# Scheduling

Instructor: Linyi Li
*Slides adapted from Dr. B. Fraser*

# Topics

1) Computers seem magically able to do more than one thing at once.

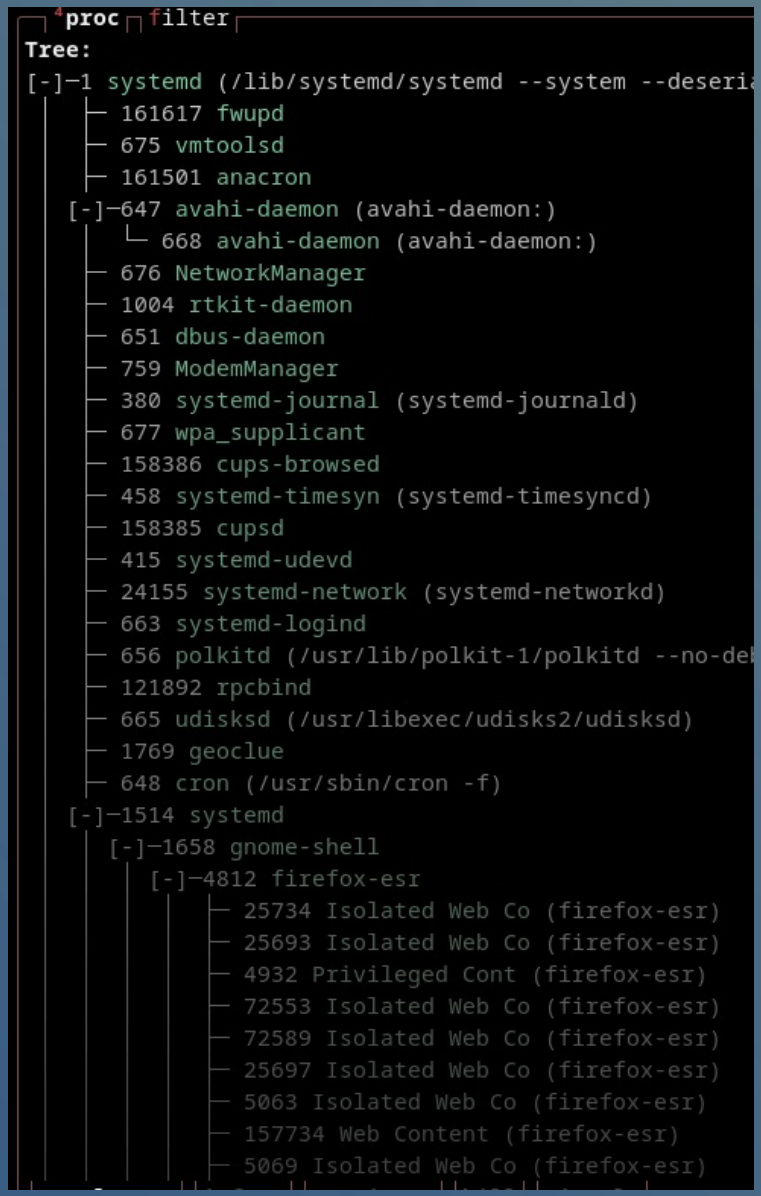
a) How do they have multiple programs running "at once"?

b) How can multiple users log into a computer at once?

2) How does the kernel decide who's turn it is to use the CPU?

# The story so far...

- *"In the beginning"...* CPUs had a single core and one program running.
- Then "*back in the day...*" computers had a single core but many users.
  - Each user might have a terminal and want to run programs
  - How do they share the same CPU?
- "*These days...*" CPUs have many cores, but..
  many more processes than cores.
- Things that need to run:
  .. processes, tasks, jobs, threads



# More Depth

- We will cover scheduling a little to understand the problem.
  - CMPT301 teaches it in depth
- Can read more in OSTEP
  https://pages.cs.wisc.edu/~remzi/OSTEP/

