CE 440 Introduction to Operating System

Lecture 5: Scheduling
Fall 2025

Prof. Yigong Hu

BOSTON UNIVERSITY

Slides courtesy of Manuel Egele, Ryan Huang and Baris Kasikci

Administrivia

Lab 0

- Due this Friday
- Done individually (cannot share with or copy form your to-be-teammates)

Find your project group member soon

- So you can get started with Lab 1 without delay
- Fill out Google form of group info (will upload on Piazza)
    - https://docs.google.com/forms/d/e/1FAIpQLScqr0QdmoruMu_w7-FizeQ9OYaijg9-d9Y58zOV28wivnYp5A/viewform?usp=dialog

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)

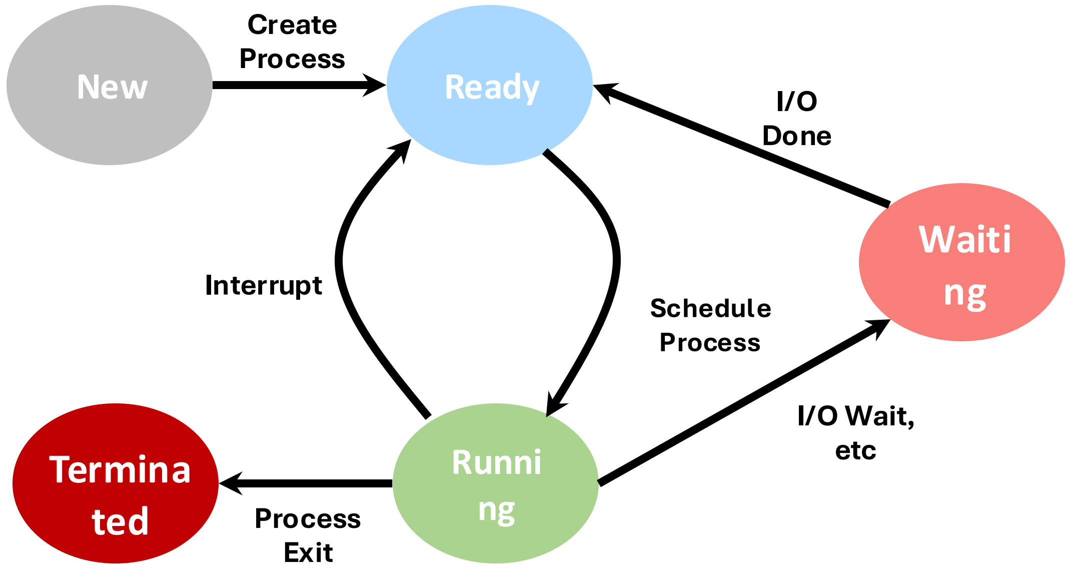
Process vs. thread

Process/thread states and APIs

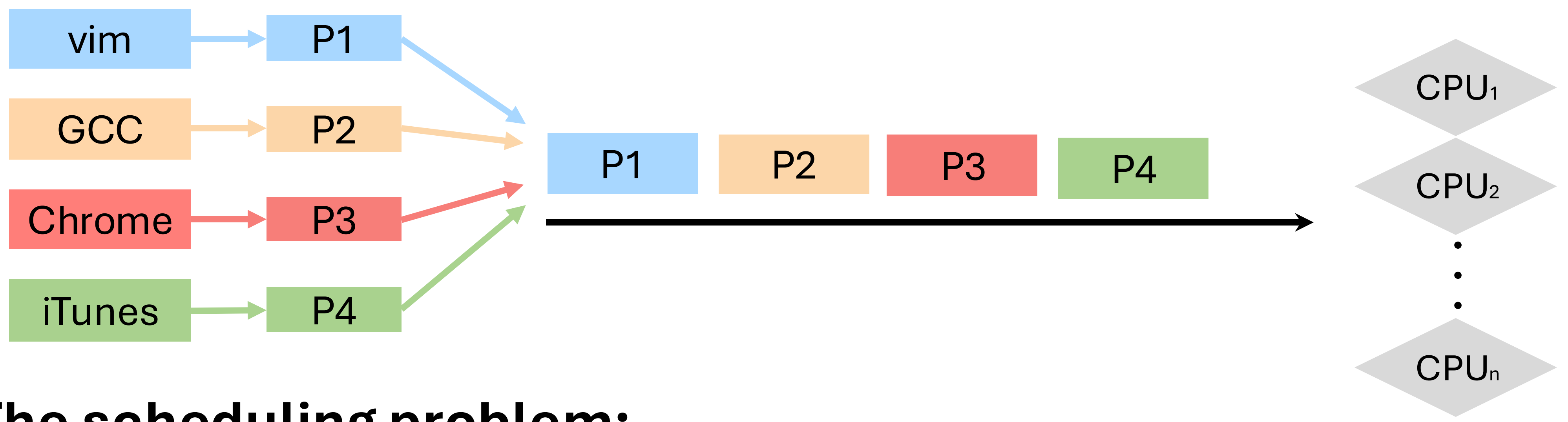
- State graph and queues
- Process creation, deletion, waiting

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

Mechanism: context switch, process state queues

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.

