CS 318 Principles of Operating Systems

Fall 2022

Lecture 5: Scheduling

Prof. Ryan Huang

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Administrivia

Project group

- Fill out the group information form
- If you don't have a group, let us know ASAP

Lab 1

- Overview session today 7-9 pm at Malone 228

Attend office hours to get help

- Don't wait until the lab deadline to ask questions
- You can check your design with TAs/instructor before implementation

Recap: Processes, Threads

Process is the OS abstraction for execution

- own view of machine

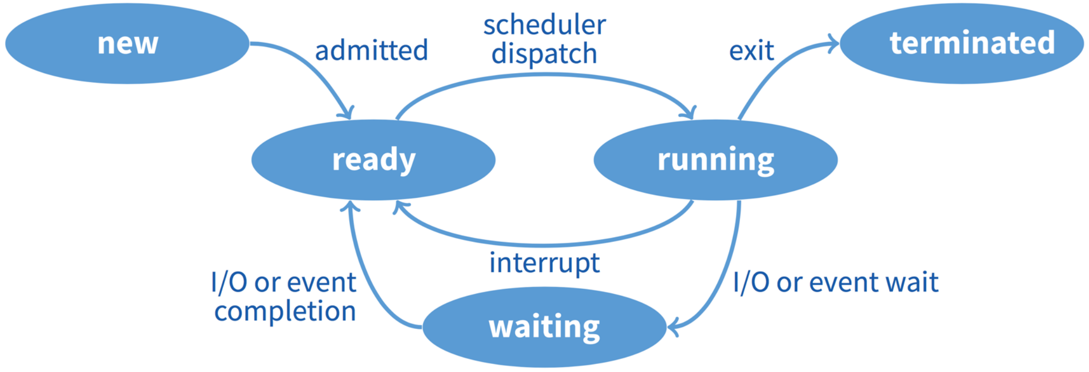
Process components

- address space, program counter, registers, open files, etc.
- kernel data structure: Process Control Block (PCB)

Process vs. thread

Process/thread states and APIs

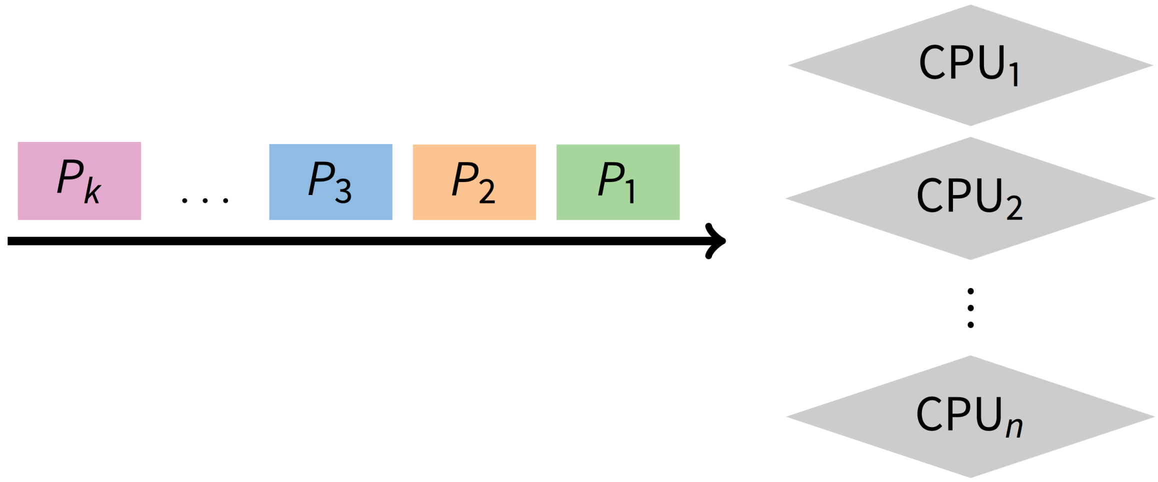
- state graph and queues
- process creation, deletion, waiting



Multiple processes/threads

- overlapping I/O and CPU activities
- context switch

Scheduling Overview



The scheduling problem:

- Have K jobs ready to run
- Have $N \geq 1$ CPUs

Policy: which jobs should we assign to which CPU(s), for how long?

- we'll refer to schedulable entities as jobs – could be processes, threads, people, etc.

Mechanism: context switch, process state queues

Scheduling Overview

1. Goals of scheduling

2. Textbook scheduling

3. Priority scheduling

4. Advanced scheduling topics (not required)

Scheduling Goals

Scheduling works at two levels in an operating system

- To determine the multiprogramming level – # of jobs loaded into memory
  - Moving jobs to/from memory is often called swapping
- To decide what job to run next to guarantee "good service"
  - Good service could be one of many different criteria

Known as long-term and short-term scheduling decisions

- Long-term scheduling happens relatively infrequently (Virtual memory lecture)
  - Significant overhead in swapping a process out to disk
- Short-term scheduling happens relatively frequently (this lecture)
  - Want to minimize the overhead of scheduling
    - Fast context switches, fast queue manipulation

Scheduling "Non-goal": *Starvation*

Starvation is when a process is prevented from making progress because some other process has the resource it requires

- Resource could be the CPU, or a lock (recall readers/writers)

Starvation usually a side effect of the sched. algorithm

- A high priority process always prevents a low priority process from running
- One thread always beats another when acquiring a lock

Starvation can be a side effect of synchronization

- Constant supply of readers always blocks out writers

Scheduling Criteria

Why do we care?

- How do we measure the effectiveness of a scheduling algorithm?

Scheduling Criteria

Throughput – # of processes that complete per unit time

- $\#\, jobs/time$
- Higher is better

Turnaround time – time for each process to complete

- $T_{finish} - T_{start}$
- Lower is better

Response time – time from request to *first* response

- $T_{response} - T_{request}$ i.e., , time between *waiting*→ *ready* transition and *ready*→ *running*
  - e.g., key press to echo, not launch to exit
- Lower is better

Above criteria are affected by secondary criteria

- *CPU utilization* – $\%CPU$ fraction of time CPU doing productive work
- *Waiting time* – $Avg(T_{wait})$ time each process waits in the ready queue

What Criterial Should We Use?

