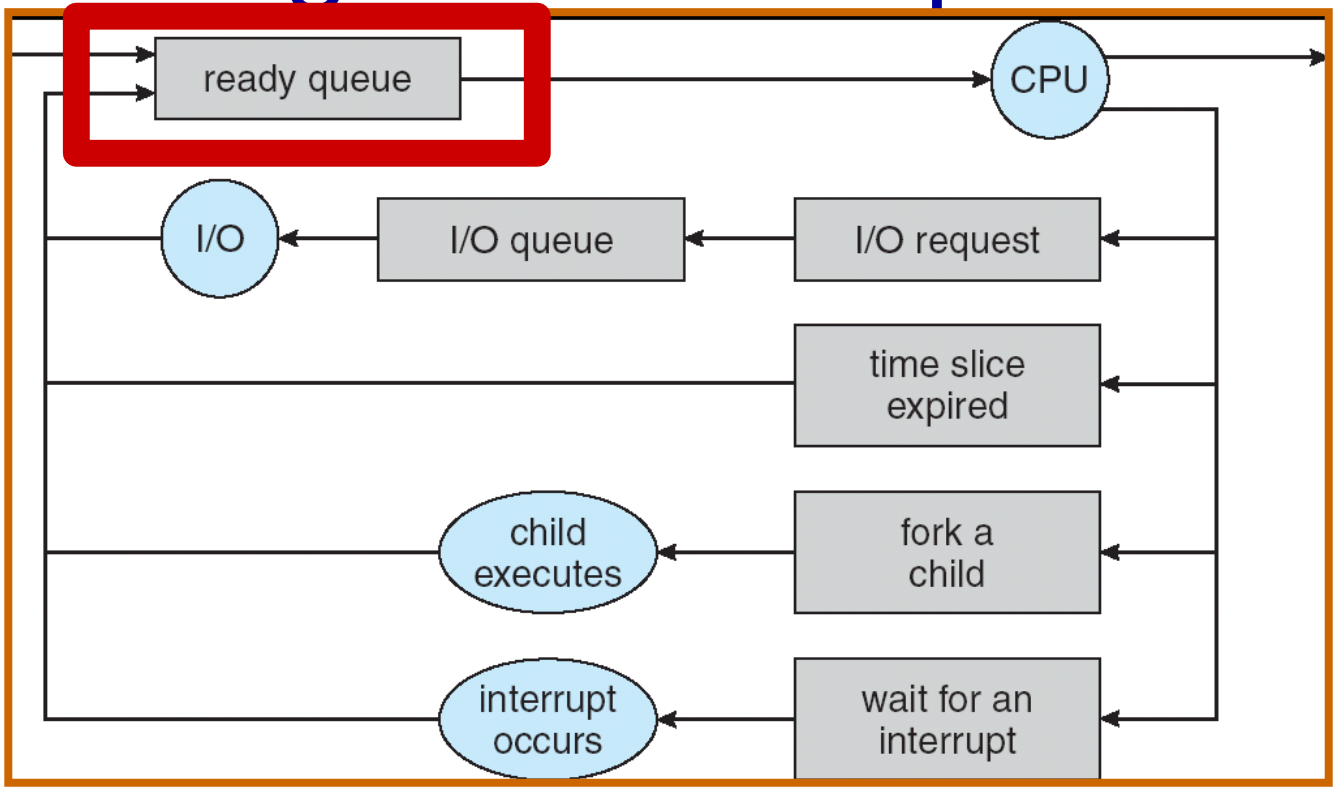
Scheduling: All about queues



PCBs move from queue to queue

Scheduling: which order to remove from queue

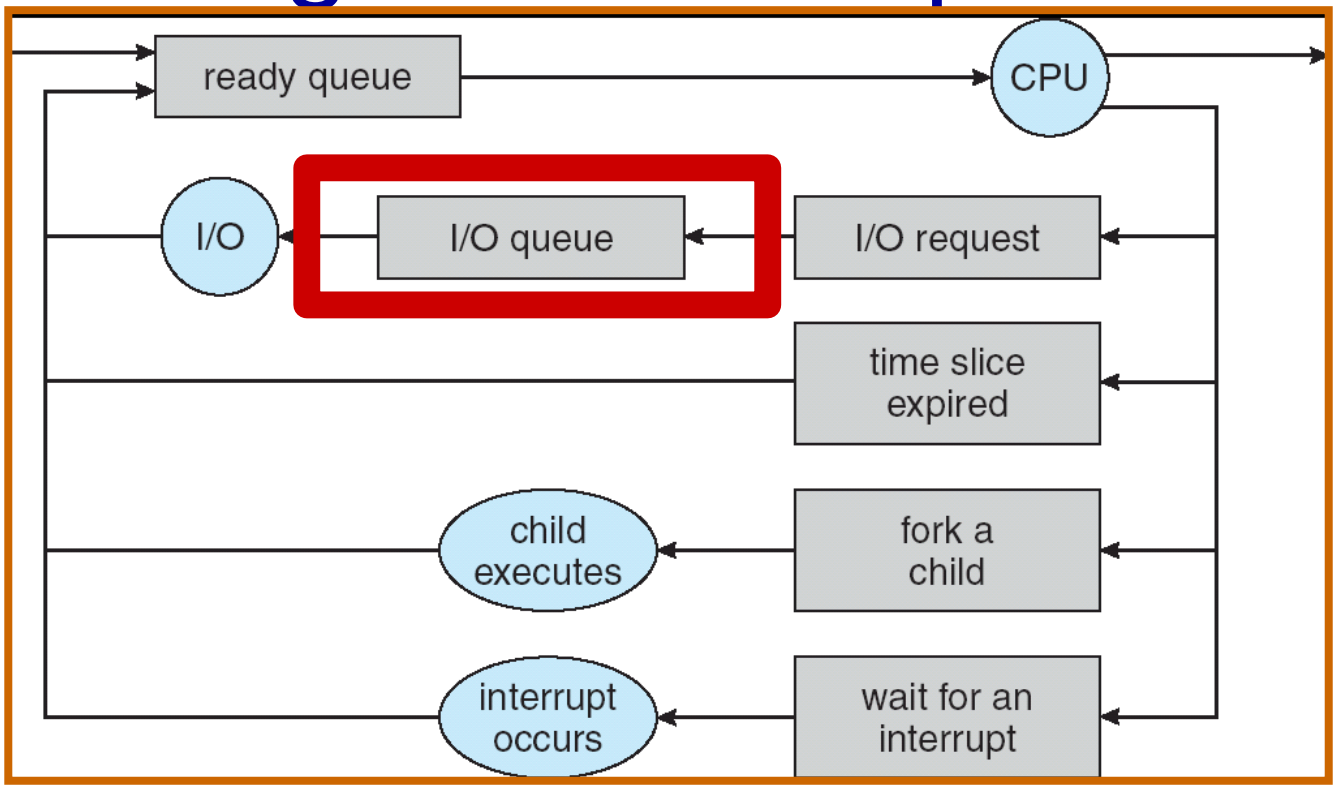
Scheduling: All about queues



PCBs move from queue to queue

Scheduling: which order to remove from queue

Scheduling: All about queues



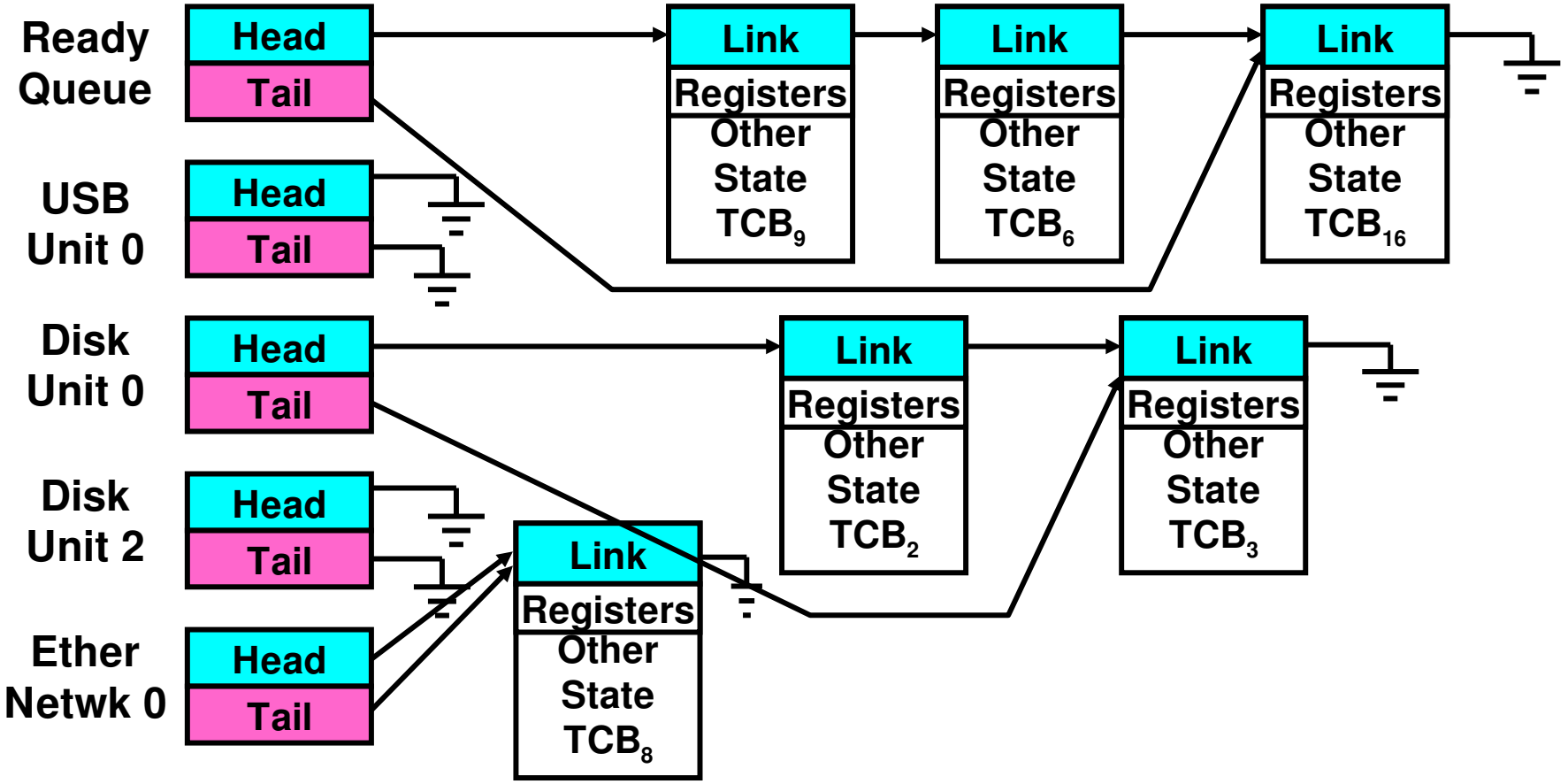
PCBs move from queue to queue

Scheduling: which order to remove from queue

All About Queues

Process not running → on some queue

Different **policy** for each queue

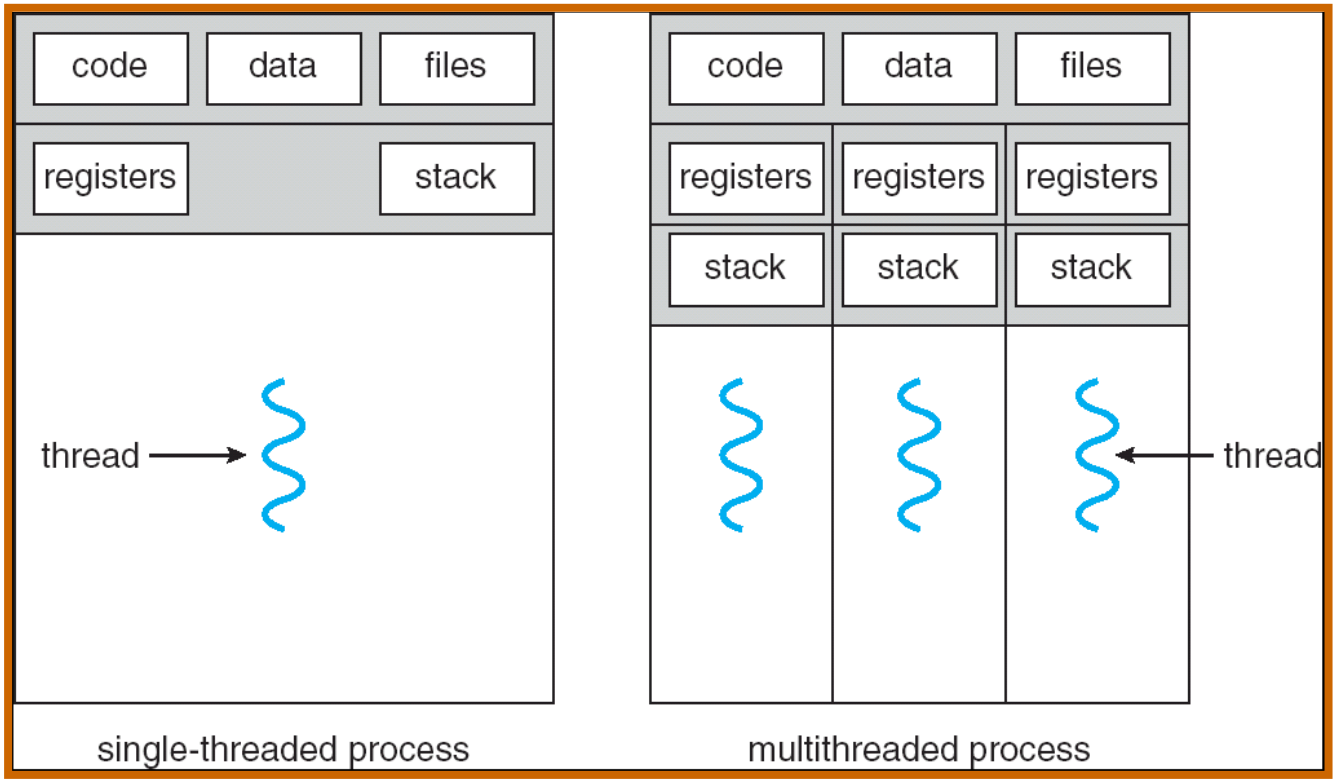


What about threads?

