

# CS161: Operating Systems

Matt Welsh  
mdw@eecs.harvard.edu



Lecture 8: Scheduling  
February 27, 2007

# Scheduling

Have already discussed context switching

- Have not discussed how the OS decides which thread to run next
- Context switching is the *mechanism*
- Scheduling is the *policy*

Which thread to run next?

How long does it run for (*granularity*)?

How to ensure every thread gets a chance to run (*fairness*)?

How to prevent starvation?

# Scheduler

The *scheduler* is the OS component that determines which thread to run next on the CPU

The scheduler operates on the ready queue

- Why does it not deal with the waiting thread queues?

When does the scheduler run?

- 
- 
- 
-

# Scheduler

The *scheduler* is the OS component that determines which thread to run next on the CPU

The scheduler operates on the ready queue

- Why does it not deal with the waiting thread queues?

When does the scheduler run?

- When a thread voluntarily gives up the CPU (yield)
- When a thread blocks on I/O, timer, etc.
- When a thread exits
- When a thread is *preempted* (e.g., due to timer interrupt)

Scheduling can be *preemptive* or *non-preemptive*

- Preemptive: Timer interrupt can force context switch
- Non-preemptive: Process must yield or block voluntarily

Batch vs. Interactive Scheduling

- Batch: Non-preemptive **and no other jobs run if they block**
- Interactive: Preemptive and other jobs do run if they block

# Scheduling Policy Goals

Goal of a scheduling policy is to achieve some “optimal” allocation of CPU time in the system

- According to some definition of “optimal”

Possible goals of the scheduling policy??

- 
- 
- 
- 
-

# Scheduling Policy Goals

Goal of a scheduling policy is to achieve some “optimal” allocation of CPU time in the system

- According to some definition of “optimal”

Possible goals:

[Note – Different texts use different meanings for these terms:]

- Maximize CPU utilization (% of time that CPU is running threads)
- Maximize CPU throughput (# jobs per second)
- Minimize job *turnaround* time ( $T_{\text{job-ends}} - T_{\text{job-starts}}$ )
- Minimize job *response* time (total time jobs spend on **ready** queue)
  - *How is this related to the “interactive response” of the system?*
- Minimize job *waiting* time (total time jobs spend on **wait** queue)
  - *How can scheduling policy affect waiting time???*

These goals often conflict!

- Batch system: Try to maximize job throughput and minimize turnaround time
- Interactive system: Minimize response time of interactive jobs (i.e., editors, etc.)

The choice of scheduling policy has a huge impact on performance

# Starvation

## Schedulers often try to eliminate thread starvation

- e.g., If a high priority thread always gets to run before a low-priority thread
- We say the low priority thread is *starved*

## Not all schedulers have this as a goal!

- Sometimes starvation is permitted in order to achieve other goals

## Example: Real time systems

- Some threads must run under a specific *deadline*
- e.g., Motor-control task must run every 30 ms to effectively steer robot
- In this case it is (sometimes) OK to starve other threads
- We saw how starvation occurred in the Mars Pathfinder due to *priority inversion*

# First-Come-First-Served (FCFS)

Jobs are scheduled in the order that they arrive

- Also called First-In-First-Out (FIFO)

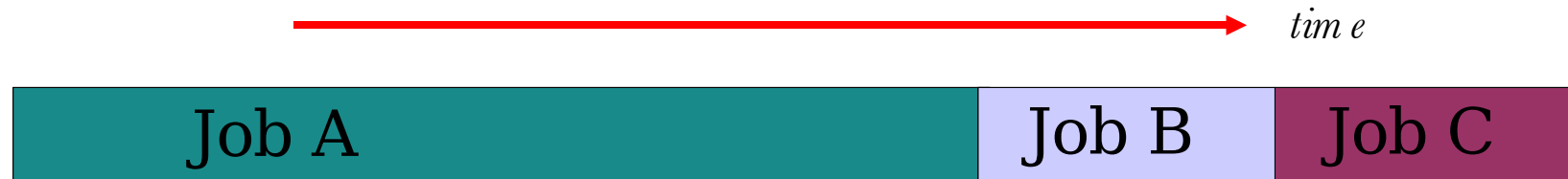
Used only for batch scheduling

- Implies that job runs to completion – never blocks or gets context switched out

Jobs treated equally, no starvation!

- As long as jobs eventually complete, of course

What's wrong with FCFS?



Short jobs get stuck behind long ones!

# Round Robin (RR)

Essentially FCFS with preemption

A thread runs until it blocks or its *CPU quantum* expires

- How to determine the ideal CPU quantum?



Job A: 13 time units, Job B & C: 4 time units

- Turnaround time with FCFS: Job A = 13, Job B = (13+4), Job C = (13 + 4 + 4)
  - *Total turnaround time = 51, mean = (51/3) = 17*
- Turnaround time with RR: Job A = 21, Job B = 11, Job C = 12
  - *Total turnaround time = 44, mean = (44/3) = 14.667*

**Job A**

# Shortest Job First (SJF)

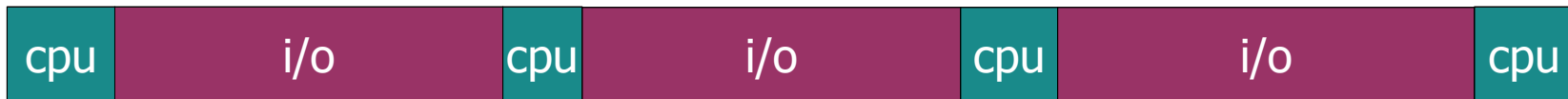
Schedule job with the shortest expected *CPU burst*

Two broad classes of processes: *CPU bound* and *I/O bound*

- CPU bound:



- I/O bound:



Examples of each kind of process?

- 
- 
-

# Shortest Job First (SJF)

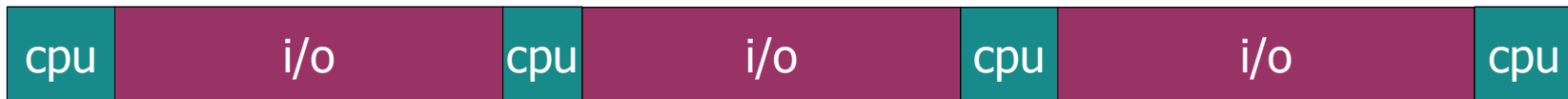
Schedule job with the shortest expected *CPU burst*

Two broad classes of processes: *CPU bound* and *I/O bound*

- CPU bound:



- I/O bound:



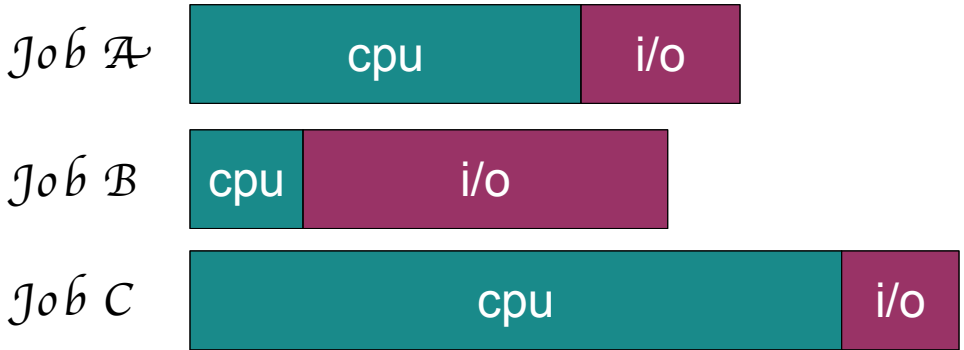
Examples of each kind of process?