Scheduling Goals

Goal 1: guarantee "good service"

- To decide what job to run next and for how long
- Good service could be one of many different criteria
  - Fairness – giving each process a fair share of the CPU
  - Throughput – maximize jobs per second
  - Response time - respond to requests quickly

Known as short-term scheduling decision

- Happens relatively frequently
- Want to minimize the overhead of scheduling
  - Fast context switches, fast queue manipulation

Scheduling Goals

Goal 2: loaded jobs into memory

- To determine the multiprogramming level: how many jobs to run simultaneously
- Moving jobs to/from memory is often called swapping

Known as long-term scheduling decision

- Happens relatively infrequently
- Significant overhead in swapping a process out to disk

Virtual Memory Lecture (Lecture 9-12)

What Is “Good Service”?

How do we measure the effectiveness of a scheduling algorithm?

Batch systems strive for

- 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
- *CPU utilization* – $\% CPU$ fraction of time CPU doing productive work

What Is “Good Service”?

Interactive systems strive to

- Minimize response time for interactive jobs (PC)
  - $T_{response} - T_{request}$:time between *waiting* → *ready* transition and *ready* → *running*
  - Lower is better
- Proportionality – meet users’ expectations
  - Service-level objective(SLO)
- Utilization and throughput are often traded off for better response time

Real-time systems

- Meeting deadlines: avoid losing data
- Predictability: avoid quality degradation in multimedia systems

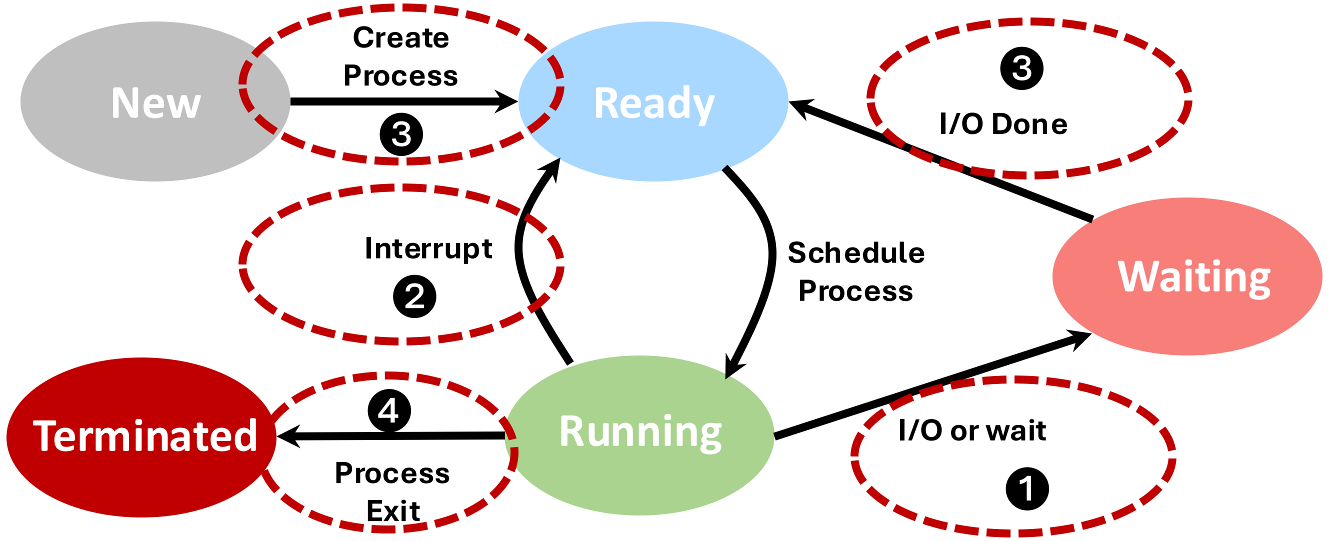
Tradeoffs

Improving on one metric can hurt another

For example:

- We want to improve throughput, so we decide to only schedule short jobs
- But now longer jobs never get run, so their turnaround time is effectively infinite

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 Overviews

- Textbook scheduling

- Priority scheduling

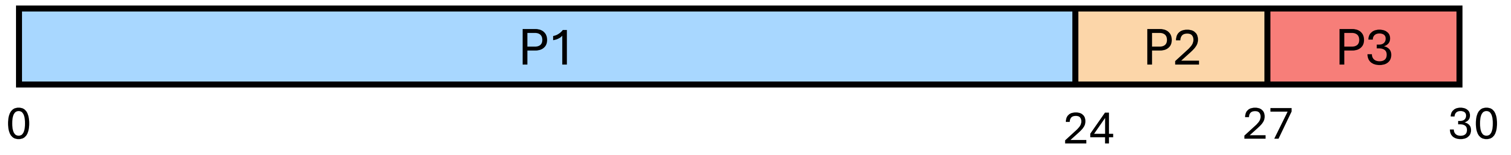
- Advanced scheduling topics (not covered)

FCFS Scheduling

"First-come first-served" (FCFS): **Run jobs in order that they arrive**

Examples:

- Say P1 needs 24 sec, while P2 and P3 need 3.
- Say P2, P3 arrived immediately after P1



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

Turnaround Time: P1 : 24, P2 : 27, P3 : 30

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

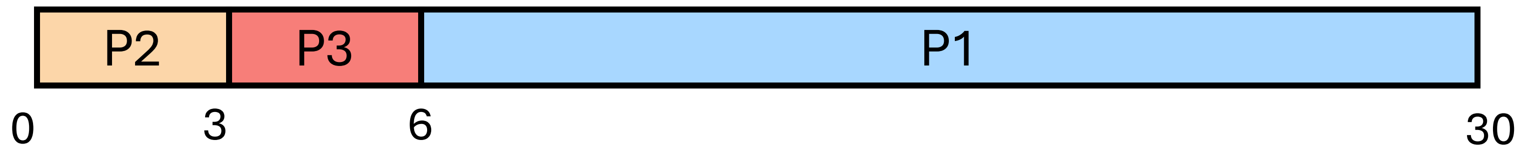
Waiting Time: P1 : 0, P2 : 24, P3 : 27

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

Can we do better with FCFS?

