# Scheduling

Adapted by Joseph Lunderville
from slides by Dr. Brian Fraser
and course material by Dr. Steve Ko

# Topics

- *Computers appear to do more things at the same time than there are CPUs*
  - How do multiple processes run “at once”?
  - How can multiple users log into a computer at once?
- *Sharing execution means taking turns*
  - How do we decide who to prioritize right now?

# 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
- *Many names for different kinds of things that run*
  - Jobs, processes, tasks, threads

# More Depth

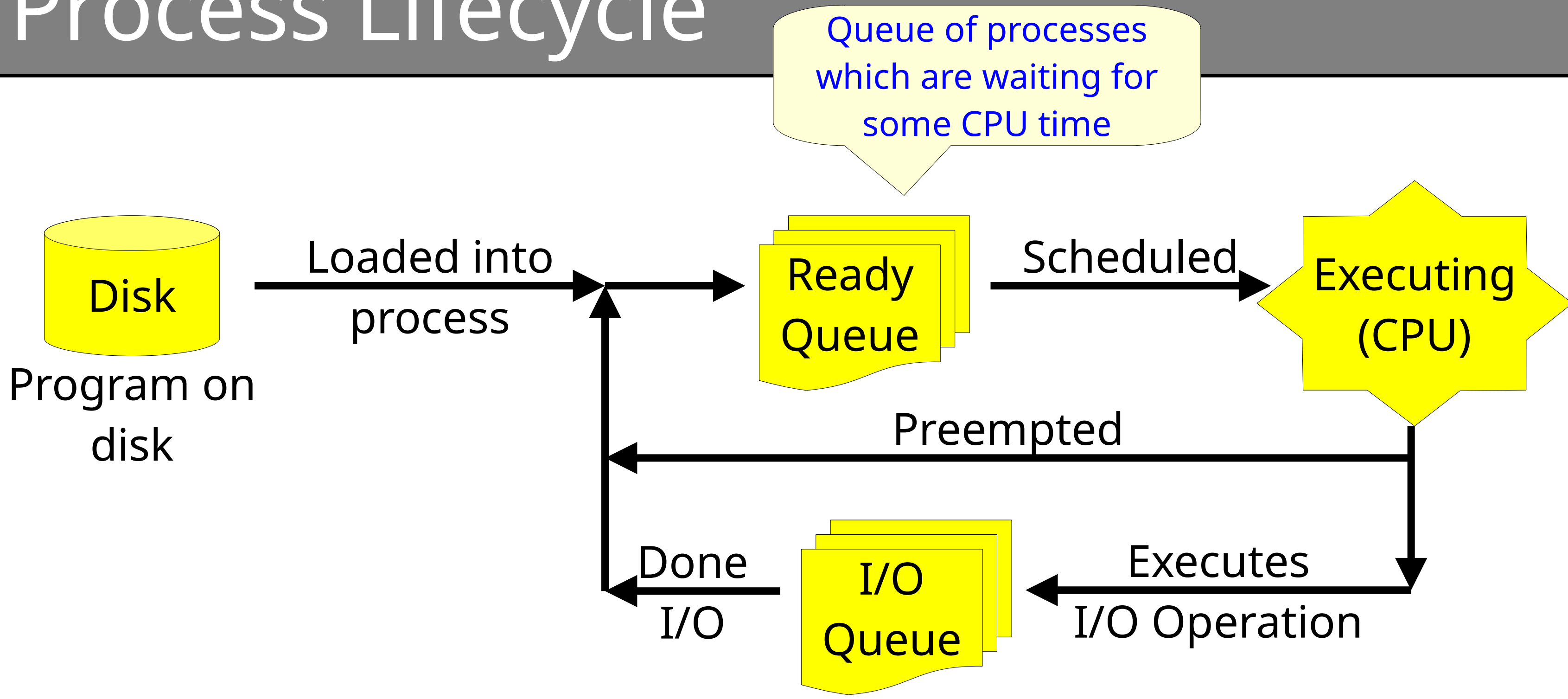
- *We will cover scheduling a little to understand the problem*
  - CMPT301 teaches it in depth
- *Can read more in OSTEP (has in-depth discussions beyond scope of this course)* https://pages.cs.wisc.edu/~remzi/OSTEP/
  - Chapter 7 Scheduling: Introduction https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched.pdf
  - 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 among multiple processes**
  - Or, sharing multiple cores among multiple processes (beyond the scope of this course)
- *Context switch*
  - **Stop running one process, and start running another process**
  - There is overhead (work) when the CPU does this switch, so don't do it too frequently – can we guess why?
  - 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
    - waits for an OS operation that takes an indefinite amount of time
      - e.g.: wait() for child, file or network I/O, thread synchronization
      - AKA “blocking”
    - yields voluntarily (sleep())
- *Preemptive scheduling*
  - The kernel stops a *process* at any time
  - The kernel itself (e.g., syscalls, scheduler) might not be preemptible
- *Preemptible kernel*
  - (almost) all of the kernel can be preempted!
  - Necessary (but not sufficient) for “real-time” operation