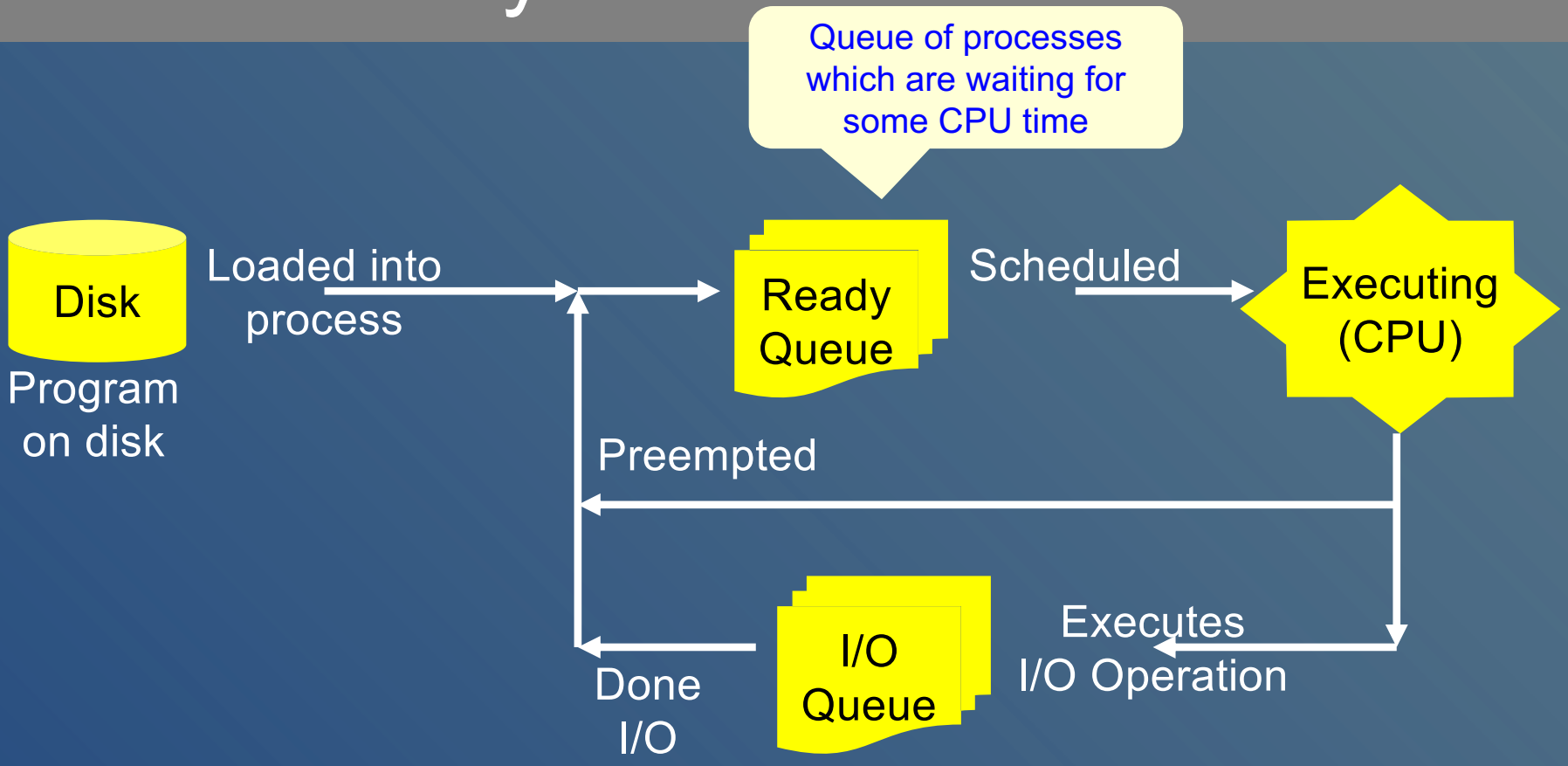
  Has in-depth discussions beyond scope of this course.
  - Chapter 7 Scheduling: Introduction
    https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched.pdf
  - Chapter 8 Scheduling: The Multi-Level Feedback Queue
    https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf
  - Chapter 9.7 The Linux Completely Fair Scheduler (CFS)
    https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-lottery.pdf

# CPU Scheduling

# CPU Scheduling

- CPU Scheduling
  - .. Sharing a core by multiple processes
  - Or, sharing multiple cores by multiple processes (beyond the scope of this course)
- Context switch
  - .. Stop running one process and start running another process.
  - This has overhead (work) to do this switch, so don't do it too frequently.
  - Stopped process can later be resumed exactly where it left off once it has another turn on the CPU.

# Process Lifecycle




- Scheduling
  - Picking one process to run next (from ready queues).
- Scheduler
  Component of the kernel that picks the next process to run.

# Types of Scheduling Algorithm

- Non-preemptive Scheduling
  - A process gives up the core when it:
    - ...terminates or
    - ..."waits" for an operation that takes a long time,

    e.g., I/O, child wait, sleep
- Preemptive Scheduling
  - The kernel stops a *process* at any time.
- Preemptable Kernel
  - (almost) all syscalls into the kernel itself can be preempted!
  - Linux Real-time kernel (PREEMPT_RT) merged to mainline kernel Sept 2024!