FCFS Scheduling Continued

Suppose we scheduled P2, P3, then P1



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

Turnaround Time: P1 : 30, P2 : 3, P3 : 6

- Average TT: (30 + 3 + 6) / 3 = 13

Observations: 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

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

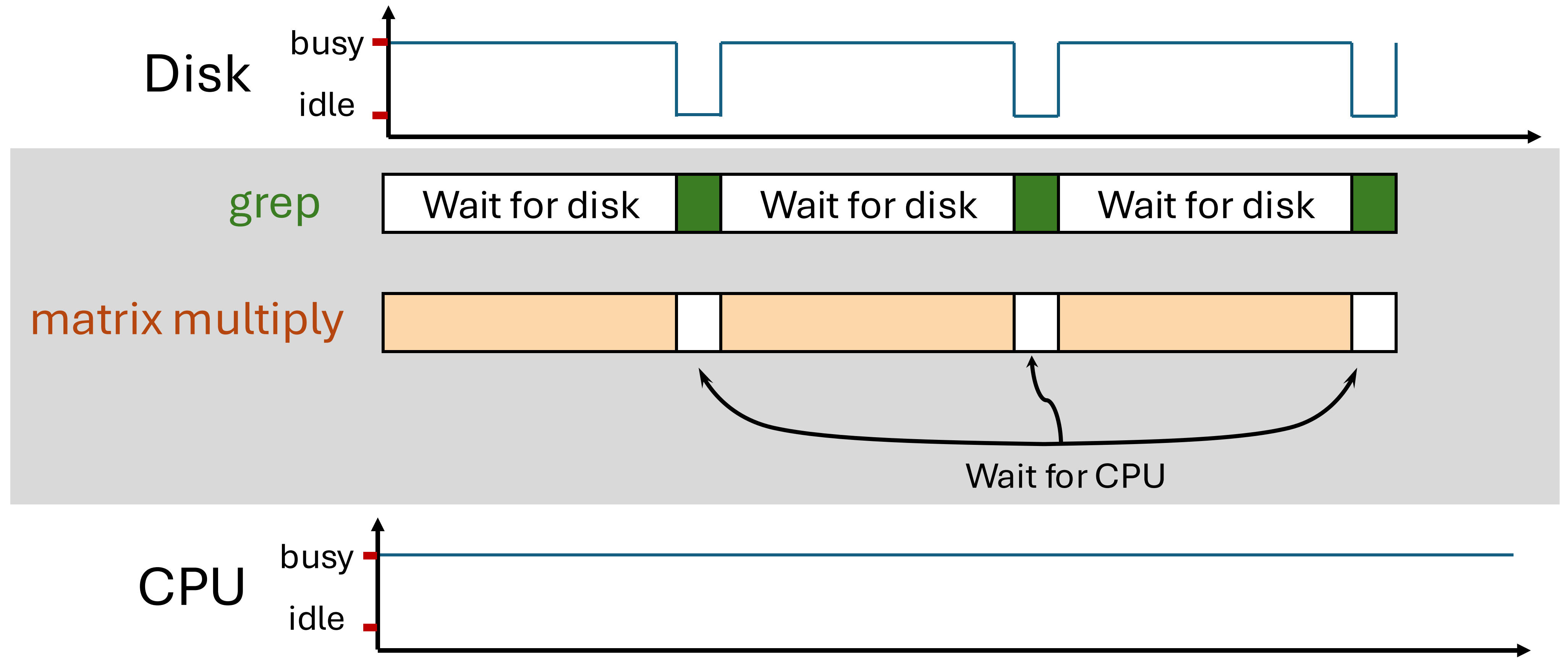
Scheduling 1**-CPU system with** n **I/O devices like scheduling asymmetric** (n + 1)**-CPU multiprocessor**

- Result: (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

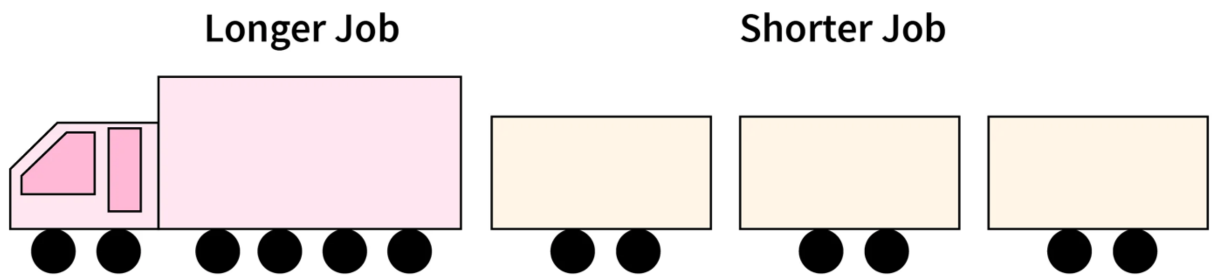


FCFS Limitations

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*



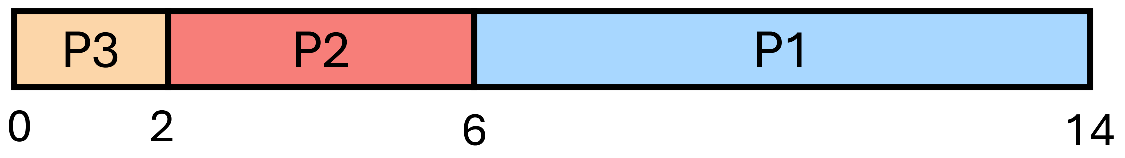
Shortest Job First (SJF)

Shortest Job First (SJF)

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

Examples:

- Say P1 needs 8 sec, P2 4 sec and P3 2 sec.