- CPU bound: compiler, number crunching, games, MP3 encoder, etc.
- I/O bound: web browser, database engine, word processor, etc.

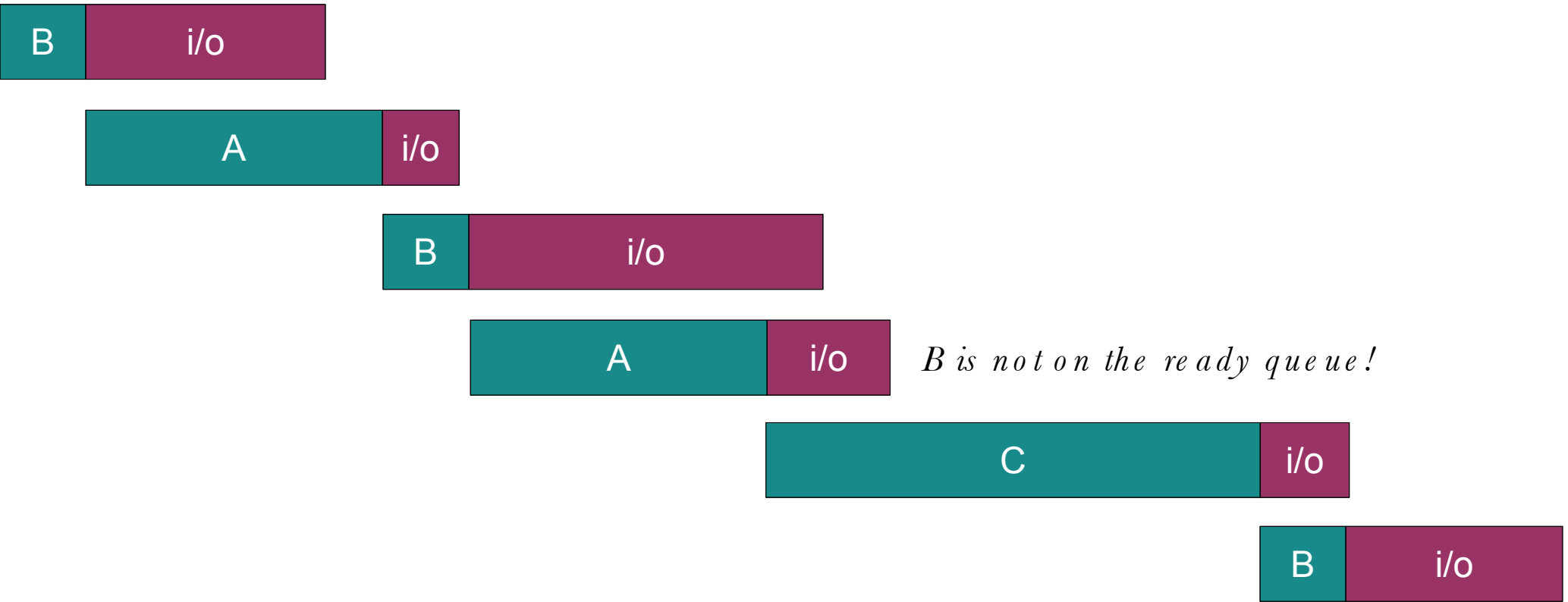
How to predict a process's CPU burst?

- Can get a pretty good guess by looking at the past history of the job
- Track the CPU burst each time a thread runs, track the average
- *CPU bound* jobs will tend to have a long burst
- *I/O bound* jobs will tend to have a short burst

# SJF Example



Resulting schedule:



# Shortest Job First (SJF)

Schedule job with the shortest expected *CPU burst*

- This policy is nonpreemptive. Job will run until it blocks for I/O.

SJF scheduling prefers I/O bound processes. Why?

Idea: A long CPU burst “hogs” the CPU.

- Running short-CPU-burst jobs first gets them done, and out of the way.
- Allows their I/O to overlap with each other: more efficient use of the CPU
- Interactive programs often have a short CPU burst: Good to run them first
  - *To yield “snappy” interactive performance, e.g., for window system or shell.*

We all do this. It is called “procrastination.”

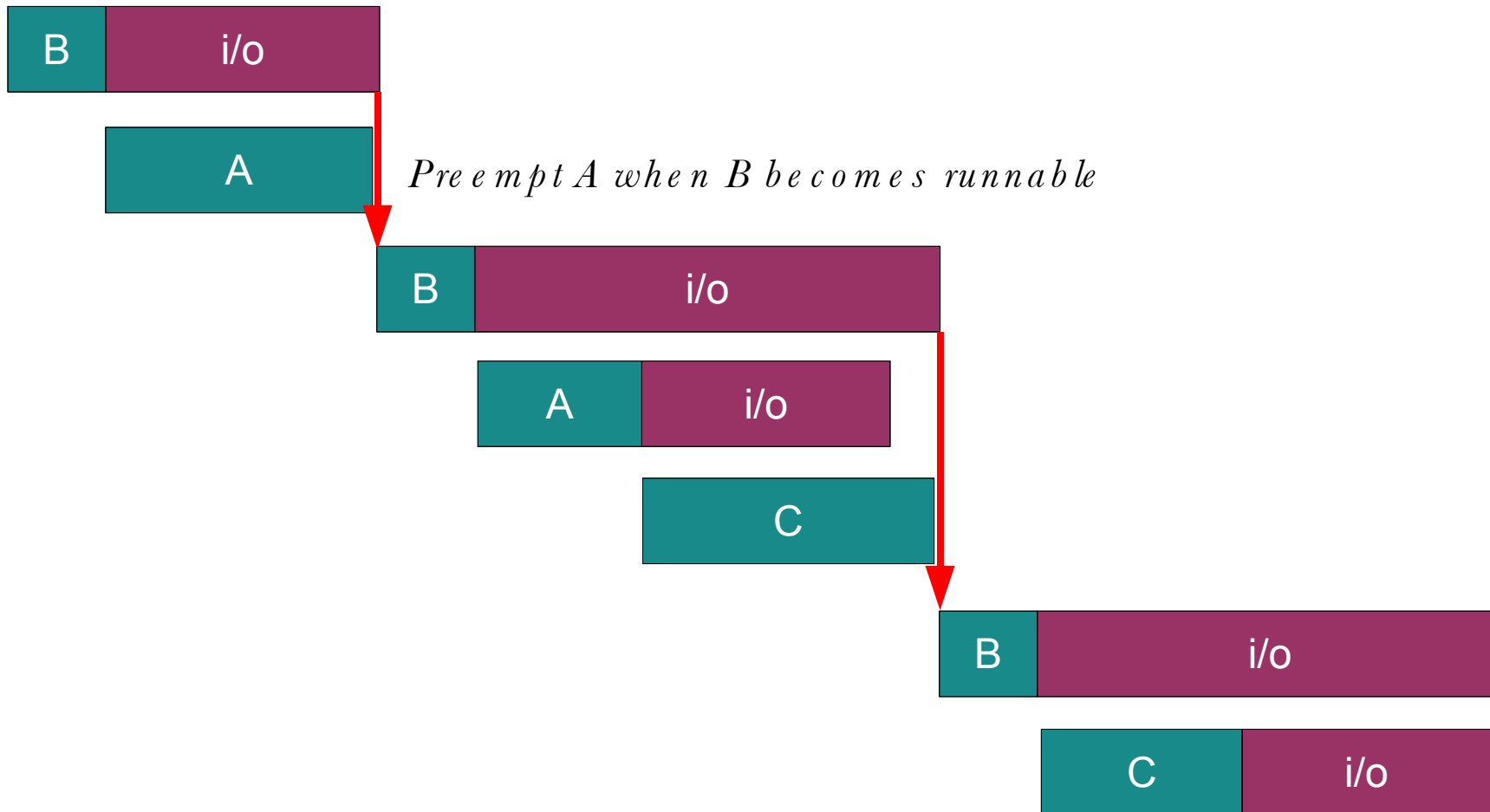
- When faced with too much work, easier to do the short tasks first, get them out of the way.
- Leave the big, hard tasks for later.

