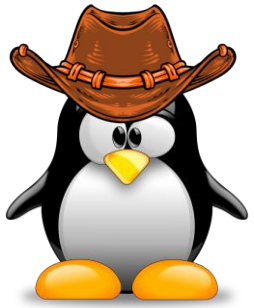
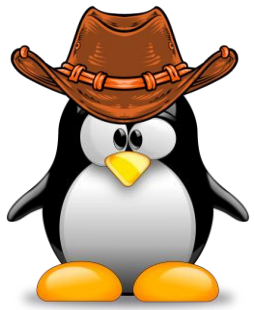


CSE 330: Operating Systems

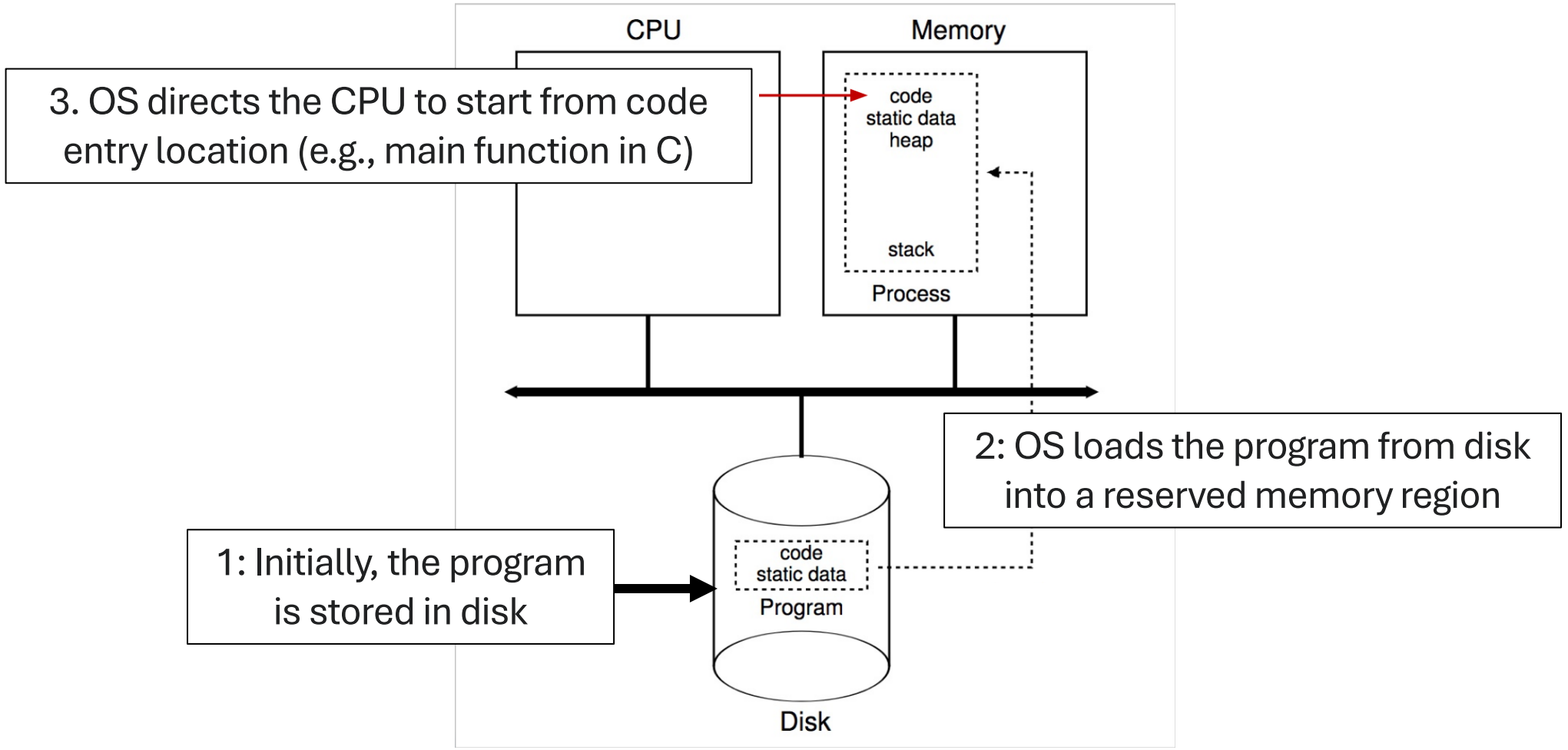
Adil Ahmad

Lecture #5: Scheduling concepts and algorithms



Recap of last week’s lecture

An illustration of typical process creation

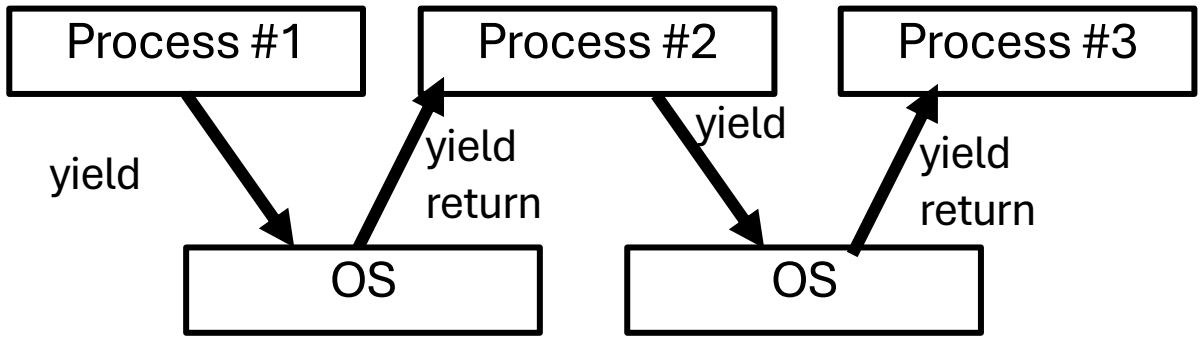


Process creation from parents

- During system boot, the kernel creates a “grandparent” process called “init” for all user-space (unprivileged) processes
- This process creates every other user-space process (e.g., your Linux terminal, etc.)
- Information about parent → child relationships is stored by Linux kernel using a tree structure
- Two system calls: **fork** and **exec**

Cooperative scheduling through a system call

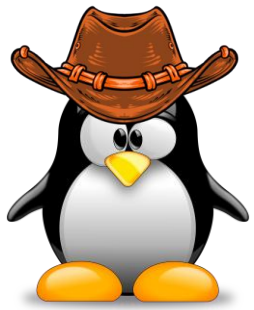
- yield() system call



- Super-simple scheduling approach and it is common on tiny “embedded” systems today