Thread: execution stream within a process

- Sometimes called "lightweight process"
- **Shares address space** with other threads in process

Single and Multithreaded Processes



Thread(s) + Shared Address Space

Multiple stacks in one address space

Thread versus Process State

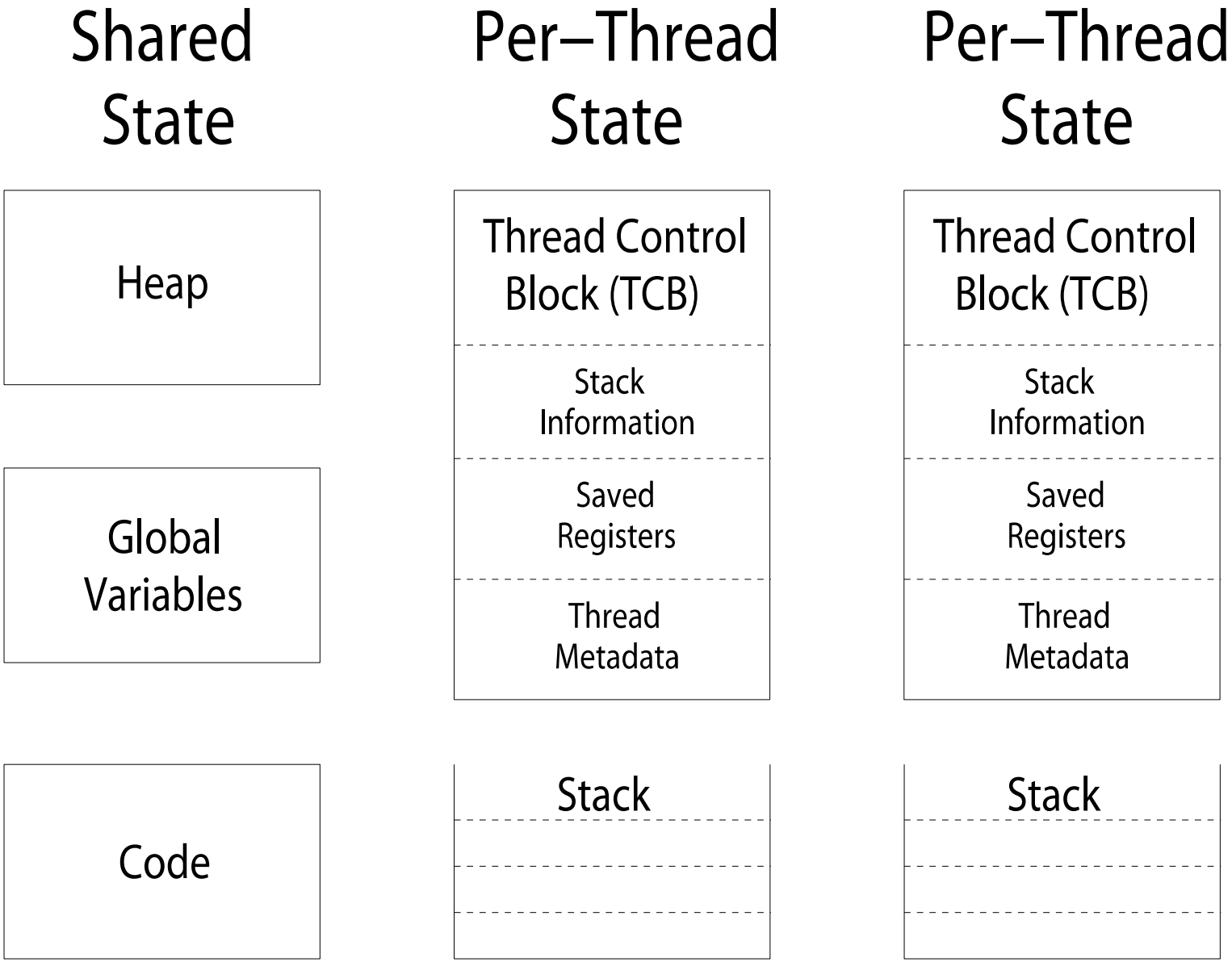
Process-wide state:

- Memory contents
- I/O (file descriptors, etc.)

Thread-"private" state:

- CPU registers including program counter
- Kept in **thread control block**

Thread versus Process State

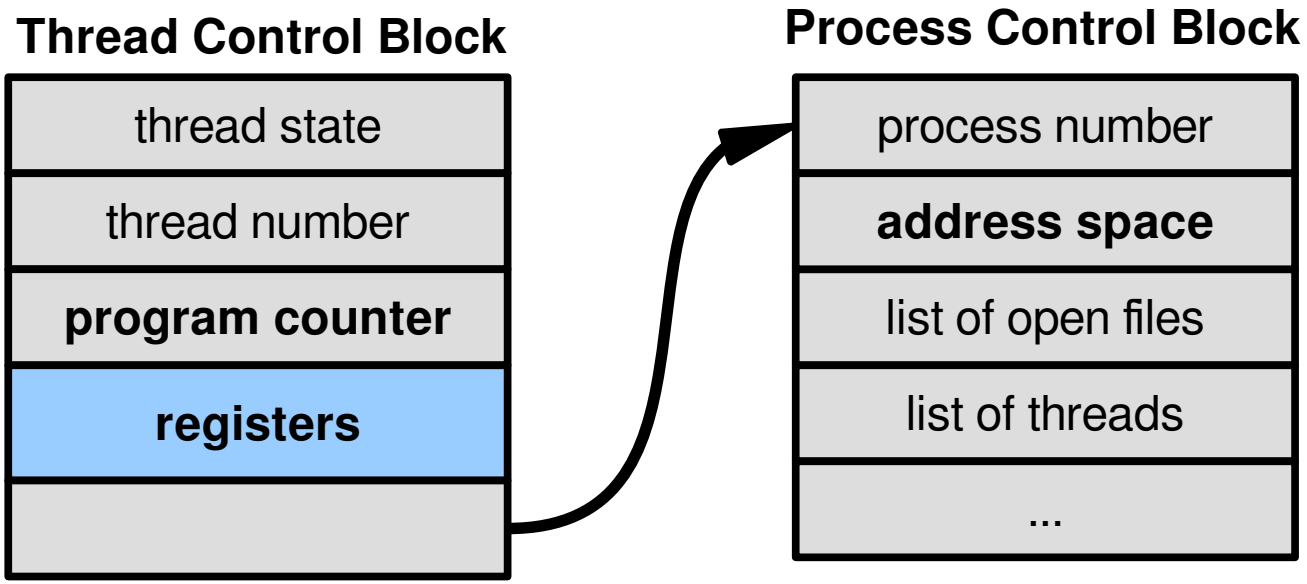


Kernel-supported threads

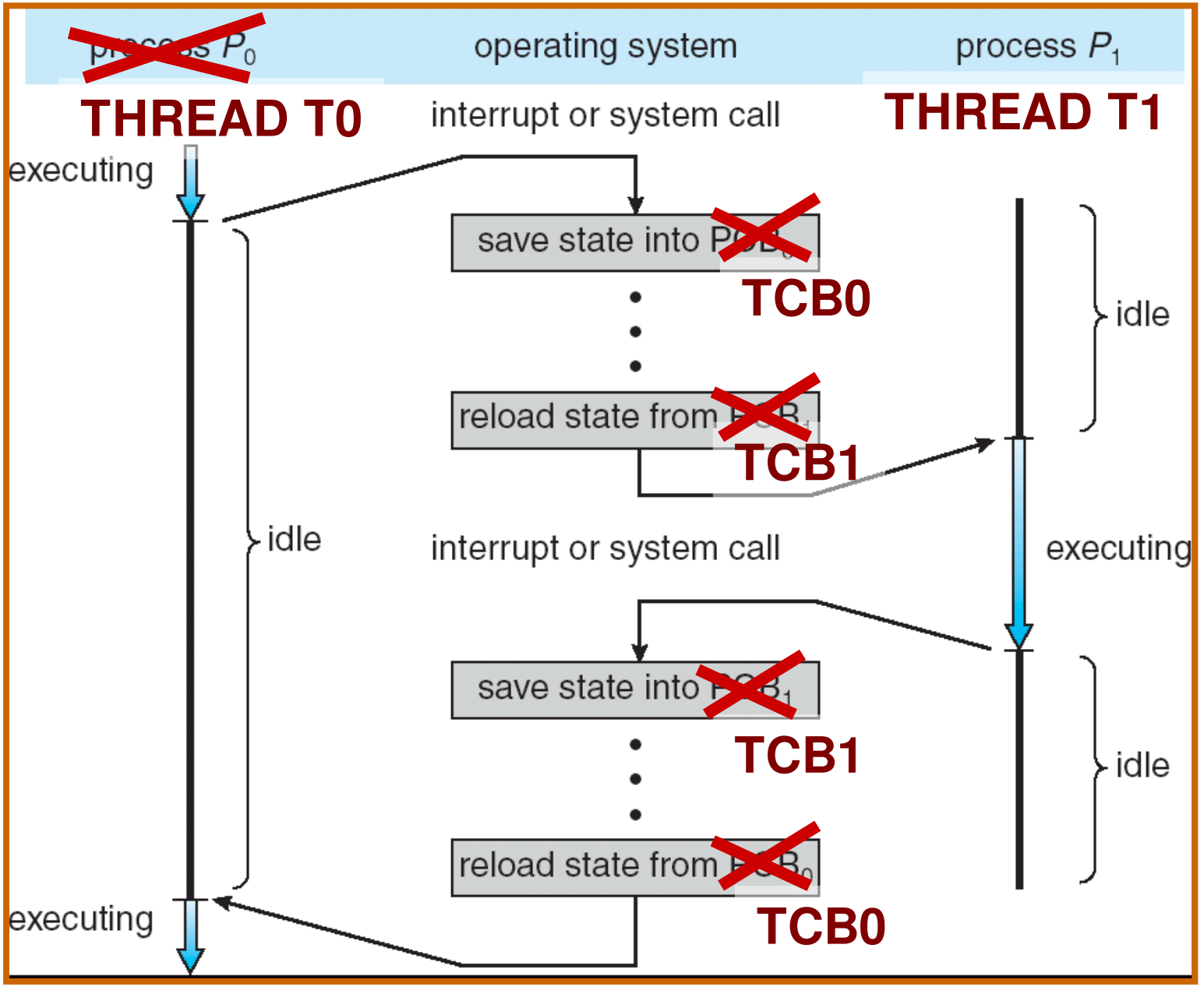
Each thread has a **thread control block**

- CPU registers, with program counter, pointer to stack
- Scheduling info: priority, etc.
- Pointer to the **process control block**

OS schedules using *TCBs instead of PCBs*



Kernel-supported threads



Example multithreaded programs (1)

Embedded systems

- Sometimes: 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 (2)

Network Servers

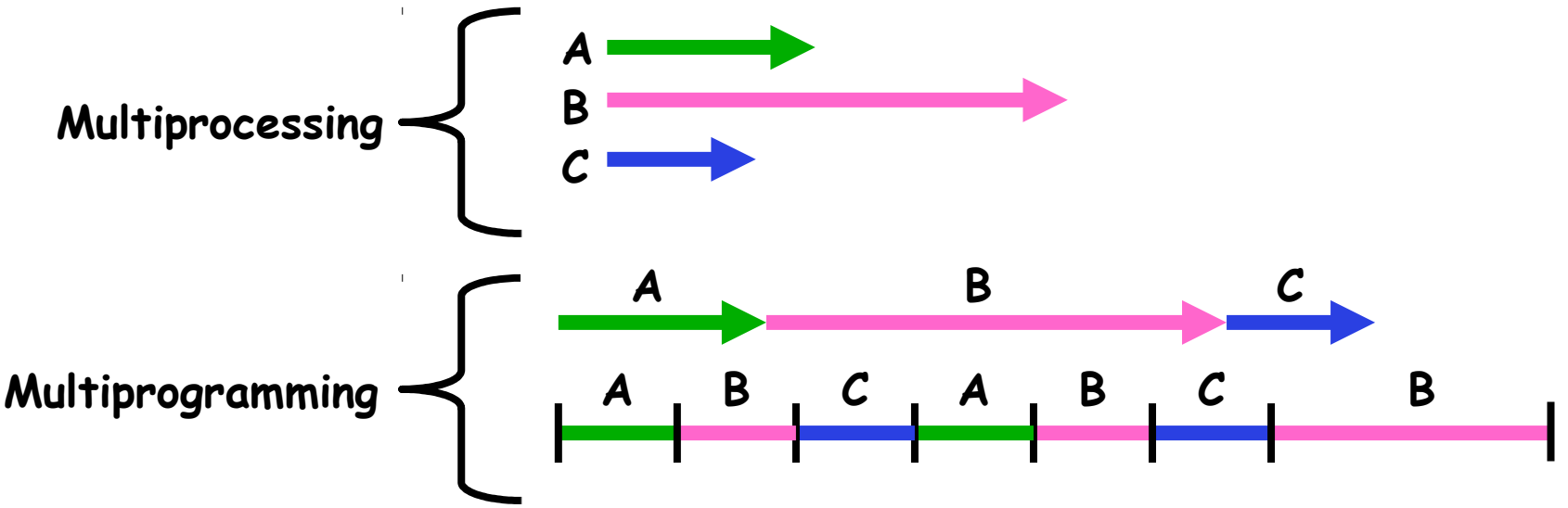
- Concurrent requests from network
- Single program, multiple concurrent operations
- File server, web server, airline reservation systems