# Shortest Remaining Time First (SRTF)

SJF is a nonpreemptive policy.

Preemptive variant: **Shortest Remaining Time First (SRTF)**

- If a job becomes **runnable** with a shorter expected CPU burst, **preempt** current job and run the new job

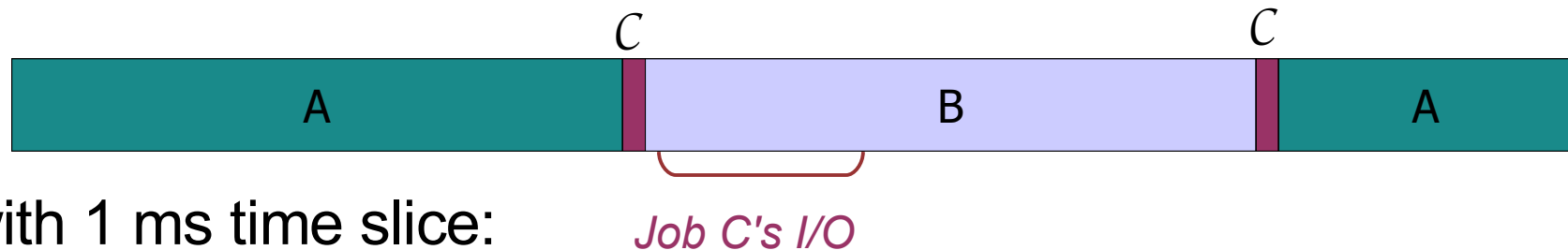


# SRTF versus RR

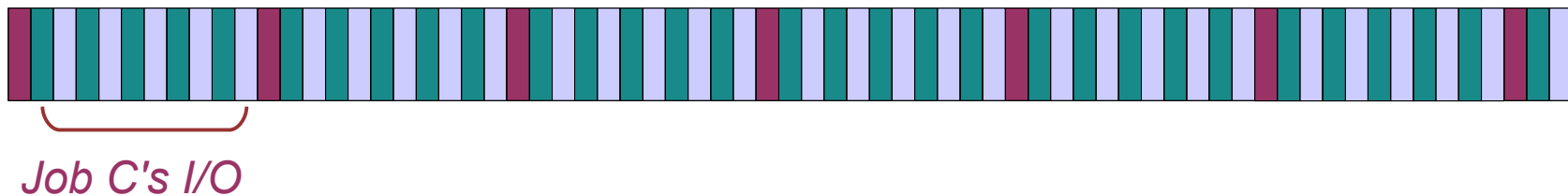
Say we have three jobs:

- Job A and B: both CPU-bound, will run for hours on the CPU with no I/O
- Job C: Requires a 1ms burst of CPU followed by 10ms I/O operation

RR with 25 ms time slice:



RR with 1 ms time slice:



- Lots of pointless context switches between Jobs A and B!

SRTF:



- Job A runs to completion, then Job B starts
- C gets scheduled whenever it needs the CPU

# Priority Scheduling

Assign each thread a *priority*

- In Linux, these range from 0 (lowest) to 99 (highest)
- UNIX “nice()” system call lets user adjust this
  - *But note, scale is inverted: -20 is highest priority and +20 is lowest*

Priority may be set by user, OS, or some combination of the two

- User may adjust priority to bias scheduler towards a thread
- OS may adjust priority to achieve system performance goals

When scheduling, simply run the job with the highest priority

Usually implemented as separate “priority queues”

- One queue for each priority level
- Use RR scheduling within each queue
- If a queue is empty, look in next lowest priority queue

What's the problem with this policy?

# Priority Scheduling

Assign each thread a *priority*

- In Linux, these range from 0 (lowest) to 99 (highest)
- UNIX “nice()” system call lets user adjust this
  - *But note, scale is inverted: -20 is highest priority and +20 is lowest*

Priority may be set by user, OS, or some combination of the two

- User may adjust priority to bias scheduler towards a thread
- OS may adjust priority to achieve system performance goals

When scheduling, simply run the job with the highest priority

Usually implemented as separate “priority queues”

- One queue for each priority level
- Use RR scheduling within each queue
- If a queue is empty, look in next lowest priority queue

Problem: **Starvation**

- High priority threads always trump low priority threads

# Lottery Scheduling

A kind of *randomized* priority scheduling scheme!