Batch systems

- Strive for job throughput, turnaround time (supercomputers)

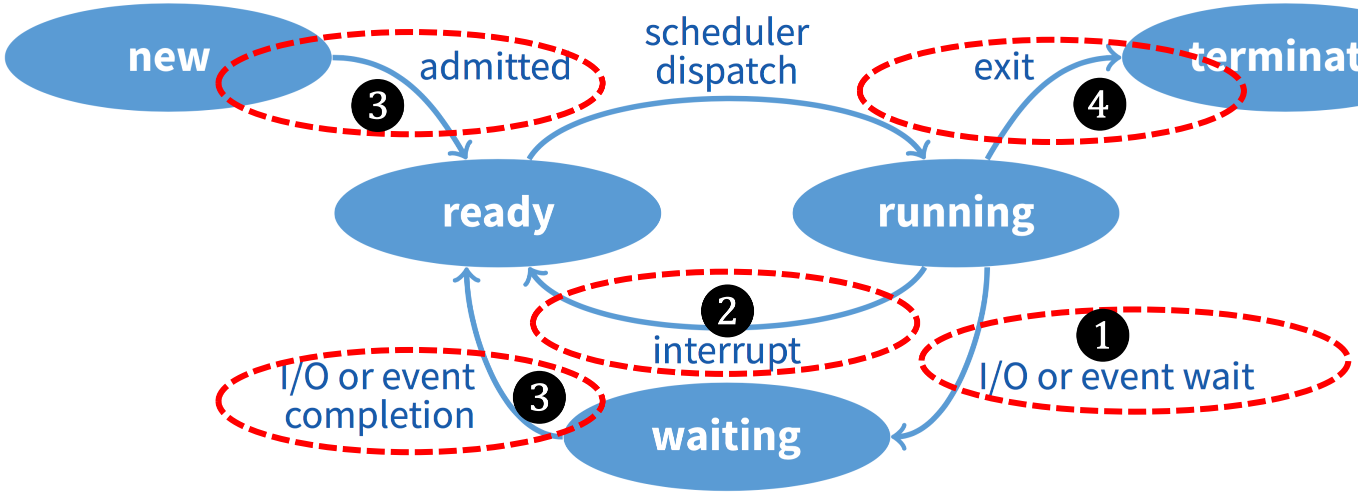
Interactive systems

- Strive to minimize response time for interactive jobs (PC)
  - Utilization and throughput are often traded off for better response time

Usually optimize average measure

- Sometimes also optimize for min/max or variance
  - e.g., minimize the maximum response time
  - e.g., users prefer predictable response time over faster but highly variable response time

When Do We Schedule CPU?



Scheduling decisions may take place when a process:

- ❶ Switches from running to waiting state
- ❷ Switches from running to ready state
- ❸ Switches from new/waiting to ready
- ❹ Exits

Non-preemptive schedules use ❶ & ❹ only

Preemptive schedulers run at all four points

Scheduling Overview

1. Goals of scheduling
2. Textbook scheduling
3. Priority scheduling
4. Advanced scheduling topics (not required)

Example: FCFS Scheduling

Run jobs in order that they arrive

- Called "First-come first-served" (FCFS)
- E.g., Say P_1 needs 24 sec, while P_2 and P_3 need 3.
- Say P_2, P_3 arrived immediately after P_1, get:

Throughput: 3 jobs / 30 sec = 0.1 jobs/sec

Turnaround Time: P_1 : 24, P_2 : 27, P_3 : 30

- Average TT: (24 + 27 + 30) / 3 = 27

Waiting Time: P_1 : 0, P_2 : 24, P_3 : 27

- Average WT: (0 + 24 + 27) / 3 = 17

Can we do better?

FCFS Continued

Suppose we scheduled P_2, P_3, then P_1

- Would get:

Throughput: 3 jobs / 30 sec = 0.1 jobs/sec

Turnaround Time: P_1 : 30, P_2 : 3, P_3 : 6

- Average TT: (30 + 3 + 6) / 3 = 13 – much less than 27

Lesson: scheduling algorithm can reduce TT

- Minimizing waiting time can improve RT and TT

Can a scheduling algorithm improve throughput?

- Yes, if jobs require both computation and I/O

Scheduling Jobs with Computation & I/O (1)

Can a scheduling algorithm improve throughput?

- Yes, if jobs require both computation and I/O

CPU is one of several devices needed by users' jobs

- CPU runs compute jobs, Disk drive runs disk jobs, etc.
- With network, part of job may run on remote CPU

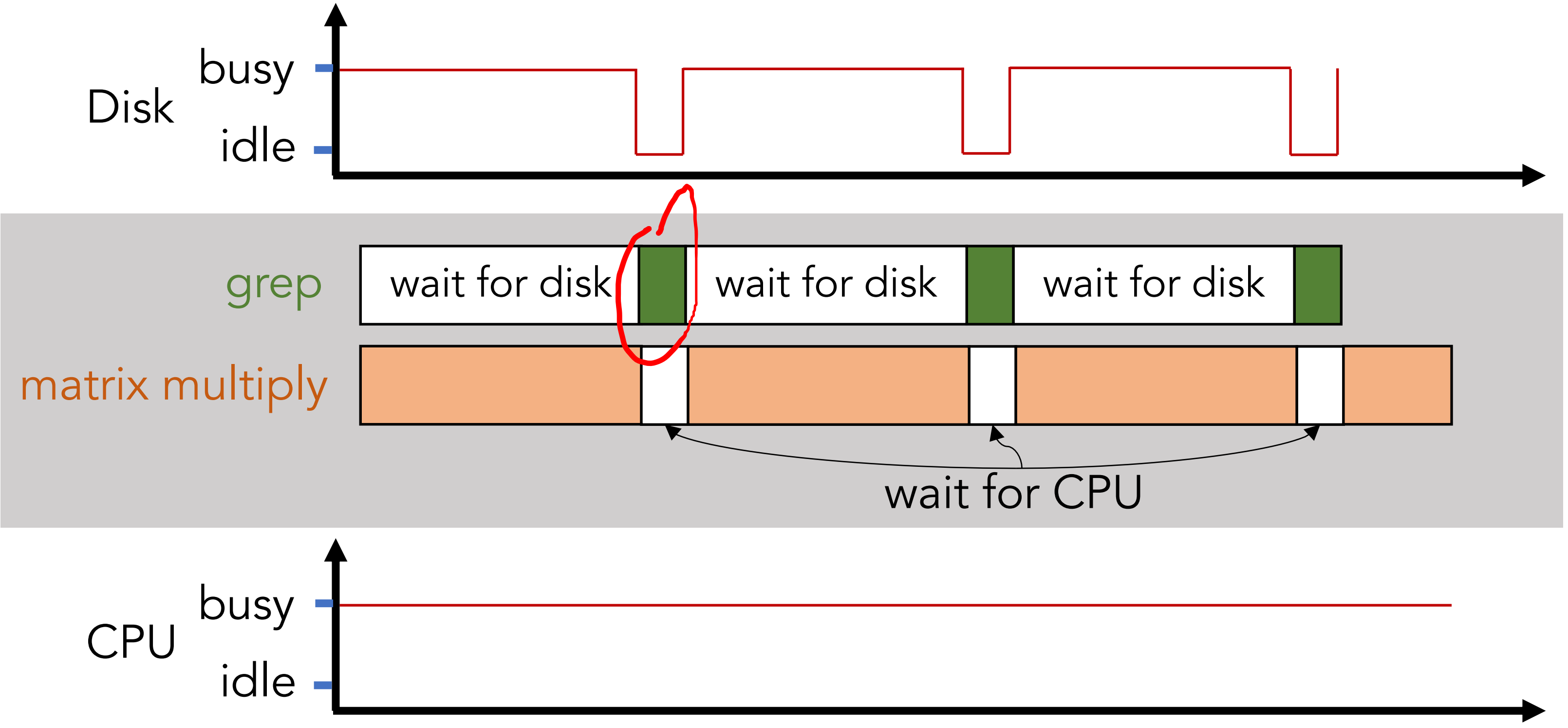
Scheduling 1-CPU system with n I/O devices like scheduling asymmetric $(n+1)$-CPU multiprocessor