Multiprocessing v Multiprogramming

Multiprocessing ↔ multiple cores

Multiprogramming ↔ multiple jobs/processes

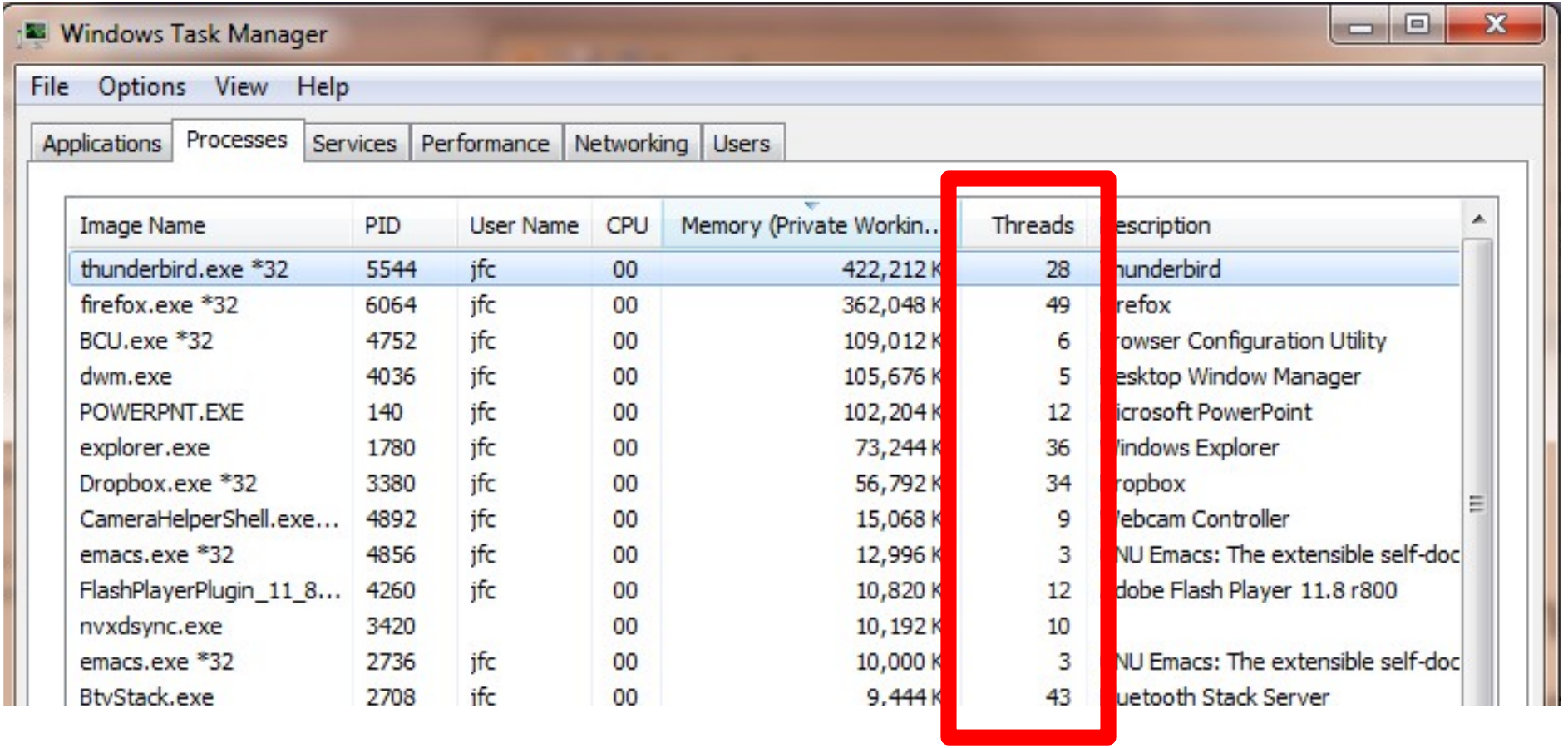
Multithreading ↔ multiple threads/process



Example multithreaded programs (3)

All over the place!

Within-process or **across-process** context switches



Threads in the Kernel

For each user process/thread

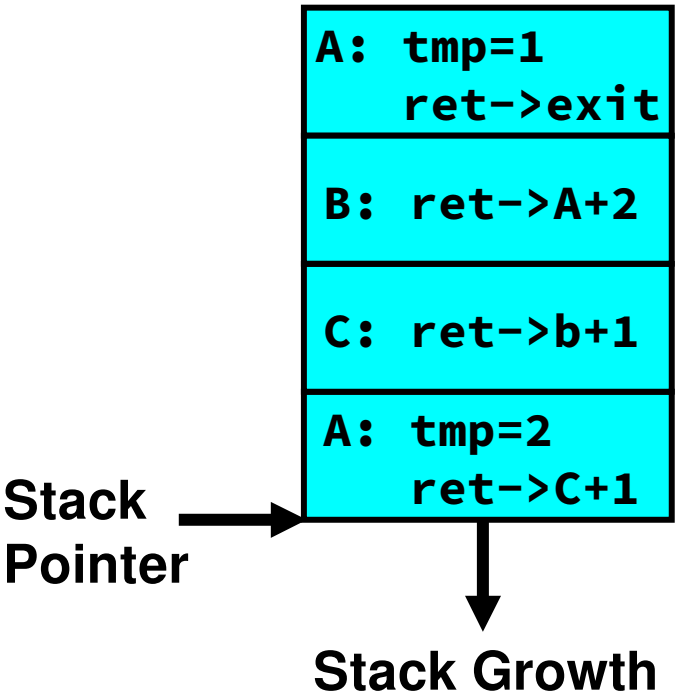
- Can always resume thread into the kernel mode

For steps in processing I/O

For device drivers

Review: Execution Stack

```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```



Stack holds temporary results

Used to implement recursion

Silly Use Case For Threads

Consider the following program:

```
main() {
    ComputePI("pi.txt");
    PrintClassList("clist.text");
}
```

Never prints out class list...

Adding Threads

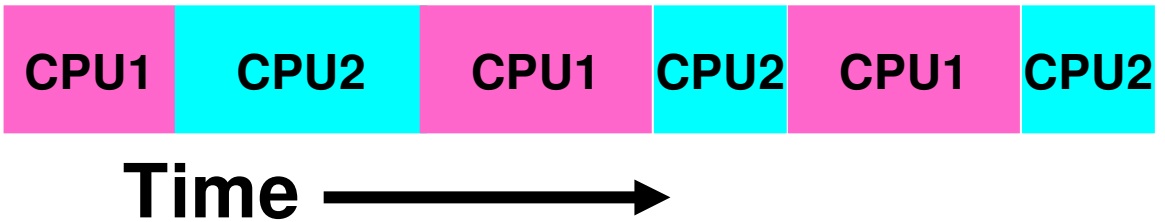
Program with Threads (loose syntax):

```
main() {
    thread_fork(ComputePI, "pi.txt");
    thread_fork(PrintClassList, "clist.text");
}
```

`thread_fork`: Start an thread running given procedure

Behavior:

- Class list gets printed out
- As if two CPUs:



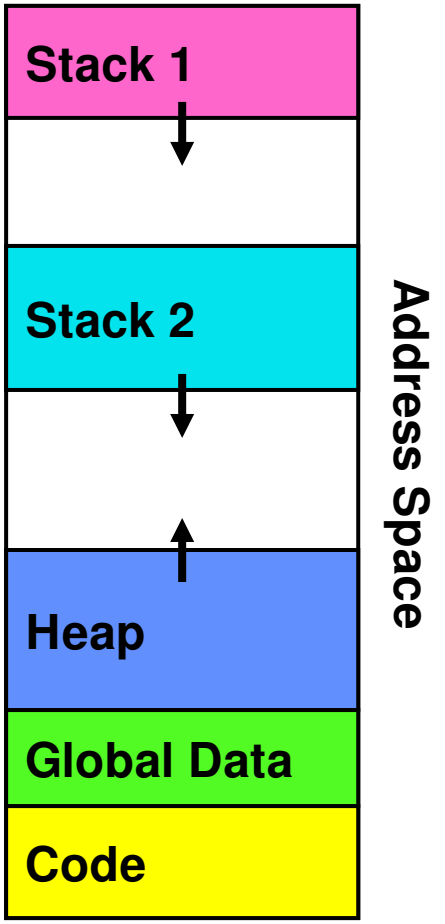
Memory Footprint: Two Threads

Two sets of CPU registers

Two sets of stacks

Issues:

- Placing stacks
- Size of stacks (limited!)
- What if threads run into each other?



Actual thread operations in Pintos

`thread_create(name, priority, func, args)`

- Create a thread to run func(args)

`thread_yield()`

- Relinquish processor voluntarily

`thread_join(thread)`

- Wait (put on queue) until thread exits, then return

`thread_exit()`

- Quit/cleanup current thread, waking up any joiner

Dispatch Loop

```
Loop {
        RunCurrentThread();
        ChooseNextThread();
        SaveStateOfCPU(curTCB);
        LoadStateOfCPU(nextTCB);
      }
```

Conceptually all the OS executes

Infinite loop

- "all" the OS does

Dispatch Loop

```
Loop {
        RunCurrentThread();
        ChooseNextThread();
        SaveStateOfCPU(curTCB);
        LoadStateOfCPU(nextTCB);
      }
```

How?

- Load its registers into the CPU
- Load its environment (address space)
- Jump to its PC

How does dispatch loop get control again?

- Thread returns control voluntarily – yield(), IO
- External events: thread is *preempted*

Voltunarily giving up control

I/O

- e.g. Keypress

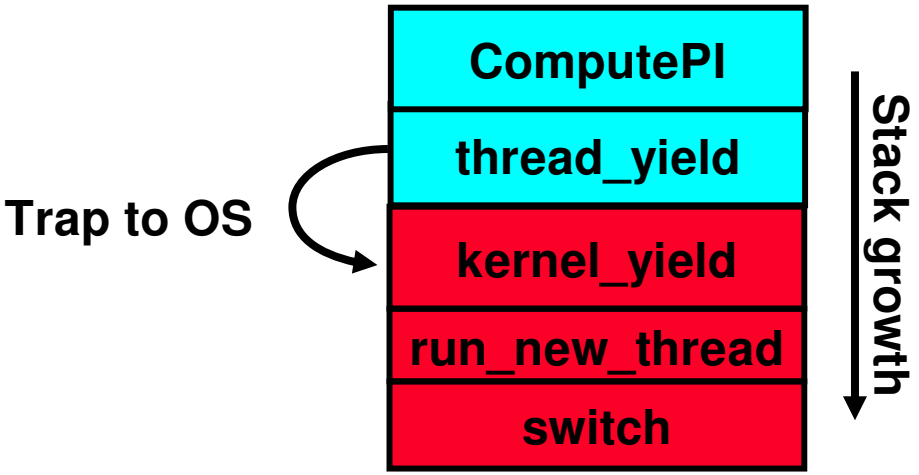
Waiting for a signal from another thread

- Thread makes system call to wait

Thread executes `thread_yield()`

- Gives up CPU, but puts back on ready queue

Yielding Thread

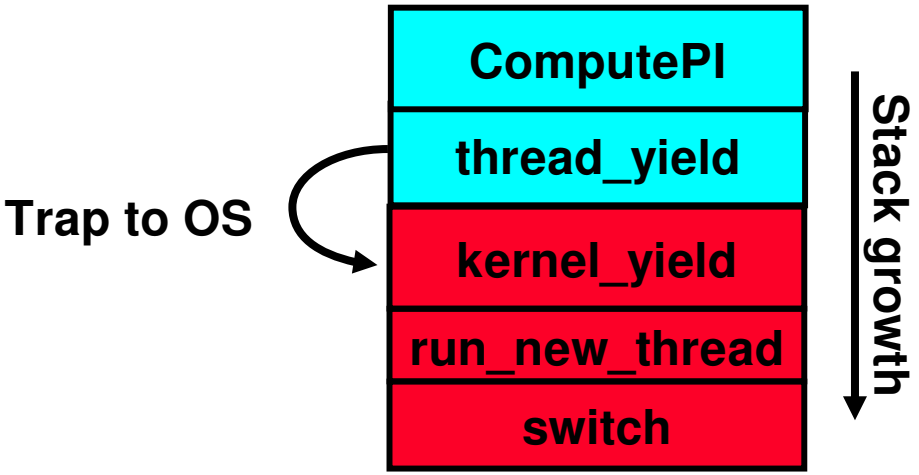



```
computePI() {
  while(TRUE) {
    ComputeNextDigit();
    thread_yield();
  }
}
```

Cyan = user stack; Red = kernel stack