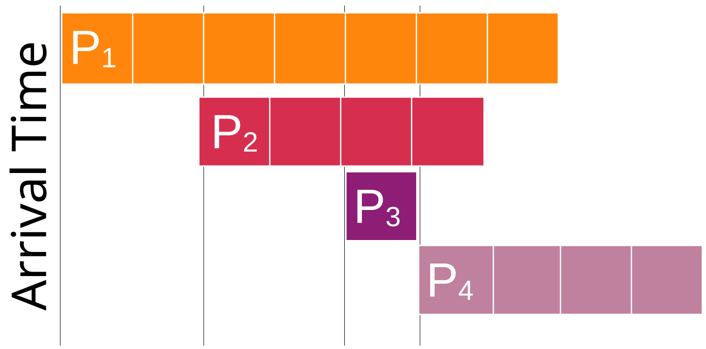
# Scheduling Goals

- *We want to **maximize***
  - **CPU utilization:** keep the CPU as busy as possible
  - **Throughput:** units of work, i.e. number of processes, completed per unit time
- *We want to **minimize***
  - **Turnaround time:** time taken to execute a particular process (from *submission* to *termination*)
  - **Wait time:** 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
- These are not orthogonal! They overlap

# Scheduling Algorithms

# Simplifying Assumptions

- 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
  - Preemptive algorithms generally don’t use this information
- Often processes are long lived, but only need the CPU in short bursts
  - We'll just look at one burst of activity from each process, during one short time interval
  - In reality this kind of scheduling would recur at irregular intervals



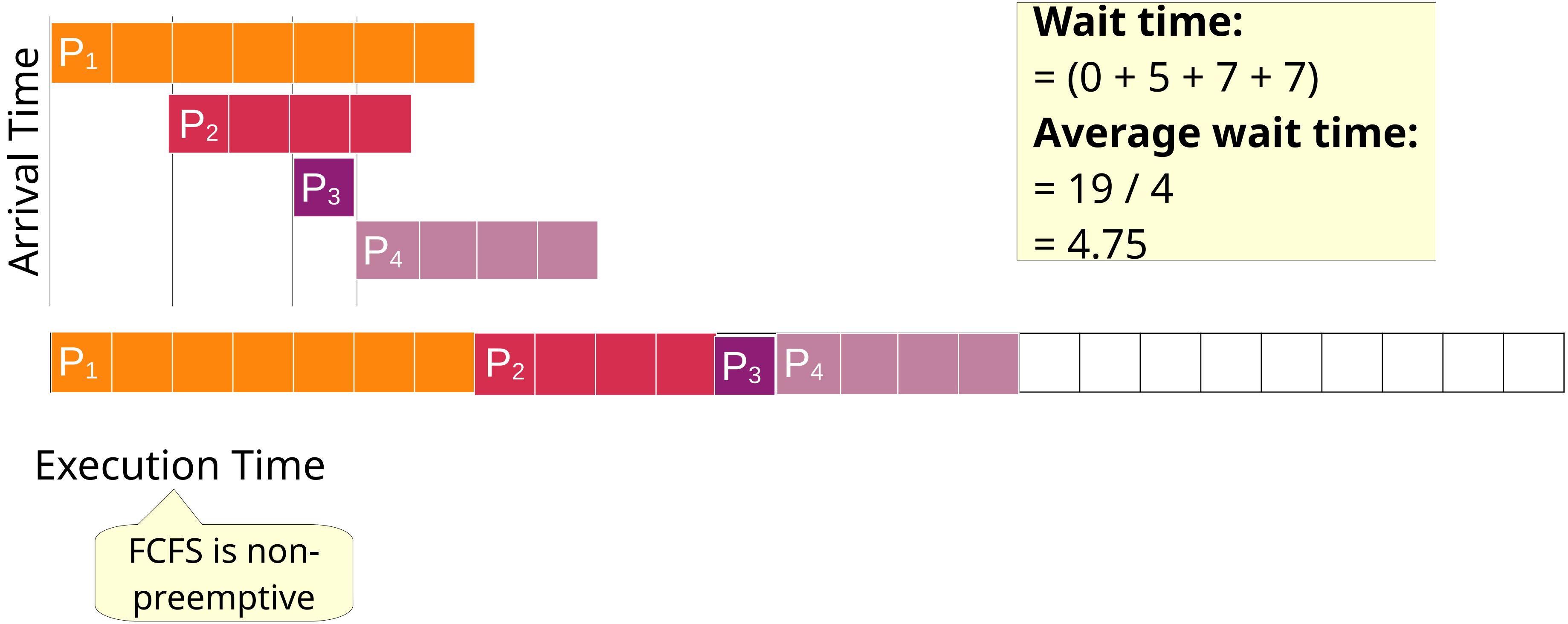
# First Come, First Served (FCFS)