- Result: all I/O devices + CPU busy ➔ $(n+1)$-fold throughput gain!

Scheduling Jobs with Computation & I/O (2)

Example: disk-bound `grep` + CPU-bound `matrix_multiply`

- Overlap them just right, throughput will be almost doubled



FCFS Limitations

FCFS algorithm is non-preemptive in nature

- Once CPU time has been allocated to a process, other processes can get CPU time only after the current process has finished or gets blocked.

This property of FCFS scheduling is called *Convoy Effect*

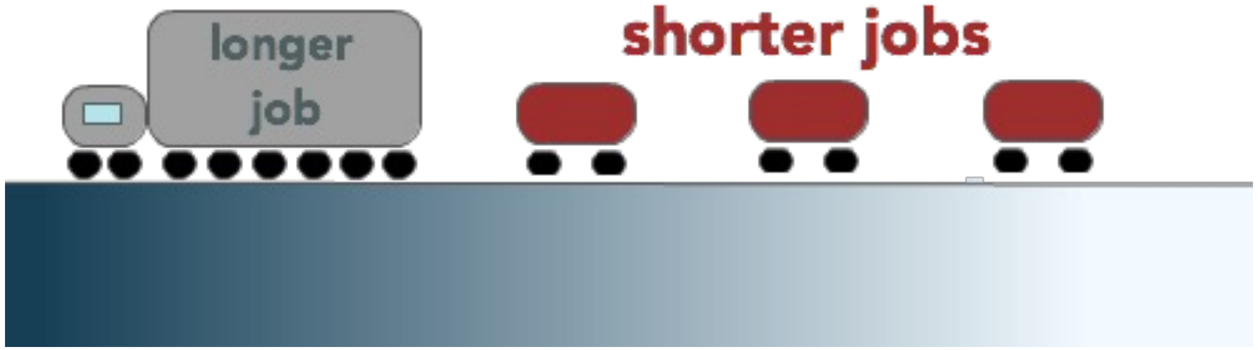


image source: http://web.cs.ucla.edu/classes/fall14/cs111/scribe/7a/convoy_effect.png

Shortest Job First (SJF)

Shortest Job First (SJF)

- Choose the job with the smallest expected CPU burst
  - Person with smallest # of items in shopping cart checks out first

Example

- Three jobs available, CPU bursts are P_1 8 sec, P_2 4 sec, P_3 2 sec

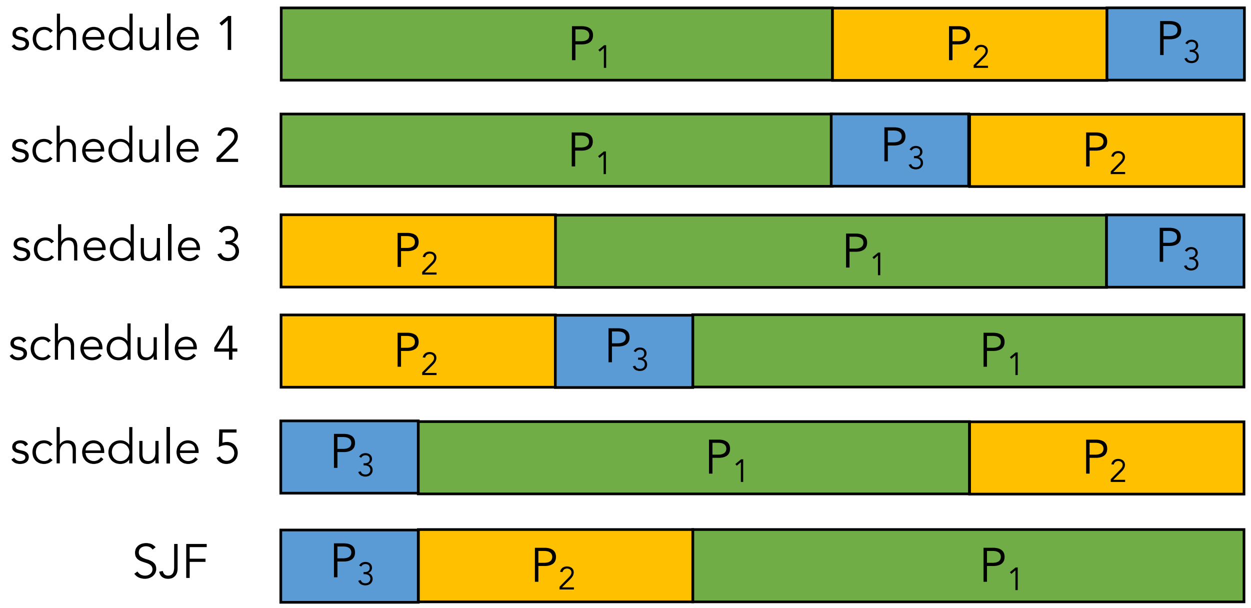
Average Waiting Time: (0 + 2 + 6) / 3 = 2.67

SJF Has Optimal Average Waiting Time

SJF has *provably* optimal minimum average *waiting* time (AWT)

Previous example: P_1 8 sec, P_2 4 sec, P_3 2 sec

- How many possible schedules?