```
run_new_thread() {
   newThread = PickNewThread();
   switch(curThread, newThread);
   ThreadHouseKeeping(); /* Do any cleanup */
}
```

Yielding Thread



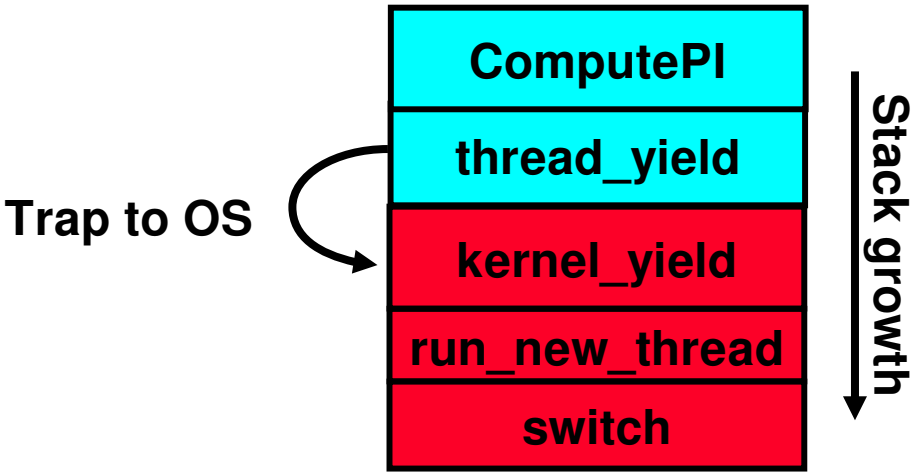

```
computePI() {
  while(TRUE) {
    ComputeNextDigit();
    thread_yield();
  }
}
```

Cyan = user stack; Red = kernel stack

```
run_new_thread() {
  newThread = PickNewThread();
  switch(curThread, newThread);
  ThreadHouseKeeping(); /* Do any cleanup */
}
```

Scheduler: Later

Yielding Thread




```
computePI() {
  while(TRUE) {
    ComputeNextDigit();
    thread_yield();
  }
}
```

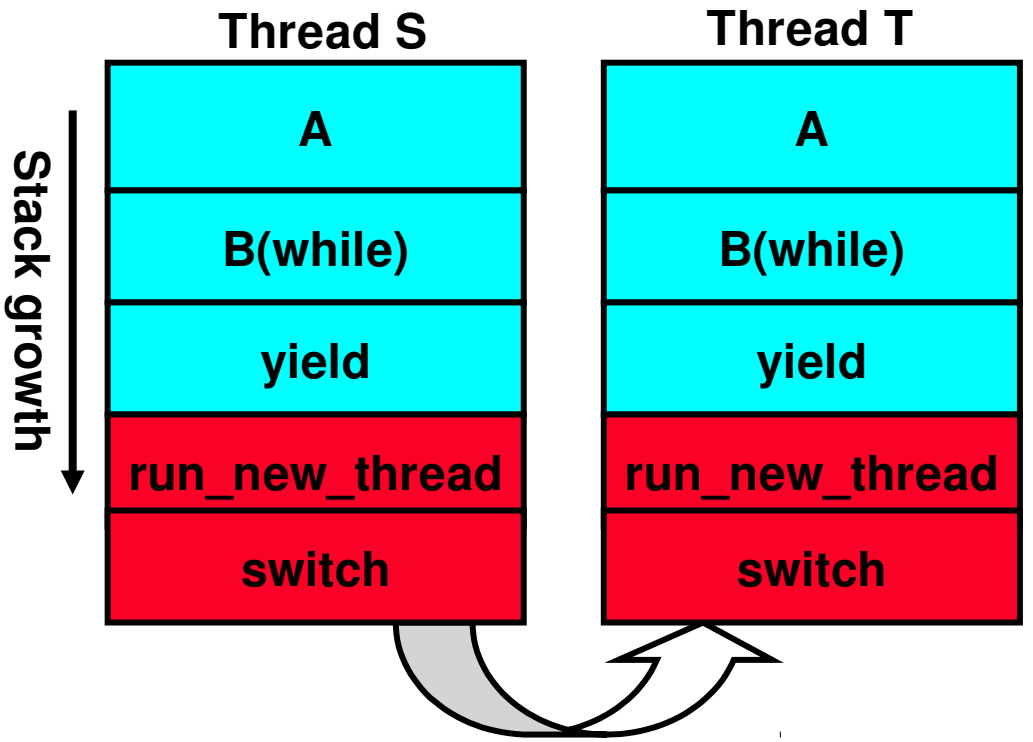
Cyan = user stack; Red = kernel stack

```
run_new_thread() {
  newThread = PickNewThread();
  switch(curThread, newThread);
  ThreadHouseKeeping(); /* Do any cleanup */
}
```

Juggling Stacks

```
proc A() {
     B();
}

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




```
switch(curThread, newThread);
```

Thread S's switch **returns to Thread T's** (and vice-versa)

The Context Switch Itself

```
switch(tCur,tNew) {
   /* Unload old thread */
   TCB[tCur].regs.r7 = CPU.r7;
      …
   TCB[tCur].regs.r0 = CPU.r0;
   TCB[tCur].regs.sp = CPU.sp;
   /* store old thread's return address from switch() */
   TCB[tCur].regs.retpc = CPU.retpc;
   /* Load and execute new thread */
   CPU.r7 = TCB[tNew].regs.r7;
      …
   CPU.r0 = TCB[tNew].regs.r0;
   CPU.sp = TCB[tNew].regs.sp;
   CPU.retpc = TCB[tNew].regs.retpc;
   return; /* Return to CPU.retpc */
}
```

Context Switch Numbers

Every 10-100ms

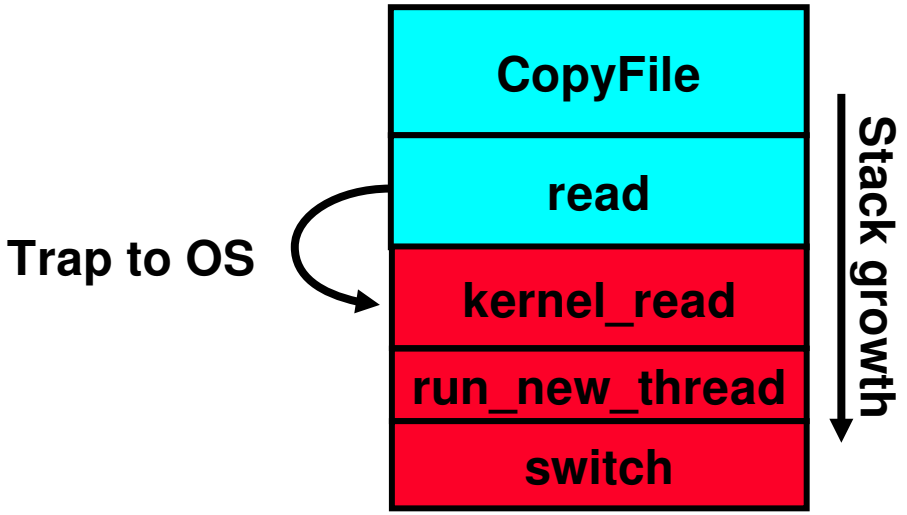
Linux context switch: 3-4 microsecs (Intel i7 & E5)

- Thread context switch 100 ns

But slower caches are real cost:

- Caches are "cold"
- Cost depends on working set size
  (amount of memory used by process "soon")

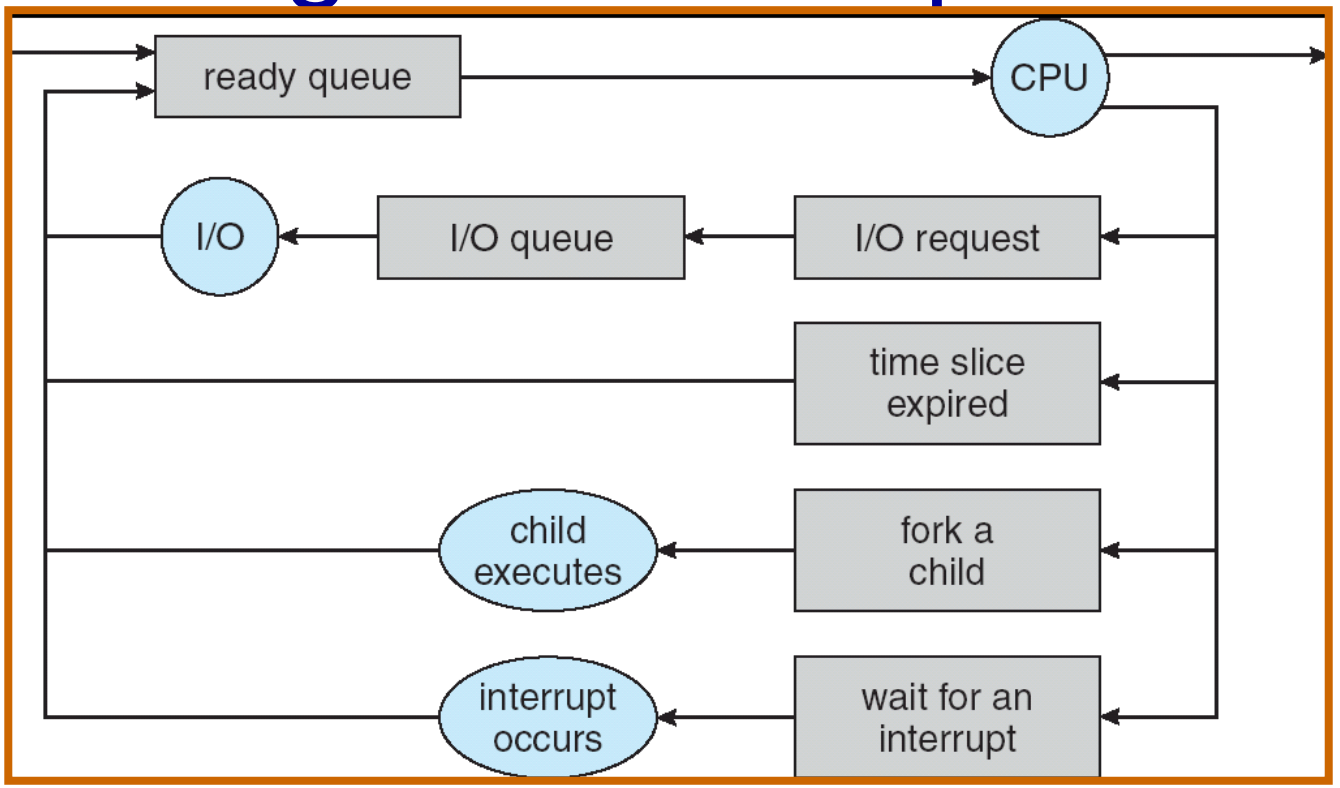
How about blocking on I/O?