- **First Come, First Served:**
  - ***Run in the order of arrival***
  - Simplest functioning algorithm
  - 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 a scheduling algorithm is

There are other metrics are based on scheduling goals above, but for now waiting time is easy to calculate

# First Come, First Served 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

# Audience Participation - FCFS

- What is the total wait time for the following processes using FCFS?

| | 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

# Audience Participation – FCFS 2

- What is the total wait time for the following processes using FCFS?

| | 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***
  - Non-preemptive:
    - Once running, a job runs to completion

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

# Shortest Job First Example

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

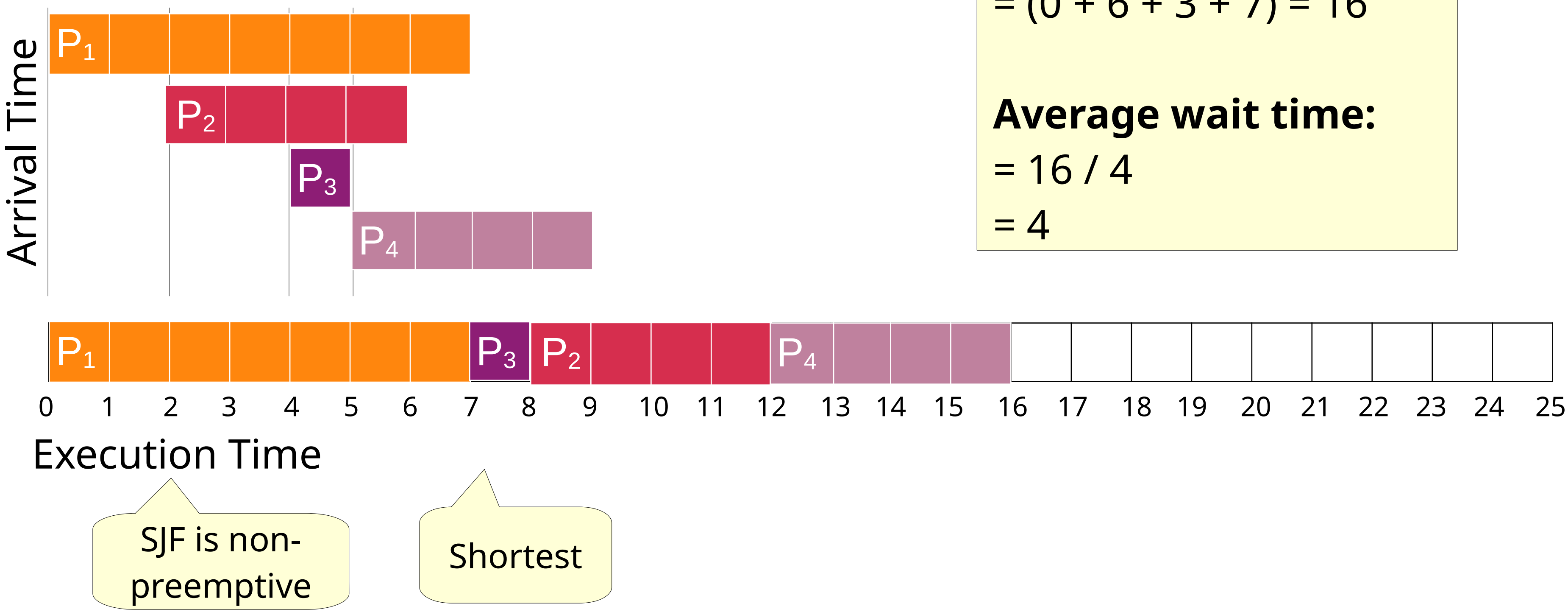
**Total Wait Time:**

= (0 + 6 + 3 + 7) = 16

**Average wait time:**

= 16 / 4

= 4



# Shortest Remaining Time First (SRTF)

- **Shortest Remaining Time First Scheduling Algorithm:**
  - ***Schedule the process with the shortest remaining execution time***
  - This is **preemptive:** when a new job arrives, *it can interrupt a 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**

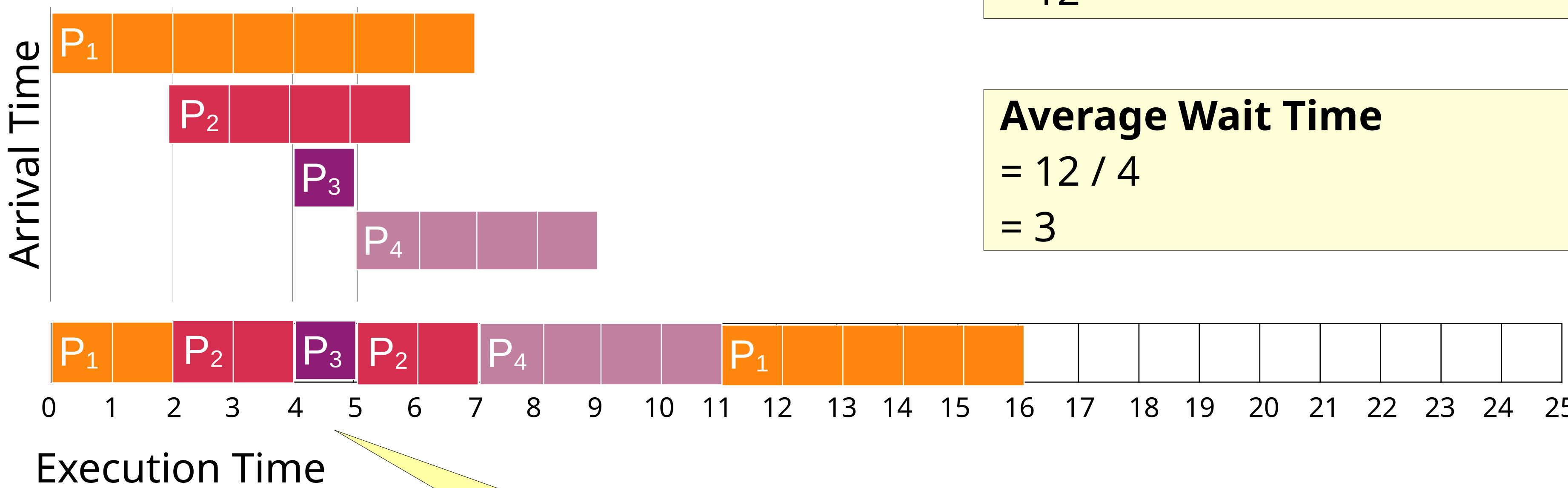
| **P1** | **P2** | **P3** | **P4** |
|---|---|---|---|
| 0+9 | 0+1 | 0 | 2 |