AWT = (0+8+12)/3 = 6.67

AWT = (0+8+10)/3 = 6

AWT = (0+4+12)/3 = 5.33

AWT = (0+4+6)/3 = 3.33

AWT = (0+2+10)/3 = 4

AWT = (0+2+6)/3 = 2.67

Shortest Job First (SJF)

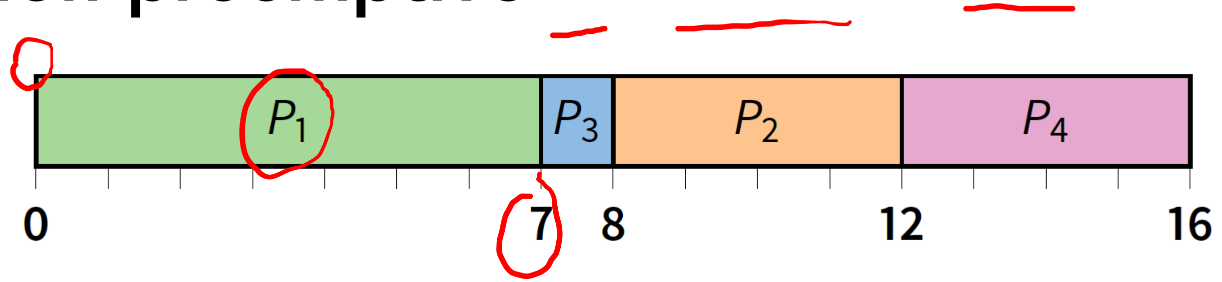
Two schemes

- Non-preemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
- Preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt current process
    - Known as the *Shortest-Remaining-Time-First* or ***SRTF***

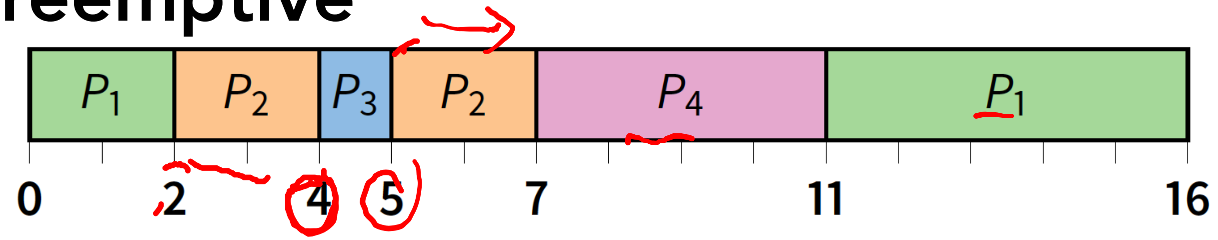
Examples

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Non-preemptive



Preemptive



What is the AWT?

SJF Limitations

Can potentially lead to unfairness or starvation

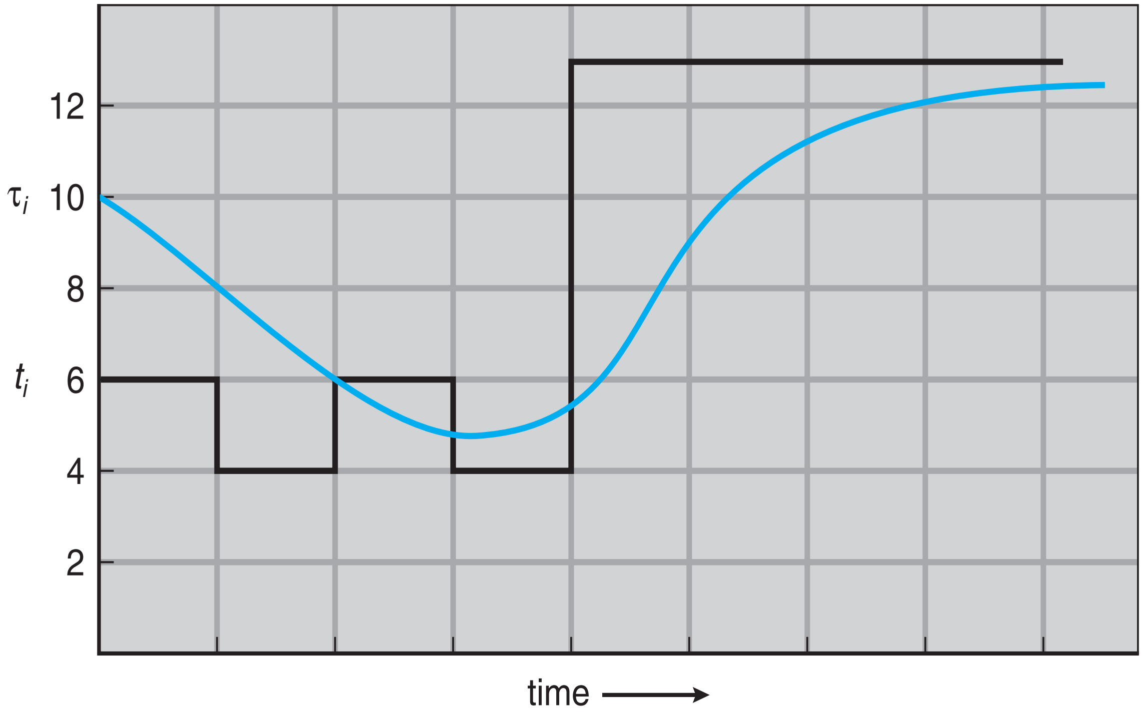
Impossible to know size of CPU burst ahead of time

- Like choosing person in line without looking inside cart

How can you make a reasonable guess?

- Estimate CPU burst length based on past
- E.g., exponentially weighted average
  - t_n actual length of process's n^{th} CPU burst
  - τ_{n+1} estimated length of proc's $(n+1)^{st}$ CPU burst
  - Choose parameter α where $0 < \alpha \leq 1$, e.g., $\alpha = 0.5$
  - Let $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

Exp. Weighted Average Example



CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Round Robin (RR)

Solution to fairness and starvation

- Each job is given a time slice called a quantum
- Preempt job after duration of quantum
- When preempted, move to back of FIFO queue

Advantages:

- Fair allocation of CPU across jobs
- Low average waiting time when job lengths vary
- Good for responsiveness if small number of jobs

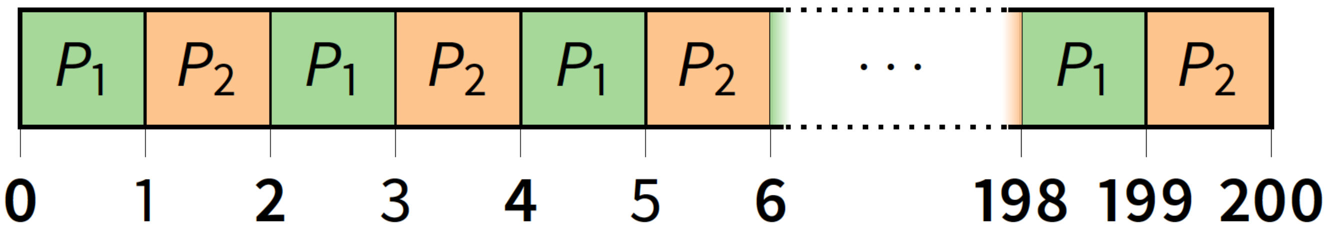
Disadvantages?