Same thing

Difference: Which queue thread is put on

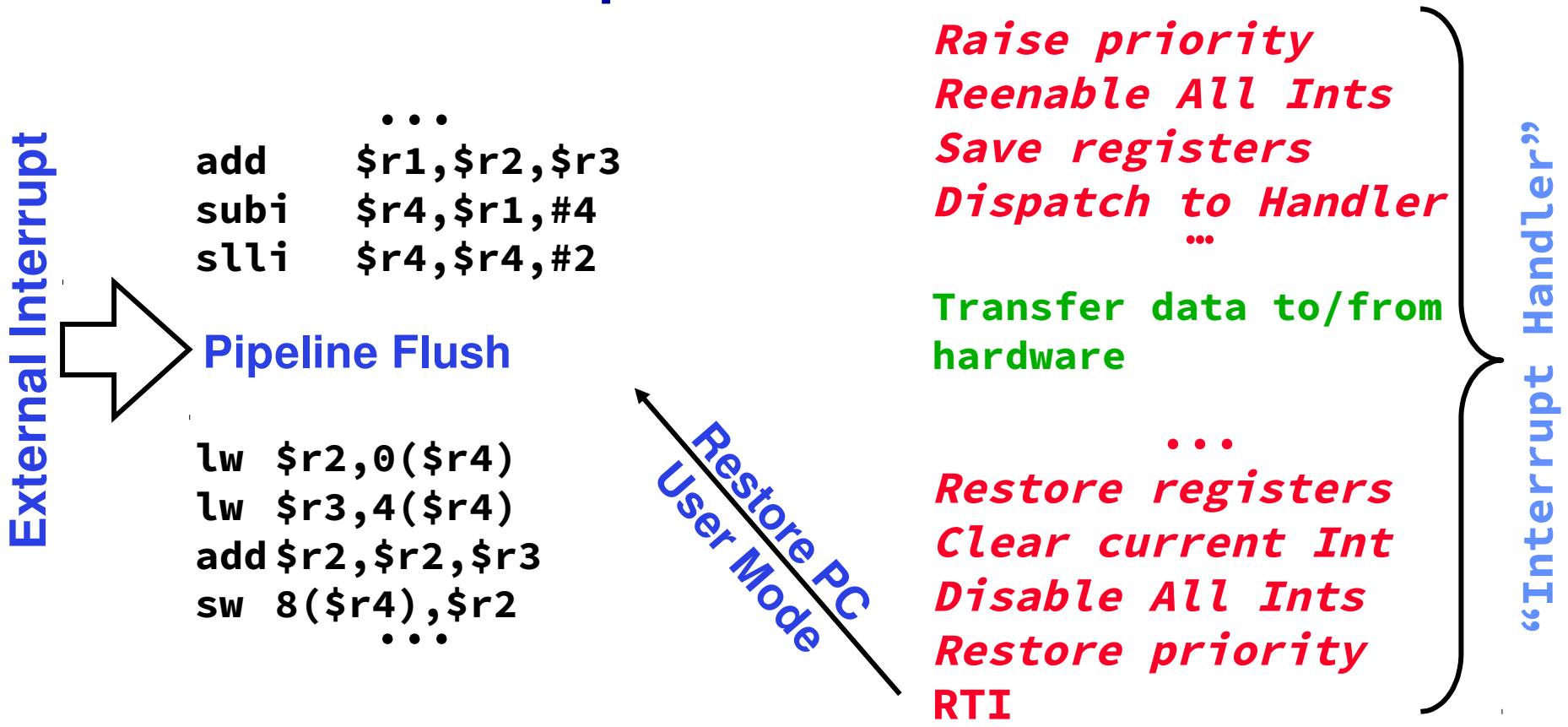
Scheduling: All about queues



PCBs move from queue to queue

Scheduling: which order to remove from queue

Recall: Interrupt

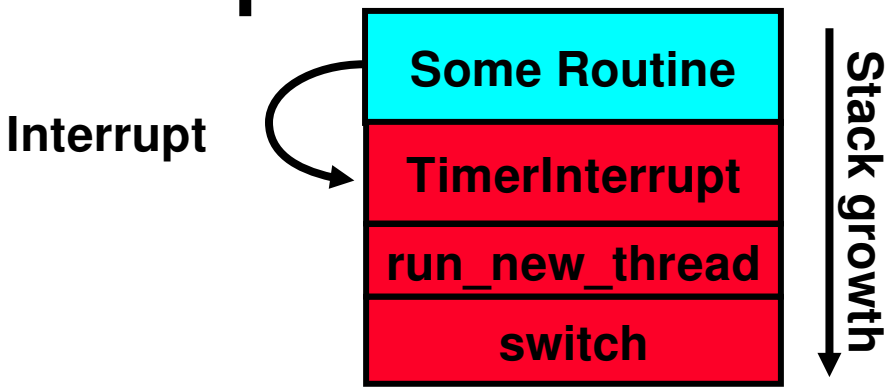


Hardware invoked mode switch

- Handled immediately, no scheduling required

Switching Threads from Interrupts

Prevent threads from running forever with **timer interrupt**




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

Same thing from IO interrupts

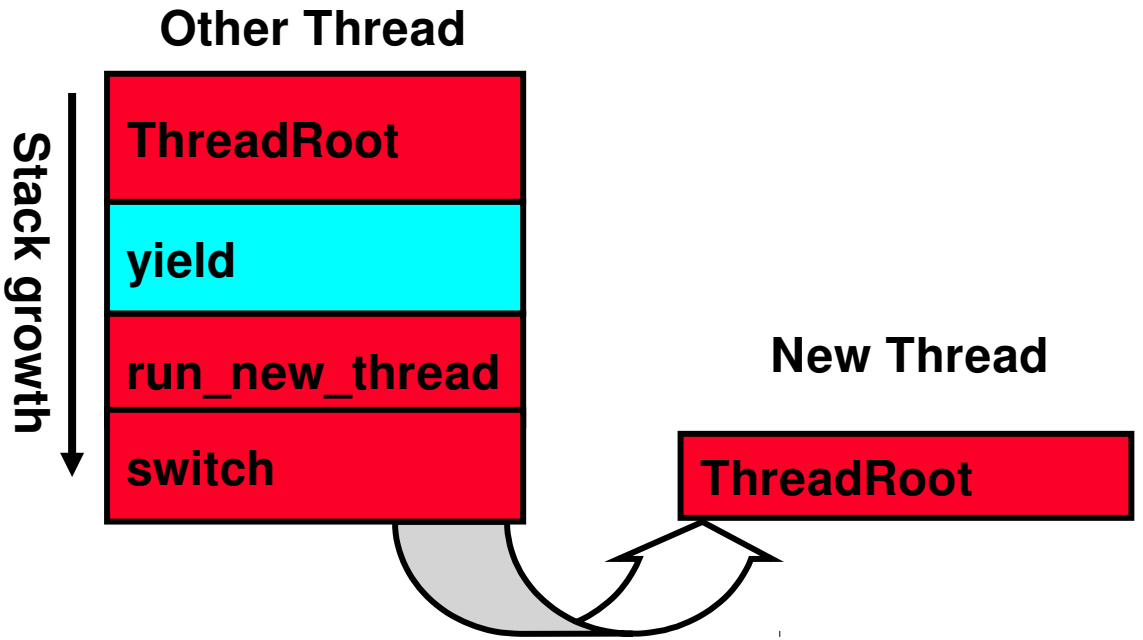
- Example: immediately start process waiting for keypress

Bootstrapping Threads

Can't call **`switch()`** without starting a thread

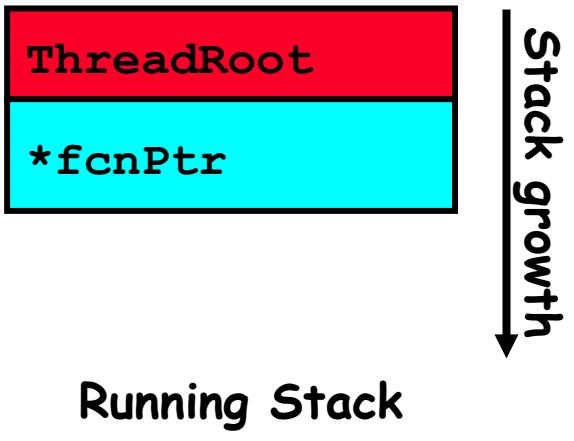
How do we make a ***new*** thread?

```
SetupNewThread(tNew) {
  ...
  TCB[tNew].regs.sp =
        newStack;
  TCB[tNew].regs.retpc =
        &ThreadRoot;
}
```



Bootstraping Threads: `ThreadRoot`

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



`ThreadRoot()` ***never returns***

- **`ThreadFinish()`** destroys thread, calls scheduler

Logistics

Homework 0 – due Friday 11:59PM

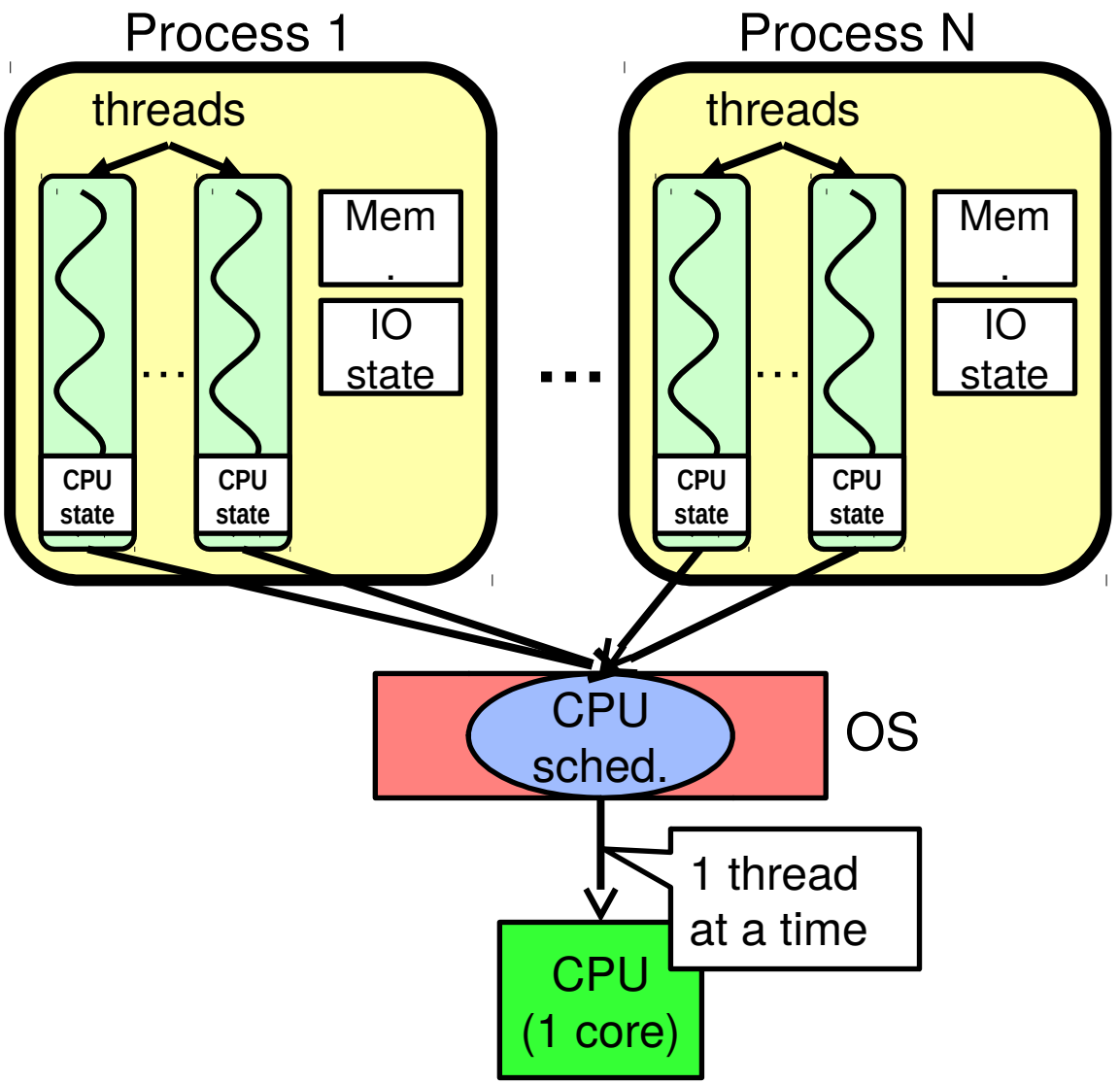
- Vagrant problems? Help after lecture (651 Soda)
- Office Hours Tomorrow: 12p-2p (651 Soda)
  - Alex: 12-1p; Charles: 1-2p

Project groups – due Friday

- instructions on Piazza

Break

Putting it Together: Threads/Proceses



Switch overhead:

- Same process: **low**
- Different process: **high**

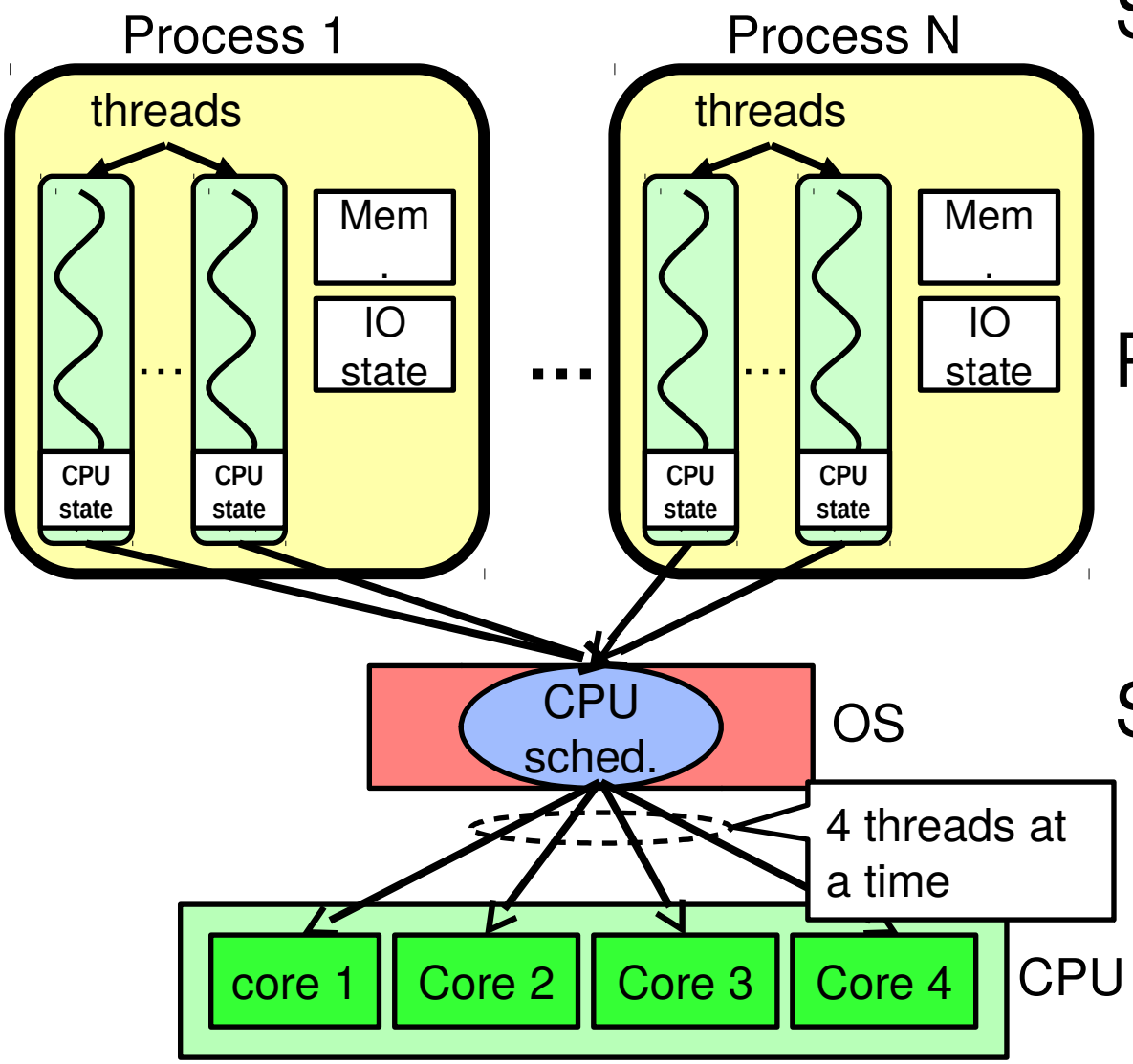
Protection:

- Same process: **low**
- Different process: **high**

Sharing overhead:

- Same process: **low**
- Different process: **high**

Putting it Together: Threads/Proceses



Switch overhead:

- Same process: **low**
- Different process: **high**

Protection:

- Same process: **low**
- Different process: **high**

Sharing overhead:

- Same process: **low**
- Different process: **high**

Kernel v. User Threads

So far: threads in the kernel

- TCB kept in kernel
- Run/block (on IO, etc.) independently
- **Two mode switches *for every context switch – expensive (?)***
- Creating threads requires system call(s)

Alternative: User-mode threads

Kernel v. User Threads

User program contains its own thread scheduler

Several *user threads* per kernel thread

User scheduler switches whenever threads yield

Context switches very cheap!