# Scheduling Criteria

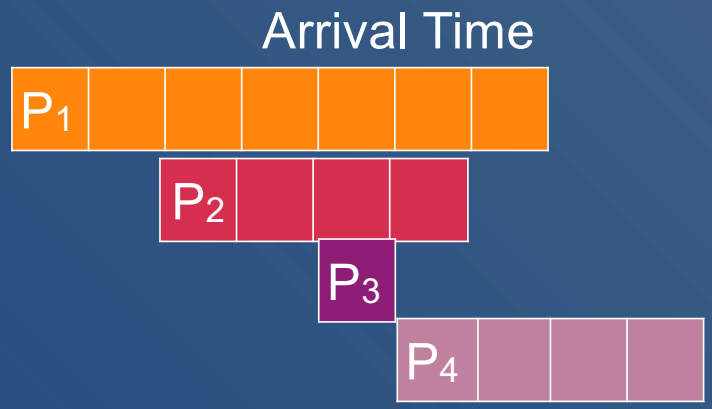
- We want to maximize:
  - CPU utilization : keep the CPU as busy as possible
  - Throughput : # of processes that complete their execution per time unit

- We want to minimize:
  - Turnaround time:
    amount of time to execute a particular process
    (time from submission to termination)
  - Waiting time : amount of time a process has been waiting in ready queue
  - Response time : amount of time it takes from when a request is submitted until the first response is produced

# Scheduling Algorithms

# Scheduling Simplifications

- Each process needs the CPU for a certain amount of time.
  - We'll assume we know how much time it needs at the start, but could be estimated.
- Often processes are long lived, but only need the CPU in short bursts of CPU time.
  - Here we'll just look at one such burst of time, but in reality a process often has many such bursts.



# First Come, First Served (FCFS)

- Simplest algorithm
  - Run in the order of arrival.
  - Non-preemptive

(once running, a process keeps running)

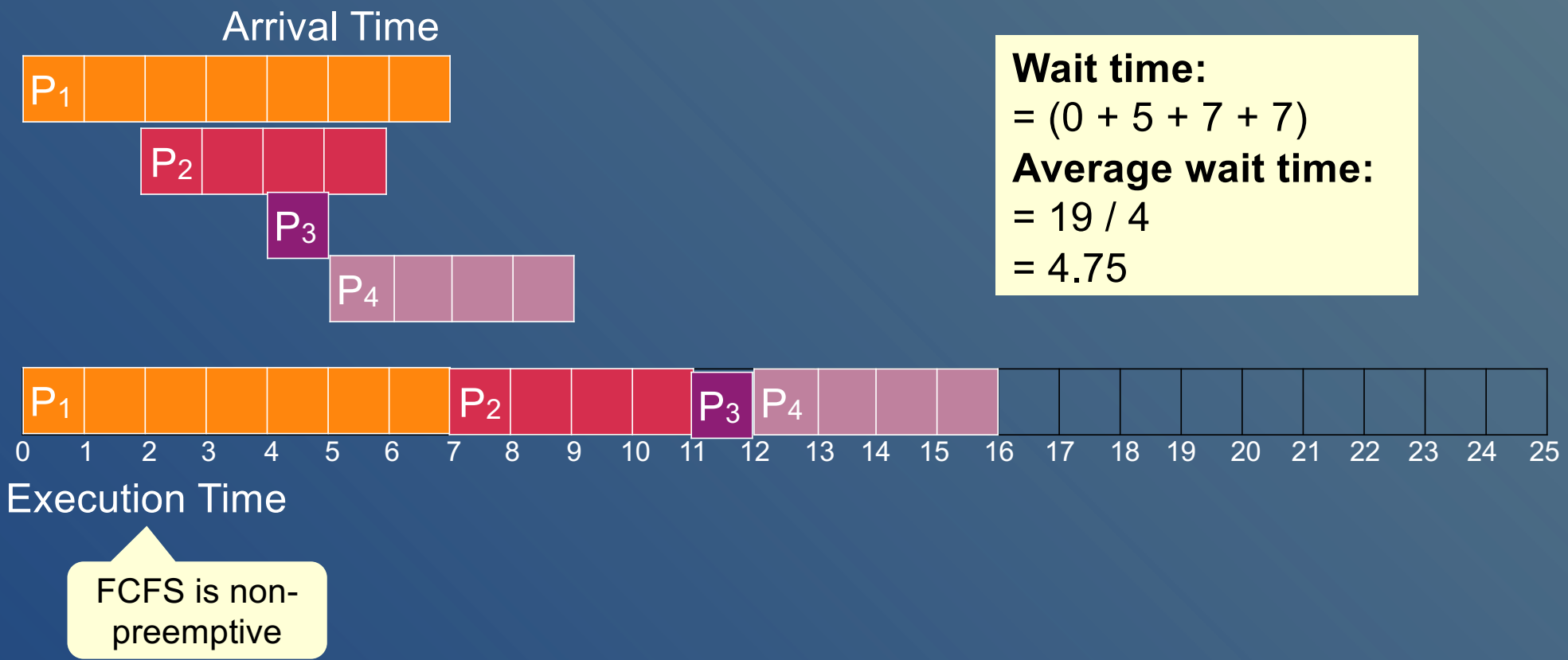
- Waiting time
  - Sum of how long each process is in the ready queue.