RR Disadvantages

Context switches are frequent and need to be very fast

Varying sized jobs are good …what about same-sized jobs?

Assume 2 jobs of time=100 each:



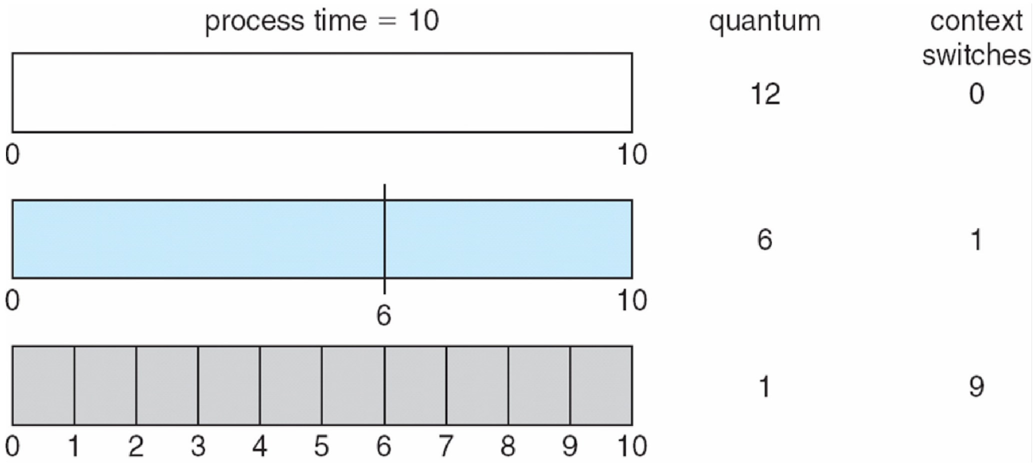
Even if context switches were free…

- What would average turnaround time be with RR?
- How does that compare to FCFS?

P1 = 199
P2 = 200
199.5
150

P1 = 100
P2 = 200

Time Quantum



How to pick quantum?

- Want much larger than context switch cost
- Majority of bursts should be less than quantum
- But not so large system reverts to FCFS

Typical values: 1–100 msec

Scheduling Overview

1. Goals of scheduling
2. Textbook scheduling
3. Priority scheduling
4. Advanced scheduling topics (not required)

Priority Scheduling

Priority Scheduling

- Associate a numeric priority with each process
  - E.g., smaller number means higher priority (Unix/BSD)
  - Or smaller number means lower priority (Pintos)
- Give CPU to the process with highest priority
  - Airline check-in for first class passengers
  - Can be done preemptively or non-preemptively
- Can implement SJF, priority = 1/(expected CPU burst)

Problem: starvation – low priority jobs can wait indefinitely

Solution? "Age" processes

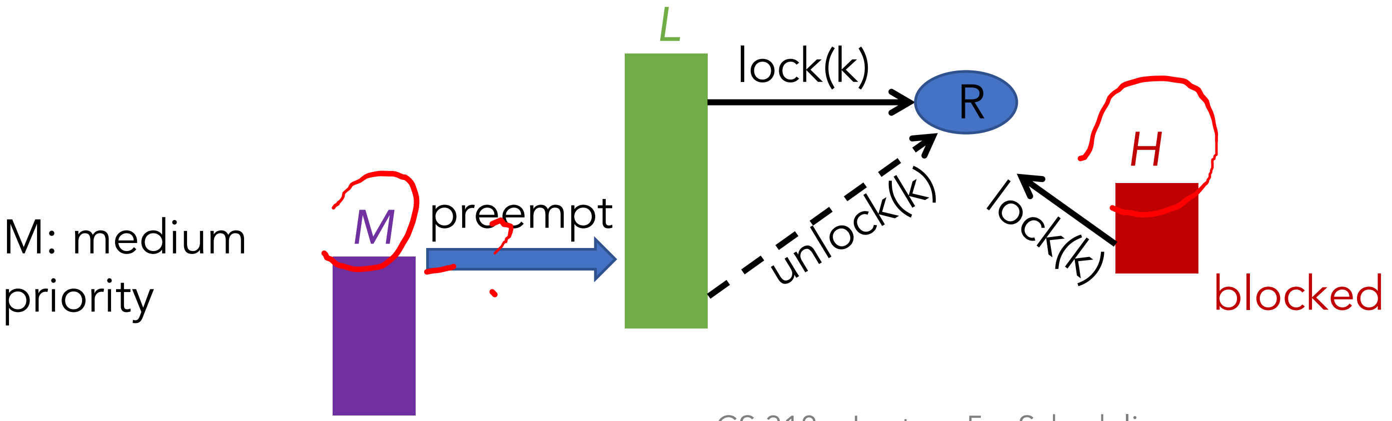
- Increase priority as a function of waiting time
- Decrease priority as a function of CPU consumption

Priority Inversion (1)

Caveat using Priority Scheduling w/ Synch Primitives

- Priority scheduling Rule
  1) Always pick highest-priority thread
  2) …*unless* a lower-priority thread is holding a resource the highest-priority thread wants to get
- Potential *Priority Inversion* Problem

Two tasks: *H* at high priority, *L* at low priority



Priority Inversion (2)

Two tasks: *H* at high priority, *L* at low priority

- *L* acquires lock `k` for exclusive use of a shared resource R
- If *H* tries to acquire `k`, blocked until *L* release resource R
- *M* enters system at medium priority, preempts *L*
  - *L* unable to release R in time, *H* unable to run, despite having higher priority than *M*

Not just a hypothetical issue, it happened in real-world software!

- The root cause for a famous Mars PathFinder failure in 1997
- low-priority data gathering task and a medium-priority communications task prevented the critical bus management task from running

Solution: Priority Donation

"Donate" our priority if we get blocked

- Whenever a high-priority task has to wait for some shared resource that currently held by an executing low priority task,
- the low-priority task is *temporarily* assigned the priority of the highest waiting priority task for the duration of its use of the shared resource

Why this helps?