Preemptive scheduling in general-purpose OSs

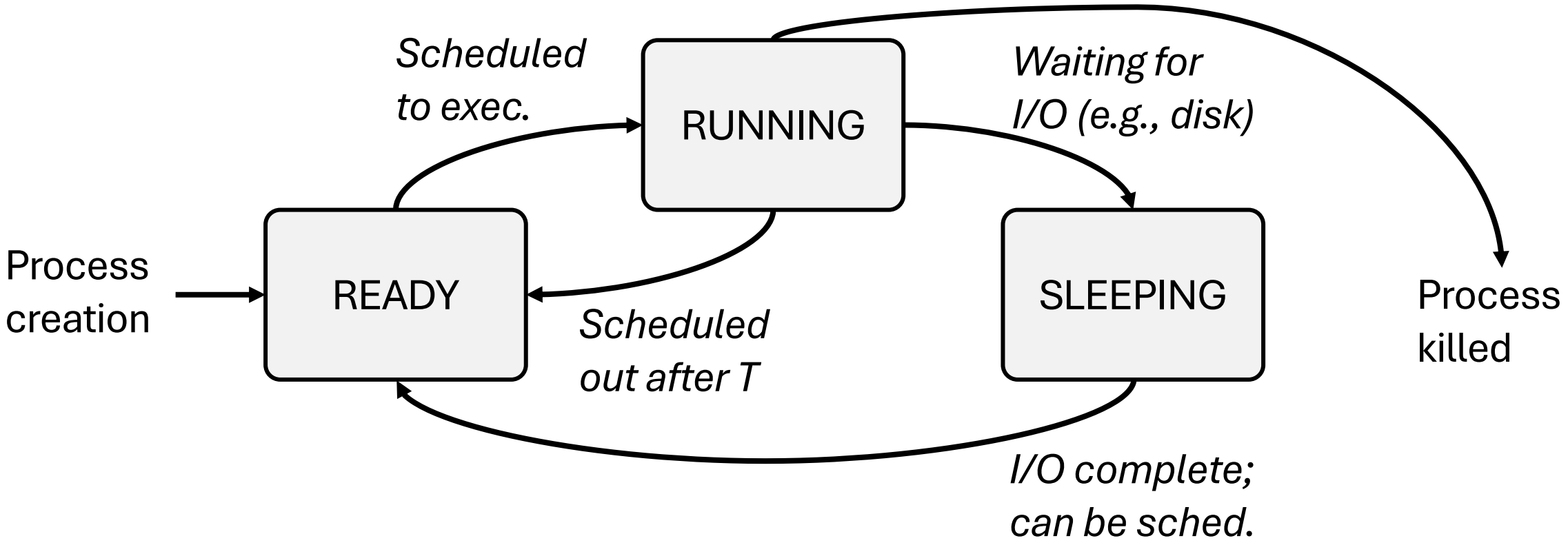
- OSs programs a periodic ‘timer’ interrupt that is fired and forces a switch to the OS’ context
- OS at the interrupt decides which process should then be resumed from all the processes that are RUNNABLE



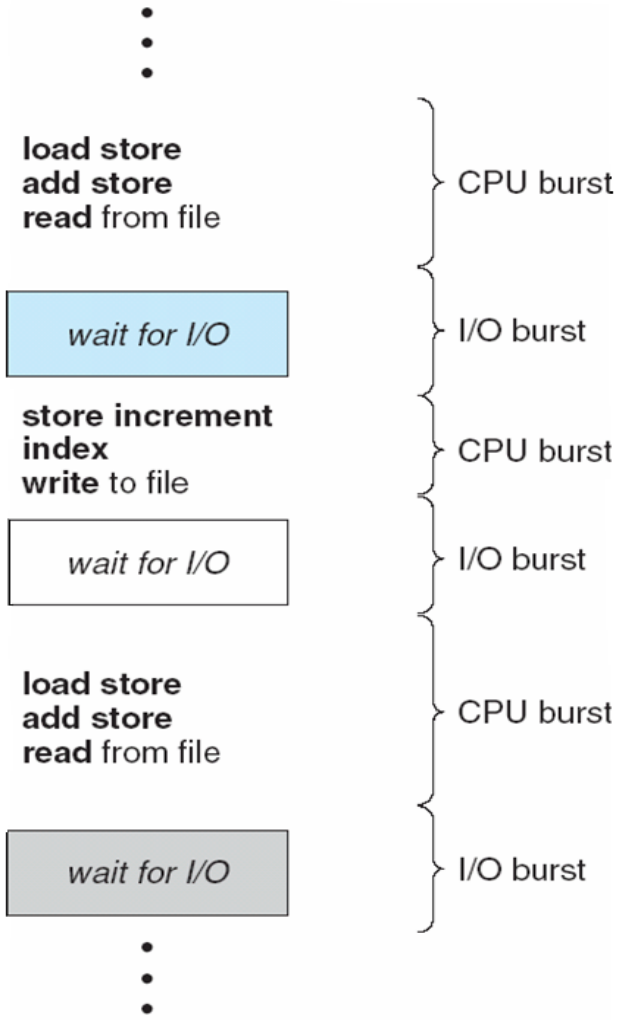
Scheduling concepts and criteria

Once created, a process can have different states

Three main states of a process during its execution



RUNNING process (job) has two kinds of “bursts”



CPU burst → computing results in memory

I/O burst → computing results on data in storage

Four important criteria that OS considers for scheduling

- Turnaround time:
  - amount of time to execute a particular process
- Waiting time:
  - amount of time a process has been waiting in the ready queue
- Response time:
  - amount of time it takes from when a request was submitted until the first response is produced, not the complete output
- Deadlines
  - Certain task must not be delayed more than “X” time

Understanding *waiting* and *turnaround* time

Waiting time:

- For one process: $T_{waiting} = T_{start} - T_{arrival}$
- Average over all procs: $Sum(T_{waiting}) / \#\ of\ processes$

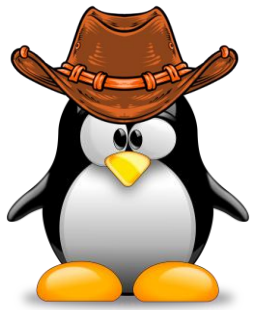
Turnaround time:

- For one process: $T_{turnaround} = T_{complete} - T_{arrival}$
- Average over all procs: $Sum(T_{turnaround}) / \#\ of\ processes$

Different scheduling algorithms we will talk about

- First-In-First-Out (FIFO)
- Shortest Job First (SJF)
- Shortest Remaining Time First (SRTF)

Let’s see how waiting and turnaround times are affected!