- Used to assess how good an algorithm is.

There are other metrics are based on scheduling criteria above, but waiting time is easy to calculate, so we'll use it for comparing scheduling algorithms.

# First Come, First Serve Example

| | P1 | P2 | P3 | P4 |
|---|---|---|---|---|
| Execution Time | 7 | 4 | 1 | 4 |
| Arrival Time | 0 | 2 | 4 | 5 |




**Wait time:**
= (0 + 5 + 7 + 7)
**Average wait time:**
= 19 / 4
= 4.75

# ABCD: First Come, First Served (FCFS)

- What is the total wait time for the following processes using FCFS? (if same arrival time, order by index)

| | P1 | P2 | P3 | P4 |
|---|---|---|---|---|
| Execution Time | 40 | 20 | 8 | 10 |
| Arrival Time | 0 | 0 | 0 | 0 |

(a) 40 + 20 + 8 = 68

(b) 40 + 20 + 8 + 10 = 78

(c) 40 + 60 + 68 = 168

(d) 40 + 60 + 68 + 78 = 246

# ABCD: First Come, First Served (FCFS)

- What is the total wait time for the following processes using FCFS? (if same arrival time, order by index)

| | P1 | P2 | P3 | P4 |
|---|---|---|---|---|
| Execution Time | 10 | 20 | 8 | 40 |
| Arrival Time | 0 | 0 | 0 | 0 |

(a) 10 + 30 + 38 = 78

(b) 10 + 30 + 38 + 78 = 156

(c) 10 + 20 + 8 = 38

(d) 10 + 20 + 8 + 40 = 78

- What is the problem with FCFS?
  - A long process can sabotage all other processes.

# Shortest Job First (SJF)

- Let's try something where a long process doesn't sabotage all other processes.

- Shortest Job First Scheduling Algorithm
  - Among the remaining processes, pick the process with the shortest execution time.

Assume for the sake of discussion, we know how long each process takes.

Not always possible but can be estimated.

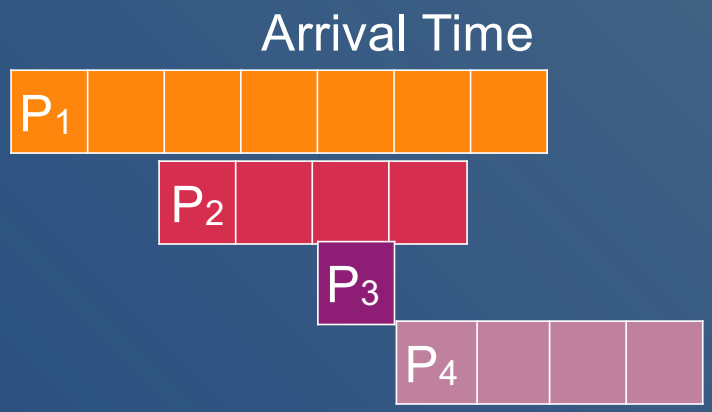
- Non-preemptive:

Once running, a job runs to completion

- * Theoretically optimal waiting time, if all processes arrive at the beginning
  - Prove by reordering

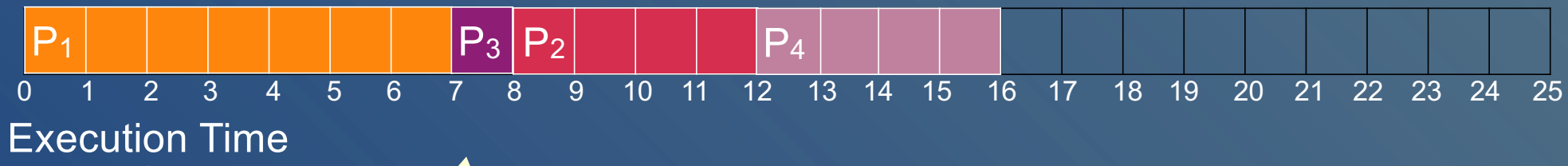
# Shortest Job First Example

| | P1 | P2 | P3 | P4 |
|---|---|---|---|---|
| Execution Time | 7 | 4 | 1 | 4 |
| Arrival Time | 0 | 2 | 4 | 5 |




**Total Wait Time:**
= (0 + 6 + 3 + 7) = 16

**Average wait time:**
= 16 / 4
= 4




SJF is non-preemptive

Shortest

# Shortest Remaining Time First (SRTF)

- Schedule the process with..
  the shortest remaining execution time.