- Since the low-priority task gets temporarily boosted priority, it keeps medium priority tasks from pre-empting the (originally) low priority task
- Once resource released, low-priority task continues at its original priority

Priority Donation Example

Say higher number = higher priority (like Pintos)

Example 1: *L* (prio 2), *M* (prio 4), *H* (prio 8)

- *L* holds lock `k`
- *M* waits on `k`, *L*'s priority raised to L_1 = max(*M*; *L*) = 4
- Then *H* waits on `k`, *L*'s priority raised to max(*H*; L_1) = 8

Example 2: Same *L*, *M* , *H* as above

- *L* holds lock `k`, *M* holds lock k_2
- *M* waits on `k`, *L*'s priority now L_1 = 4 (as before)
- Then *H* waits on k_2
  - *M*' s priority goes to M_1 = max(*H*; *M*) = 8, and *L*'s priority raised to max(M_1; L_1) = 8

Pintos Lab 1 Exercise 2.2

Combining Algorithms

Different types of jobs have different preferences

- Interactive, CPU-bound, batch, system, etc.
- Hard to use one size to fit all

Combining scheduling algorithms to optimize for multiple objectives

- Have multiple queues
- Use a different algorithm for each queue
- Move processes among queues

Example: Multiple-level feedback queues (MLFQ)

Multiple-level feedback queues (MLFQ)

Developed by Fernando J. Corbató in 1962

- Corbató received the 1990 Turing Award for this work and other work in Multics

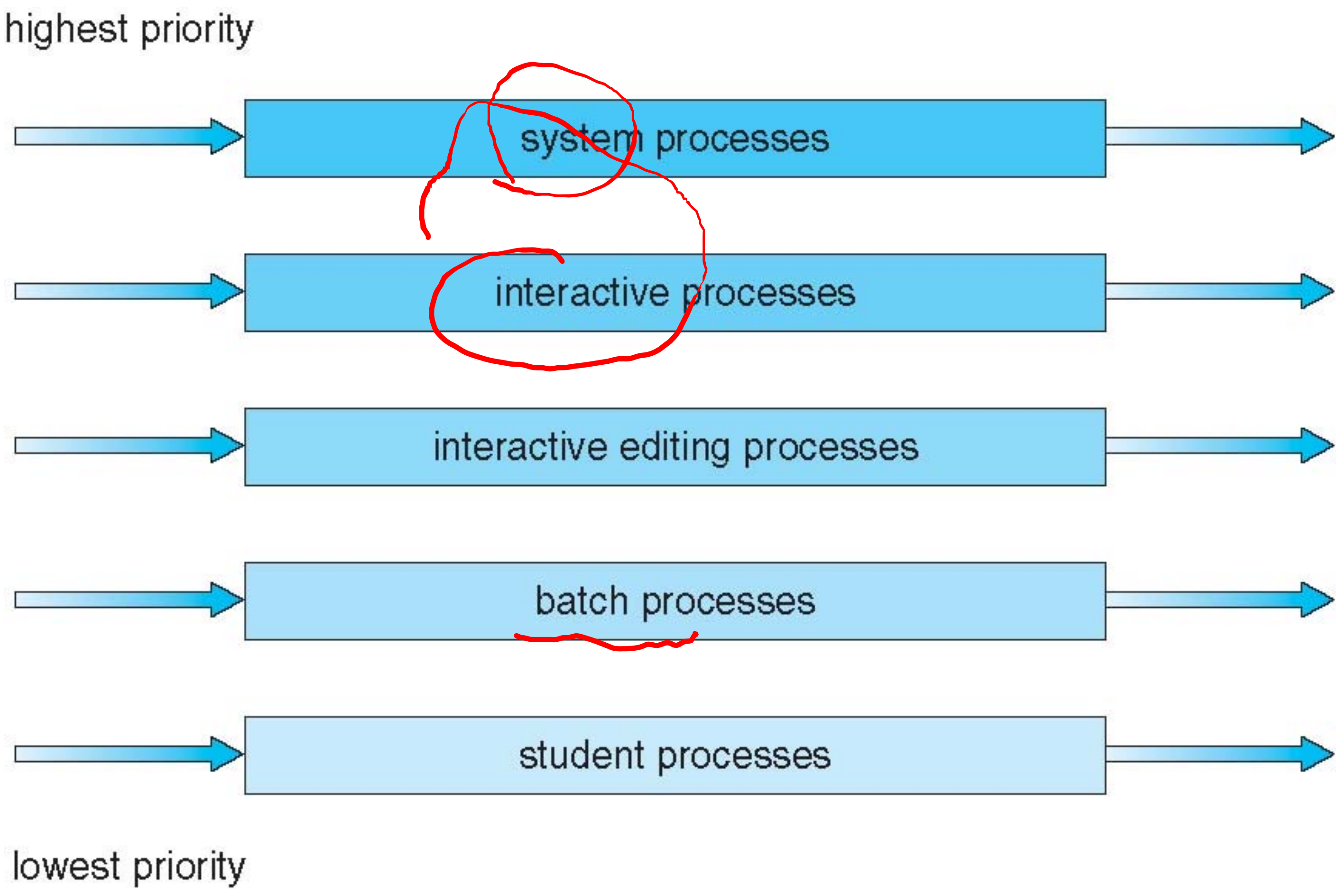
Widely used in mainstream OSes: Unix, BSD, Windows, MacOS

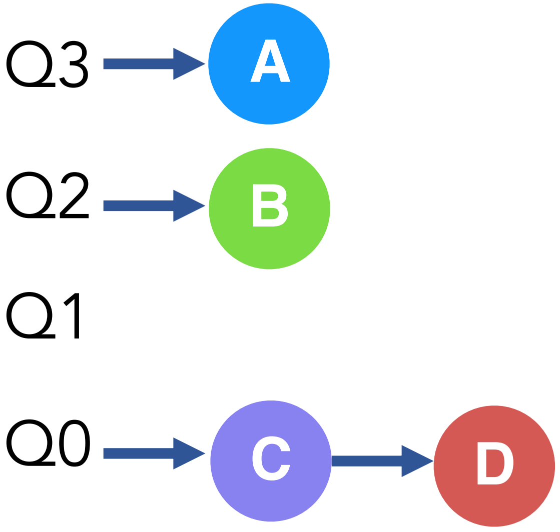
You'll get hands-on experience with it in Lab 1 ☺

Idea:

- Multiple queues representing different job types
- Queues w/ priorities: jobs in higher-priority queue preempt jobs lower-priority queue
- Jobs on same queue use the same scheduling algorithm, typically RR

Multilevel Queue Scheduling





MLFQ

Goal #1: Optimize job turnaround time for “batch” jobs

Goal #2: Minimize response time for “interactive” jobs

Challenge:

- No *a priori knowledge* of what type a job is, what the next burst is, etc.
- Let a job tells us its “niceness” (priority)?