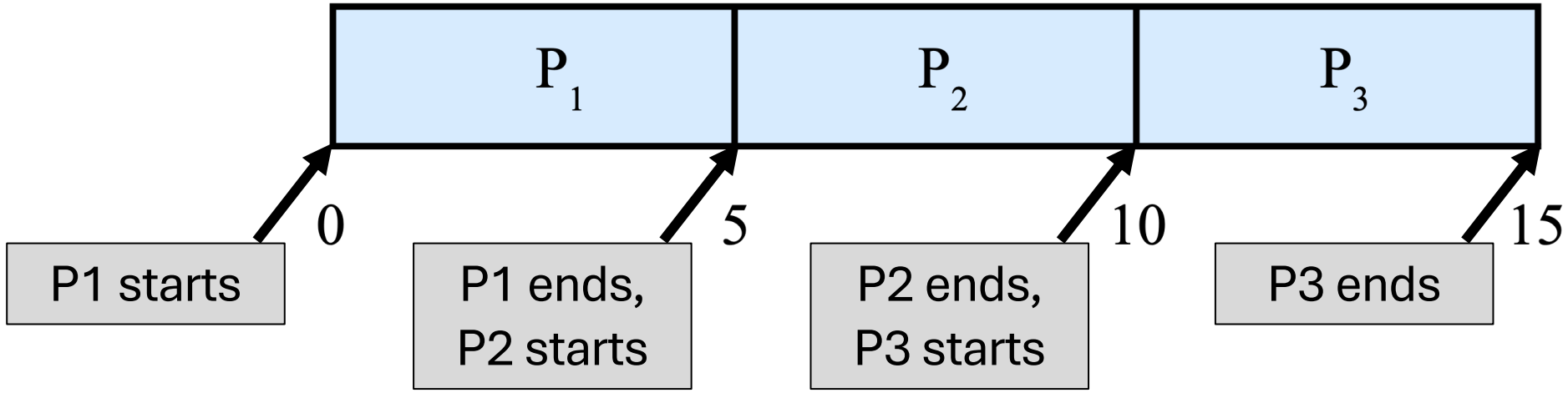
First-In-First-Out (FIFO)

Start with simple assumptions

- Each job runs for the same amount of time
- All jobs arrive at the same time
- All jobs only use the CPU (no I/O)
- The run-time of each job is known

FIFO with simple assumptions

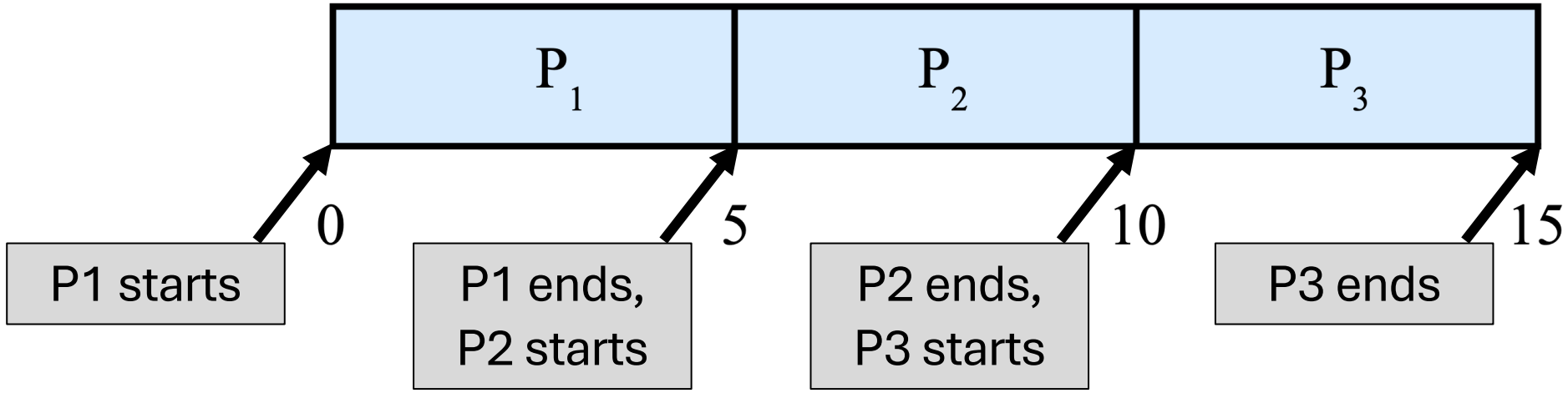
- Three processes: P1, P2, P3 (5 burst time each)
- Suppose processes arrive in order: P1 → P2 → P3 (Gannt chart)



- Waiting time: P1= 0, P2 = 5, P3 = 10
- Average waiting time: (0+5+10)/3 = 5

Now, it's your turn

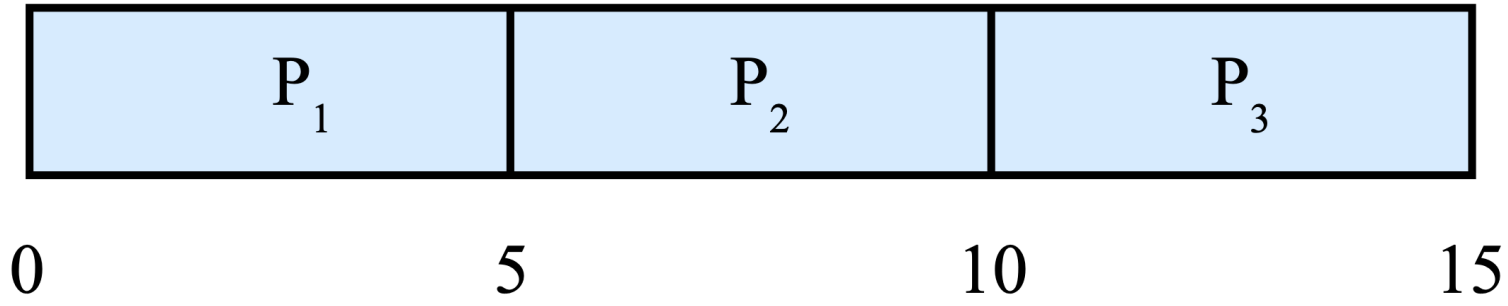
- Three processes: P1, P2, P3 (5 burst time each)
- Suppose processes arrive in order: P1 → P2 → P3 (Gannt chart)



- **What is the average turnaround time?**
  - (5 + 10 + 15)/3 = 10

Now, it's your turn

- Three processes: P1, P2, P3 (5 burst time each)
- Suppose processes arrive in order: P1 → P2 → P3 (Gannt chart)



