

CS162
Operating Systems and
Systems Programming
Lecture 11

Scheduling 1:
Concepts and Classic Policies

February 26th, 2026

Prof. John Kubiawicz

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

Recall: Basic Structure of *Mesa* Monitor Program

- Monitors represent the synchronization logic of the program
 - Wait if necessary
 - Signal when change something so any waiting threads can proceed
- Basic structure of mesa monitor-based program:

```
lock
while (need to wait) {
    condvar.wait();
}
unlock
```

} Check and/or update
state variables
Wait if necessary

do something so no need to wait

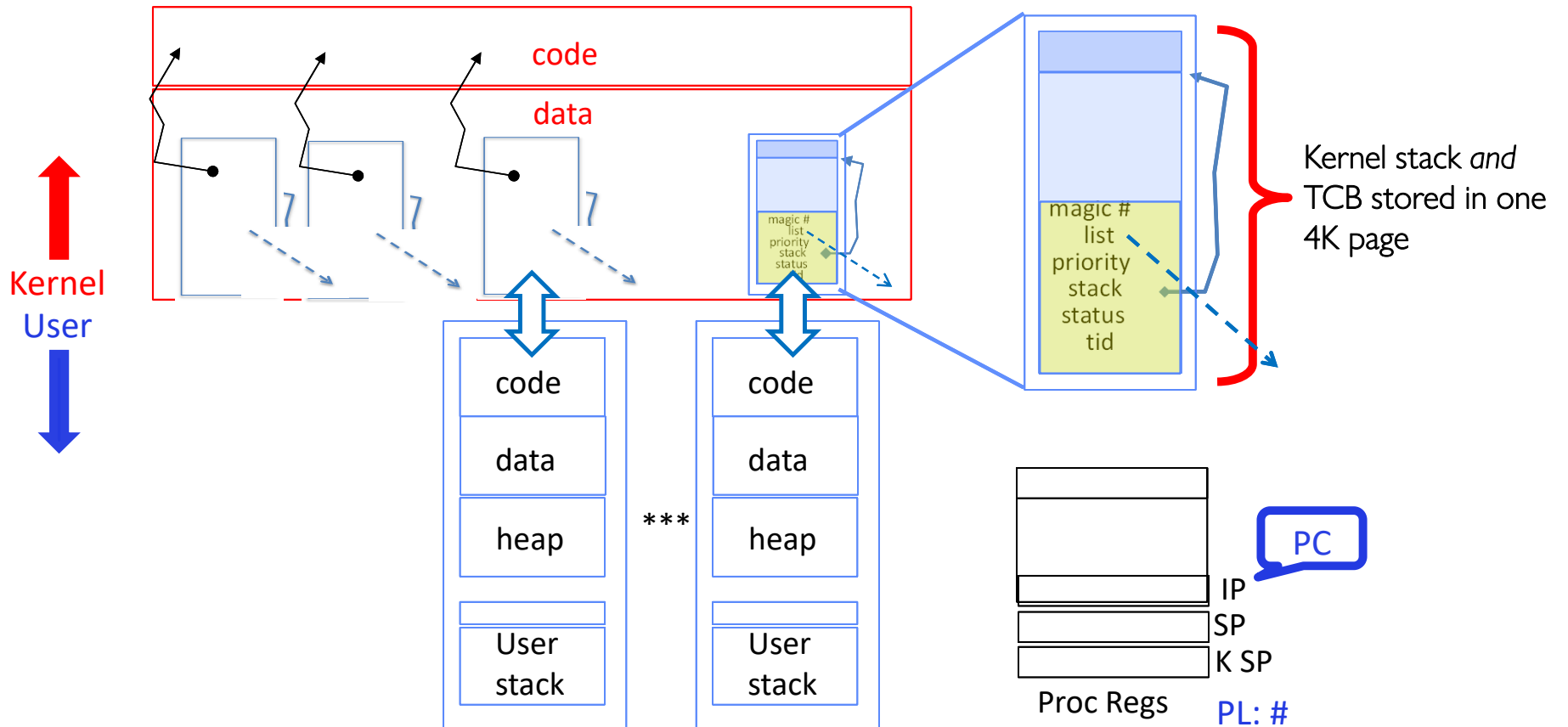
```
lock

condvar.signal();