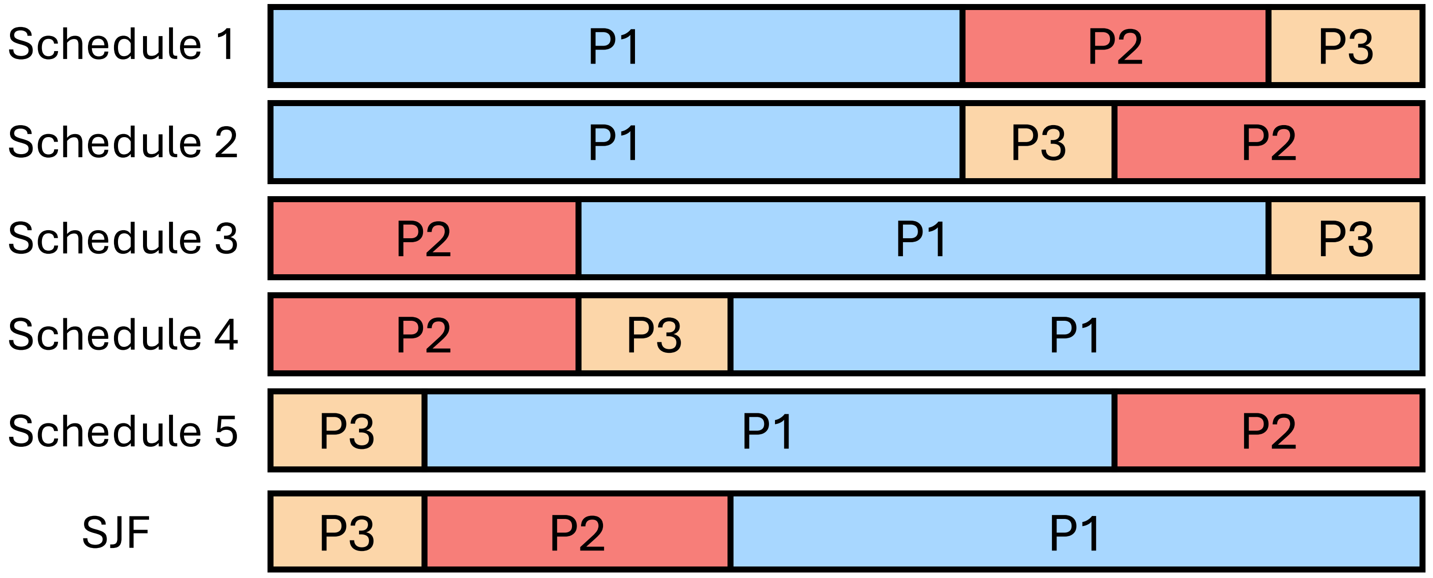
Average Waiting Time: (0 + 2 + 6) / 3 = 2.67

SJF Has Optimal Average Waiting Time

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

Previous Examples:

- P1 needs 8 sec, P2 4 sec and P3 2 sec.



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

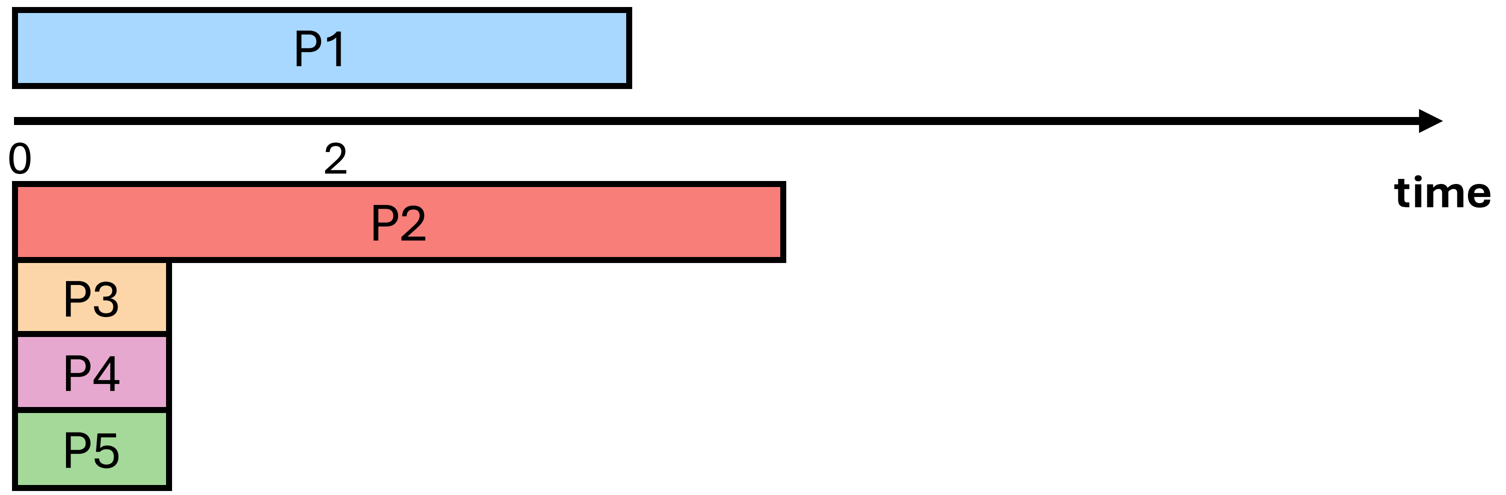
Problem: what if new jobs arrive?