Idea:

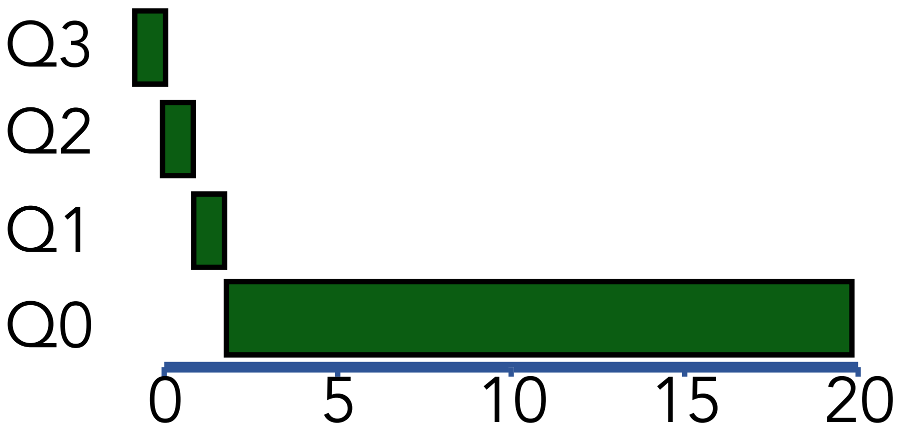
- Change a process’s priority based on how it behaves in the past (history “feedback”)

MLFQ: How to Change Priority Over Time?

Attempt

- *Rule A:* Processes start at top priority
- *Rule B:* If job uses whole slice, demote process
  - i.e., longer time slices at lower priorities
- Example 1: A long-running "batch" job

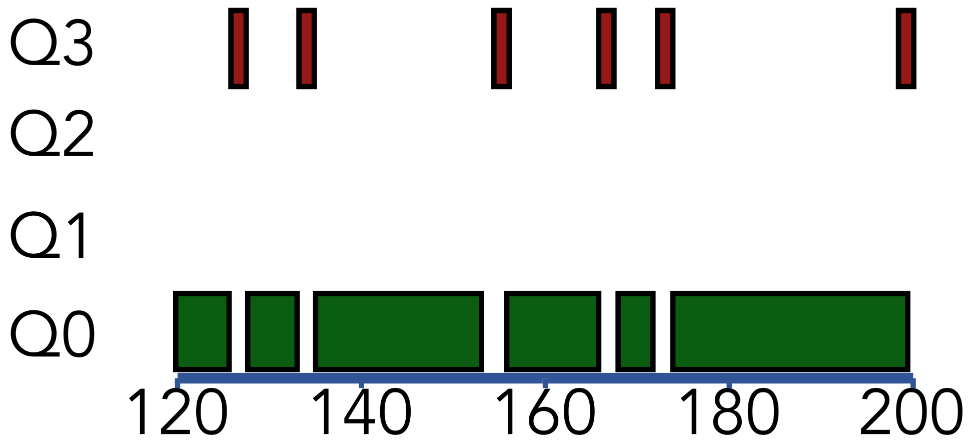
Why?



MLFQ: How to Change Priority Over Time?

Attempt

- *Rule A:* Processes start at top priority
- *Rule B:* If job uses whole slice, demote process
  - i.e., longer time slices at lower priorities
- Example 1: A long-running "batch" job
- Example 2: An "interactive" job comes along



MLFQ: How to Change Priority Over Time?

Attempt

- *Rule A:* Processes start at top priority
- *Rule B:* If job uses whole slice, demote process

Problems:

- unforgiving + starvation
- gaming the system
  - E.g., performing I/O right before time-slice ends

MLFQ: How to Change Priority Over Time?

Attempt

- *Rule A:* Processes start at top priority
- *Rule B:* If job uses whole slice, demote process

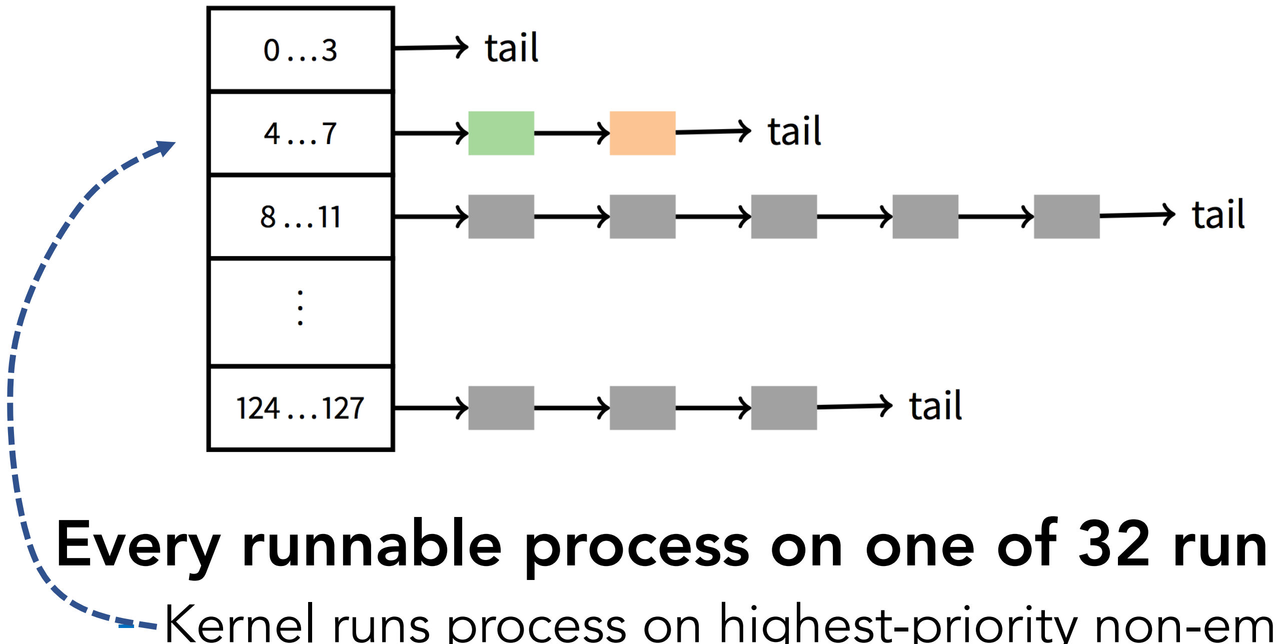
Problems:

- unforgiving + starvation
- gaming the system

Fixing the problems:

- Periodically boost priority for jobs that haven't been scheduled
- Account for job's *total* run time at priority level (instead of just this time slice)