Give each thread some number of “tickets”

- The more tickets a thread has, the higher its priority

On each scheduling interval:

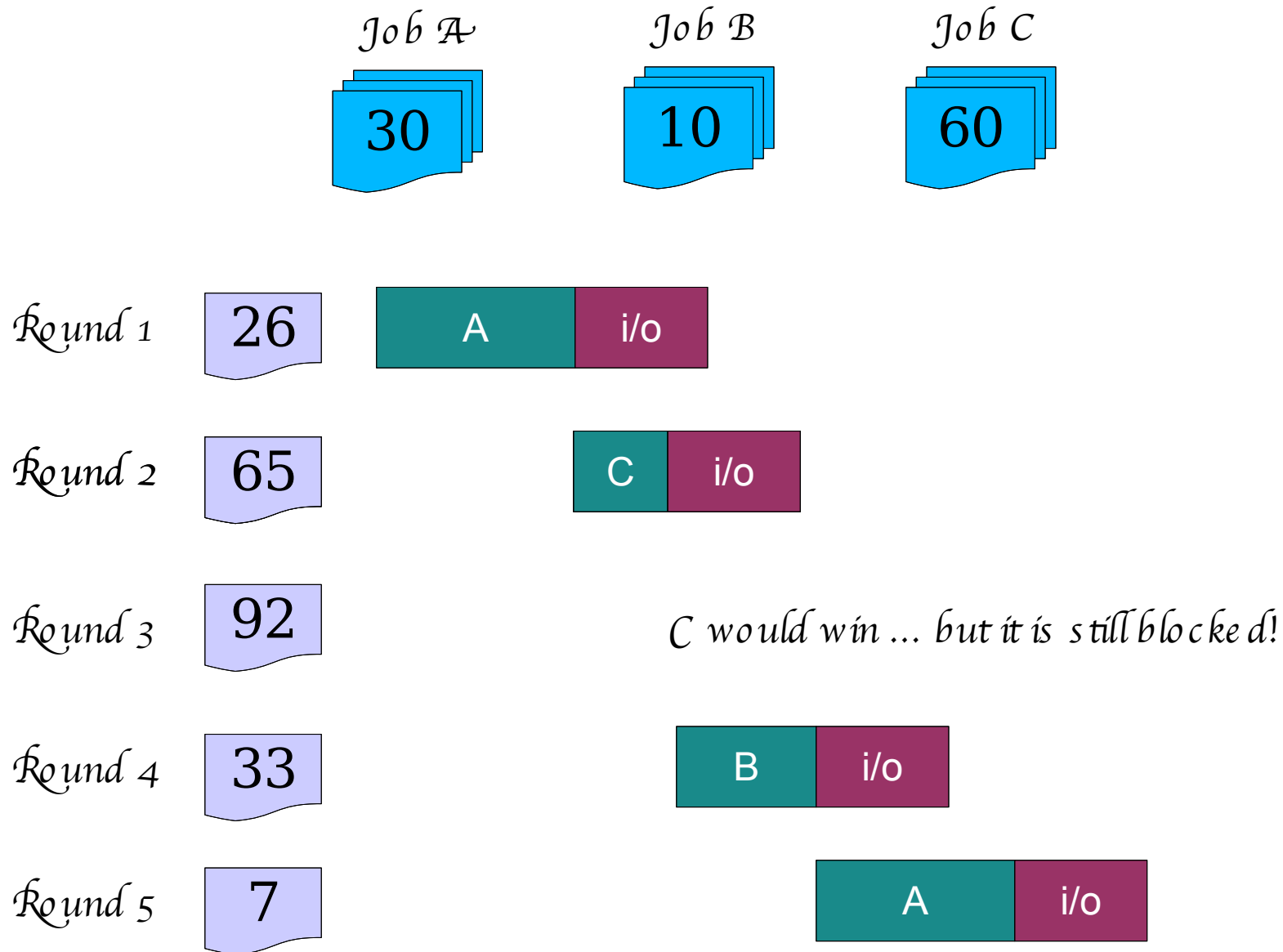
- Pick a random number between 1 and total # of tickets
- Scheduling the job holding the ticket with this number

How does this avoid starvation?

- Even low priority threads have a small chance of running!



# Lottery scheduling example



# Multilevel Feedback Queues (MLFQ)

Observation: Want to give *higher* priority to **I/O-bound** jobs

- They are likely to be interactive and need CPU rapidly after I/O completes
- However, jobs are not **always** I/O bound or CPU-bound during execution!
  - *Web browser is mostly I/O bound and interactive*
  - *But, becomes CPU bound when running a Java applet*

Basic idea: Adjust priority of a thread in response to its CPU usage

- **Increase** priority if job has a short CPU burst
- **Decrease** priority if job has a long CPU burst (e.g., uses up CPU quantum)
- Jobs with lower priorities get **longer CPU quantum**

What is the rationale for this???

# Multilevel Feedback Queues (MLFQ)

Observation: Want to give *higher* priority to **I/O-bound** jobs

- They are likely to be interactive and need CPU rapidly after I/O completes
- However, jobs are not **always** I/O bound or CPU-bound during execution!
  - *Web browser is mostly I/O bound and interactive*
  - *But, becomes CPU bound when running a Java applet*

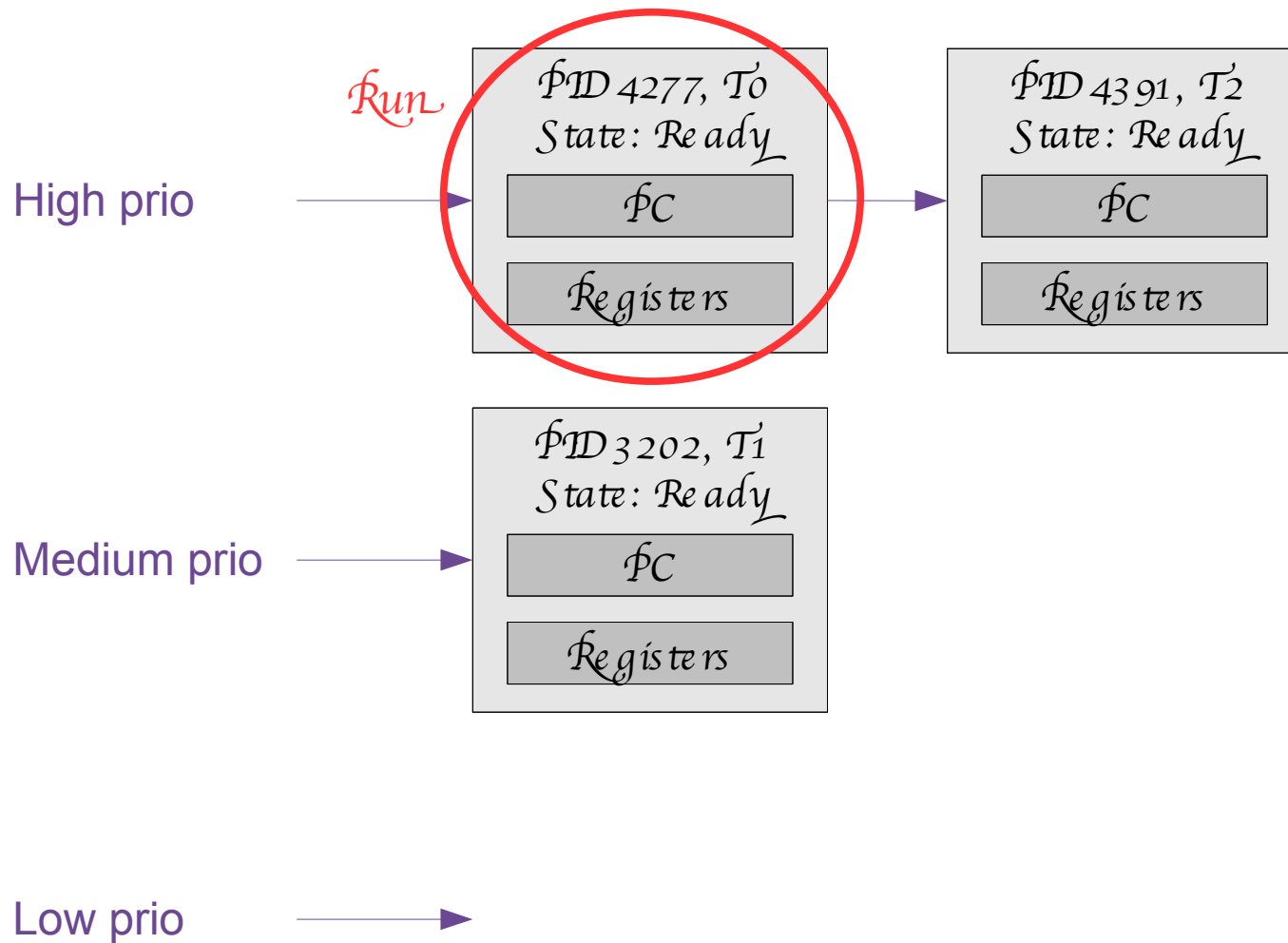
Basic idea: Adjust priority of a thread in response to its CPU usage