unlock
```

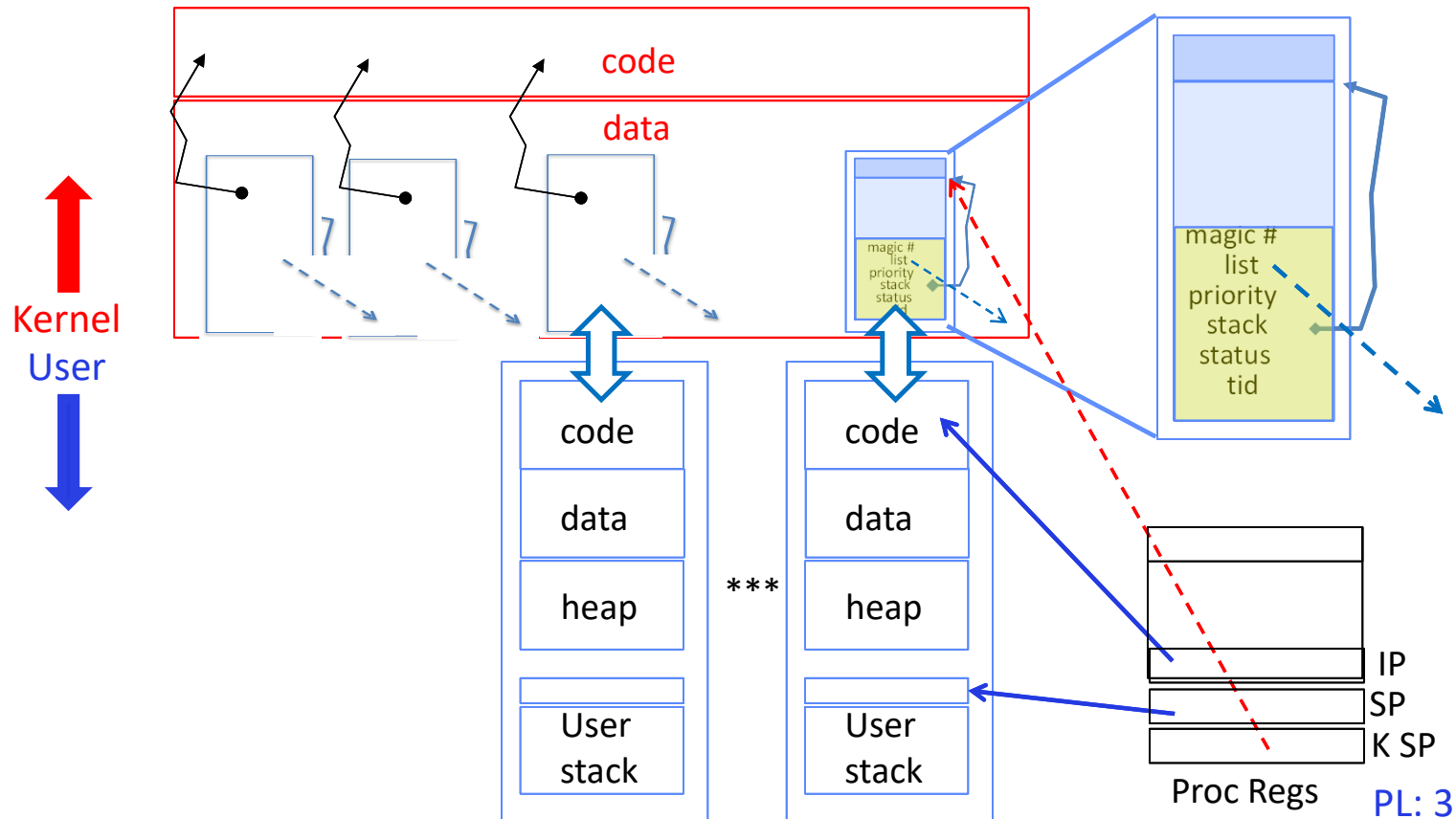
} Check and/or update
state variables

Recall: MT Kernel single Thread Process ala Pintos/x86



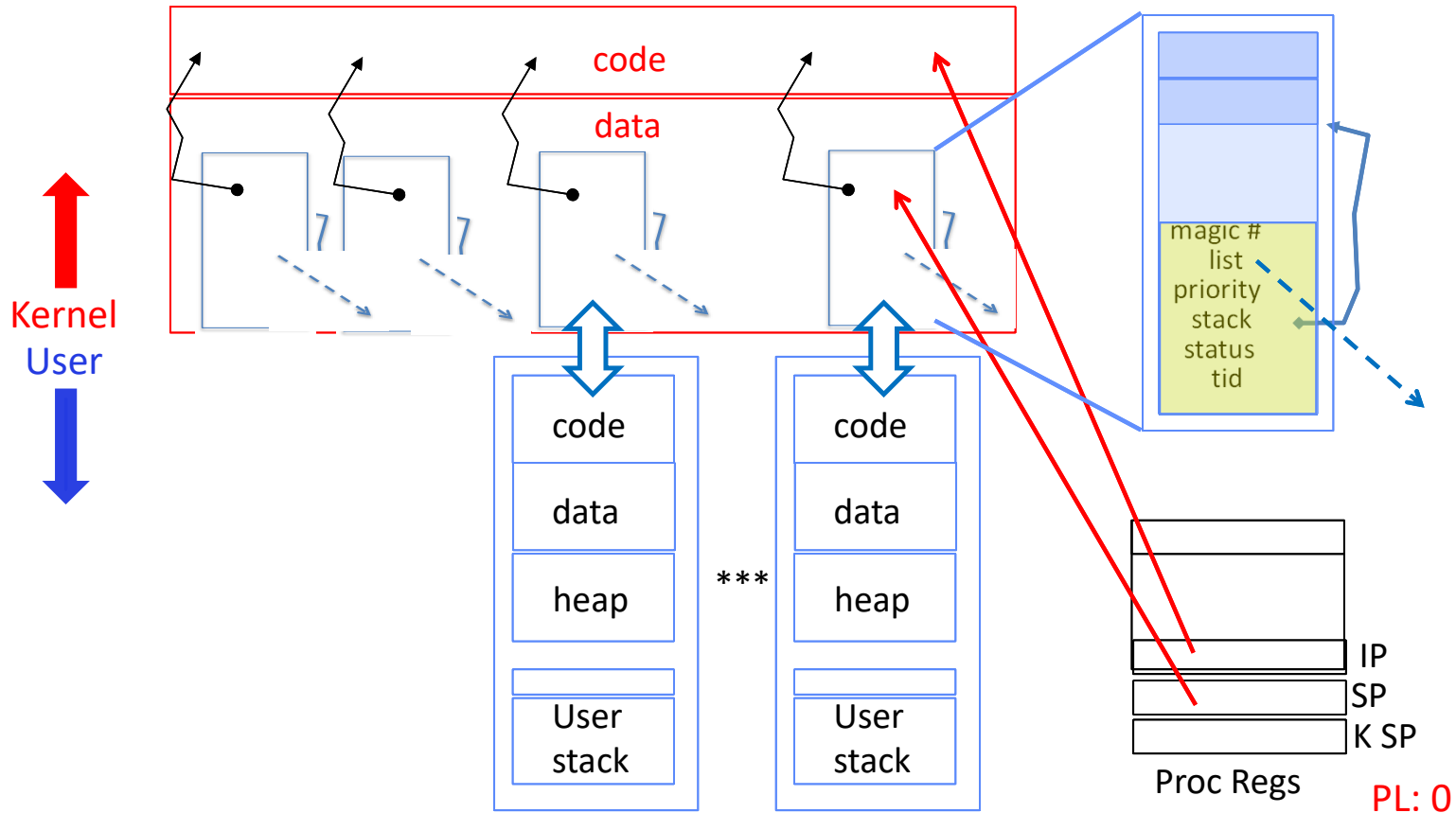
- Each user process/thread associated with a kernel thread, described by a 4KB page object containing TCB and kernel stack for the kernel thread

Running User Code with Kernel stack waiting



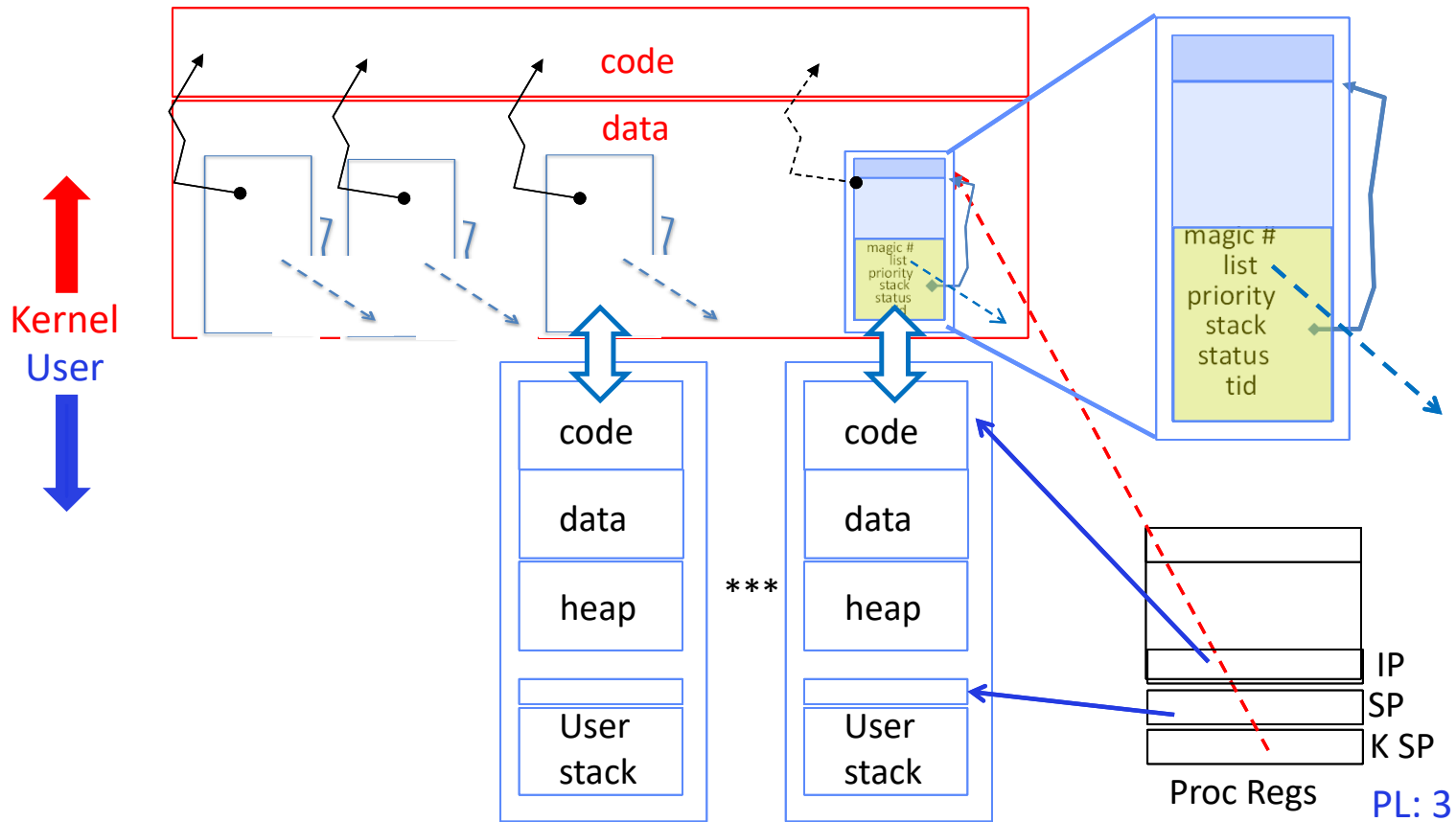
- x86 CPU holds interrupt SP in register
- During user thread execution, associated kernel thread is “standing by”

User → Kernel (interrupts, syscalls)



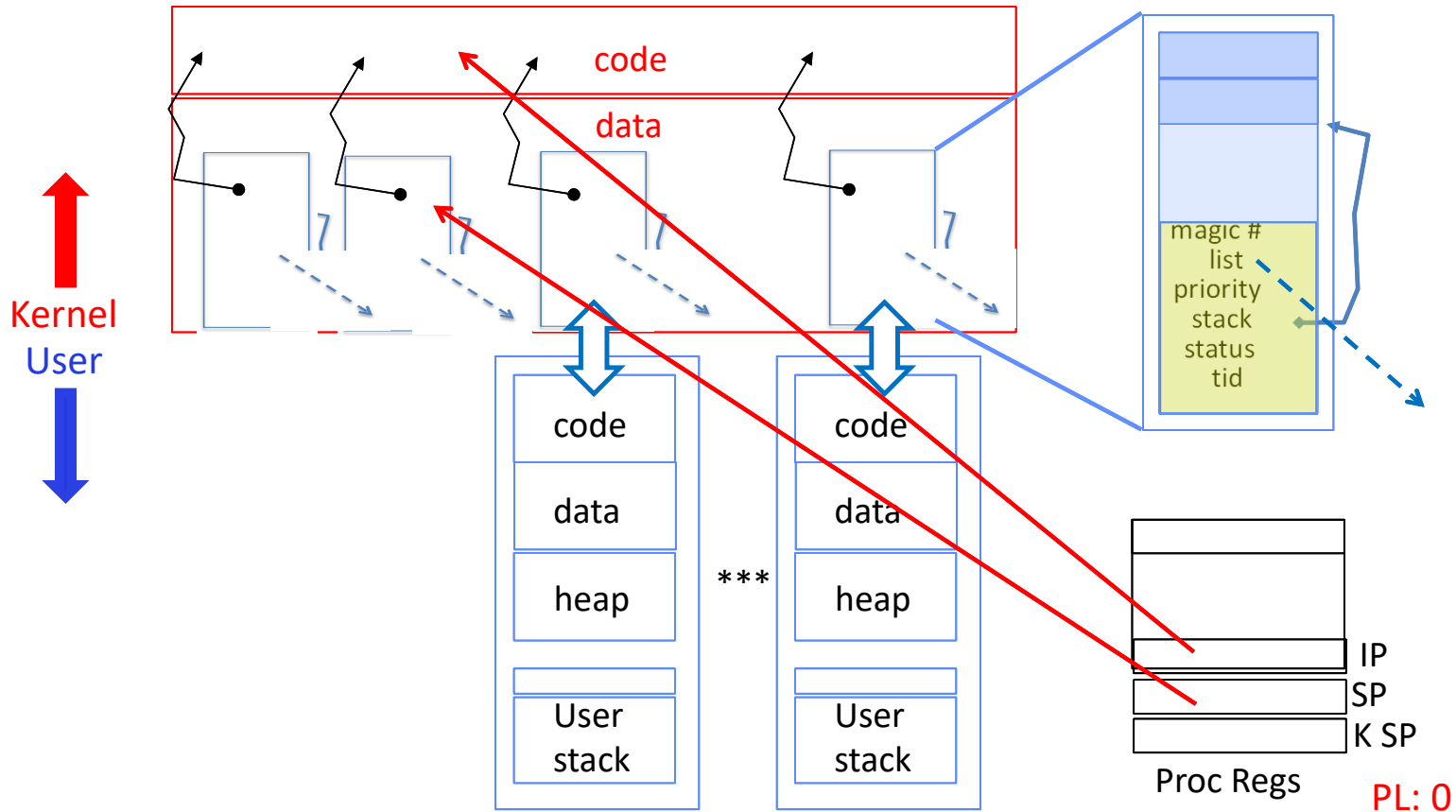
- Mechanism vectors through “interrupt vector”