- **What is the average turnaround time?**
  - (5 + 10 + 15)/3 = 10

Let’s negate one assumption to make things interesting

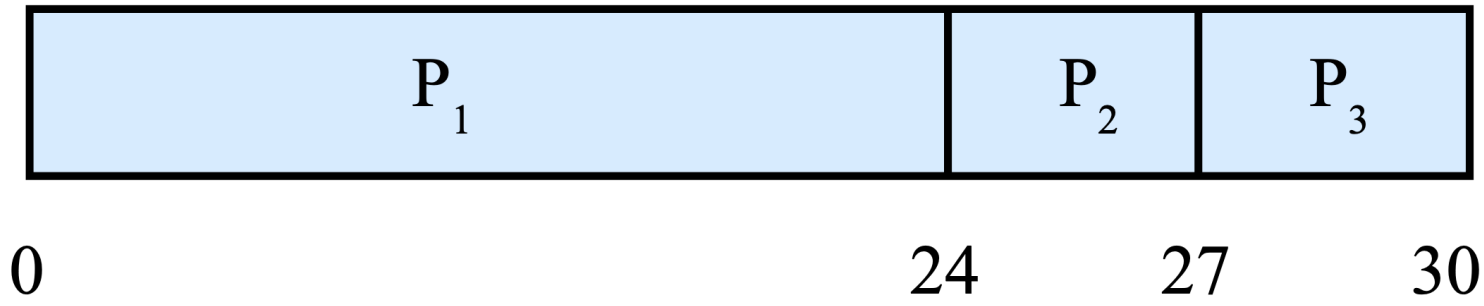
- ~~Each job runs for the same amount of time~~
- All jobs arrive at the same time
- All jobs only use the CPU (no I/O)
- The run-time of each job is known

FIFO example (version 2)

- Three processes in this order

Process	P1	P2	P3
Burst	24	3	3

- Suppose processes arrive in order: P1 → P2 → P3 (Gannt chart)



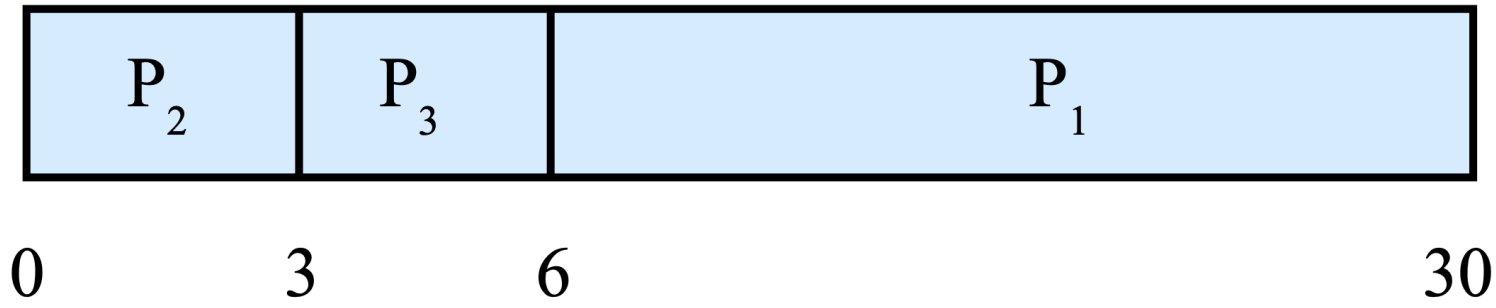
- **What is the waiting time and average?**
  - P1= 0, P2 = 24, P3 = 27
  - Average: (0+24+27)/3 = 17

FIFO example (version 3)

- Three processes in this order

Process	P1	P2	P3
Burst	24	3	3

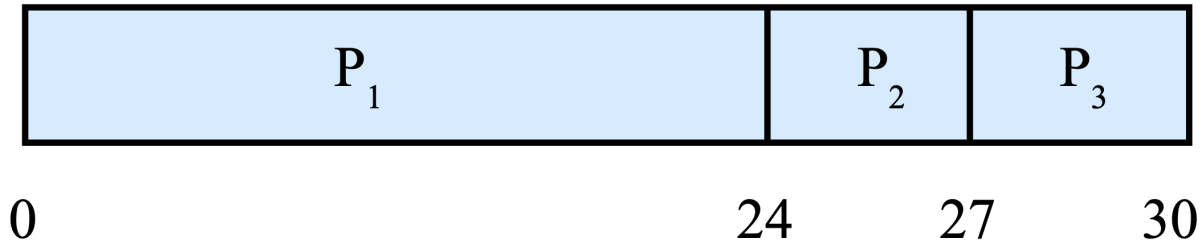
- Suppose processes arrive in order: P2 → P3 → P1 (Gannt chart)



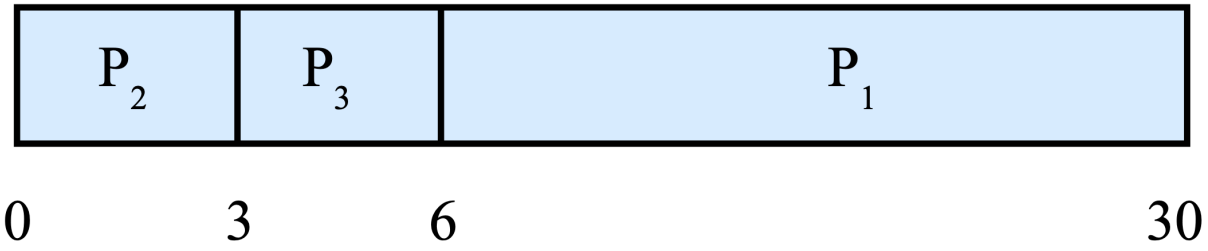
- **What is the waiting time and average?**
  - P1= 6, P2 = 0, P3 = 3 (Average = 3)

What exactly changed between the two scenarios?

- **Scenario #1:** P1= 0, P2 = 24, P3 = 27 (Average = 17)



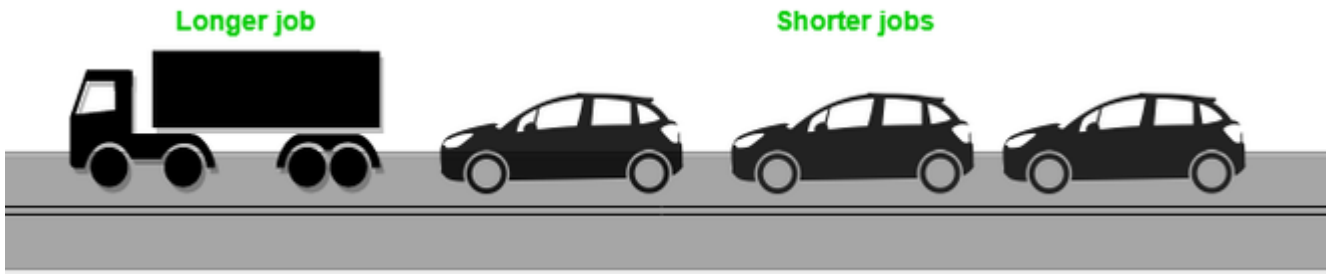
- **Scenario #2:** P1= 6, P2 = 0, P3 = 3 (Average = 3)