- **Increase** priority if job has a short CPU burst
- **Decrease** priority if job has a long CPU burst (e.g., uses up CPU quantum)
- Jobs with lower priorities get **longer CPU quantum**

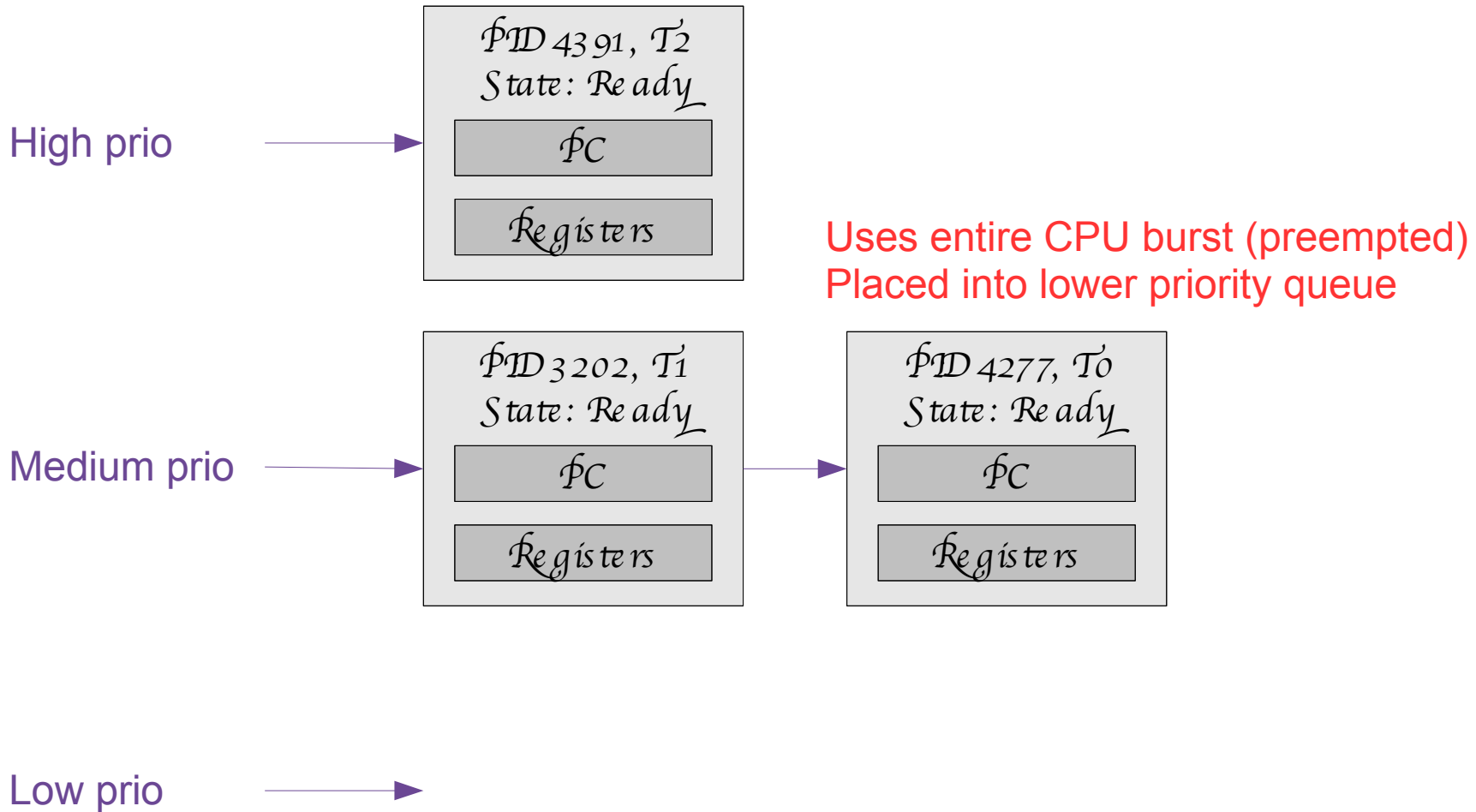
What is the rationale for this???

- Don't want to give high priority to CPU-bound jobs...
  - *Because lower-priority jobs can't preempt them if they get the CPU.*
- OK to give longer CPU quantum to low-priority jobs:
  - *I/O bound jobs with higher priority can still preempt when they become runnable.*