= 12

**Average Wait Time**

= 12 / 4

= 3




SRTF is preemptive

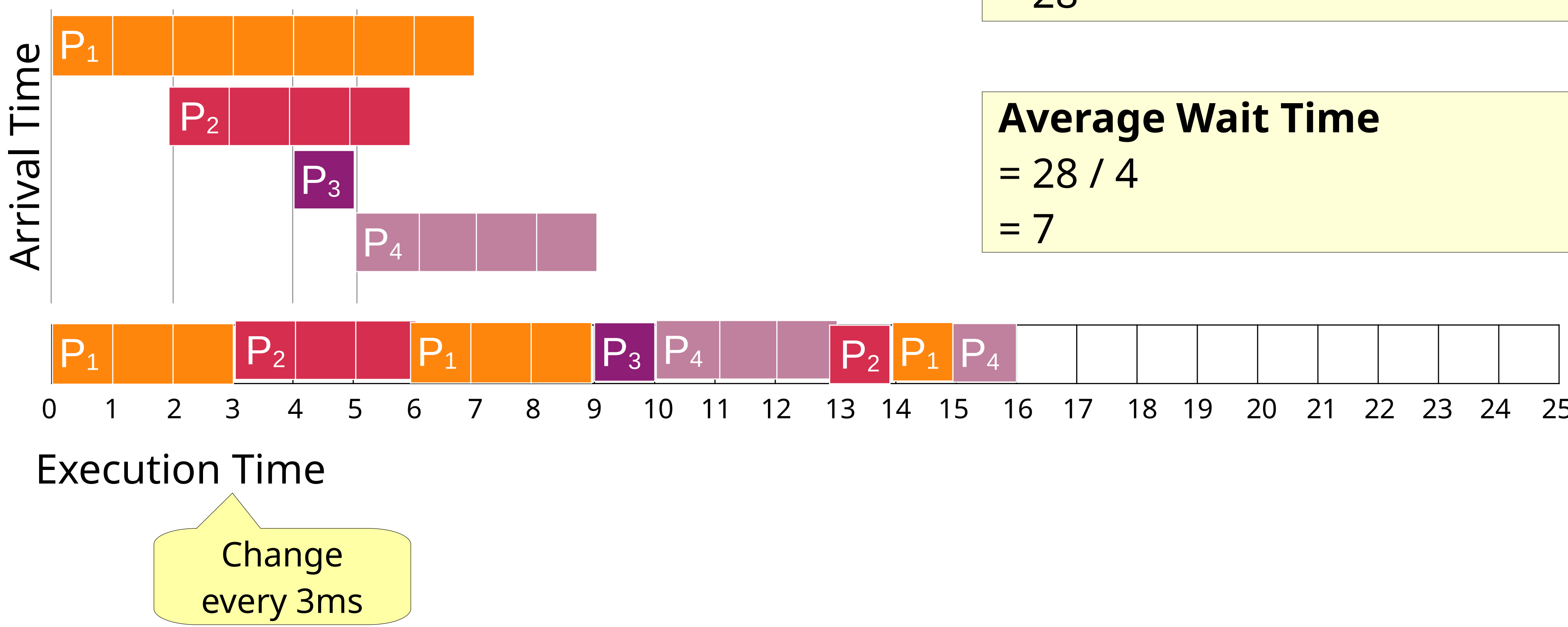
Always pick shortest remaining time

# Round Robin (RR)

- **Round Robin Scheduling Algorithm:**
  - Forget about knowing how long things take
  - Just give everyone **equal length *turns***
- *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 (tail)* of the *ready queue*
    - Take *first* process from the *front (head)* of the *ready queue* and run it
  - *Newly arrived* processes inserted at the *back* of the *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