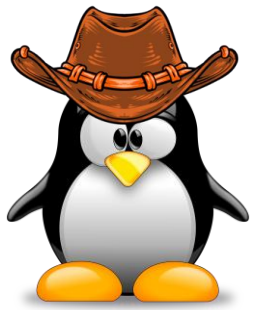
  - The shorter job(s) was scheduled earlier than the longer jobs

This brings us to the **convoy effect**

- Short-running processes are lagging (behind) long-running processes



- **Why do you think this is a problem intuitively?**
  - Shorter jobs are typically “latency-sensitive” (e.g., opening a browser) while longer-running processes are typically not (e.g., downloading a 1GB file)



Shortest Job First (SJF)

Shortest-Job-First introduction

- Associate with each process the length of its CPU burst
- CPU is assigned to the process with the smallest CPU burst (i.e., shortest job)
- The process is allowed to execute until it completes
- SJF is optimal (i.e., the best possible) to reducing average waiting time when all jobs arrive at the same time

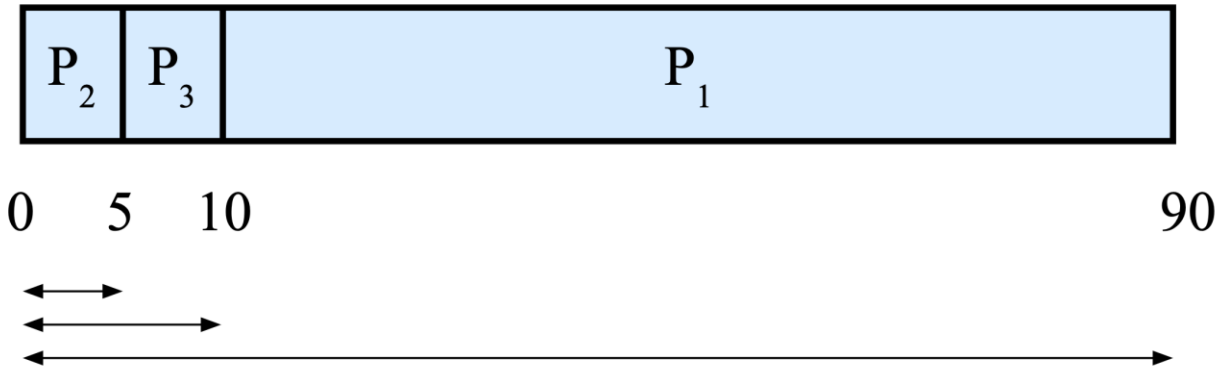
Understanding the term **optimal** in formal usage

- Optimal for something does not mean "*it's better than some alternatives*"
- Optimal for something means that "*we can formally prove it's the best that we can achieve under a given set of assumptions/scenarios*"
- If you're interested in looking at the optimality proof for SJF under the assumptions stated in slide #2, please check
  - https://courses.engr.illinois.edu/cs374/sp2017/slides/19-greedy.pdf

SJF example (version 1)

- Three processes:

Process	P1	P2	P3
Burst	80	5	5
Arrival	0	0	0

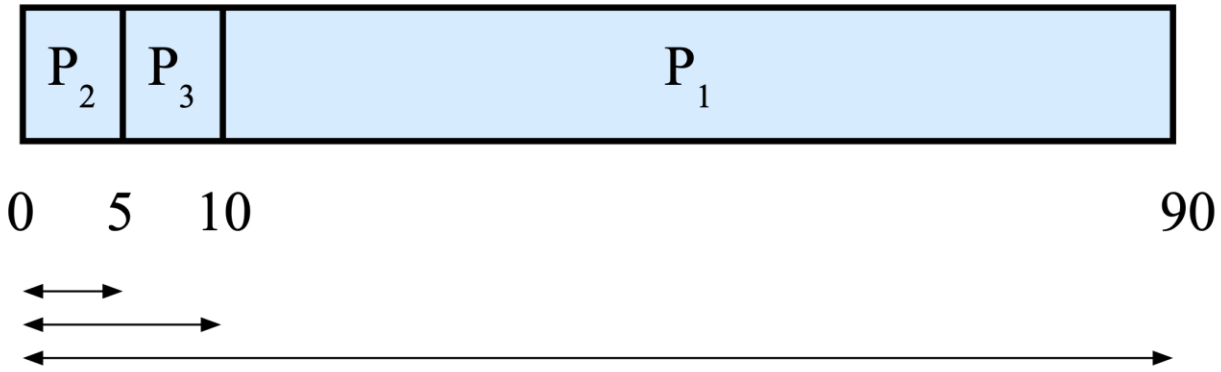


- **What is the average waiting time?**
  - (5 + 10 + 0)/3 = 5

SJF example (version 1)

- Three processes:

Process	P1	P2	P3
Burst	80	5	5
Arrival	0	0	0



- **What is the average turnaround time?**
  - (5 + 10 + 90)/3 = 35

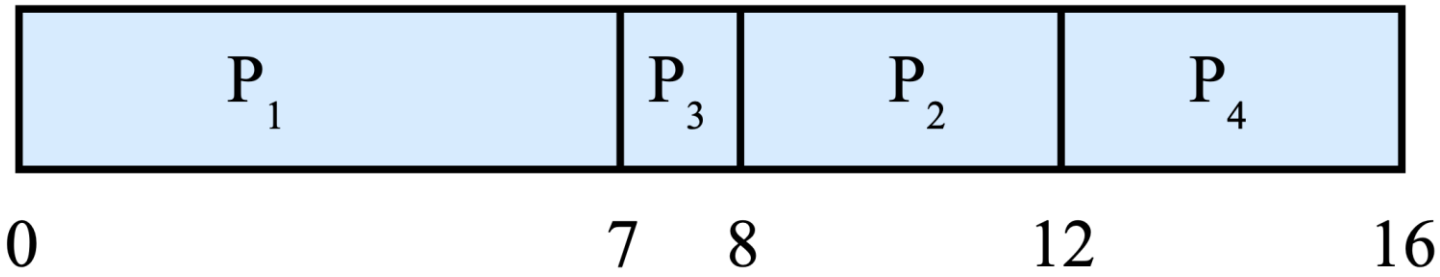
Negate the same arrival time assumption!