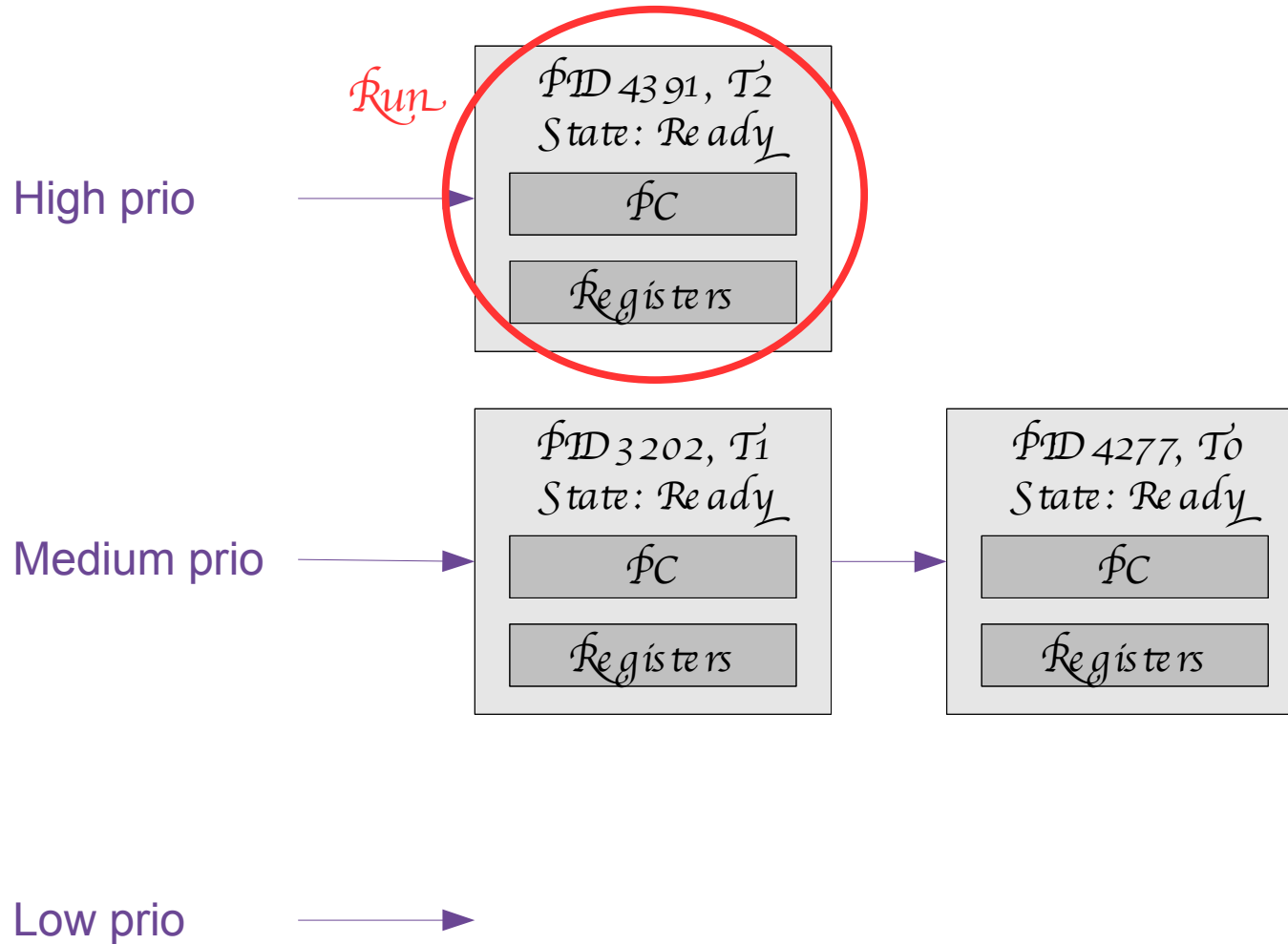
# MLFQ Implementation



# MLFQ Implementation



# MLFQ Implementation



# MLFQ Implementation

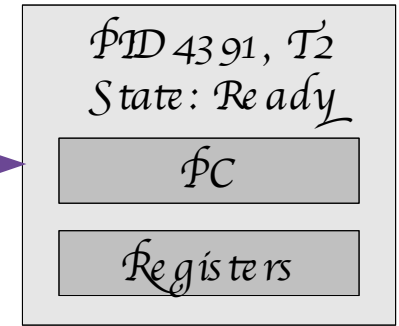
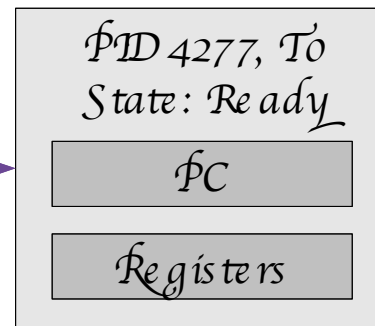
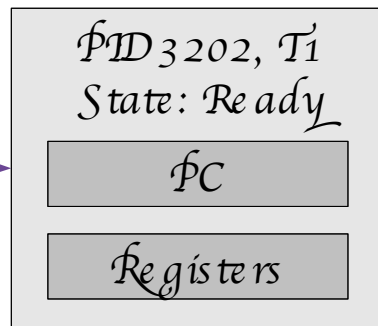
High prio