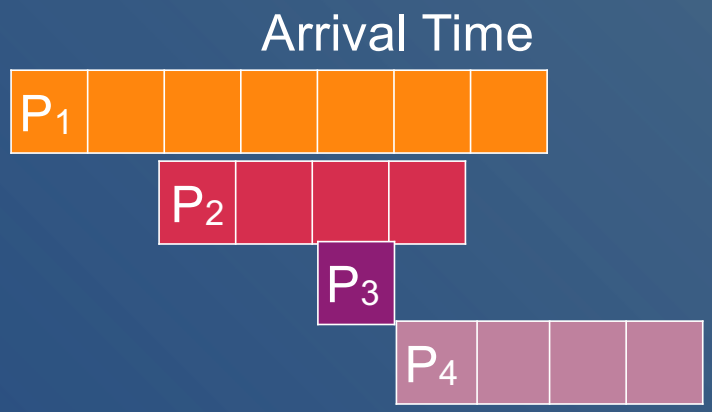
– This is preemptive:

–

When a new job arrives,
.. it can interrupt currently executing job.

# Shortest Remaining Time First Example

| | P1 | P2 | P3 | P4 |
|---|---|---|---|---|
| Execution Time | 7 | 4 | 1 | 4 |
| Arrival Time | 0 | 2 | 4 | 5 |




**Wait Times**

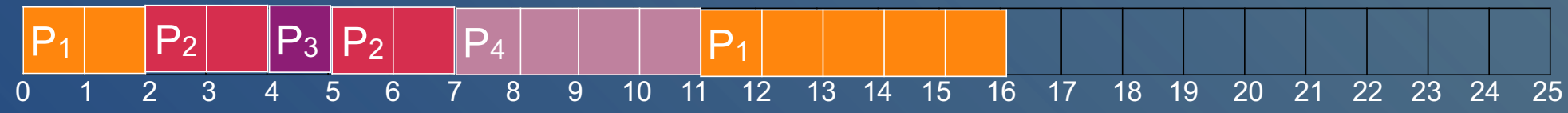
| P1 | P2 | P3 | P4 |
|---|---|---|---|
| 0+9 | 0+1 | 0 | 2 |

= 12

**Average Wait Time**

= 12 / 4

= 3




Execution Time

SRTF is preemptive

Always pick shortest remaining time

# More on SRTF and Wait Time Counting

- When preempted,
  - Wait counter starts;
  - So, for a process

**Wait Times**

| P1 | P2 | P3 | P4 |
|---|---|---|---|
| 0+9 | 0+1 | 0 | 2 |
| = 12 | | | |

Wait time = end time – execution time – arrival time

- * SRTF (Shortest Remaining Time First) minimizes waiting time
  - Why?
  - Recall SJF (Shortest Job First, the non-preemptive version) conditionally minimizes waiting time
  - SRTF is anytime SJF
  - Prove by contradiction

# Round Robin (RR)

- Forget about knowing how long things take:
  just give everyone.. equal length turns use round robin.
- Preemptive

–Quantum:
.. How long a turn each process gets on the CPU

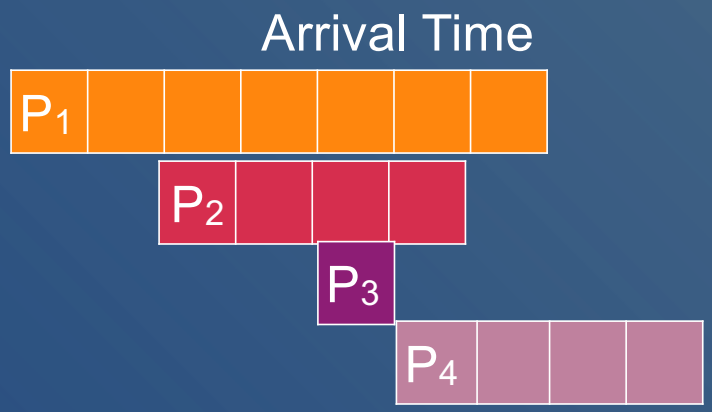
–Each x units of time (quantum) the scheduler will:

- Move currently running process
  to the back of the ready queue
- Take first process in ready queue and runs it
- Go to next process if quantum used up or current process finishes

–Newly arrived processes go at back of ready queue

# Round Robin Example (Quantum = 3ms)

| | P1 | P2 | P3 | P4 |
|---|---|---|---|---|
| Execution Time | 7 | 4 | 1 | 4 |
| Arrival Time | 0 | 2 | 4 | 5 |




**Wait Times**

| P1 | P2 | P3 | P4 |
|---|---|---|---|
| 0+3+5 | 1+7 | 5 | 5+2 |

= 28

**Average Wait Time**

= 28 / 4

= 7


Execution Time

Change every 3ms

# ABCD: Round Robin

- If the quantum is **very** long,
  then round robin is effectively the same as:

(a) First come first serve

(b) Shortest Job First

(c) Shortest remaining time first

- If the quantum is **very** short, what can go wrong?

(a)Processes do not make progress because they keep being reset when preempted

(b)Processes do not make progress because they keep being killed when preempted

(c)Context switch overhead is too high

(d)The ready queue is likely to be empty

# Priority Scheduling

- Priority Scheduling
  - .. Run the process in ready queue with the highest priority.
  - This can be either preemptive or non-preemptive.

When a new process is added to the ready queue, do we allow a context switch?

- Motivation: real-time tasks with deadlines
  - Some systems require hard or soft deadlines for their computational tasks
  - e.g, an airplane controller must respond to an outside event (e.g., an incoming bird) within a fixed (short?) time period.

# Real-Time Deadlines

- Hard real-time systems:
  - .. Strict deadlines that **must not** be missed

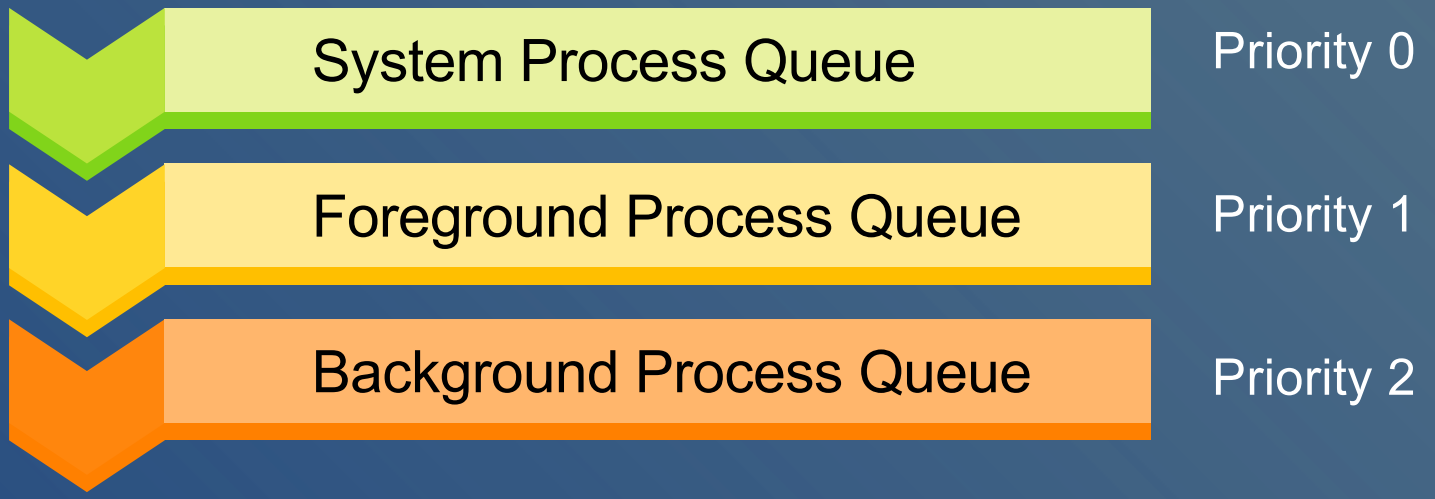
  because the consequence of missing a deadline can be catastrophic.
- Soft real-time systems:
  - Approximate deadlines that can be missed but should not be by much.
- Real-time tasks usually have higher priorities, i.e., they are more important to run.

# Priority Scheduling (cont)

- Task priority is typically expressed as a number (where a smaller number has a higher priority).
- Problem: Starvation
  - .. Lower priority processes may never run.
  - e.g, If high priority processes keep arriving, low priority processes may never run

# Multilevel Queue Scheduling

- Multilevel Queue Scheduling
  - .. Group processes based on categories
  - Each category gets its own
    .. ready queue and a priority value.