Counterexample

The optimality proof only applies when all jobs are available at time 0

Suppose we have instead:

- At time 0, P1 needs 4 sec and P2 needs 5 sec.
- At time 2 seconds, processes P3, P4, and P5 arrive, each requiring 1 second of CPU time.

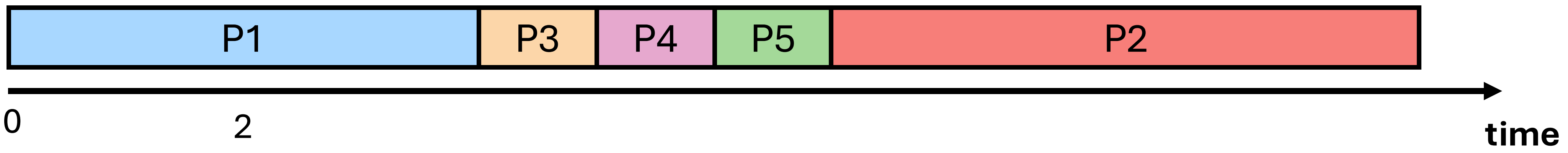


Counterexample

The optimality proof only applies when all jobs are available at time 0

Suppose we have instead:

- At time 0, P1 needs 4 sec and P2 needs 8 sec.
- At time 2 seconds, processes P3, P4, and P5 arrive, each requiring 1 second of CPU time.



What is the AWT?

Shortest Remaining Time Next

SRTF chooses the process whose remaining run time is the shortest

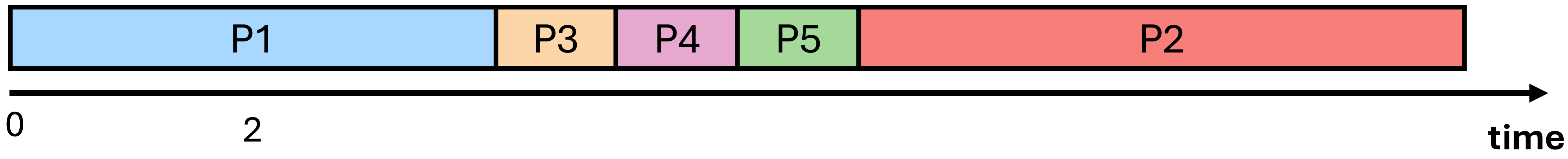
- When a new job arrives, its remaining run time is compared to the one of the currently running process
- If current process has more remaining time than the run time of new process, the current process is preempted and the new one is run

Examples with Preemptive

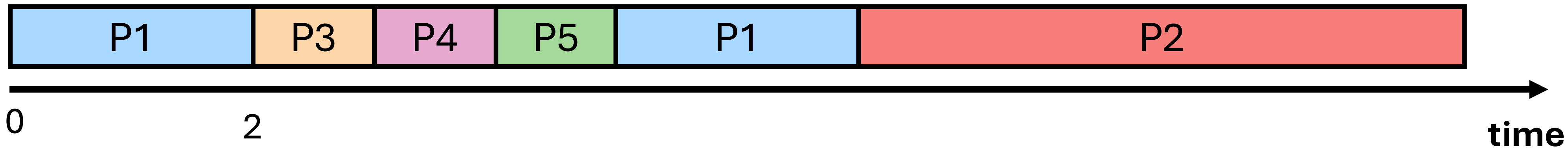
Process	Arrive Time	Burst Time
P1	0	4
P2	0	5
P3	2	1
P4	2	1
P5	2	1

What is the AWT?

Non-preemptive SJF:



Preemptive SRJF:



SJF Limitations

This algorithm also assumes that running time for all the processes to be run is known in advance