Kernel → User



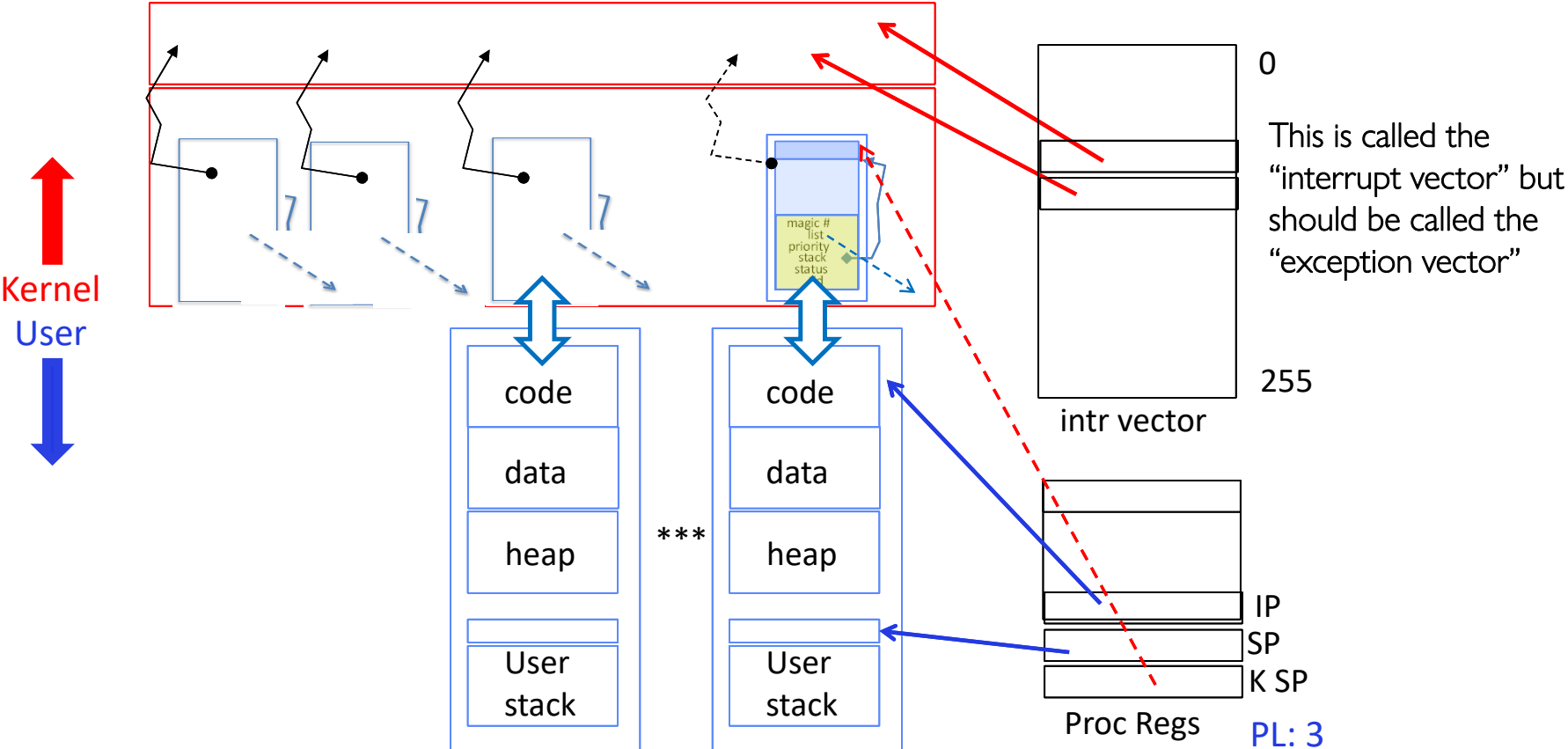
- Interrupt return (iret) restores user stack, IP, and PL

In Kernel Thread: No User Component



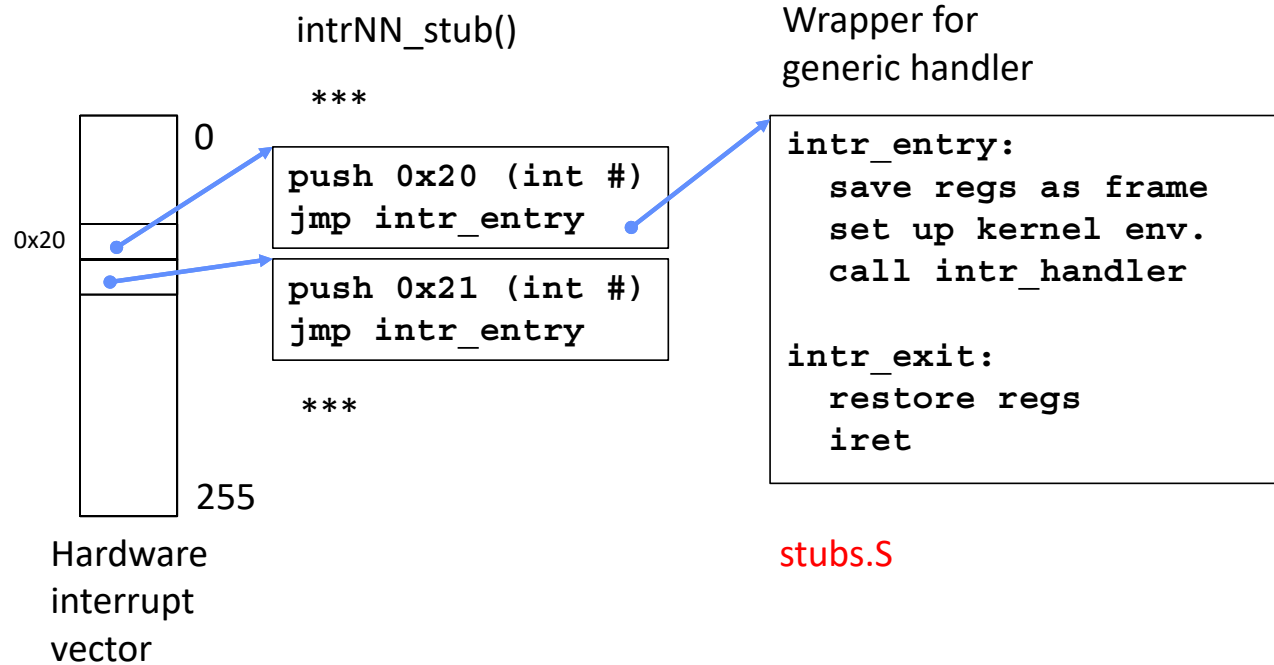
- Kernel threads execute with small stack in thread structure
- Pure kernel threads have no corresponding user-mode thread

User → Kernel via “interrupt vector” (interrupts & traps)

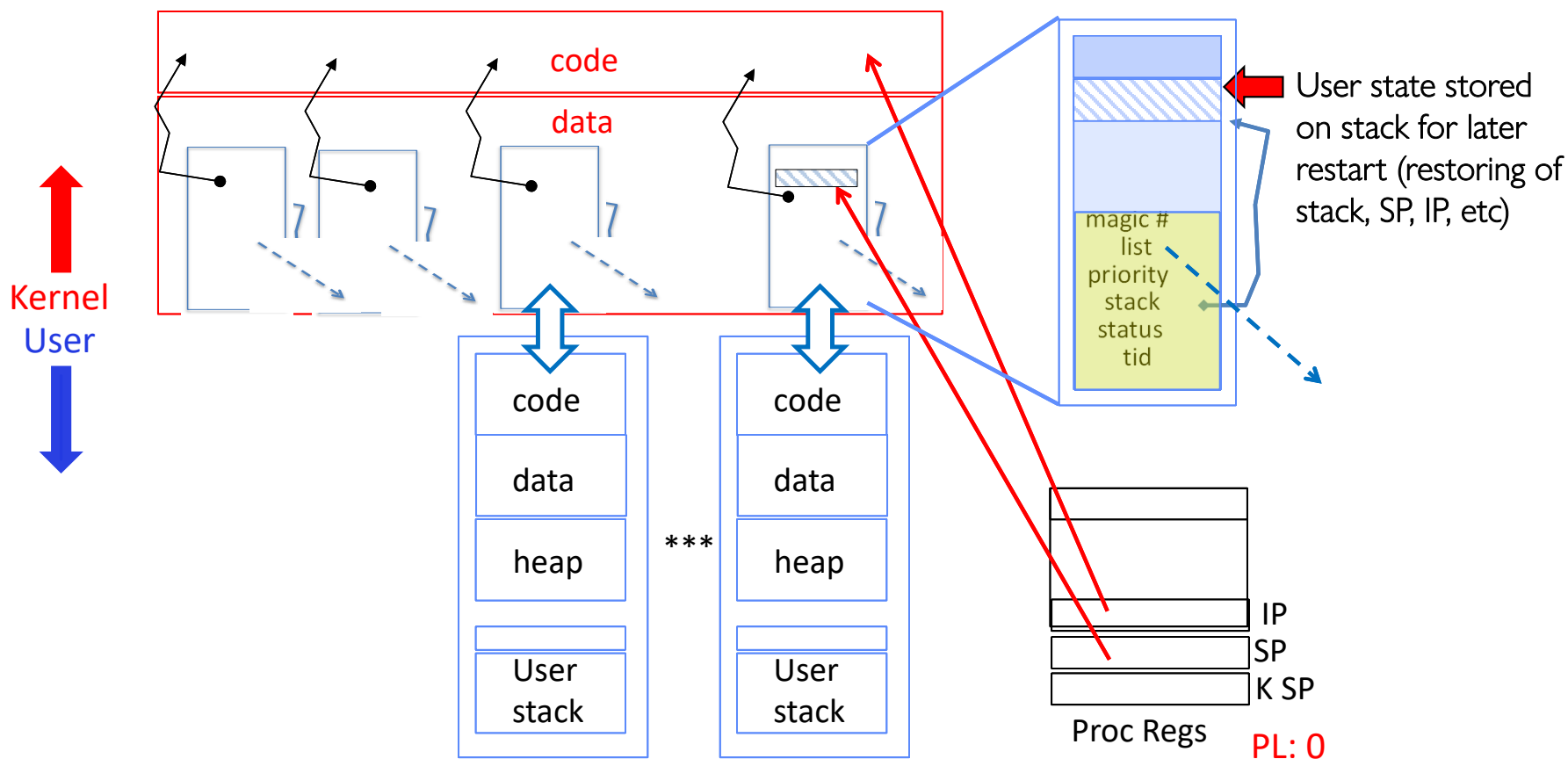


- Interrupts (timer) or trap (syscall, page fault) transfers through interrupt vector (IDT)
 - Each slot for different exception type