- Copy registers, jump – switch() in userspace

User Threading Problems

One thread blocks on I/O, everything does

Multiple cores?

Solution: never make blocking system calls?

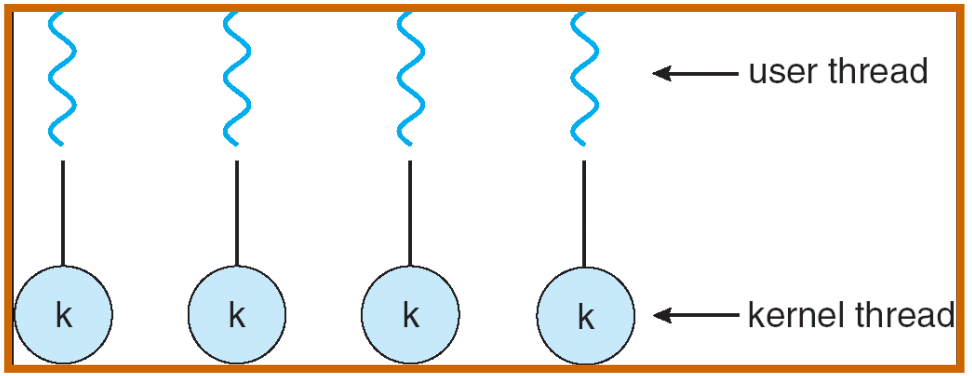
- Big changes to OS libraries and/or restrictions on programmer
- What about page faults?

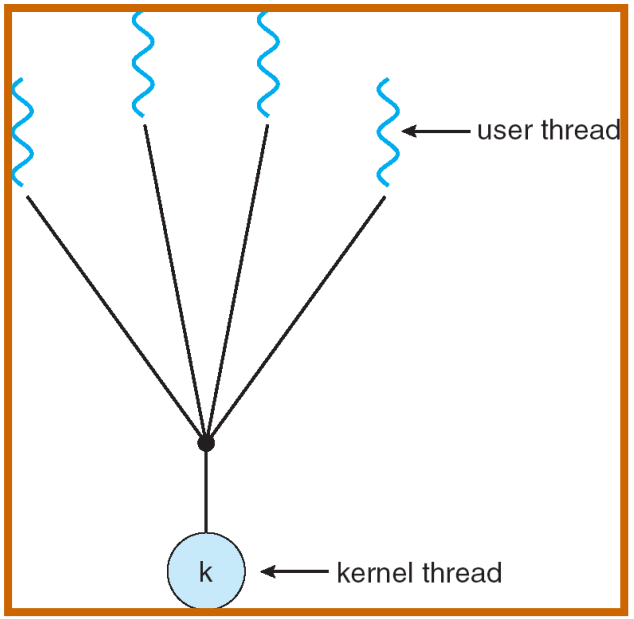
Solution: ***scheduler activations***

- Kernel makes ***upcall*** *to* user thread scheduler when thread would block (like a signal handler)
- User-level scheduler does all thread context switches

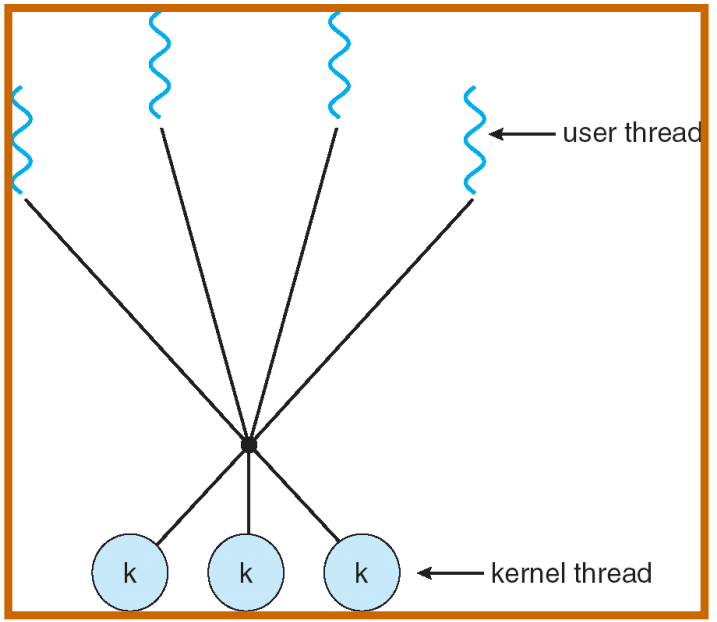
Threading Models

Simple One-to-One Threading Model





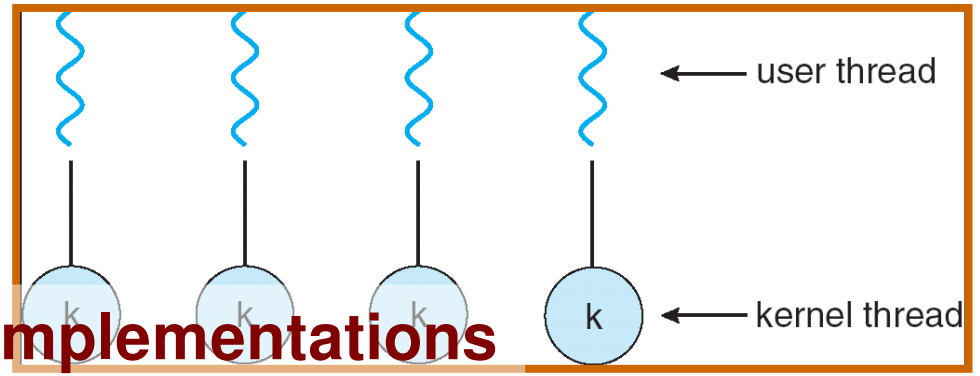
Many-to-One



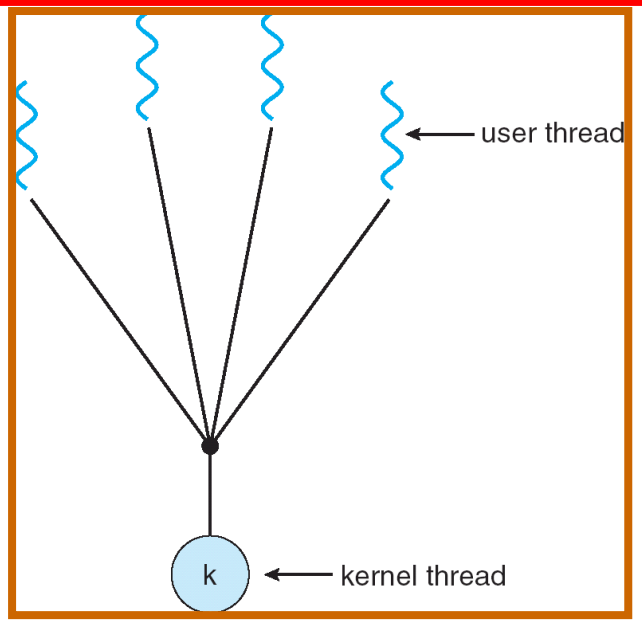
Many-to-Many

Threading Models

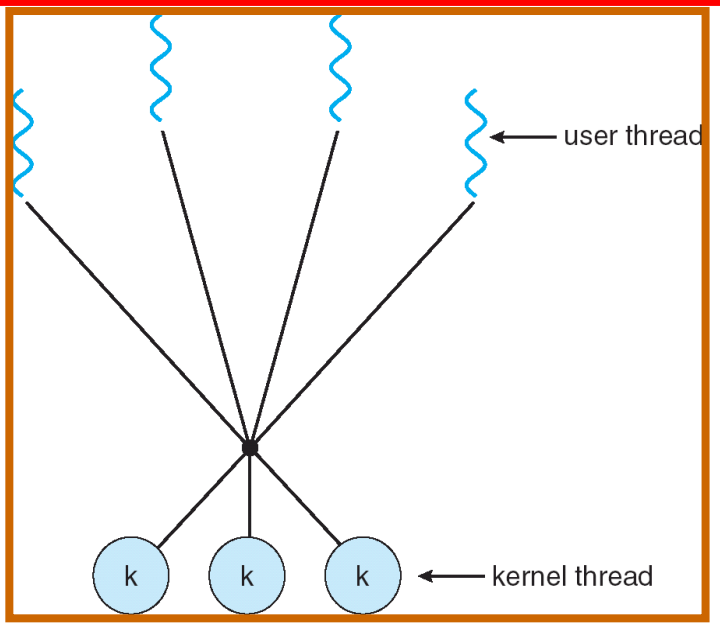
Simple One-to-One Threading Model



Almost all current implementations



Many-to-One



Many-to-Many

Threading Models

Option A: User-level library, one kernel thread (1:N)

- Early Java, ...
- Library does thread context switch
- Kernel time slices between processes, e.g. on I/O

Option B: User-level library, multiple kernel threads (M:N)

- Solaris <9, FreeBSD <7, ...
- Kernel still time slices on I/O

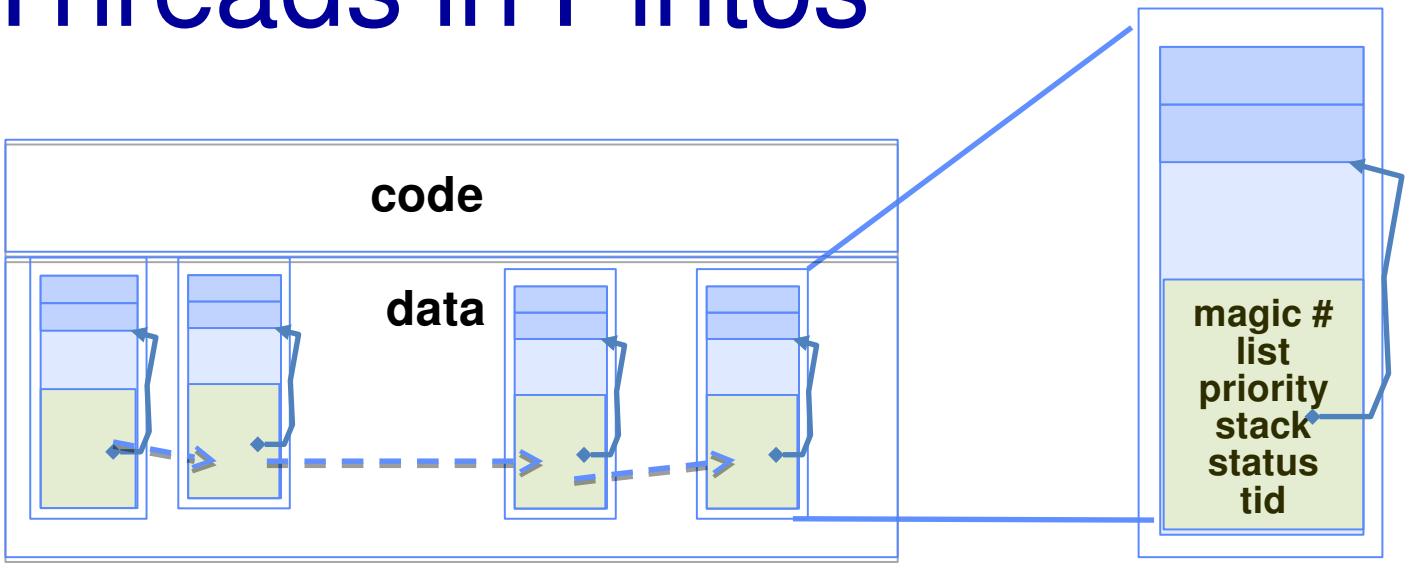
Option C: Scheduler activations, kernel thread/CPU (M:N + upcalls)

- NetBSD < 5, Windows (option), ...