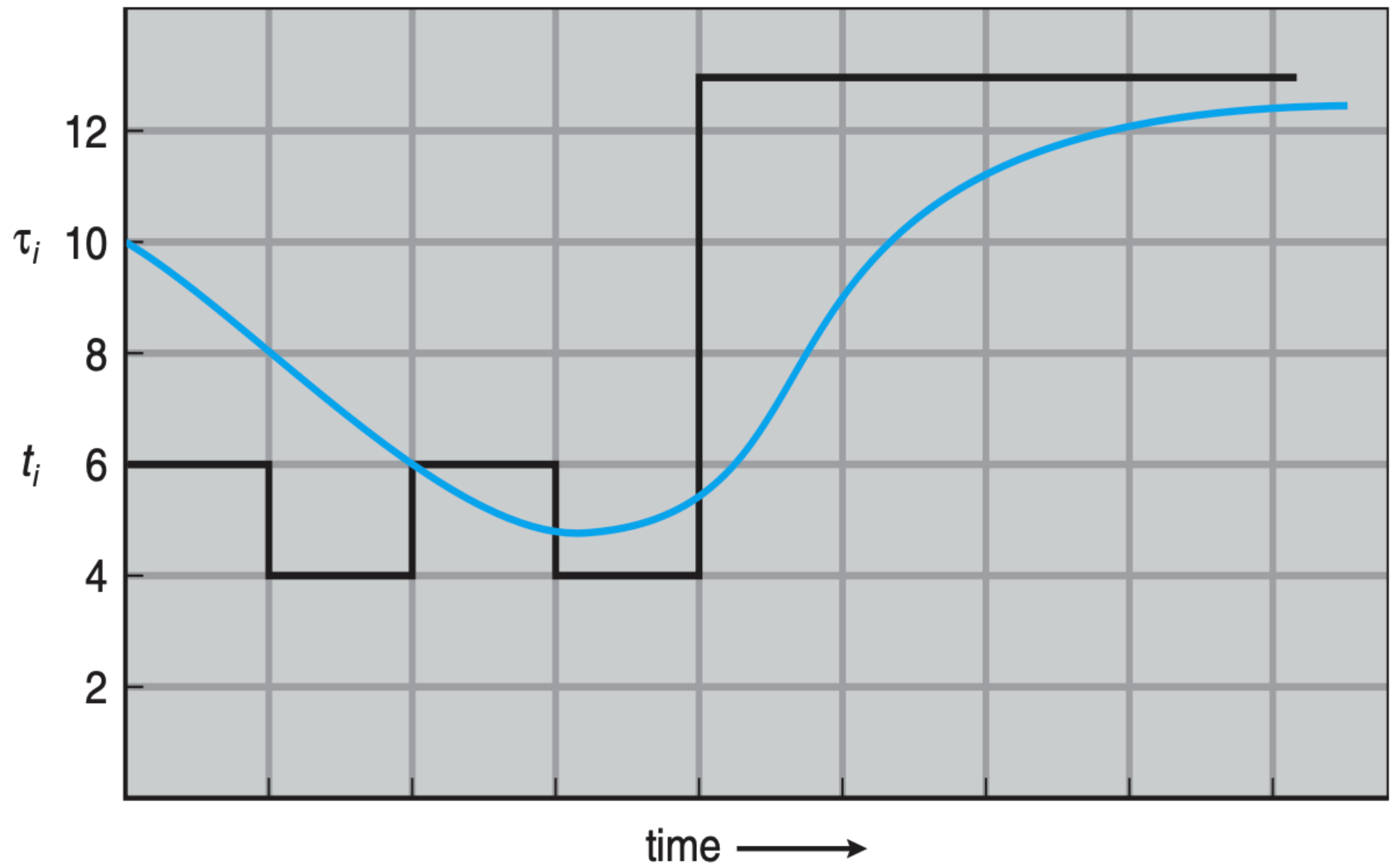
- Impossible to know size of CPU burst ahead of time

Can potentially lead to unfairness or starvation

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 $\tau_{n+1} = t_n + (1 - \alpha)\ \tau_n$

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)

Now, since we have preemptive scheduling:

- Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds
- Run first process until its quantum is used up
- Move that process to the end and run the next process
- Simple, fair
  - No process waits forever

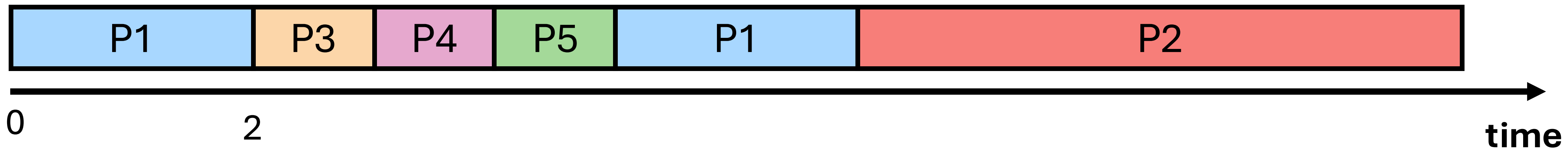
Solution to fairness and starvation

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

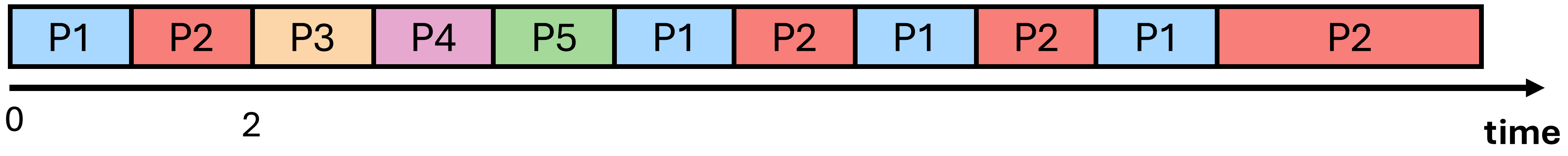
Examples with Round Robin

Process	Arrive Time	Burst Time
P1	0	4
P2	0	5
P3	2	1
P4	2	1
P5	2	1

Preemptive SRJF:



Round Robin with quantum as 1 second



Advantage of Round Robin

Advantages:

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

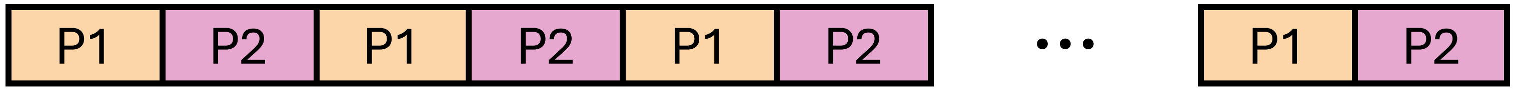
Disadvantages?