Pintos Interrupt Processing for Timer (0x20)

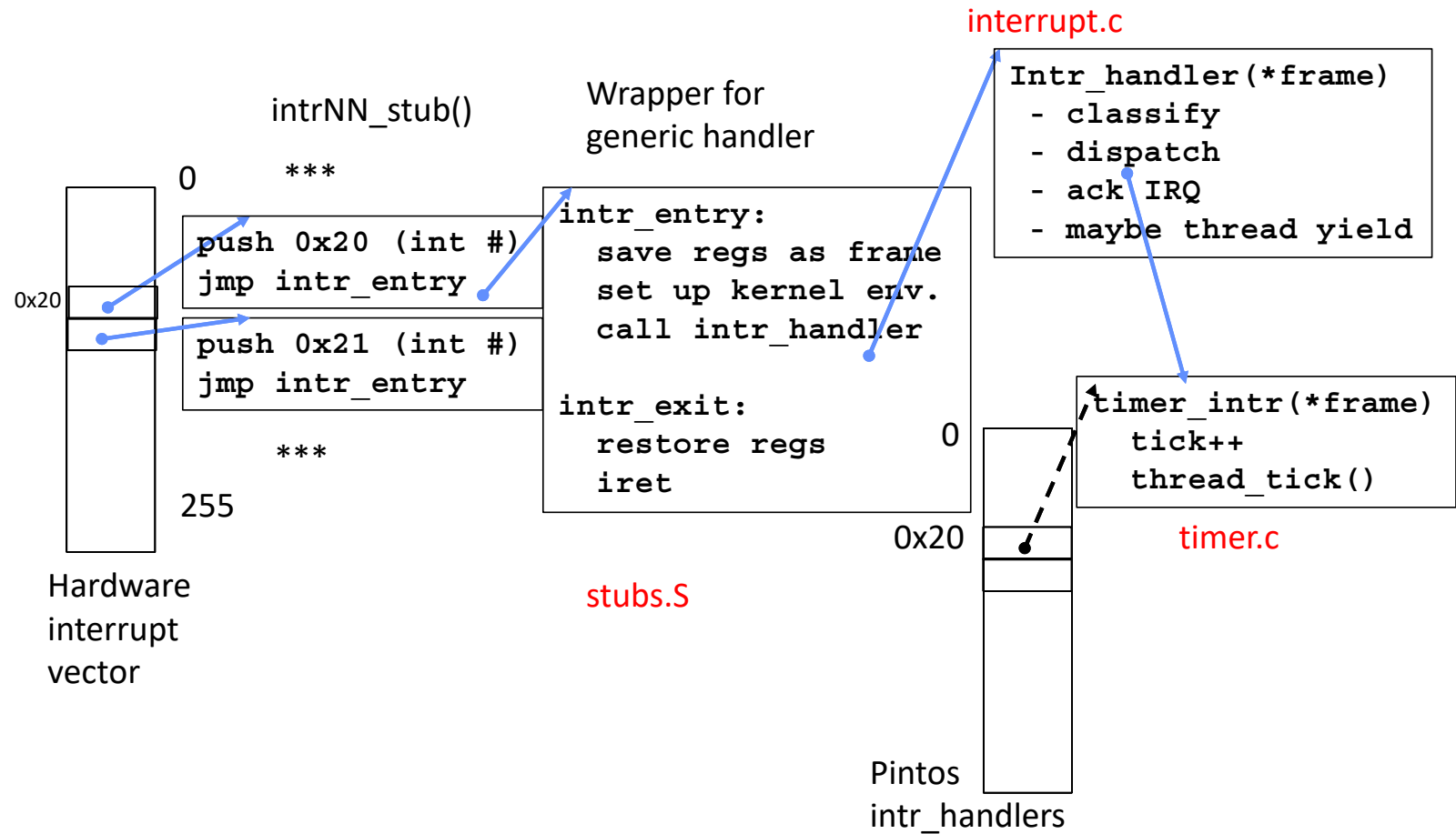


Switch to Kernel Stack for Thread

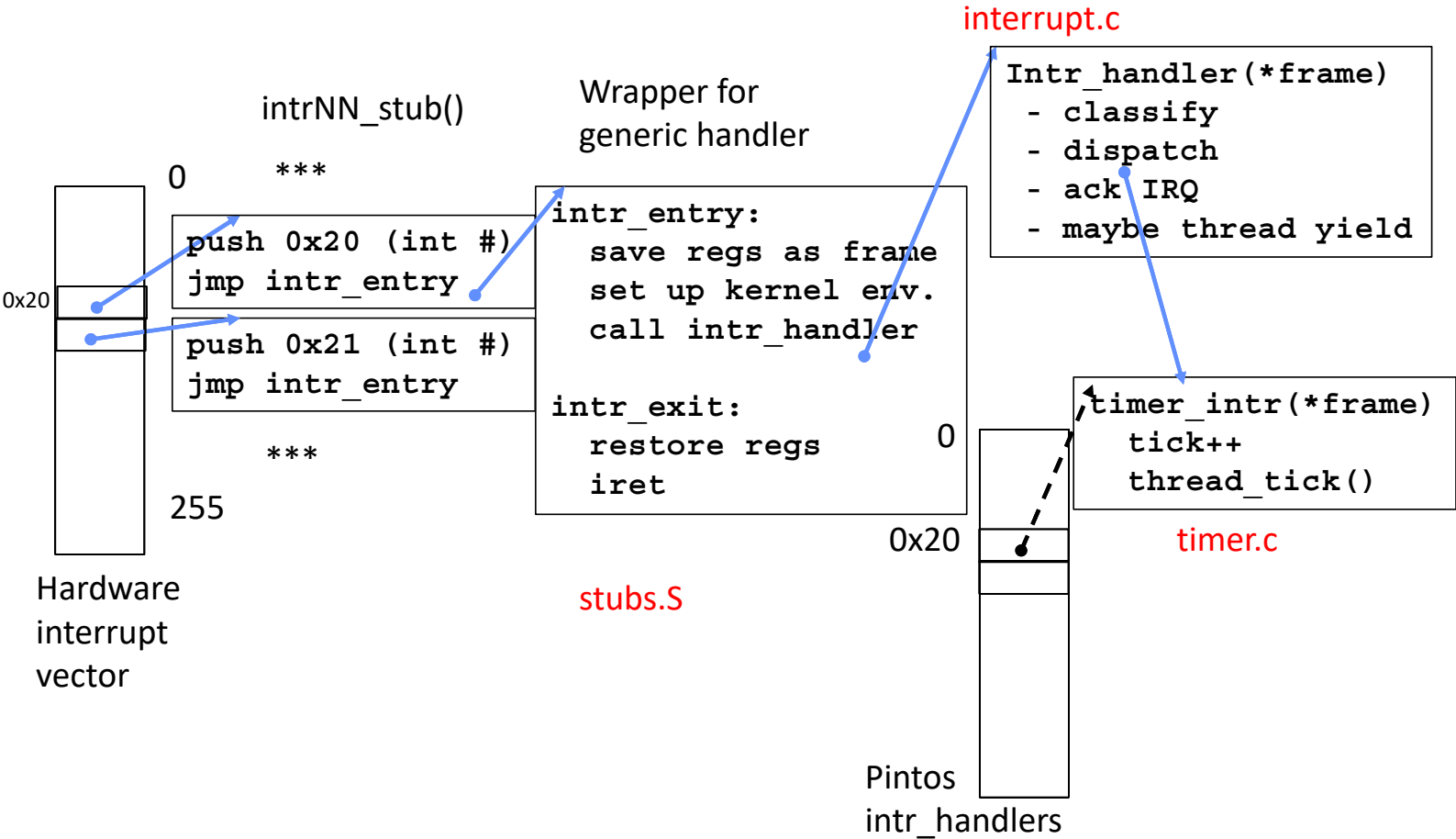


- Information required to restart thread stored on kernel stack
 - Switching becomes simple to different kernel stack and restoring

Pintos Interrupt Processing for Timer (0x20)