Medium prio



Low prio



# MLFQ Implementation

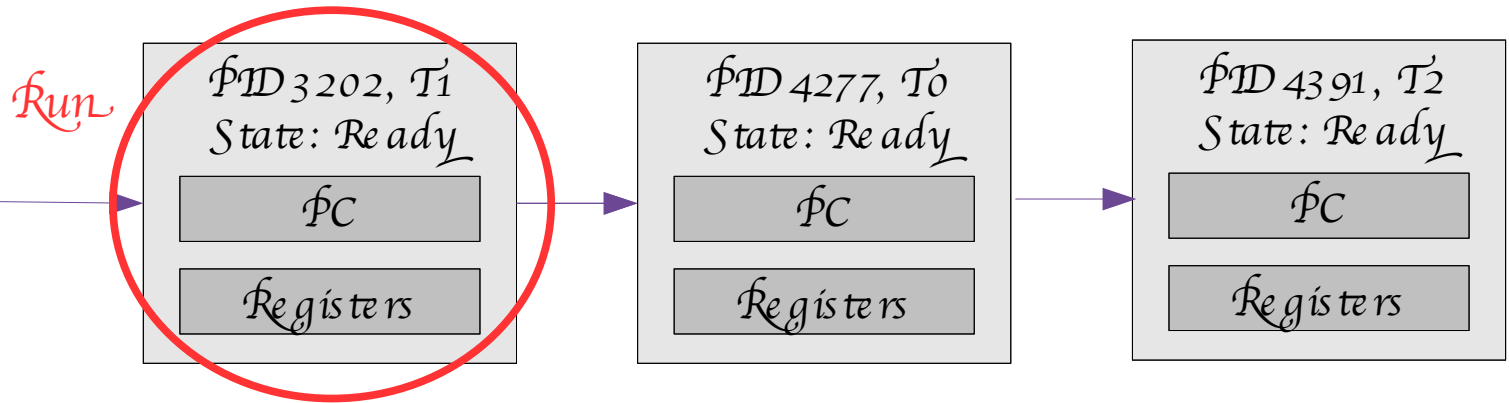
High prio



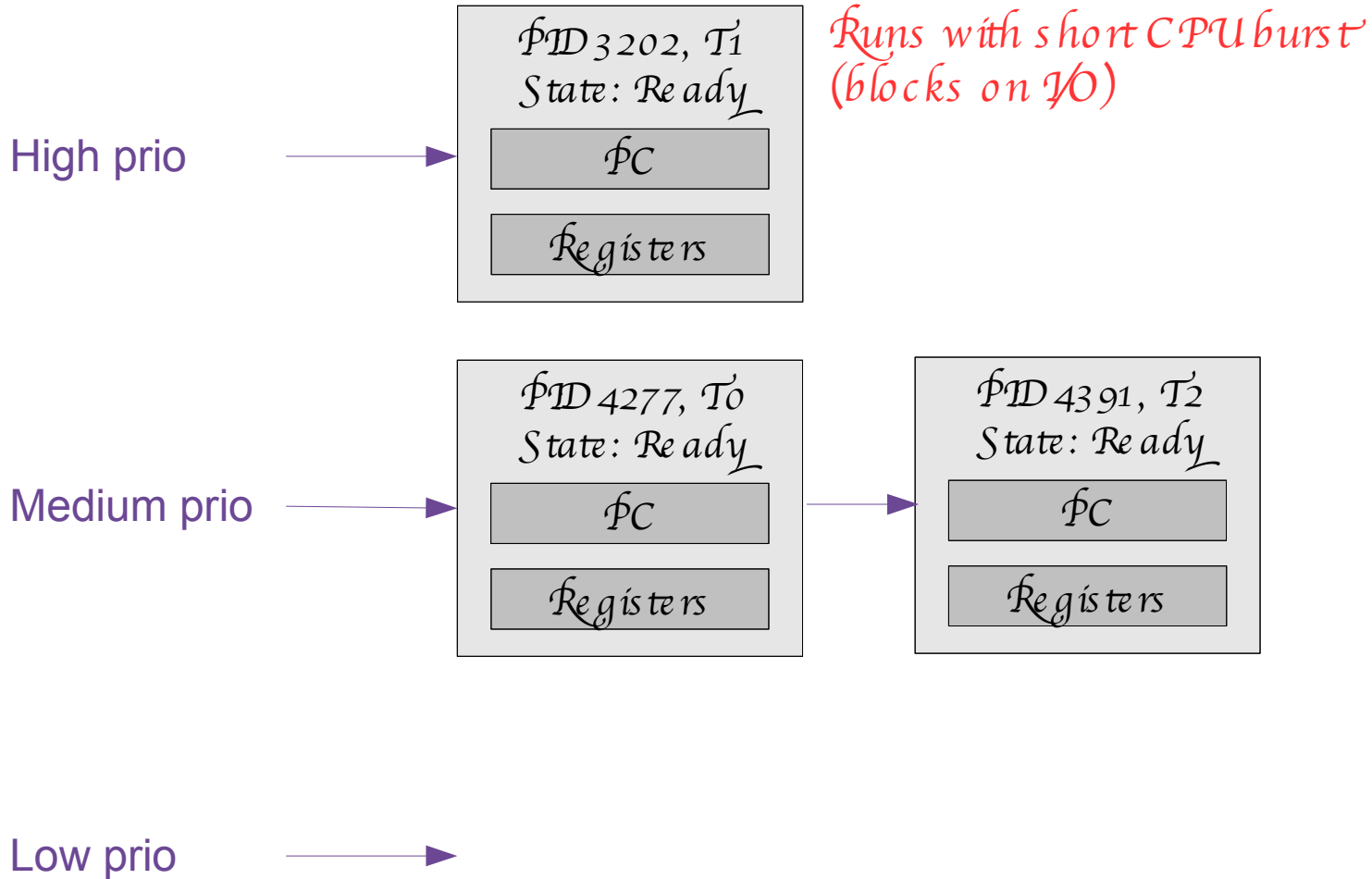
Medium prio



Low prio



# MLFQ Implementation



# Linux Scheduling Policy (pre-2.6)

Caveat: I am eliding some details here!

Each thread has a *different* CPU quantum

- CPU quantum for each thread calculated after all threads have exhausted their quantum – this is called an *epoch*
- If a thread blocks before its quantum has expired, it can use the leftover quantum during the same epoch
  - *Think “rollover minutes”*

Threads assigned initial quantum (about 210ms)

- Can be adjusted by setting thread priority

Scan over all runnable processes and calculate “goodness” for each

- Sum of static process priority plus “dynamic priority”
- Dynamic priority increases as threads wait on I/O
- Give a small bonus to a thread in the same address space as the previously running thread – *why??*

Schedule process with the highest “goodness” value

# Linux O(1) Scheduler (post-2.6)

## Original Linux scheduler did not scale well

- Had to recalculate goodness on all threads every epoch
- As the number of threads gets large, this overhead is serious!

## New O(1) scheduler introduced in Linux 2.5 by Ingo Molnar

- Fancy O(1) priority queue and bitmap scheme to get highest-priority thread
- 140 separate run queues
- Update thread priority when it is descheduled

## No bias for switching between threads in same addr space!

- Ingo says “no workload I know shows any sensitivity to this”\*

\*<http://www.ussg.iu.edu/hypermail/linux/kernel/0201.0/0810.html>

# Linux 2.6 Scheduler Details

Each task has two priorities: **static priority** and **dynamic priority**

## Static priority

- Ranges from 100-139, default value is 120
- Only changed using the nice() system call – change by -20 (higher) to +19 (lower)

## Dynamic priority

- Represents static priority plus a dynamic “bonus”
- This is the value actually used by the scheduler when deciding which task to run

## How the “bonus” is calculated

- Bonus range is between -5 (higher priority) to +5 (lower priority)
- I/O bound tasks given boost of up to -5
- CPU bound jobs given penalty of up to +5
- Bonus calculated by taking ratio of task's “wait time” to “running time”
  - *Idea: Task that waits more often is probably I/O bound*