- ~~Each job runs for the same amount of time~~
- ~~All jobs arrive at the same time~~
- All jobs only use the CPU (no I/O)
- The run-time of each job is known

SJF example (version 2)

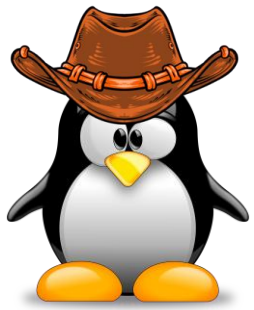
- Four processes:

Process	P1	P2	P3	P4
Burst	7	4	1	4
Arrival	0	2	4	5



- **What is the waiting time and average?**
  - P1 = 0;
  - P2 → 8 – 2 = 6;
  - P3 → 7 – 4 = 3;
  - P4 → 12 – 5 = 7
  - (0 + 6 + 3 + 7) / 4 = 4

- The **optimal** average waiting time in this example is much lower



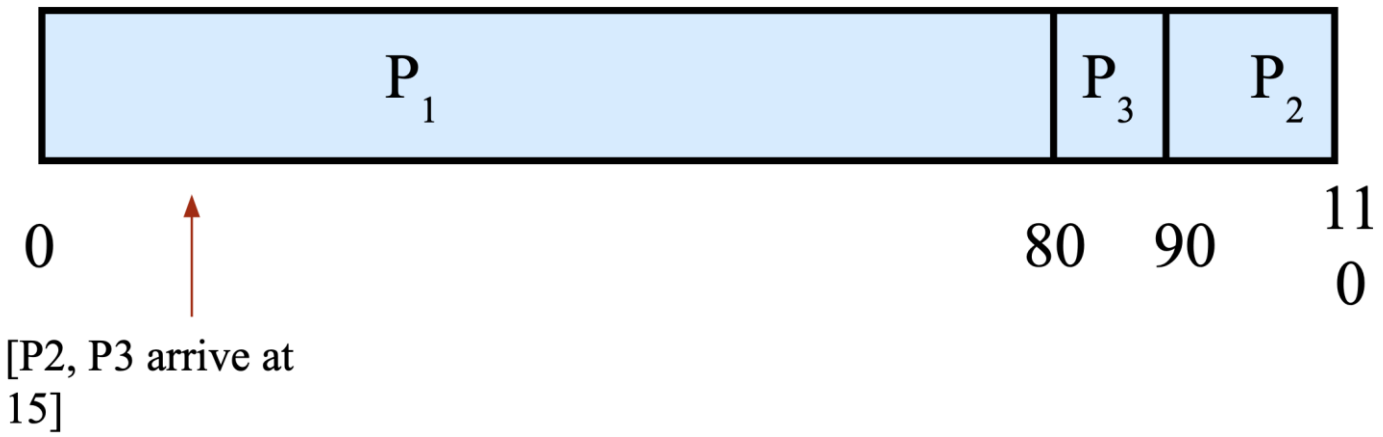
Shortest Remaining Time First (SRTF)

SRTF: Adding the concept of *preemption* to SJF

- General idea of SRTF:
  - ➢ Keep track of the **remaining time** of each process
  - ➢ **Switch jobs** to complete the one that will complete fastest
    - i.e., schedule the job with the shortest remaining time
- Also called "**Preemptive** Shortest-Job-First"
- Unlike previous schemes (**non-preemptive FIFO and SJF**), a process will not always run to completion in this scheme

Let's revise with an example of *(non-preemptive)* SJF first

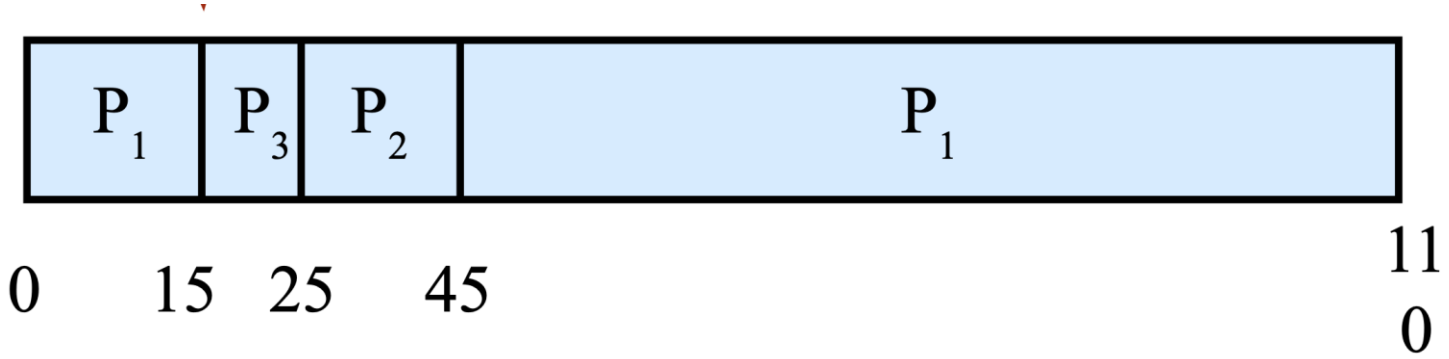
Process	P1	P2	P3
Burst	80	20	10
Arrival	0	~15	~15



- **Average waiting time:** (0 + 80 + 90) / 3 = ~57
- **Average turnaround time:** (80+75+105) / 3 = ~87

Let’s look at the same example with SRTF

Process	P1	P2	P3
Burst	80	20	10
Arrival	0	~15	~15



- **Average waiting time:** (30 + 10 + 0) / 3 = ~13.3
- **Average turnaround time:** (110 + 30 +10) / 3 = ~50

On the combined optimality of SJF and SRTF

- SJF is “optimal” (i.e., the best possible) to reducing waiting time when all jobs arrive at the same time

- SRTF combines SJF with the concept of preemption to become optimal when jobs do not arrive at the same time

What are some potential problems with SJF and SRTF?

- Starvation can practically happen
  - If shorter jobs keep arriving, a long-running job would simply not be scheduled
  - We say that the long-running job is “starved”

- Requires knowledge of each job’s “burst or execution time”
  - It’s not always easy to precisely tell how long a job will execute

Let’s tackle this one first!

Let’s take away another assumption!

- ~~Each job runs for the same amount of time~~
- ~~All jobs arrive at the same time~~
- ~~The run-time of each job is known~~
- All jobs only use the CPU (no I/O)

Estimating the length of jobs

- *Idea:* Based on the observations in the past, we can try to predict

- **Simple average** based on combining the observations in the past
  - t_i: actual length of the i^{th} CPU burst
  - $t_{n+1} = \frac{1}{n} * \sum_{i=1}^{n}(t_i)$

- **Potential problem with a simple average?**
  - Gives every old process the same weight and does not consider recent workloads

Estimating the length of “recent” jobs

- *Idea:* Based on the observations in the *recent* past, we can try to predict
- **Exponential averaging** reduces the weight of older in time measurements
- Exponential averaging:
  - t_i: actual length of the i^{th} CPU burst
  - $t_{n+1} = \sum_{i=0}^{n-1} (1-a)^i a t_{n-i}$
  - (a) is a smoothing (or aging) value

What are some potential problems with SJF and SRTF?

Let's tackle this one now!