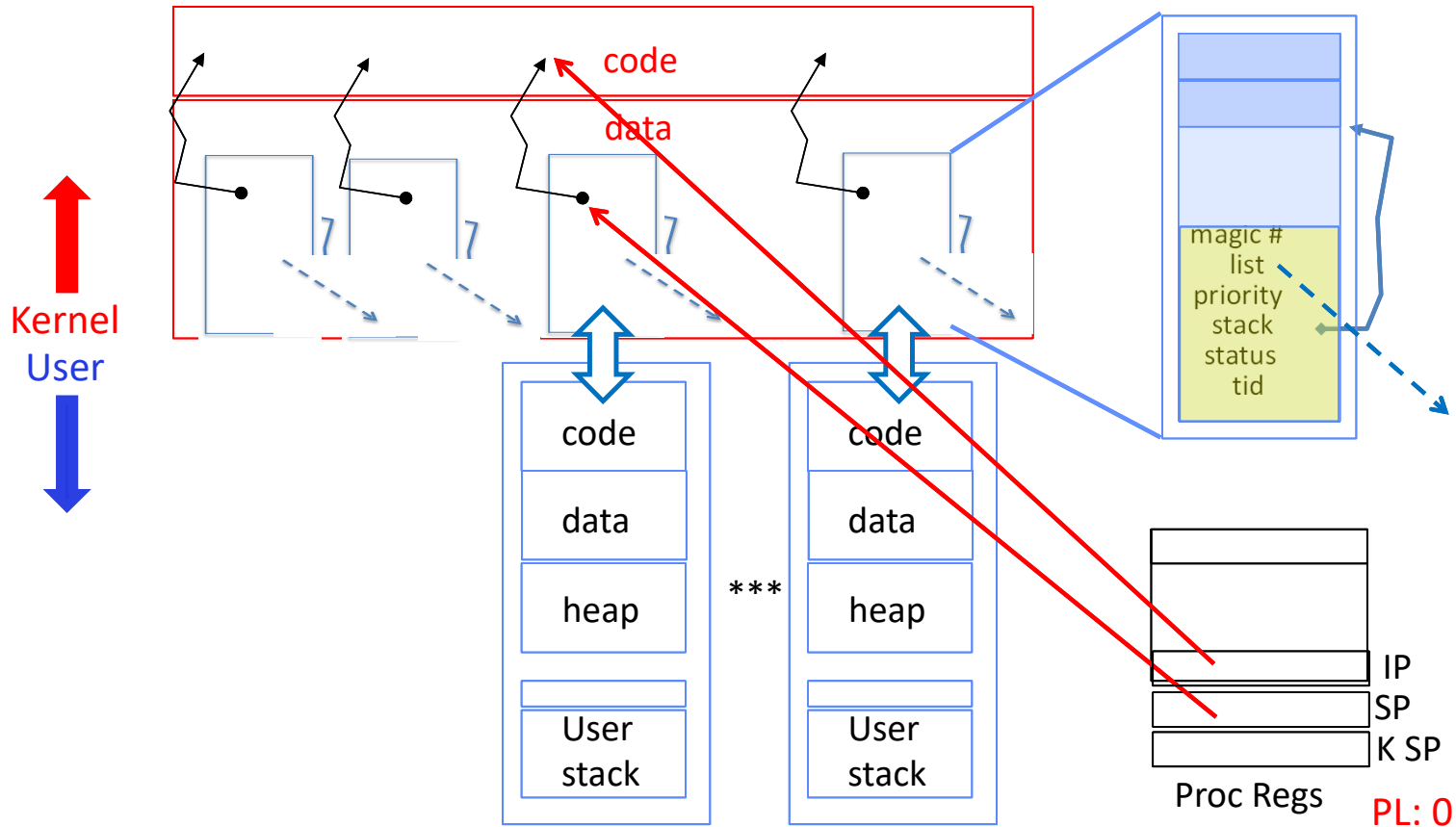
Pintos Interrupt Processing for Timer (0x20)



Timer may trigger thread switch

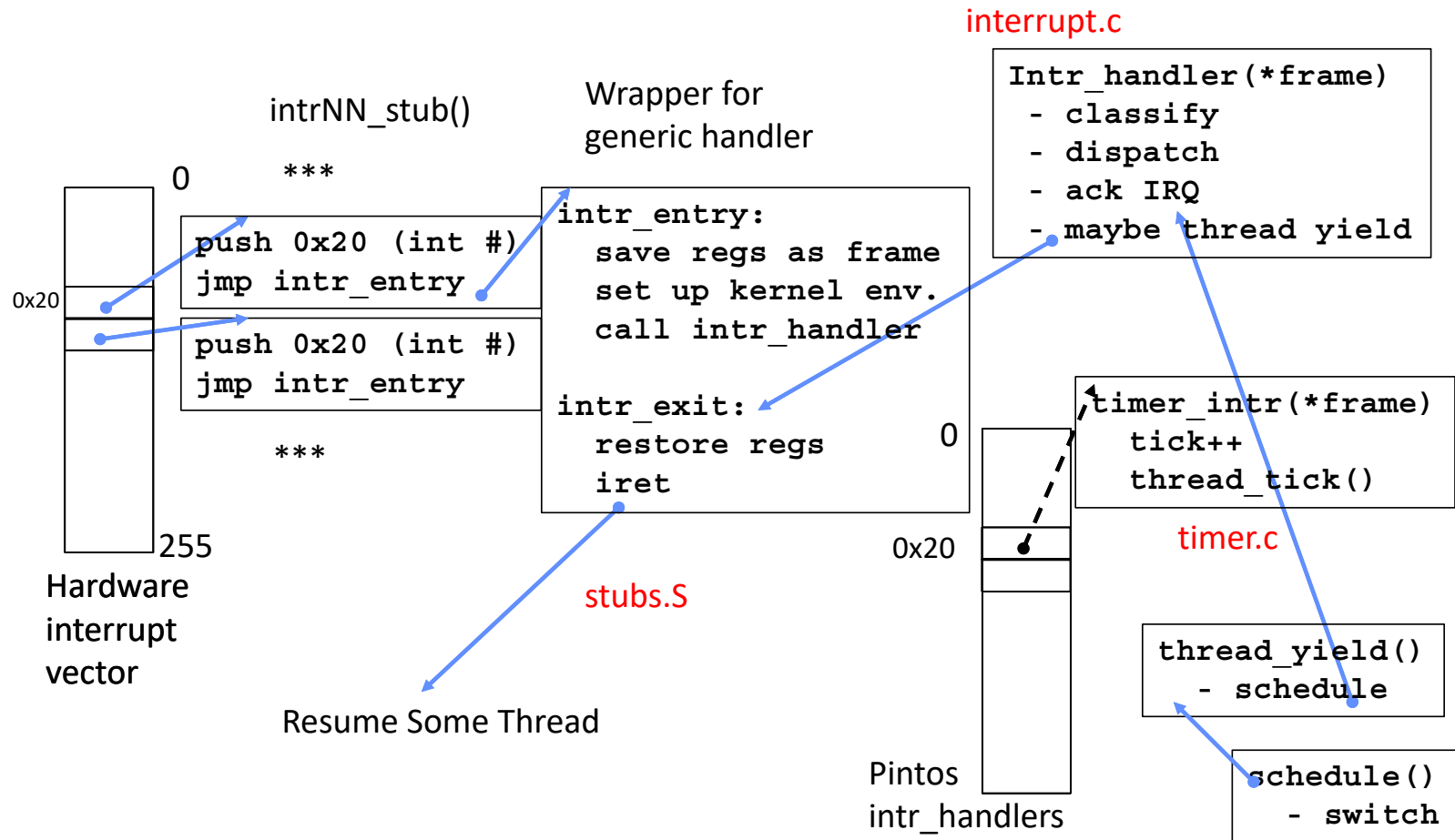
- thread_tick
 - Updates thread counters
 - If quanta exhausted, sets yield flag
- thread_yield
 - On path to rtn from interrupt
 - Sets current thread back to READY
 - Pushes it back on ready_list
 - Calls schedule to select next thread to run upon iret
- Schedule
 - Selects next thread to run
 - Calls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

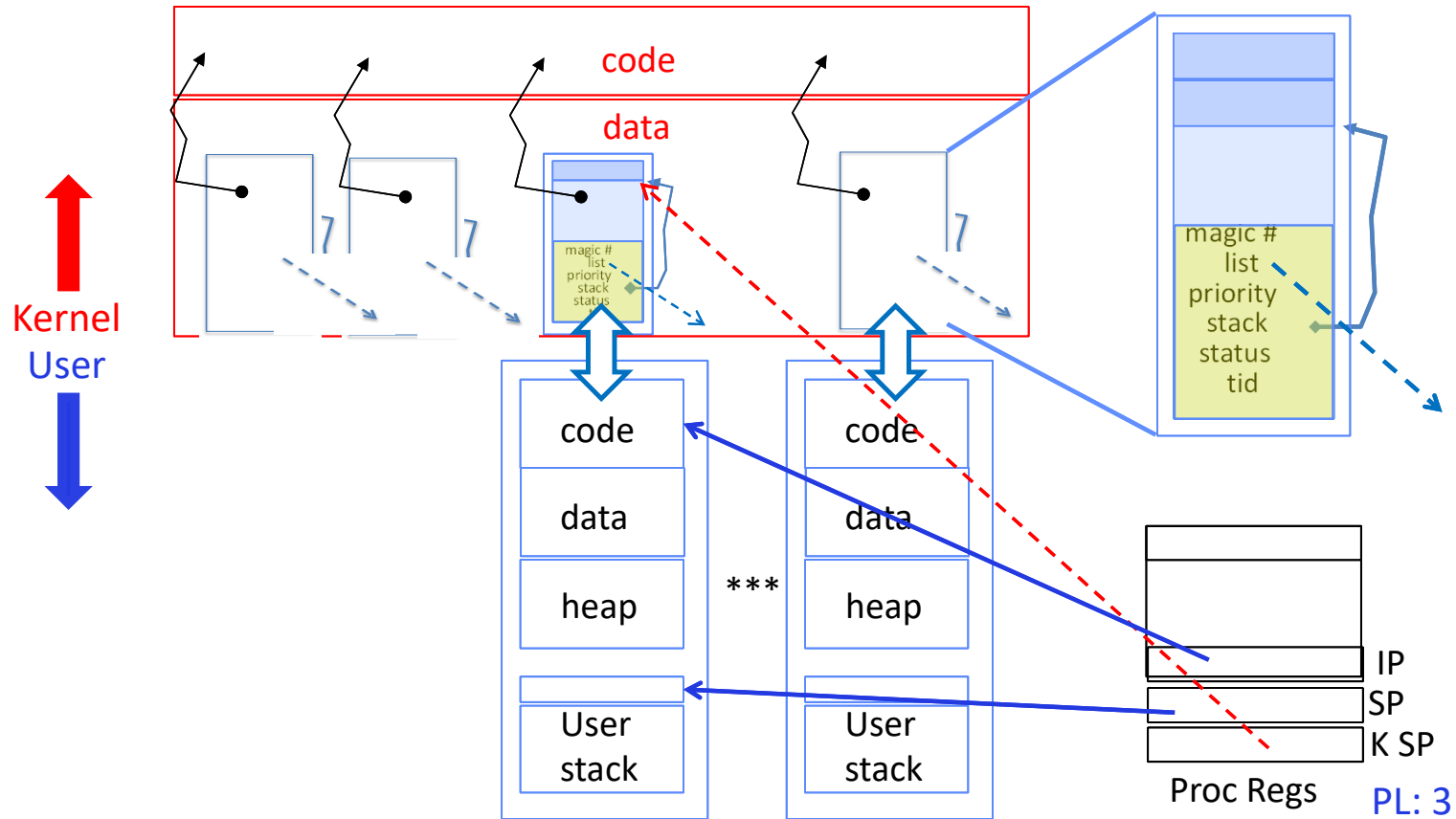


- switch_threads: save regs on current small stack, change SP, return from destination threads call to switch_threads

Pintos Return from Processing for Timer (0x20)



Kernel → Different User Thread



- `iret` restores user stack and priority level (PL)

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.