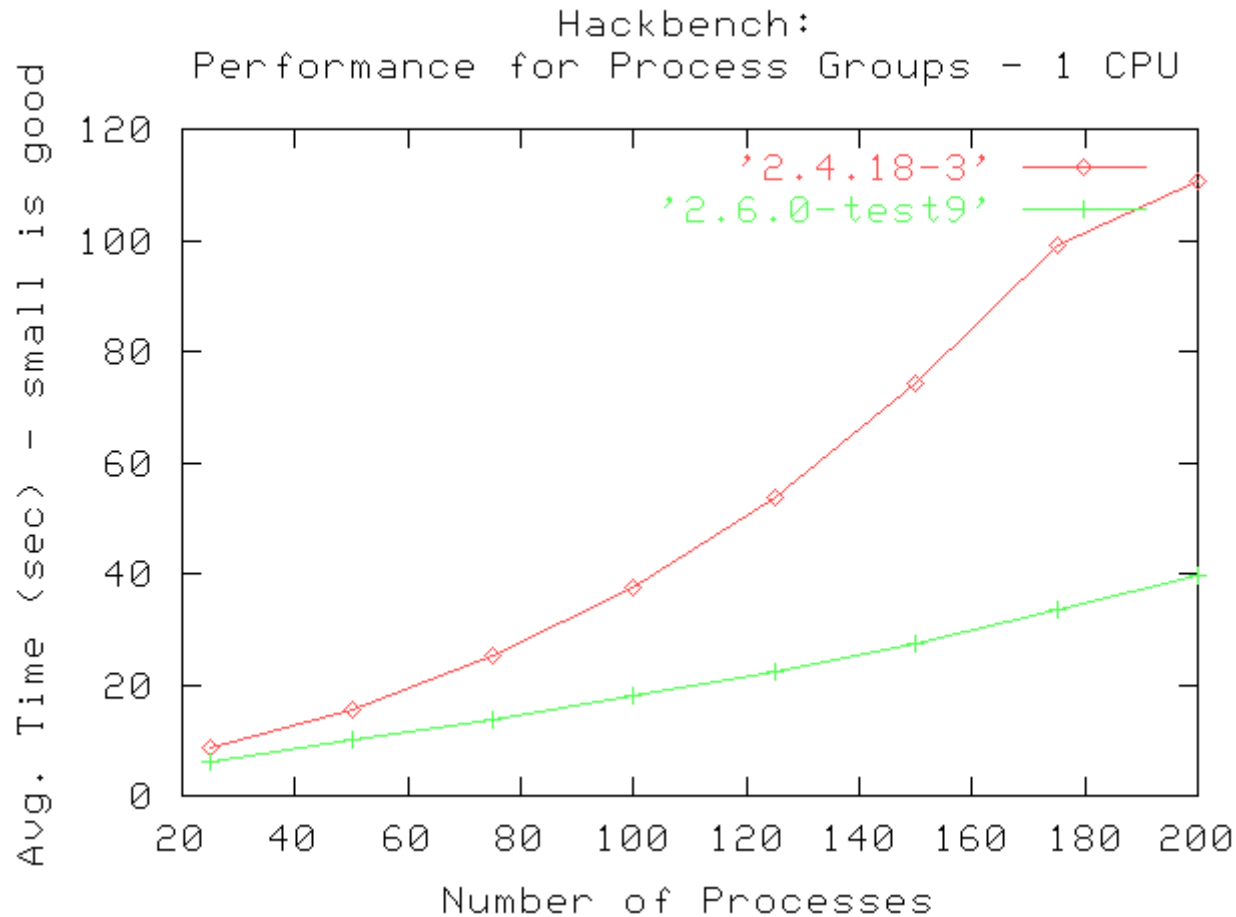
# Linux 2.6 Scheduler Details

<i>Priority Level</i>	<i>Static priority</i>	<i>Nice value</i>	<i>Time quantum</i>
Highest	100	-20	800 ms
Higher	110	-10	600 ms
Normal (default)	120	0	100 ms
Lower	130	+10	50 ms
Lowest	139	+19	5 ms

# Linux 2.6 Scheduler Performance

2-way SMP system can do almost 1 million ctx switches a second!

- Older scheduler could achieve around 240,000 switches/sec

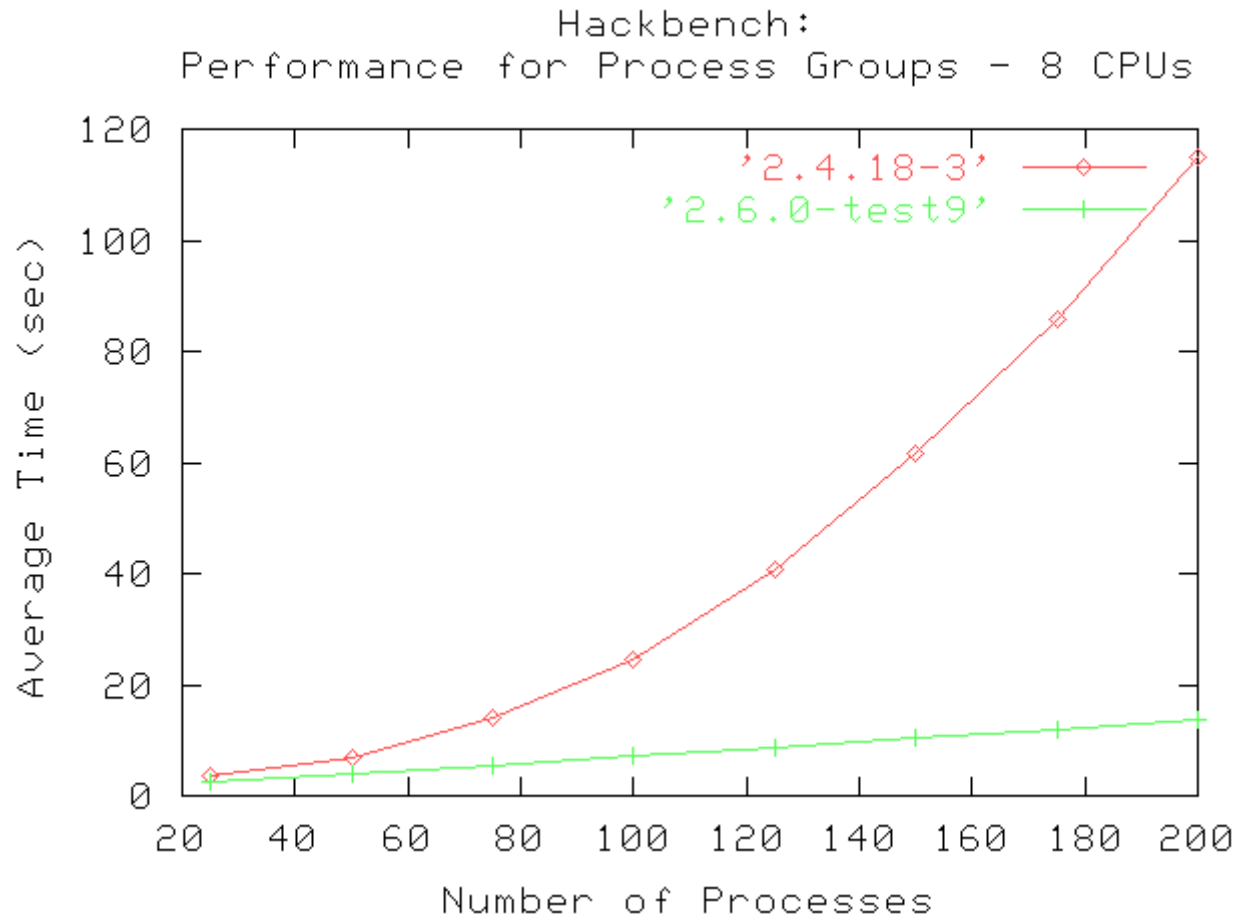


<http://developer.osdl.org/craiger/hackbench/>

# Linux 2.6 Scheduler Performance

2-way SMP system can do almost 1 million ctx switches a second!

- Older scheduler could achieve around 240,000 switches/sec



<http://developer.osdl.org/craiger/hackbench/>

# Scheduling on multiprocessor systems

## Load balancing

- Want to exploit multiple CPUs efficiently
- Try to run threads on different CPUs that *do not interfere* with each other
- For example, threads in different processes
  - *Why???*

## Space sharing

- Try to run threads from the *same process* on different CPUs simultaneously
  - *Why???*

## CPU affinity

- Generally desirable to run *a thread on the same CPU* each time
  - *Why???*

## These different goals are opposing

- Difficult to implement an SMP scheduler that gets the balance right

# Next Lecture

We will start talking about **virtual memory** (my favorite part of the class.)

- How does the system allocate physical RAM to multiple processes?
- How does the system prevent processes from accessing each other's memory?
- How does the system make each process believe it has 4GB of memory when you are really running on a 1990-era '386 with only 128 MB of RAM?
- What does the system do when you really run out of physical RAM?

Read Tanenbaum 4.1-4.3