# Audience Participation - 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 (usually short) time period.*

# Real-Time Deadlines

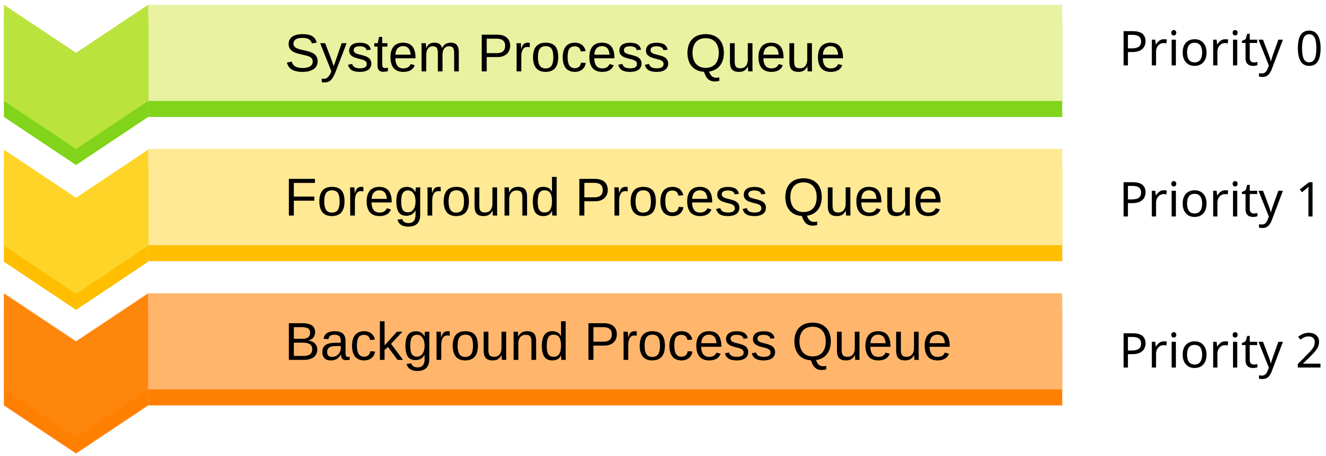
- **Hard real-time systems:**
  - ***Strict deadlines*** which cause system failure if missed
  - Failure may have real-world consequences (destruction, injury)
    - Control systems: car ECU, braking, power supplies
- **Firm real-time systems:**
  - ***Strict deadlines*** but system can tolerate some amount of misses
    - Media, telephony: dropped frames are okay, but not too many
- **Soft real-time systems:**
  - ***Approximate deadlines*** where late completion reduces value
    - Most interactive systems: lag sucks
    - At a longer timescale, reporting/prediction e.g. weather forecast
- Real-time tasks usually have higher priorities (should run first)
  - Beware priority inversion

# Priority Scheduling (cont'd)

- **Task priority** is typically expressed as a number (where a *smaller* number has a *higher* priority).
- *Problem:* ***Starvation***
  - Lower priority processes may never run (how?)
  - E.g, if high priority processes keep arriving…

# Multilevel Queue Scheduling

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

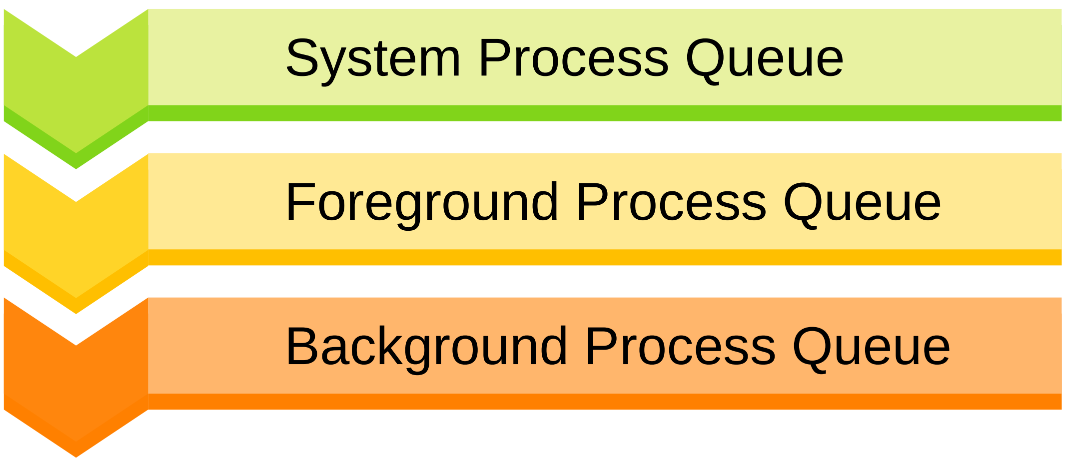


# Multilevel Queue Scheduling

- *Each queue gets CPU time based on priority*
  - (One idea) **Weighted Round Robin:** 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.