Option D: Kernel thread per user thread (1:1)

- Linux, OS X, Windows (default), FreeBSD >= 7, Solaris >= 7, NetBSD >= 5

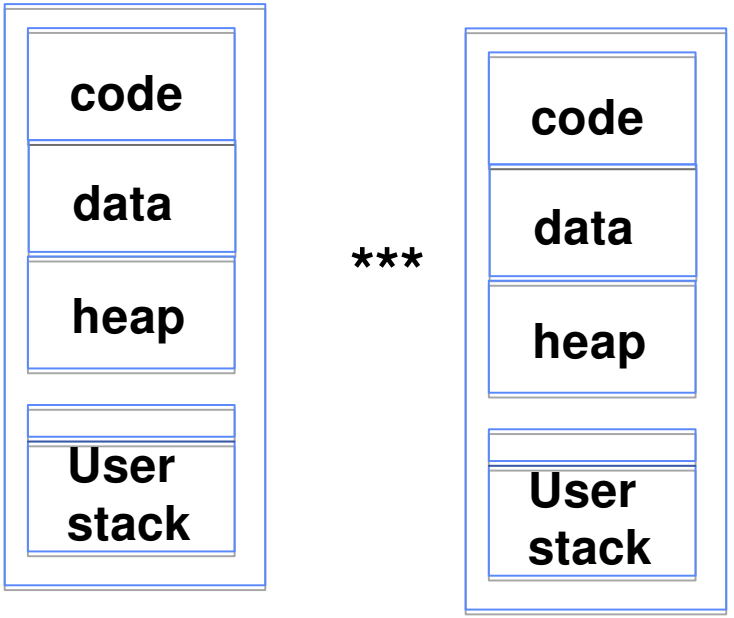
Threads in Pintos



4KiB Thread Stack **and TCB**

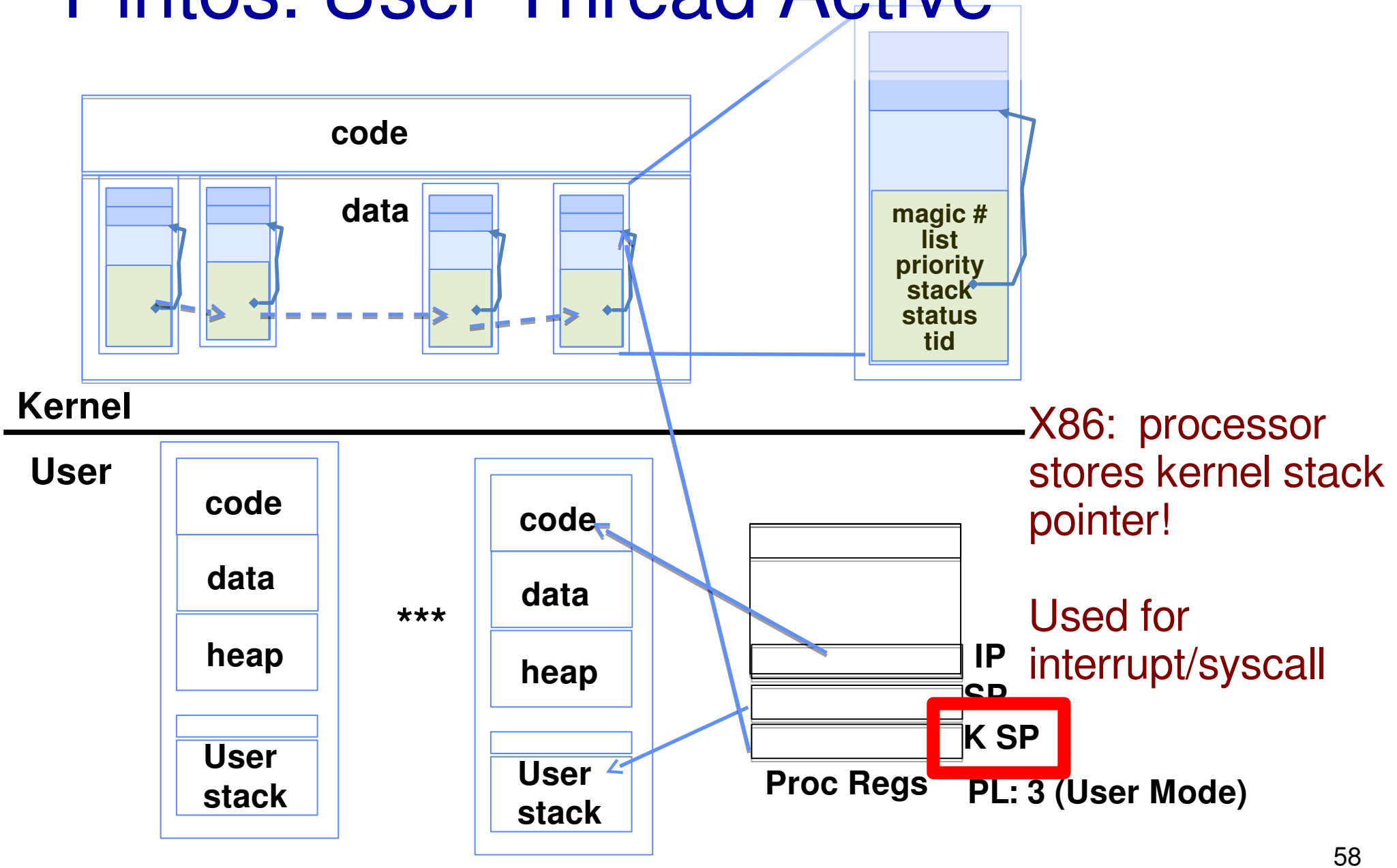
Kernel

User

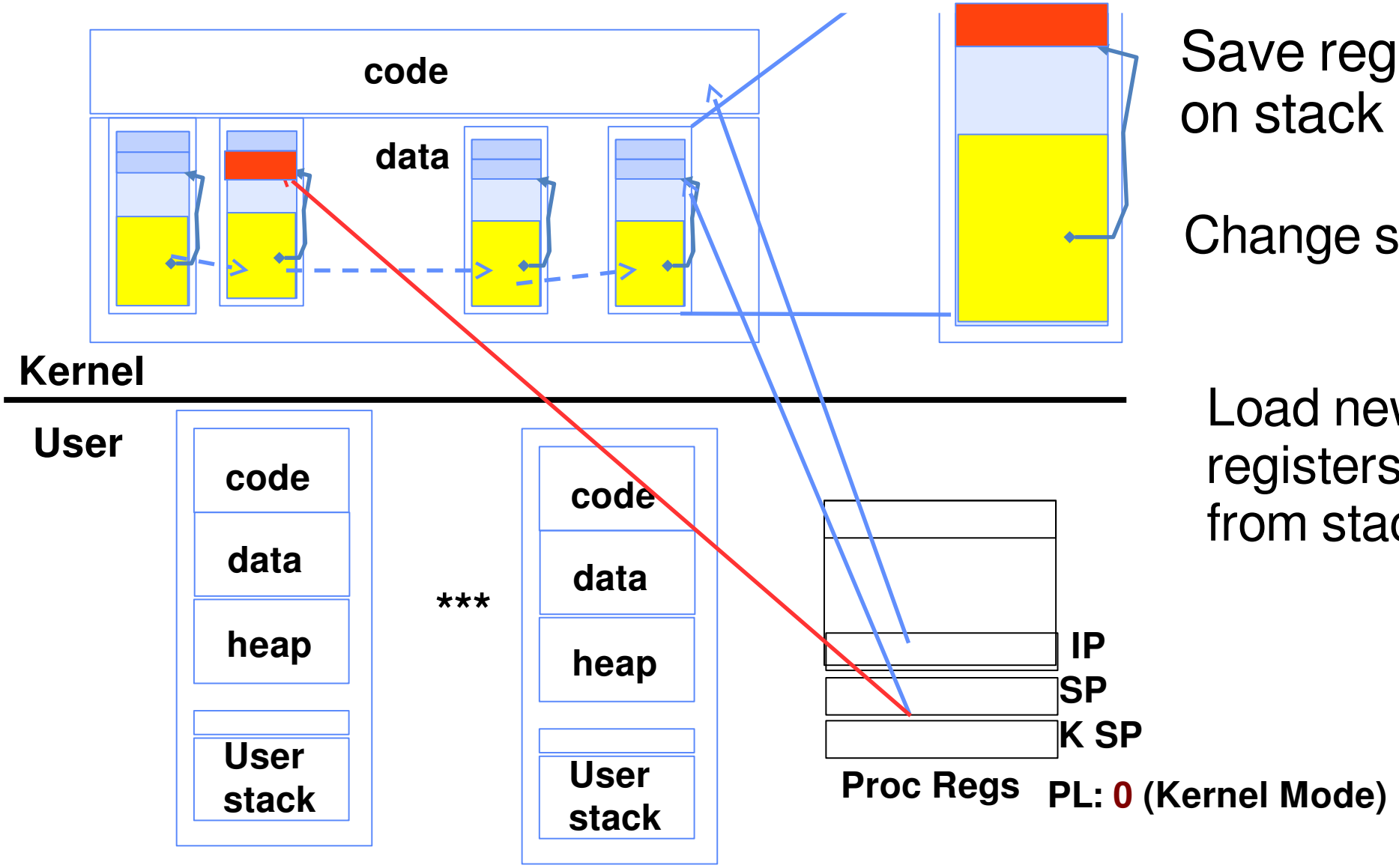


Only single-threaded processes

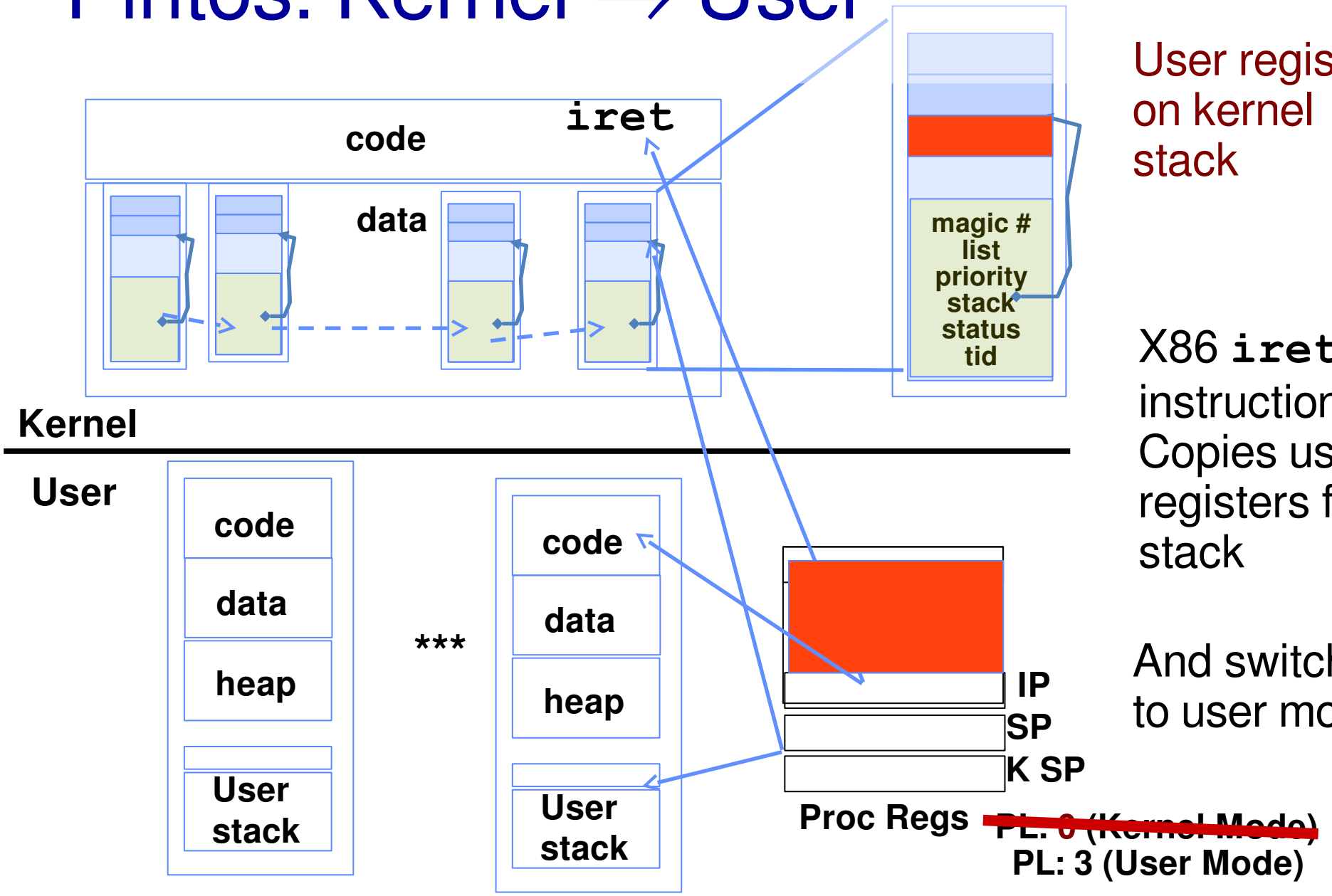
Pintos: User Thread Active



Pintos: Thread Switch (switch.S)