Disadvantages of Round Robin

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?
- Even worse than FCFS

Round Robin Discussion

How to pick quantum?

- What if too big?
  - Response time can be very bad
- What if time slice too small?
  - A notable percentage of the CPU time is spent in switching contexts

Actual choices of time slice:

- Initially, UNIX time slice 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

Scheduling Overviews

- Textbook scheduling
- Priority scheduling
- Advanced scheduling topics (not covered)

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

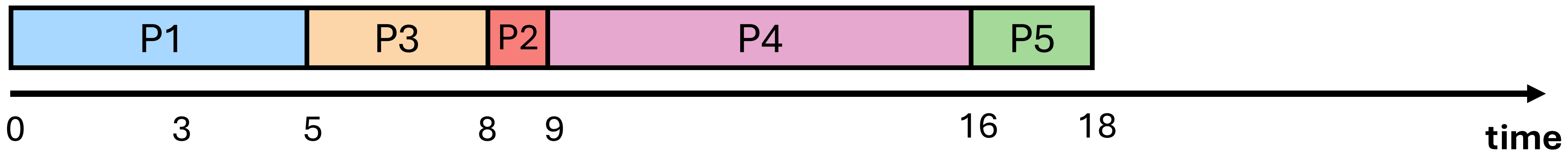
Solution? "Age" processes

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

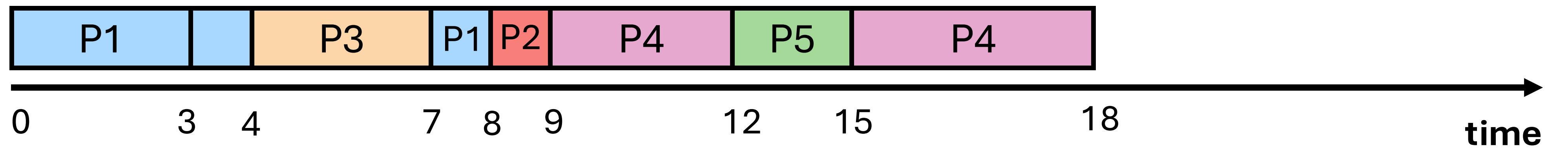
Examples with Priority Scheduling

Process	Arrive Time	Burst Time	Priority
P1	0	5	2
P2	3	1	1
P3	4	3	4
P4	8	7	0
P5	12	2	3

Non-preemptive priority scheduling:



Preemptive priority scheduling

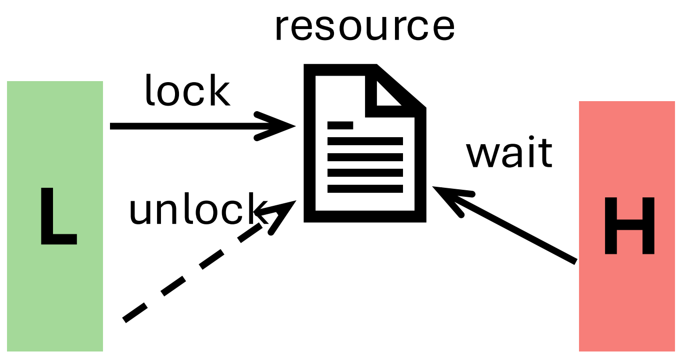


Priority Inversion (1)

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



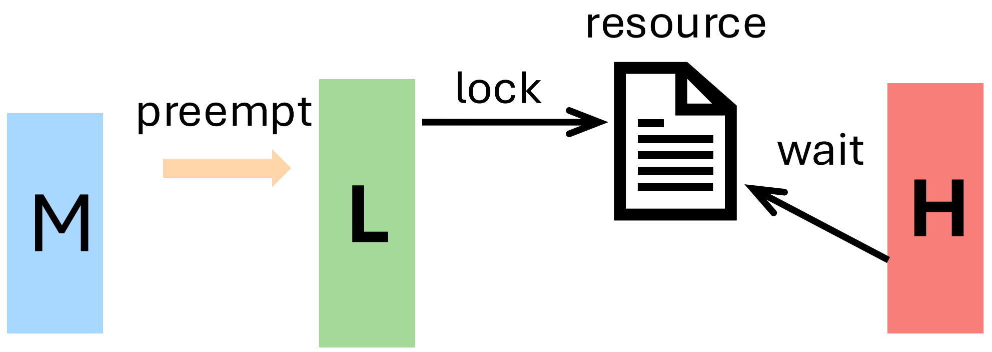
Priority Inversion (2)