- *During each **queue's** turn*
  - Scheduling algorithm (chosen per queue) picks which process *in that queue* to run
- *Avoids queue starvation*
  - Each *queue* gets a chance to run



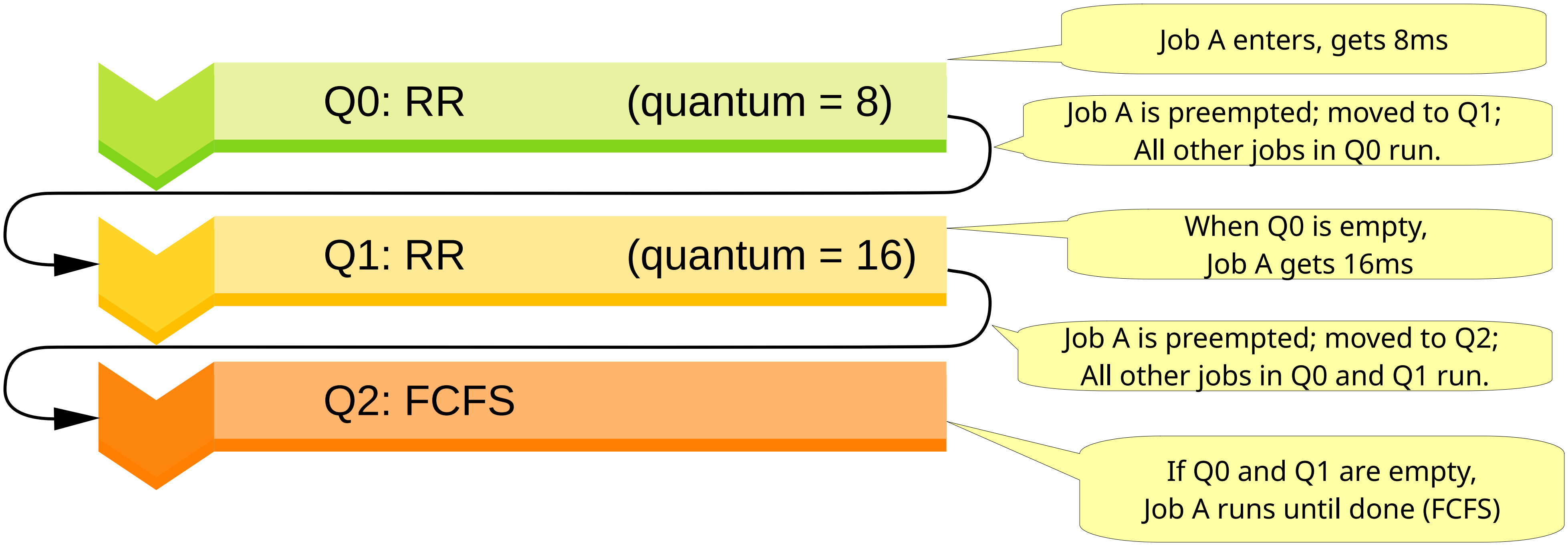

Priority 0
RR

Priority 1
FCFS

Priority 2
SJF

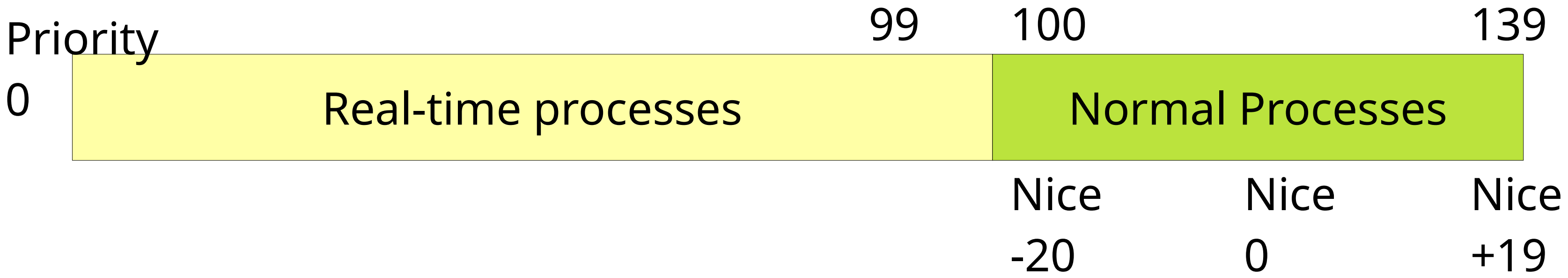
# Multilevel Feedback Queue Scheduling

- *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)
- *Nice value assigns a priority for a normal process*
  - Nice values range from -20 to +19
    (*lower* nice == *higher* priority – greedier)
  - The default nice value is 0
  - 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 ensure 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., “decay formula” is bigger for a lower priority
  - Stored internally in a balanced 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.

# Summary

- **Scheduler** picks what job to run next.
- *Algorithms*
  - First Come, First Served
  - 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