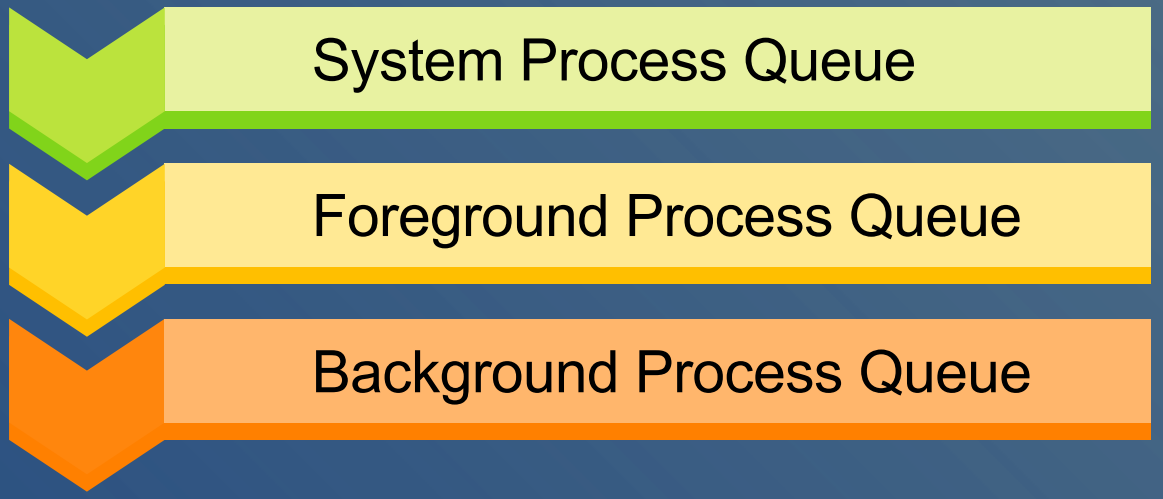
# Multilevel Queue Scheduling

- Each queue gets CPU time based on priority
  - One idea: Weighted Round Robin

    It's RR, but give more turns to higher-priority queues.
  - e.g., Schedule turns for each priority:

    0, 1, 2, 0, 1, 0, 0, 1, 2, 0, 1, 0, 0, 1, 2, 0, 1, 0, ....

Priority 0 gets more turns than 1 or 2.