Pintos: Kernel → User

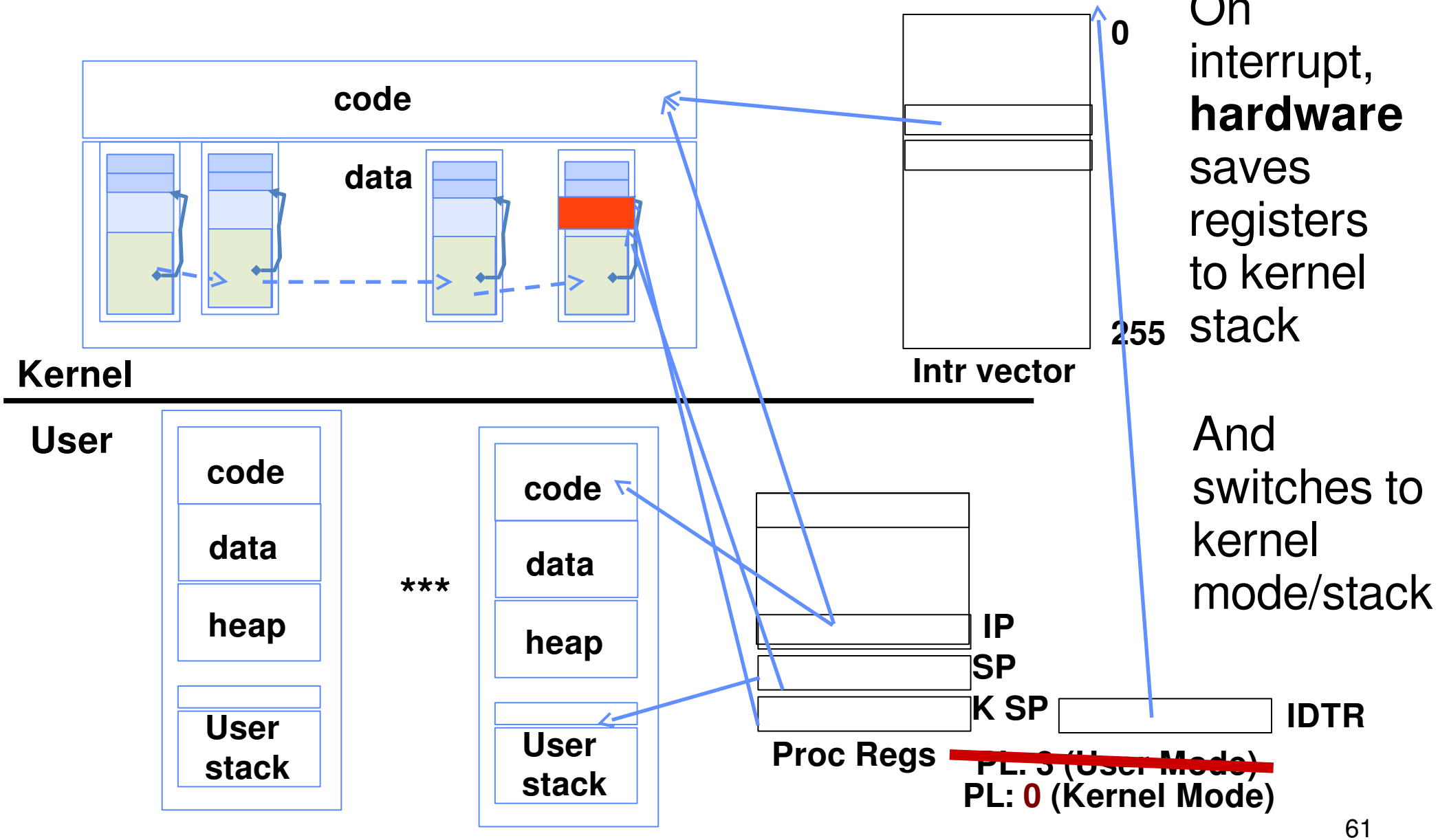


User registers on kernel stack

X86 `iret` instruction – Copies user registers from stack

And switches to user mode

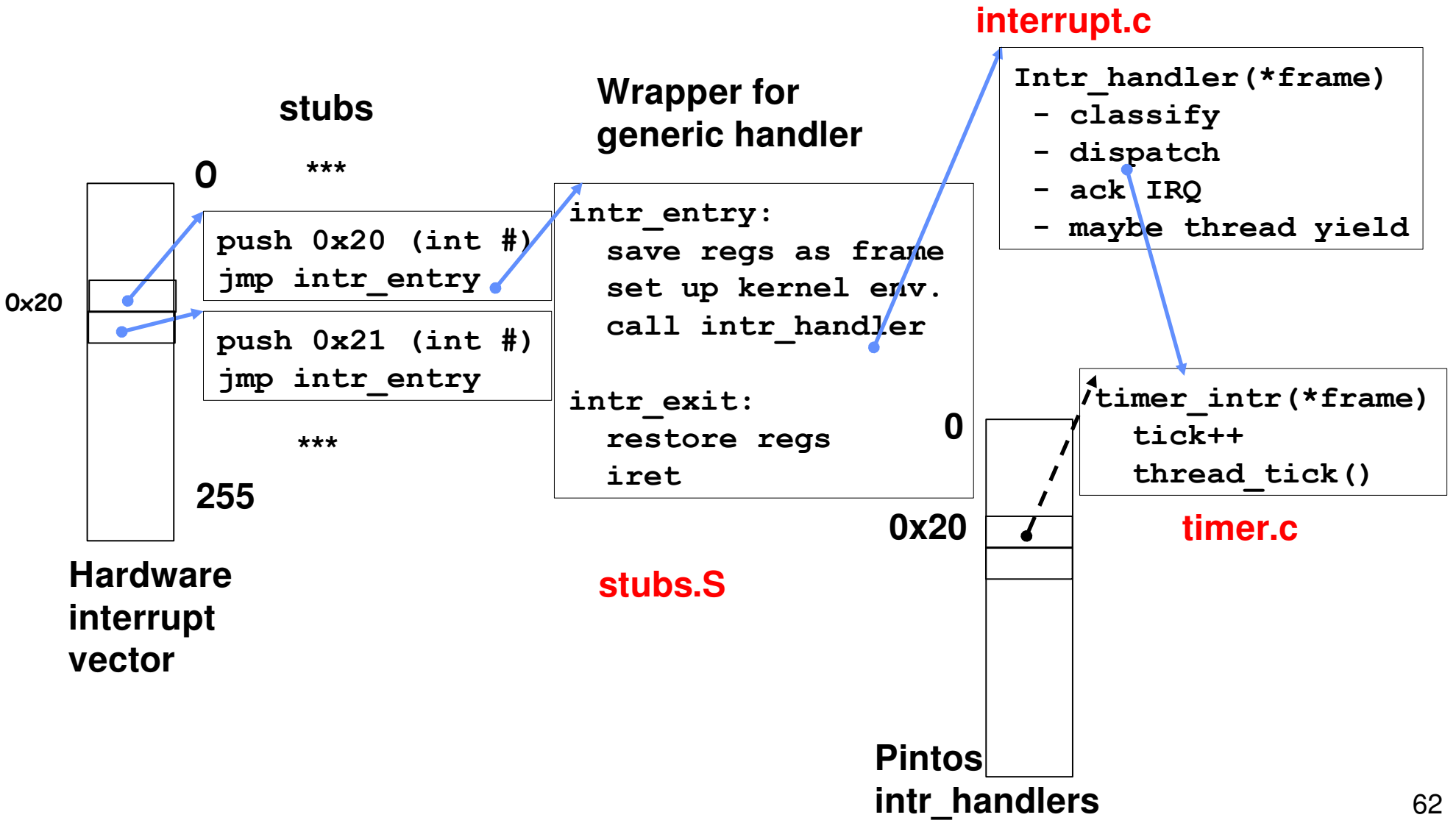
Pintos: User → Kernel



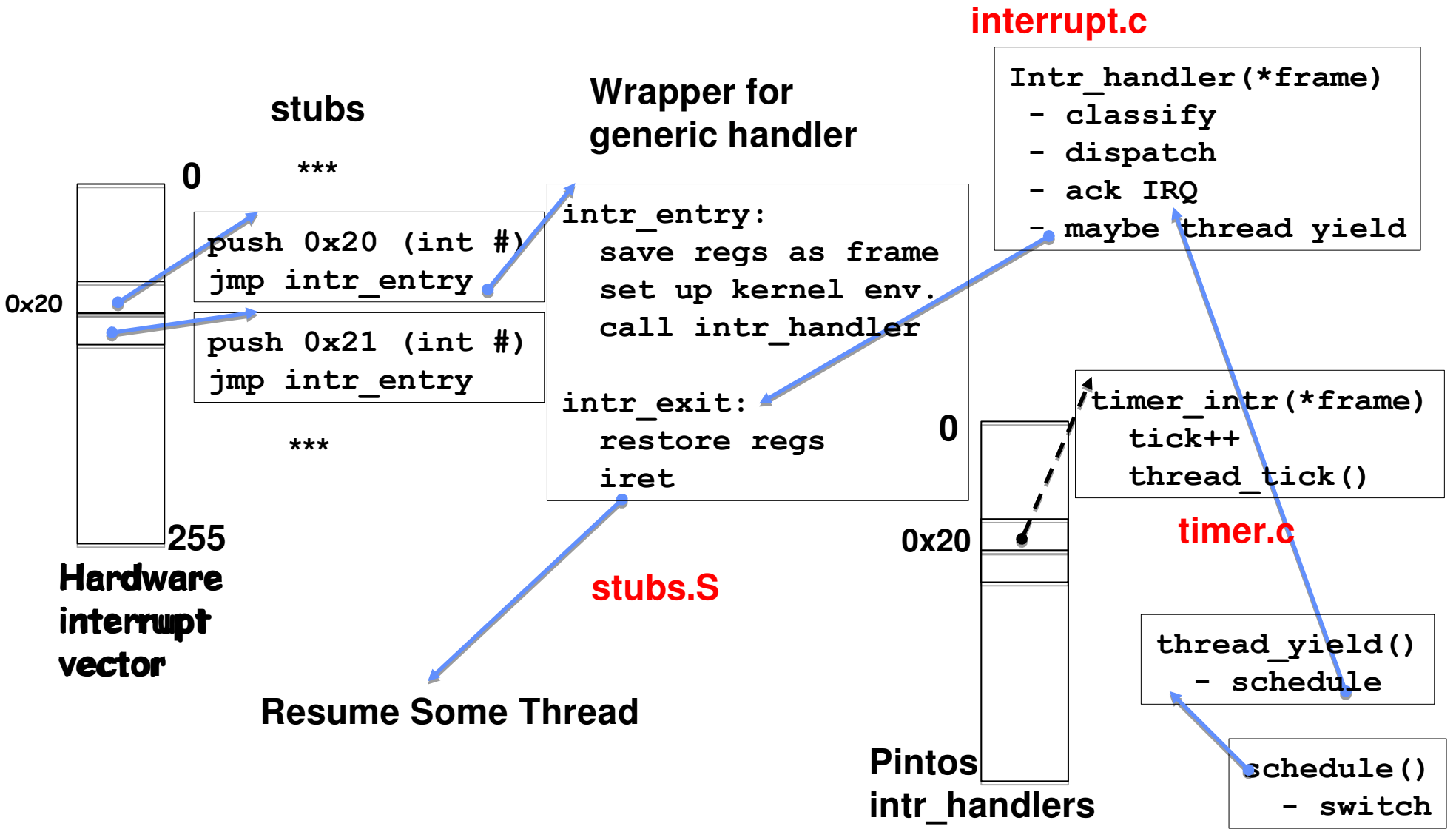
On interrupt, **hardware** saves registers to kernel stack

And switches to kernel mode/stack

Pintos Interrupt Handling (1)



Pintos Interrupt Handling (2)



A Note on Terminology

Threads versus Process

Many textbooks talk about **processes** to mean either

If concurrency, think about a "thread"

If protection, think "address space"

Summary

Kernel-supported threads

- Multiple stacks per address space
- Context switch – save registers, load other, return from "other" switch routine
- Scheduler uses Thread Control Blocks not PCBs
- Pointer from TCBs to PCBs

User-mode threads

-

Classifying Operating Systems

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

Threaded Web Server

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

One thread per connection

Problem: How fast is creating threads? (Better than **`fork()`**, but...)

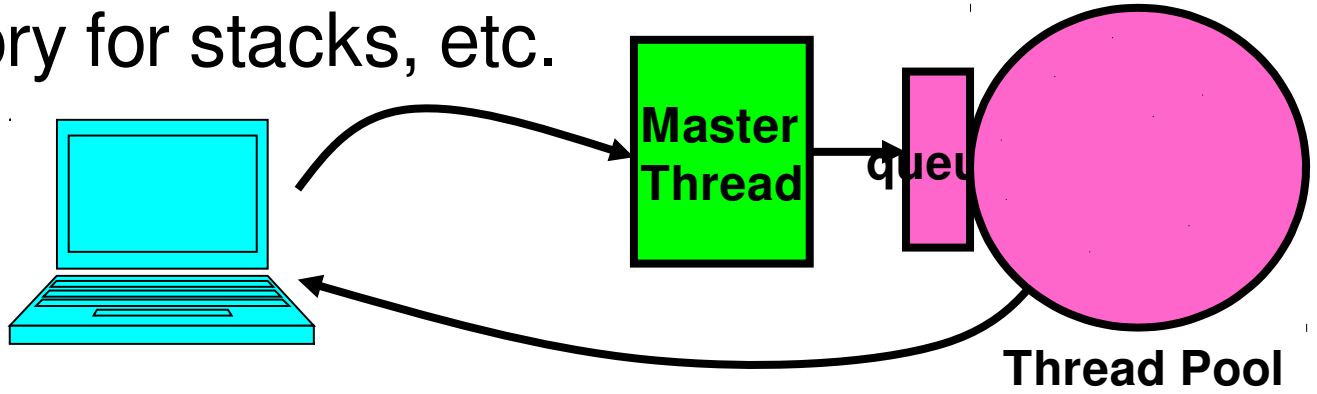
Problem: What if we get a lot of requests?

- Might run of memory (thread stacks)
- Schedulers usually have trouble with too many threads

Threaded Web Server: Thread Pools

Bounded pool of worker threads

- Allocated **in advance**: no thread creation overhead
- **Queue** of pending requests instead of running of memory for stacks, etc.




```
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);
    }
}
```