MLFQ in BSD



Every runnable process on one of 32 run queues

- Kernel runs process on highest-priority non-empty queue
- Round-robins among processes on same queue

Process priorities dynamically computed

- Processes moved between queues to reflect priority changes

Favor interactive jobs that use less CPU

Process Priority Calculation in BSD

`p_estcpu` **– per-process estimated CPU usage**

`p_nice` **– user-settable weighting factor, value range [-20, 20]**

Process priority `p_usrpri`

- $$p\_usrpri \leftarrow 50 + \left(\frac{p\_estcpu}{4}\right) + 2 * p\_nice$$
- Calculated every 4 ticks, values are bounded to [50, 127]

Rationale: *decrease* priority linearly based on recent CPU

How to calculate `p_estcpu` **?**

- Incremented whenever timer interrupt found process running
- Decayed every second while process runnable

$$p\_estcpu \leftarrow \left(\frac{2 * load}{2 * load + 1}\right) * p\_estcpu + p\_nice$$

- Load is sampled average of length of run queue plus short-term sleep queue over last minute

Pintos Notes

Same basic idea for second half of Lab 1

- But 64 priorities, not 128
- Higher numbers mean higher priority (in BSD, higher num means lower prio)
- Okay to have only one run queue if you prefer (less efficient, but we won't deduct points for it)

Have to negate priority equation:

- Formula in BSD

$$p\_usrpri \leftarrow 50 + \left(\frac{p\_estcpu}{4}\right) + 2 * p\_nice$$

- Formula in Pintos

$$priority \leftarrow 63 - \left(\frac{recent\_cpu}{4}\right) - 2 * nice$$

Scheduling Overview

1. Goals of scheduling
2. Textbook scheduling
3. Priority scheduling
4. ~~Advanced scheduling topics~~

Multiprocessor Scheduling Issues

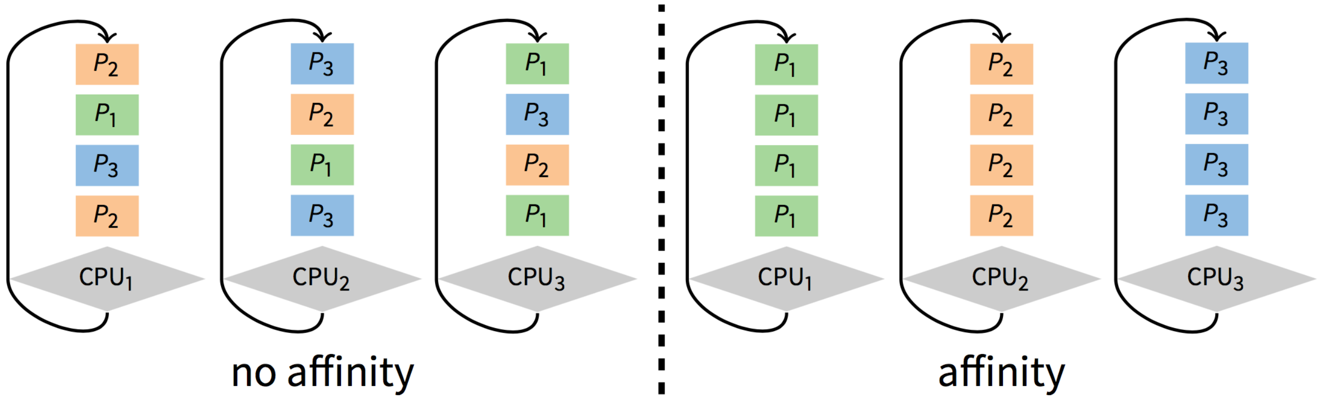
Must decide on more than which processes to run

- Must decide on which CPU to run which process

Moving between CPUs has costs

- More cache misses, depending on arch. more TLB misses too

***Affinity scheduling*—try to keep process/thread on same CPU**



- But also prevent load imbalances
- Do cost-benefit analysis when deciding to migrate…affinity can also be harmful, particularly when tail latency is critical

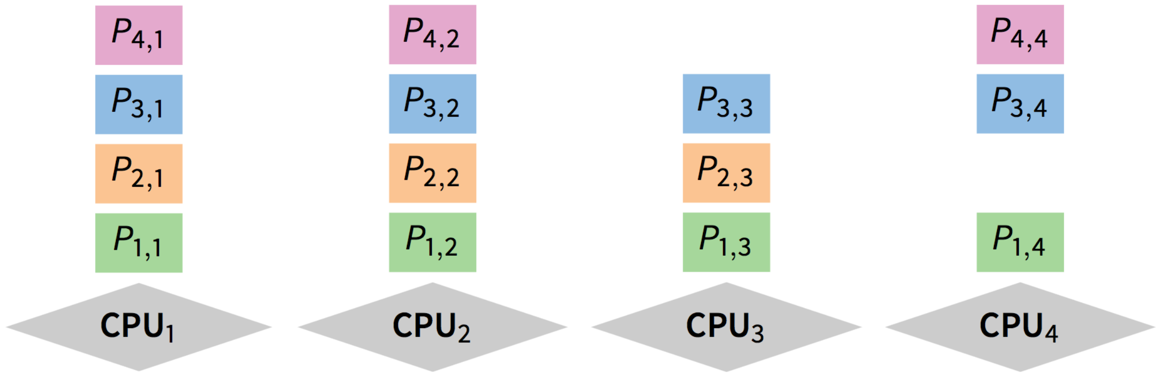
Multiprocessor Scheduling (cont)

Want related processes/threads scheduled together

- Good if threads access same resources (e.g., cached files)
- Even more important if threads communicate often, otherwise must context switch to communicate

Gang scheduling—schedule all CPUs synchronously

- With synchronized quanta, easier to schedule related processes/threads together



Real-time Scheduling

Two categories:

- Soft real time—miss deadline and CD will sound funny
- Hard real time—miss deadline and plane will crash

System must handle periodic and aperiodic events

- E.g., processes A, B, C must be scheduled every 100, 200, 500 msec, require 50, 30, 100 msec respectively
- *Schedulable* if $\sum \frac{cpu}{period} \leq 1$

Variety of scheduling strategies

- E.g., first deadline first (works if schedulable, otherwise fails spectacularly)

Scheduling Summary

Scheduling algorithm determines which process runs, quantum, priority…

Many potential goals of scheduling algorithms
- Utilization, throughput, wait time, response time, etc.

Various algorithms to meet these goals
- FCFS/FIFO, SJF, RR, Priority

Can combine algorithms
- Multiple-Level Feedback Queues (MLFQ)

Advanced topics
- *affinity scheduling, gang scheduling, real-time scheduling*

Next Time

Read Chapter 26, 27