- Starvation can practically happen
  - If shorter jobs keep arriving, a long-running job would simply not be scheduled
  - We say that the long-running job is "starved"

- Requires knowledge of each job's "burst or execution time"
  - It's not always easy to precisely tell how long a job will execute

Bring back the “response time”

Response time:

- For one process: $T_{response} = T_{first\_run} - T_{arrival}$
- Average over all procs: $Sum(T_{response})$ / # of processes

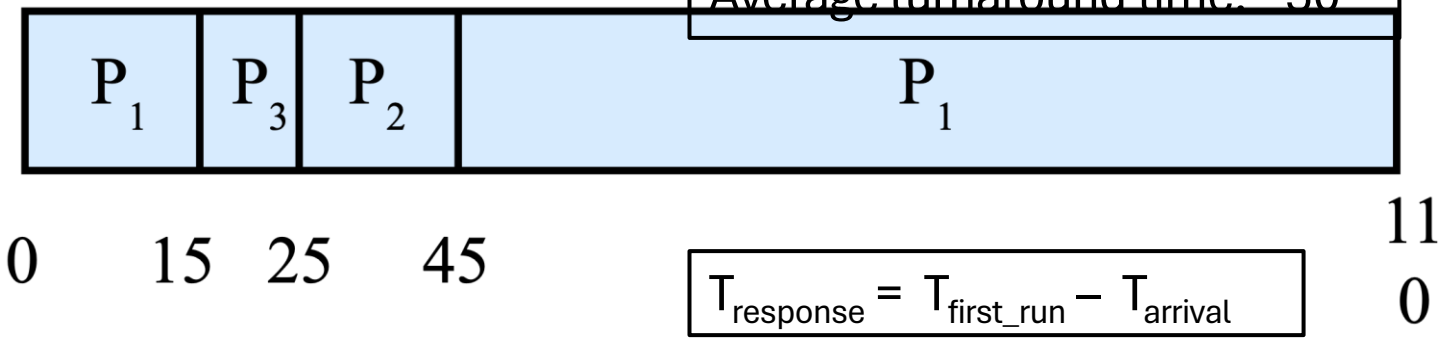
- If we want to reduce/mitigate starvation, it is important to consider *response time* as a criteria, in addition to *waiting* and *turnaround times*

Revisit a prior SJF example with response time

Process	P1	P2	P3
Burst	80	20	10
Arrival	0	~15	~15

Average waiting time: 13.3

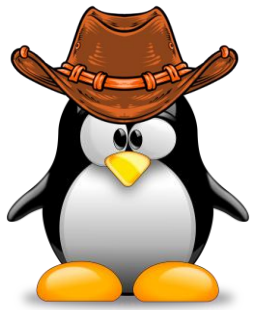
Average turnaround time: 50



$T_{response} = T_{first\_run} - T_{arrival}$

- **What is the response times of P1,P2,P3 and average?**
  - P1 → 0 – 0 = 0
  - P2 → 25 – 15 = 10
  - P3 → 15 – 0 = 0
  - Average = (0 + 10 + 0)/3 = 3.33

Very high for some jobs!



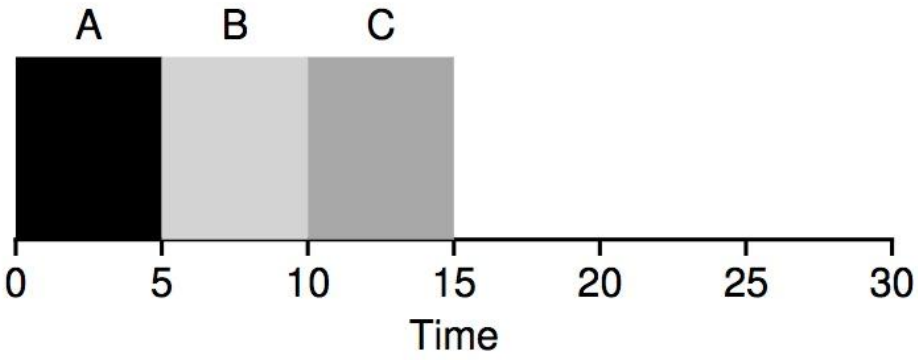
Round Robin (RR)

Round Robin (RR) introduction

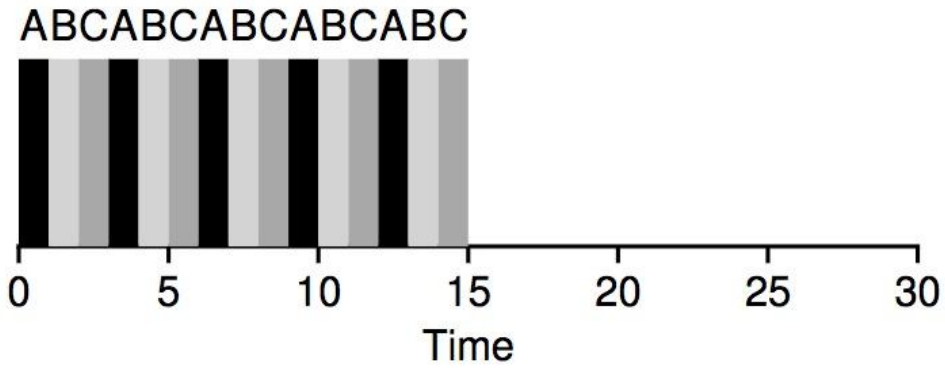
- Unlike prior algorithms, each process gets a small unit of CPU time (quantum)
- After this time is elapsed, the process is preempted (*recall the timer?*) and sent back to the READY queue
- Every time a process is added to the READY queue, it is added to the end. This applies to:
  - Newly-created processes
  - Processes that have completed their quantum
- For *n* processes and time quantum *q*, maximum response time is (n-1)/q

Take an example for RR vs SJF

- 3 processes with 5 burst time
  - SJF's average response time
    - (0 + 5 + 10)/3 = 5