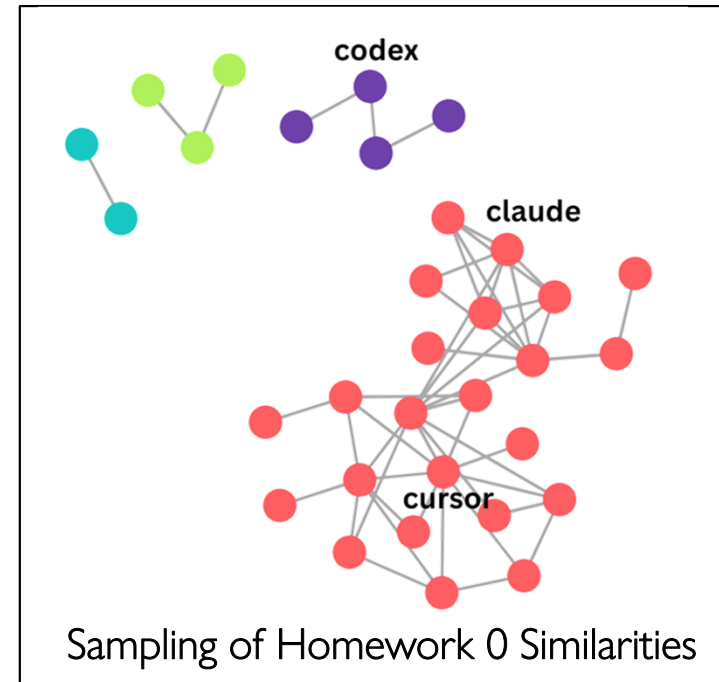
Included by Ali R. Butt in CS3204 from Virginia Tech

Kubiatowicz CS162 © UCB Spring 2026

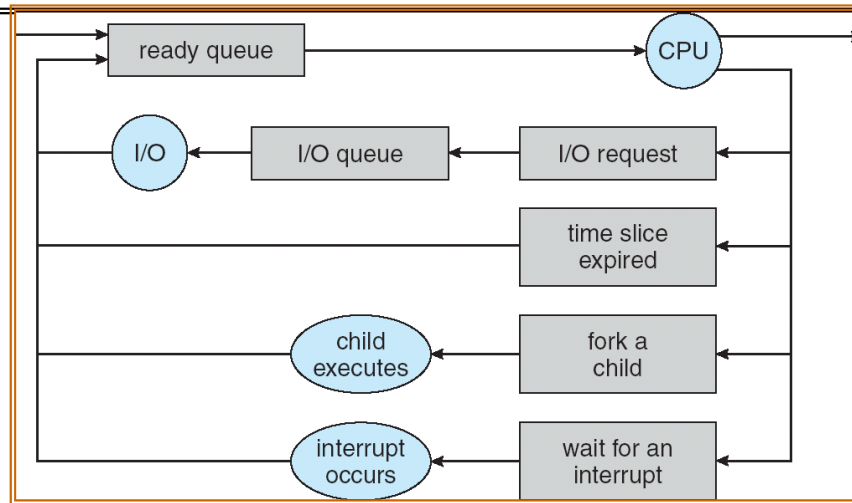
Lec 11.17

Administrivia

- Midterm I:
 - Grading still in process
 - Solutions will be up off the Resources page soon.
- Exam was long? Probably
 - Don't know results yet, but people found coding problems challenging!
 - Sorry about typos. I've corrected them in solution.
 - Hopefully you can get to where interpreting code is much easier \Rightarrow practice!
 - Please take time to understand Problems 3 and 5!
- Reminder: Use of LLMs are strictly forbidden
 - We compare all project submissions against prior year submissions and online solutions and will take actions (described on the course overview page) against offenders
 - We also compare project submissions against AI-generated solutions.
 - Do not do it! This is strictly against policy.



Recall: Scheduling



- Question: How is the OS to decide which of several tasks to take off a queue?
- **Scheduling**: deciding which threads are given access to resources from moment to moment
 - Often, we think in terms of CPU time, but could also think about access to resources like network BW or disk access

Scheduling: All About Queues

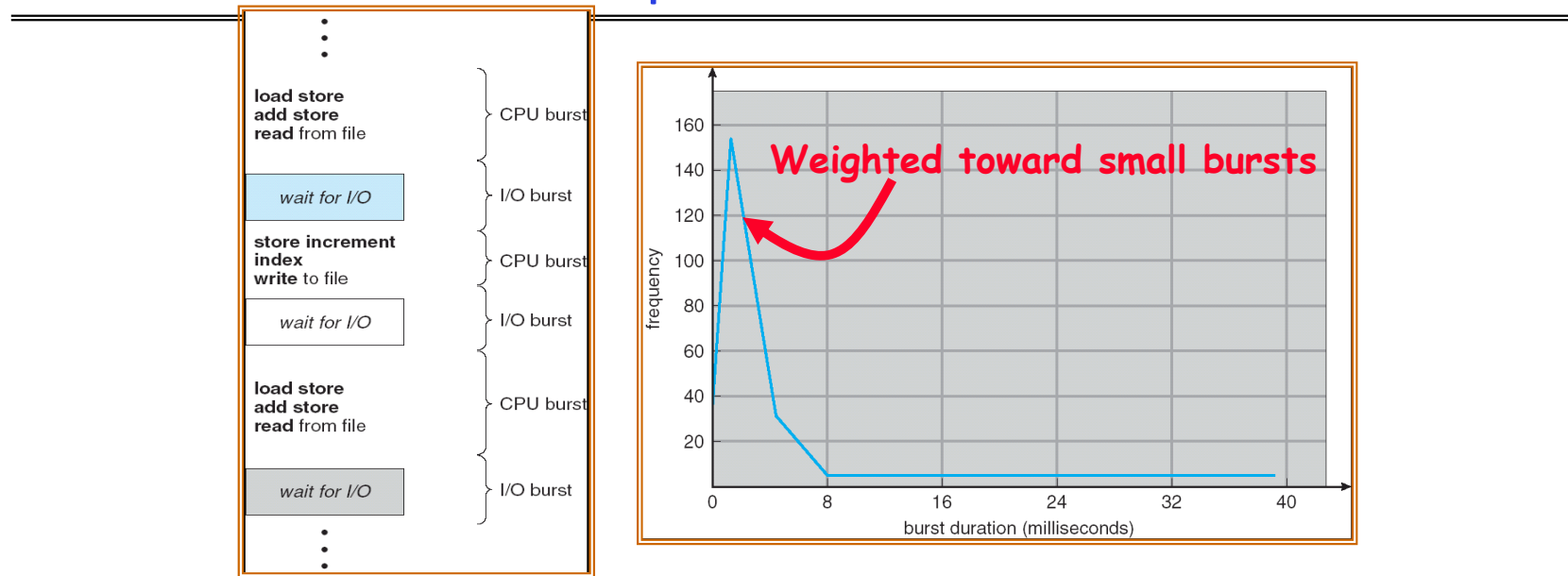


Scheduling Assumptions

- CPU scheduling big area of research in early 70's
- Many implicit assumptions for CPU scheduling:
 - One program per user
 - One thread per program
 - Programs are independent
- Clearly, these are unrealistic but they simplify the problem so it can be solved
 - For instance: is “fair” about fairness among users or programs?
 - » If I run one compilation job and you run five, you get five times as much CPU on many operating systems
- The high-level goal: Dole out CPU time to optimize some desired parameters of system



Assumption: CPU Bursts



- Execution model: programs alternate between bursts of CPU and I/O
 - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
 - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
 - With timeslicing, thread may be forced to give up CPU before finishing current CPU burst

Scheduling Policy Goals/Criteria

- Minimize Response Time
 - Minimize elapsed time to do an operation (or job)
 - Response time is what the user sees:
 - » Time to echo a keystroke in editor
 - » Time to compile a program
 - » Real-time Tasks: Must meet deadlines imposed by World
- Maximize Throughput
 - Maximize operations (or jobs) per second
 - Throughput related to response time, but not identical:
 - » Minimizing response time will lead to more context switching than if you only maximized throughput
 - Two parts to maximizing throughput
 - » Minimize overhead (for example, context-switching)
 - » Efficient use of resources (CPU, disk, memory, etc)
- Fairness
 - Share CPU among users in some equitable way
 - Fairness is not minimizing average response time:
 - » Better *average* response time by making system *less* fair

First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
 - Also “First In, First Out” (FIFO) or “Run until done”
 - » In early systems, FCFS meant one program scheduled until done (including I/O)
 - » Now, means keep CPU until thread blocks



- Example:

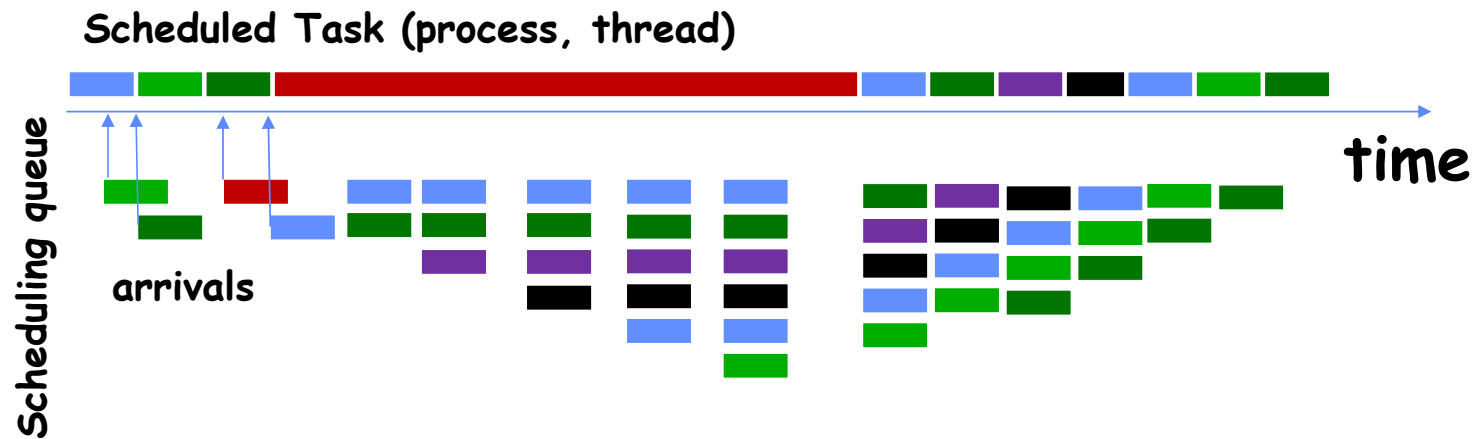
<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- Suppose processes arrive in the order: P_1, P_2, P_3
The Gantt Chart for the schedule is:



- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
- Average Completion time: $(24 + 27 + 30)/3 = 27$
- *Convoy effect*: short process stuck behind long process

Convoy effect



- With FCFS non-preemptive scheduling, convoys of small tasks tend to build up when a large one is running.

FCFS Scheduling (Cont.)

- Example continued:
 - Suppose that processes arrive in order: P2 , P3 , P1
 - Now, the Gantt chart for the schedule is:



- Waiting time for P1 = 6; P2 = 0; P3 = 3
 - Average waiting time: $(6 + 0 + 3)/3 = 3$
 - Average Completion time: $(3 + 6 + 30)/3 = 13$
- In second case:
 - Average waiting time is much better (before it was 17)
 - Average completion time is better (before it was 27)
- FIFO Pros and Cons:
 - Simple (+)
 - Short jobs get stuck behind long ones (-)
 - » Safeway: Getting milk, always stuck behind cart full of items!
 - Upside: get to read about Space Aliens!

Round Robin (RR) Scheduling

- FCFS Scheme: Potentially bad for short jobs!
 - Depends on submit order
 - If you are first in line at supermarket with milk, you don't care who is behind you, on the other hand...
- Round Robin Scheme: **Preemption!**
 - Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds
 - After quantum expires, the process is preempted and added to the end of the ready queue.
 - n processes in ready queue and time quantum is $q \Rightarrow$
 - » Each process gets $1/n$ of the CPU time
 - » In chunks of at most q time units
 - » **No process waits more than $(n-1)q$ time units**



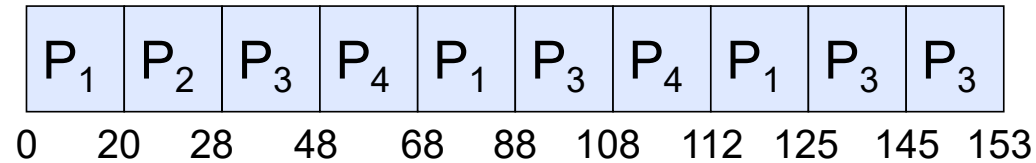
RR Scheduling (Cont.)

- Performance
 - q large \Rightarrow FCFS
 - q small \Rightarrow Interleaved (really small \Rightarrow hyperthreading?)
 - q must be large with respect to context switch, otherwise overhead is too high (all overhead)

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

– The Gantt chart is:



– Waiting time for

$$P_1 = (68 - 20) + (112 - 88) = 72$$

$$P_2 = (20 - 0) = 20$$

$$P_3 = (28 - 0) + (88 - 48) + (125 - 108) = 85$$

$$P_4 = (48 - 0) + (108 - 68) = 88$$

– Average waiting time = $(72 + 20 + 85 + 88) / 4 = 66\frac{1}{4}$

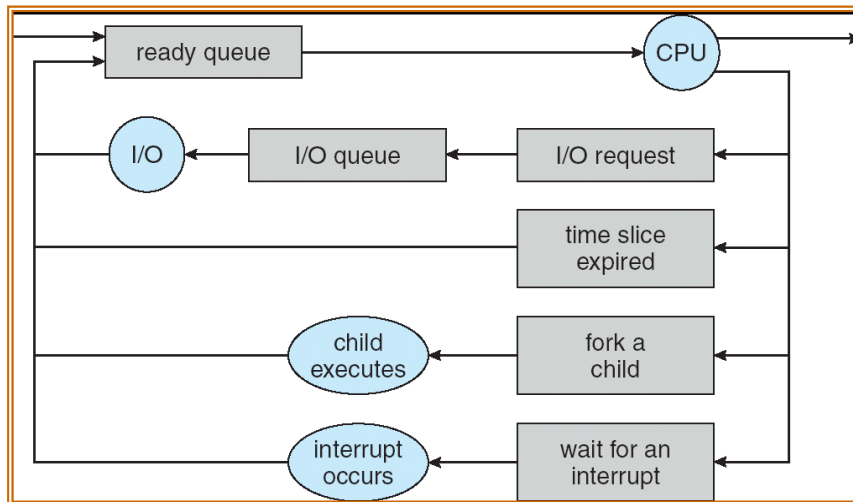
– Average completion time = $(125 + 28 + 153 + 112) / 4 = 104\frac{1}{2}$

• Thus, Round-Robin Pros and Cons:

- Better for short jobs, Fair (+)
- Context-switching time adds up for long jobs (-)

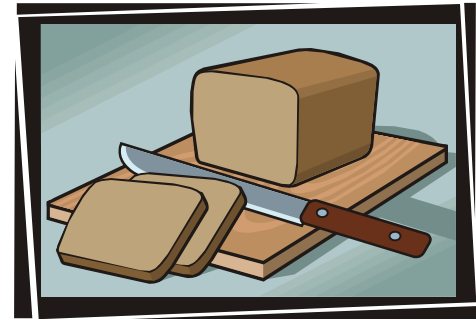
How to Implement RR in the Kernel?

- FIFO Queue, as in FCFS
- But preempt job after quantum expires, and send it to the back of the queue
 - How? Timer interrupt!
 - And, of course, careful synchronization



Round-Robin Discussion

- How do you choose time slice?
 - What if too big?
 - » Response time suffers
 - What if infinite (∞)?
 - » Get back FIFO
 - What if time slice too small?
 - » Throughput suffers!
- Actual choices of timeslice:
 - Initially, UNIX timeslice one second:
 - » Worked ok when UNIX was used by one or two people.
 - » What if three compilations going on? 3 seconds to echo each keystroke!
 - Need to balance short-job performance and long-job throughput:
 - » Typical time slice today is between 10ms – 100ms
 - » Typical context-switching overhead is 0.1ms – 1ms
 - » Roughly 1% overhead due to context-switching



Comparisons between FCFS and Round Robin

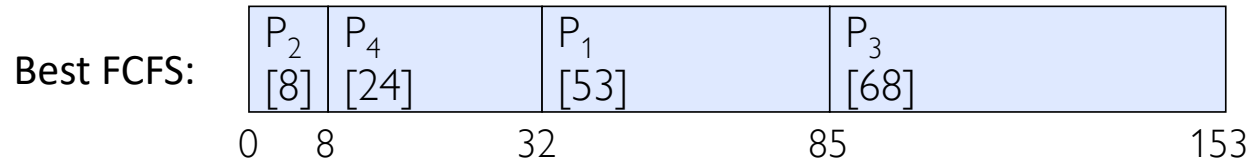
- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example: 10 jobs, each take 100s of CPU time
RR scheduler quantum of 1s
All jobs start at the same time

- Completion Times:

Job #	FIFO	RR
1	100	991
2	200	992
...
9	900	999
10	1000	1000

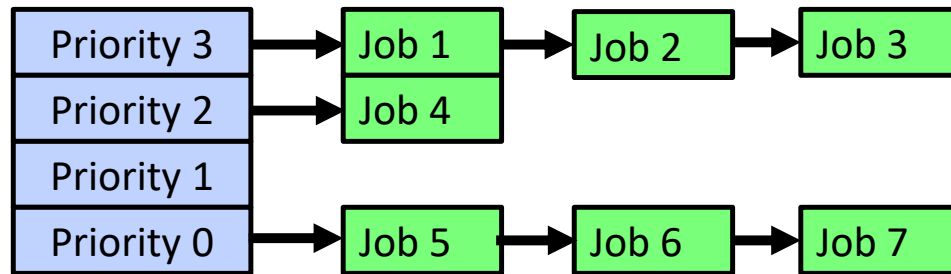
- Both RR and FCFS finish at the same time
- Average completion time is much worse under RR!
 - » Bad when all jobs same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
 - Total time for RR longer even for zero-cost switch!

Earlier Example with Different Time Quantum



	Quantum	P ₁	P ₂	P ₃	P ₄	Average
Wait Time	Best FCFS	32	0	85	8	31¼
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	61¼
	Q = 8	80	8	85	56	57¼
	Q = 10	82	10	85	68	61¼
	Q = 20	72	20	85	88	66¼
	Worst FCFS	68	145	0	121	83½
Completion Time	Best FCFS	85	8	153	32	69½
	Q = 1	137	30	153	81	100½
	Q = 5	135	28	153	82	99½
	Q = 8	133	16	153	80	95½
	Q = 10	135	18	153	92	99½
	Q = 20	125	28	153	112	104½
	Worst FCFS	121	153	68	145	121¾

Handling Differences in Importance: Strict Priority Scheduling



- Execution Plan
 - Always execute highest-priority runnable jobs to completion
 - Each queue can be processed in RR with some time-quantum
- Problems:
 - Starvation:
 - » Lower priority jobs don't get to run because higher priority jobs
 - Deadlock: Priority Inversion
 - » Happens when low priority task has lock needed by high-priority task
 - » Usually involves third, intermediate priority task preventing high-priority task from running
- How to fix problems?
 - Dynamic priorities – adjust base-level priority up or down based on heuristics about interactivity, locking, burst behavior, etc...