Priority 0
RR

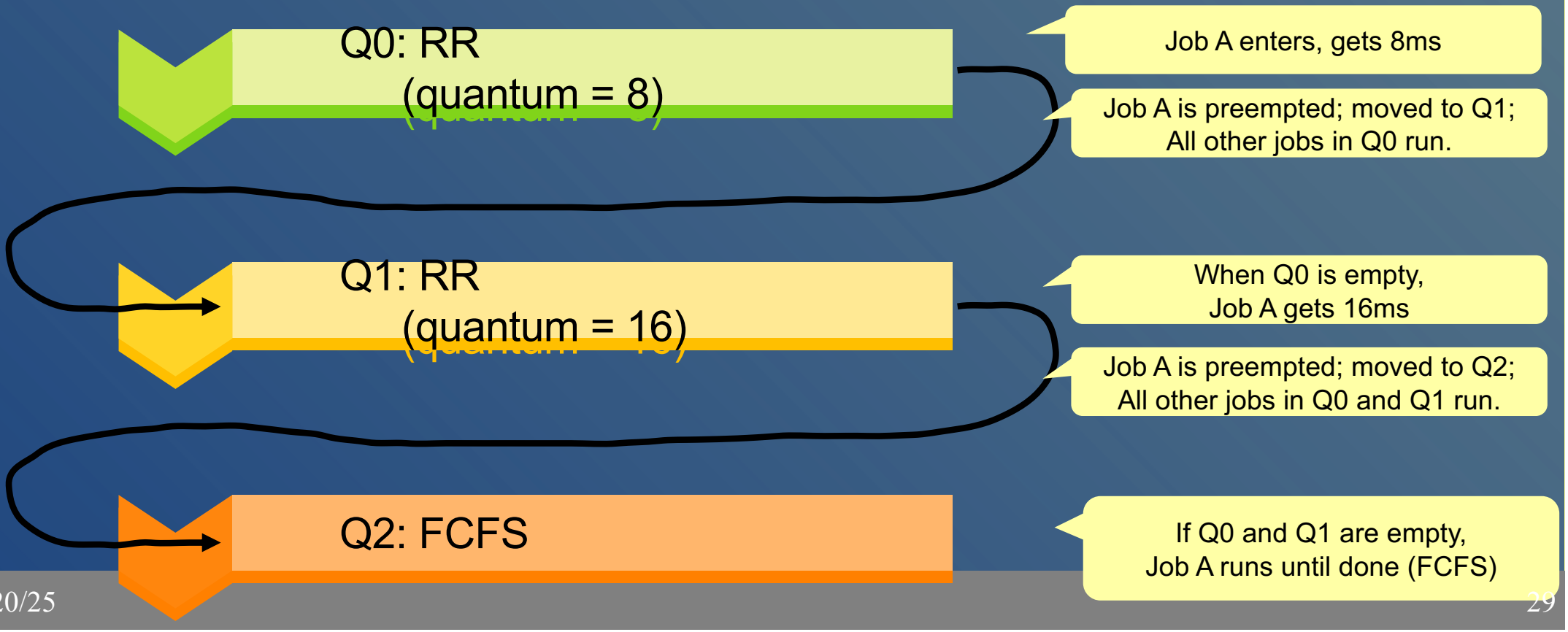
Priority 1
FCFS

Priority 2
SJF

- During each queue's turn:
  - .. Uses some scheduling algorithm to pick what process in that queue is scheduled.
- Avoids starvation: Each queue gets a chance to run.

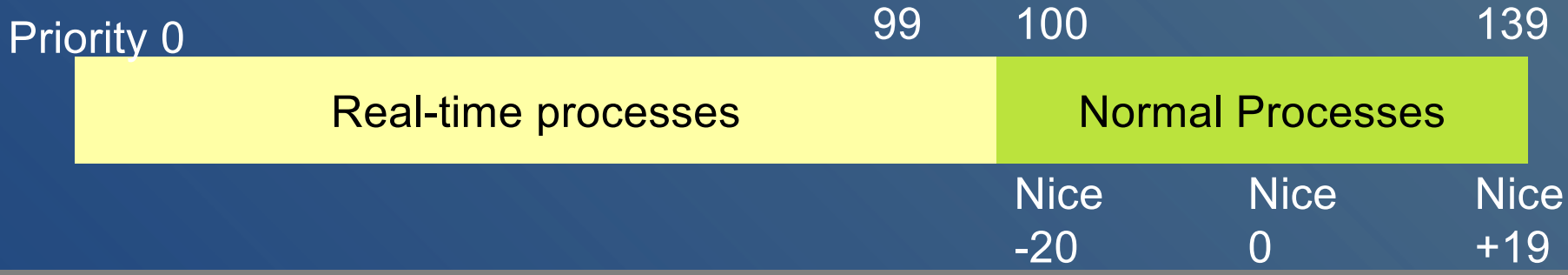
# Multilevel Feedback Queue Scheduling

- Multilevel Feedback Queue
  - Use multiple queues.
  - .. Move a process to lower priority if it takes too much CPU time.
  - Like Multilevel Queue, but processes lose priority via aging:
    Lower priority by moving to lower queue if process runs too long.



# Linux is Nice

- Linux categorizes processes into two classes
  - Real-time processes (priority values 0 to 99)
  - Normal processes (priority values 100 to 139)
- We can use nice value to..
  change/assign a priority for a normal process.
  - Nice values range from -20 to +19
  (lower nice == higher priority).
  - The default nice value is 0.
  - The nice -20 = priority 100, etc.



# Linux Completely Fair Scheduler (CFS)

- Longer running processes get a lower priority.
  - .. The longer it ran, the less chance it gets to run.
  - Older processes lose priority (aging)
- CFS tries to make sure that
  .. each process uses a similar amount of CPU time.
  - CFS uses virtual run time instead of physical (actual) run time
  - Virtual run time = physical run time + decay-formula
    - Higher decay with lower priority
    - i.e., the decay formula produces a bigger value for a lower priority
  - *Stored internally in a red-black tree based on virtual run time*

# Process Types

## Interactive vs. batch

- Interactive
  - Mainly user driven; regular desktop applications
- Batch
  - Program runs from start to end; no interaction needed.
    e.g., Compiling a program, Data analytics

## I/O bound vs. CPU bound

- I/O bound
  - More I/O than computation e.g., format change, such as CSV to XML
- CPU bound
  - More computation than I/O e.g., compression, cryptography, etc.
- (More: Memory bound, Communication bound, …)

# Summary

- Scheduler picks what job to run next.
- Algorithms:
  - First come, first serve
  - Shortest job first
  - Shortest remaining time first
  - Round Robin
  - Multilevel queue
  - Multilevel feedback queue
  - Completely Fair Schedule
- Drawing process scheduling diagrams
  - Compute Wait time, average wait time