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

- What if we have a tasks *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

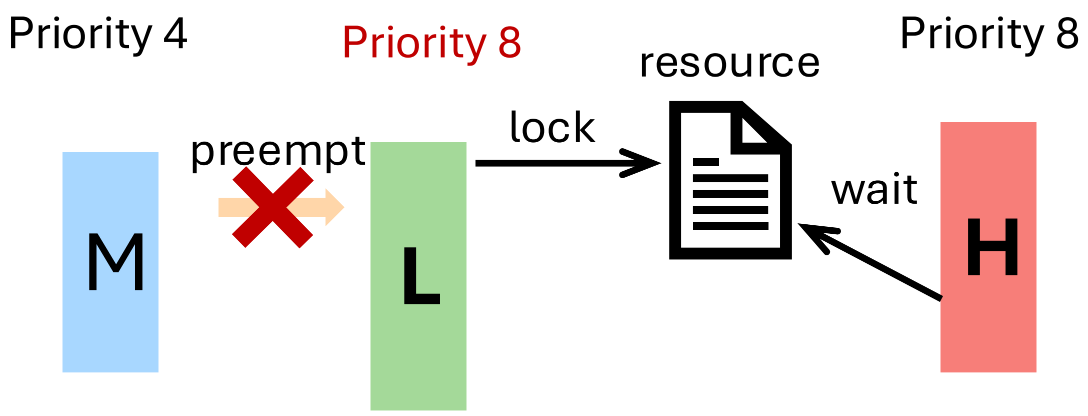
"Donate" our priority if we get blocked

- 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



Pintos Lab 1 Exercise 2.2

Details in lab 1 overview session

Combing 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

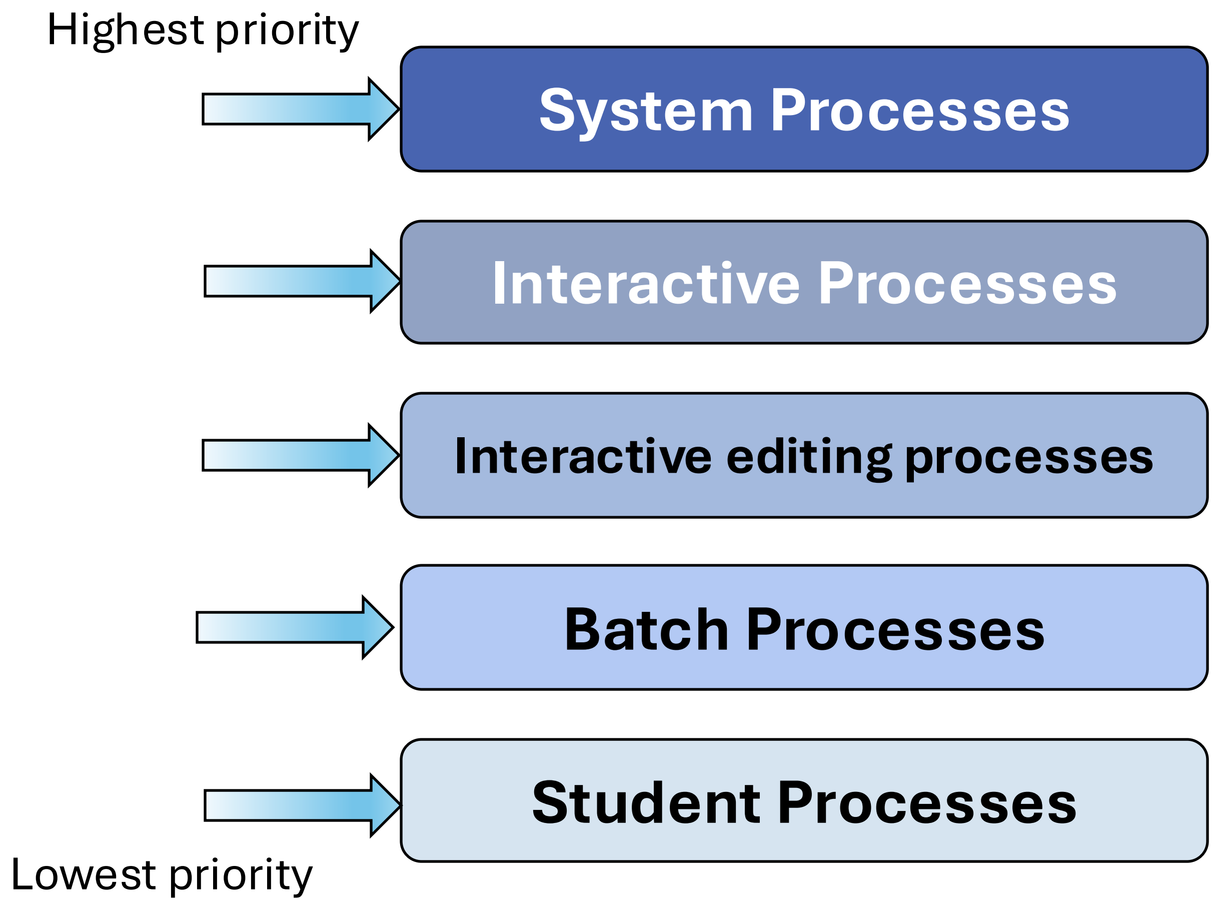
Widely used in mainstream OSes: Unix, BSD, Windows, MacOS

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

Multiple-level Queues Scheduling



Multiple-level Feedback Queues Scheduling

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")

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 : A long-running “batch” job

Problems:

- Starvation
- Gaming the system
    - E.g., performing I/O right before time-slice ends

How to Change Priority Over Time

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

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

- Calculated every 4 ticks, values are bounded to [50, 127]
- 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

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

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

Tips for Pintos

Same basic idea for second half of Lab 1

- But 64 priorities, not 128
- Higher numbers mean higher priority (in BSD, higher numbers means lower priority)
- 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:

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

In Pintos $p\_usrpri \leftarrow 63 + \left(\frac{recent\_cpu}{4}\right) + 2 \times nice$

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)

Next Time

Read Chapter 28,29