Scheduling Fairness

- What about fairness?
 - Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
 - » long running jobs may never get CPU
 - » Urban legend: In Multics, shut down machine, found 10-year-old job ⇒
Ok, probably not...
 - Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
 - Tradeoff: fairness gained by hurting avg response time!

Scheduling Fairness

- How to implement fairness?
 - Could give each queue some fraction of the CPU
 - » What if one long-running job and 100 short-running ones?
 - » Like express lanes in a supermarket—sometimes express lanes get so long, get better service by going into one of the other lines
 - Could increase priority of jobs that don't get service
 - » What is done in some variants of UNIX
 - » This is ad hoc—what rate should you increase priorities?
 - » And, as system gets overloaded, no job gets CPU time, so everyone increases in priority⇒Interactive jobs suffer

What if we Knew the Future?

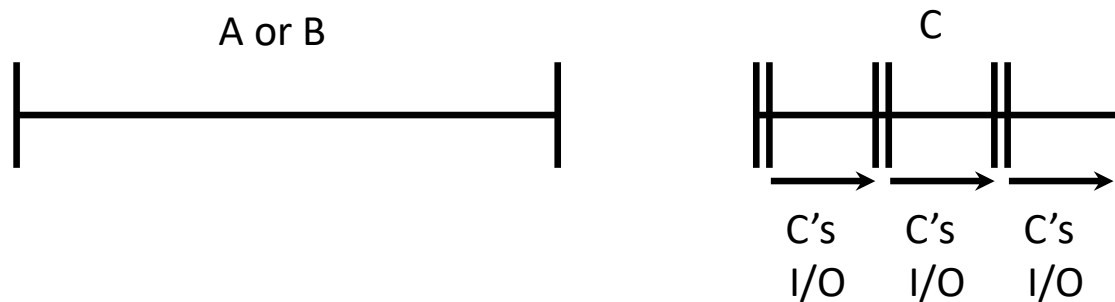
- Could we always mirror best FCFS?
- Shortest Job First (SJF):
 - Run whatever job has least amount of computation to do
 - Sometimes called “Shortest Time to Completion First” (STCF)
- Shortest Remaining Time First (SRTF):
 - Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
 - Sometimes called “Shortest Remaining Time to Completion First” (SRTCF)
- These can be applied to whole program or current CPU burst
 - Idea is to get short jobs out of the system
 - Big effect on short jobs, only small effect on long ones
 - Result is better average response time



Discussion

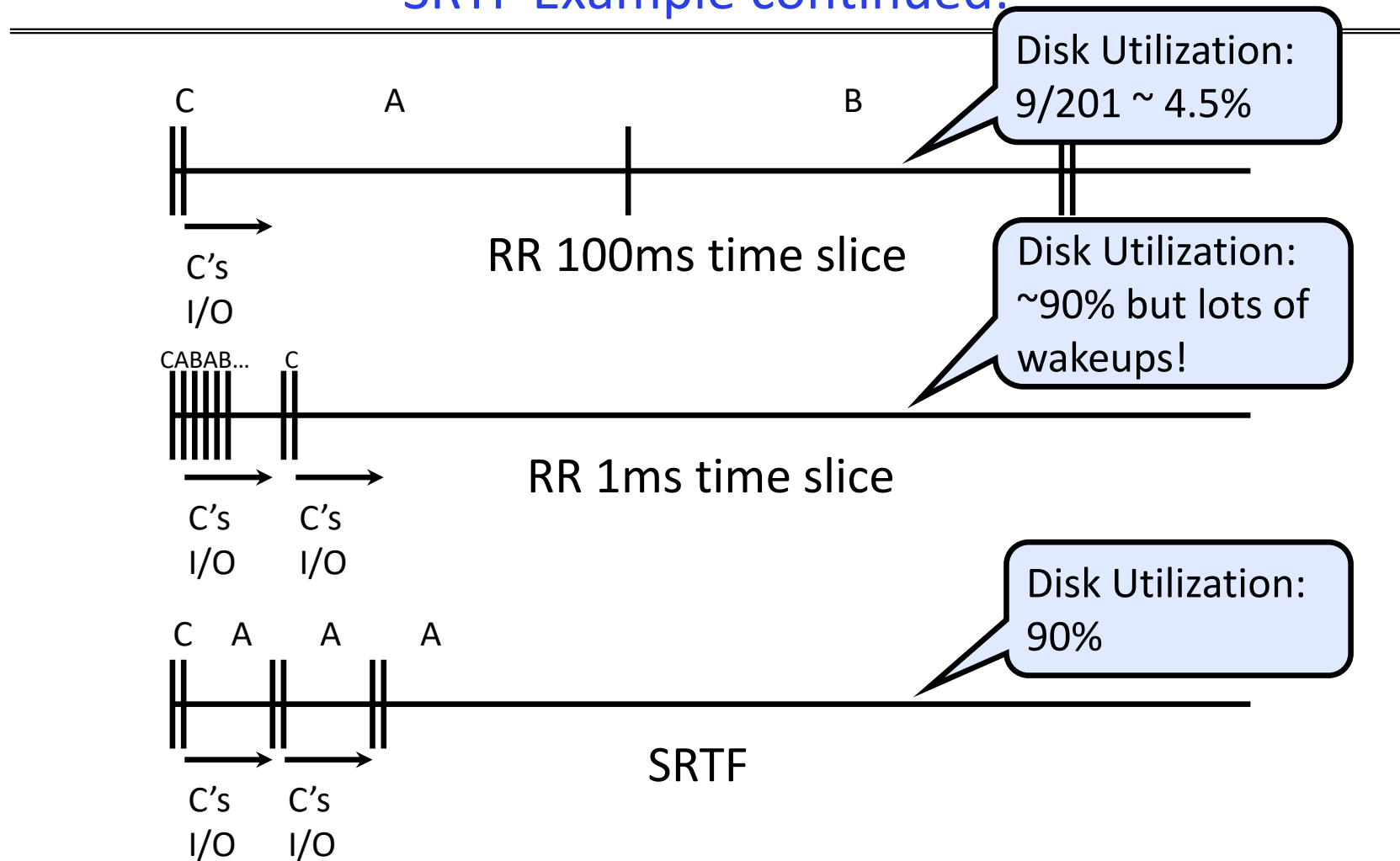
- SJF/SRTF are the best you can do at minimizing average response time
 - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
 - Since SRTF is always at least as good as SJF, focus on SRTF
- Comparison of SRTF with FCFS
 - What if all jobs the same length?
 - » SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length)
 - What if jobs have varying length?
 - » SRTF: short jobs not stuck behind long ones

Example to illustrate benefits of SRTF



- Three jobs:
 - A, B: both CPU bound, run for week
 - C: I/O bound, loop 1ms CPU, 9ms disk I/O
 - If only one at a time, C uses 90% of the disk, A or B could use 100% of the CPU
- With FCFS:
 - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
 - Easier to see with a timeline

SRTF Example continued:



SRTF Further discussion

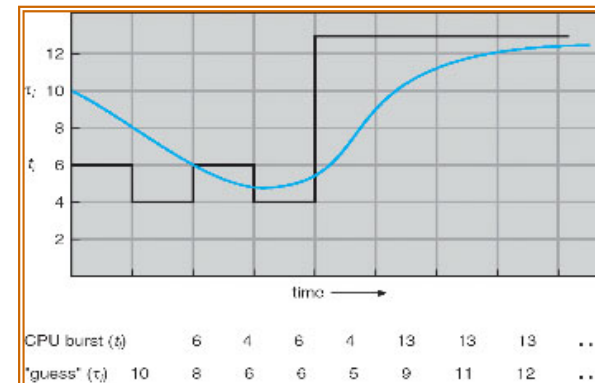
- Starvation
 - SRTF can lead to starvation if many small jobs!
 - Large jobs never get to run
- Somehow need to predict future
 - How can we do this?
 - Some systems ask the user
 - » When you submit a job, have to say how long it will take
 - » To stop cheating, system kills job if takes too long
 - But: hard to predict job's runtime even for non-malicious users
- Bottom line, can't really know how long job will take
 - However, can use SRTF as a yardstick for measuring other policies
 - Optimal, so can't do any better
- SRTF Pros & Cons
 - Optimal (average response time) (+)
 - Hard to predict future (-)
 - Unfair (-)



Predicting the Length of the Next CPU Burst

- **Adaptive:** Changing policy based on past behavior
 - CPU scheduling, in virtual memory, in file systems, etc
 - Works because programs have predictable behavior
 - » If program was I/O bound in past, likely in future
 - » If computer behavior were random, wouldn't help
- Example: SRTF with estimated burst length
 - Use an estimator function on previous bursts:
Let $t_{n-1}, t_{n-2}, t_{n-3}, \dots$ be previous CPU burst lengths.
Estimate next burst $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \dots)$
 - Function f could be one of many different time series estimation schemes (Kalman filters, etc)
 - For instance,

exponential averaging
 $\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$
with $(0 < \alpha \leq 1)$



Conclusion

- **Scheduling Goals:**
 - Minimize Response Time (e.g. for human interaction)
 - Maximize Throughput (e.g. for large computations)
 - Fairness (e.g. Proper Sharing of Resources)
 - Predictability (e.g. Hard/Soft Realtime)
- **Round-Robin Scheduling:**
 - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
 - Pros: Better for short jobs
- **Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):**
 - Run whatever job has the least amount of computation to do/least remaining amount of computation to do