  - RR with time quantum (q = 1)
    - (0 +1 + 2)/3 = 1

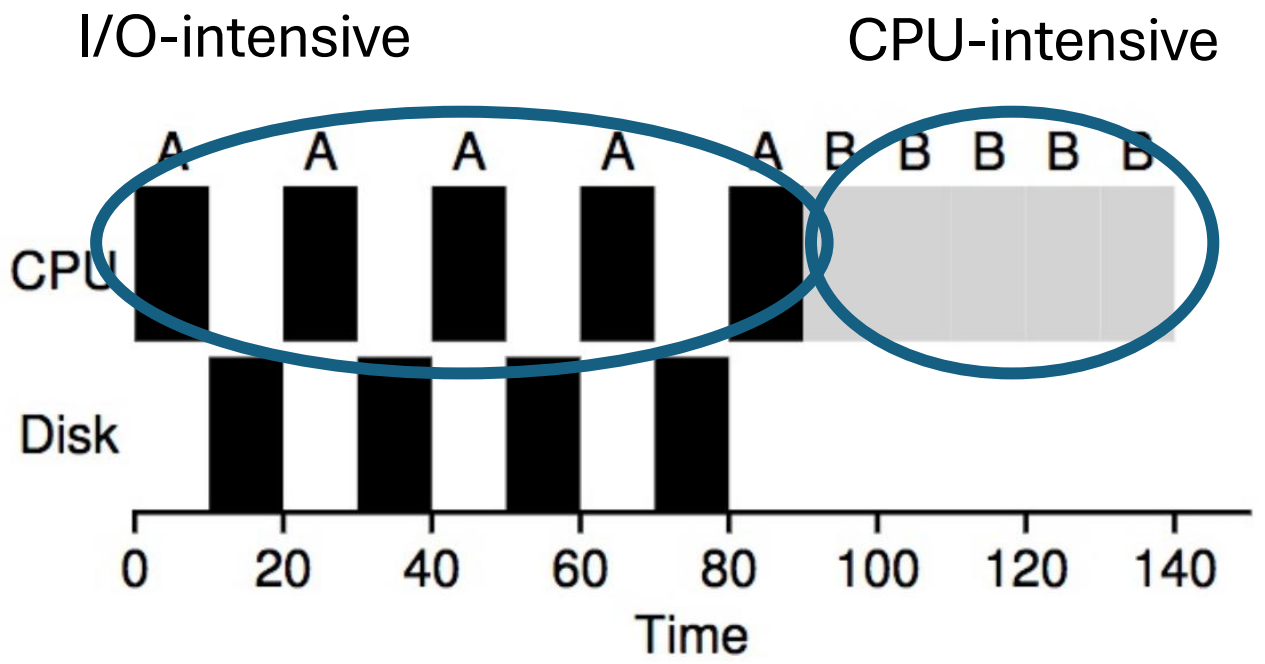


Let’s take away the final assumption!

- ~~Each job runs for the same amount of time~~
- ~~All jobs arrive at the same time~~
- ~~The run-time of each job is known~~
- ~~All jobs only use the CPU (no I/O)~~

Is there any complication with considering I/O and Round-Robin?

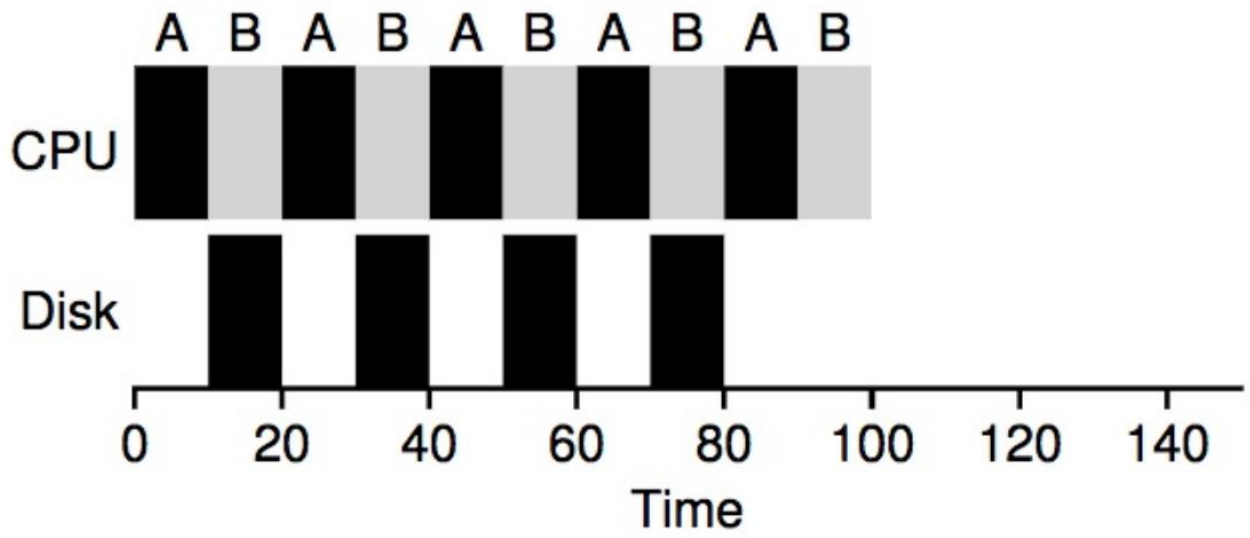
- Poor usage of resources remaining agnostic to I/O-intensive behavior



- Naïve RR gives a full time-quantum to “A” which just wastes CPU cycles

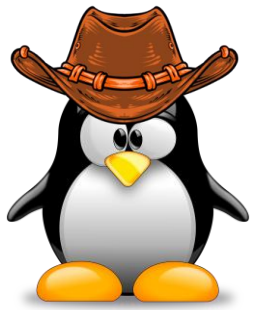
I/O-aware round robin helps use resources better!

- Keep 2 queues: one each for CPU-intensive and I/O-intensive
- *Overlap* the execution of CPU-intensive and I/O-intensive processes



Did we make any other important (implicit) assumption?

- ~~Each job runs for the same amount of time~~
- ~~All jobs arrive at the same time~~
- ~~The run-time of each job is known~~
- ~~All jobs only use the CPU (no I/O)~~
- All jobs have the same “priority”



Priority scheduling

Priority scheduling introduction

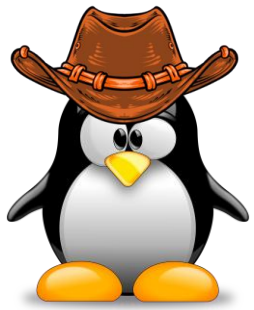
- Not all jobs/processes are created equal, some deserve to be handled before others
- Each process is assigned a “priority number” by the OS/user
  - Linux/MacOS use -20 to 20
  - -20 is highest priority (reserved for short, very critical kernel jobs)
  - 20 is lowest priority
- If processes have the same priority, they are scheduled in a “round-robin” fashion (check previous slides)

What's the obvious problem with priority scheduling?

- Like SJF, it can also lead to starvation
  - The system gets a continuous stream of very high number of high-priority jobs
  - Low-priority jobs essentially can never be served
- **How would you address this problem?**

Multi-level priority queuing with feedback

- General idea is as follows:
  - Keep a round-robin queue for each level of priority (e.g., -20 to 20) in the system
  - *Feedback:* Check if a job has remained in a lower priority queue for a configured time interval (e.g., 10s), and has not been served
  - Based on feedback, move the job to higher priority queue so that it may eventually be served (based on the RR implementation within a priority queue)
- This is basically the default implementation used in Linux today



Questions? Otherwise, see you next time!