CS 370: OPERATING SYSTEMS
[CPU SCHEDULING]

Computer Science
Colorado State University

Instructor: Louis-Noel Pouchet
Spring 2024

** Lecture slides created by: SHRIDEEP PALLICKARA

Topics covered in this lecture

- CPU Scheduling
- Scheduling Criteria
- Scheduling Algorithms
  - First Come First Serve (FCFS)
  - Shortest Job First
  - Round robin scheduling

CPU SCHEDULING

The basis of multiprogrammed Operating Systems

Multiprogramming organizes jobs so that the CPU always has one to execute

- A single program (generally) **cannot** keep CPU & I/O devices busy at all times
- A user frequently runs multiple programs
- When a job needs to **wait**, the CPU **switches** to another job
- Utilizes resources effectively
  - CPU, memory, and peripheral devices

Observed Property of Process execution: CPU-I/O burst cycle

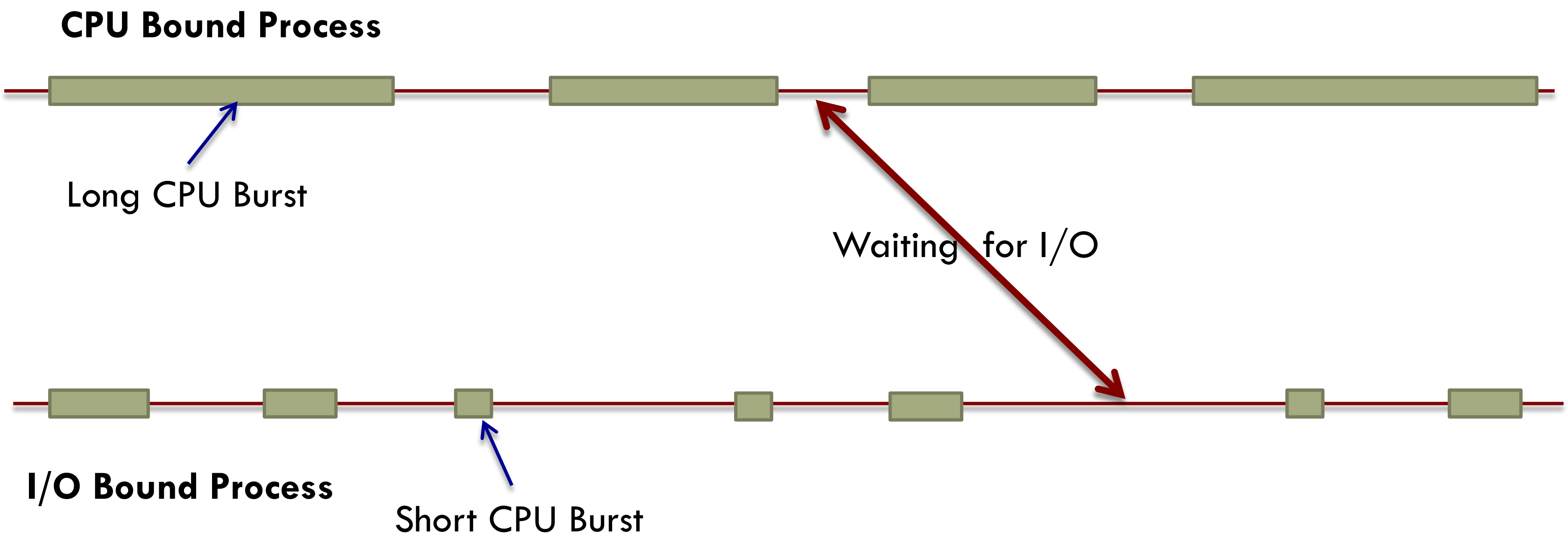
Processes **alternate** between CPU-I/O bursts

load store **add store** **read** from file	CPU burst
wait for I/O	I/O burst
store increment **index** **write** to file	CPU burst
wait for I/O	I/O burst
load store **add store** **read** from file	CPU burst
wait for I/O	I/O burst

Distribution of the duration of CPU bursts

- Large number of short CPU bursts
  - A typical **I/O bound** process
- Small number of long CPU bursts
  - A typical **CPU-bound** process

Bursts of CPU usage alternate with periods of waiting for I/O

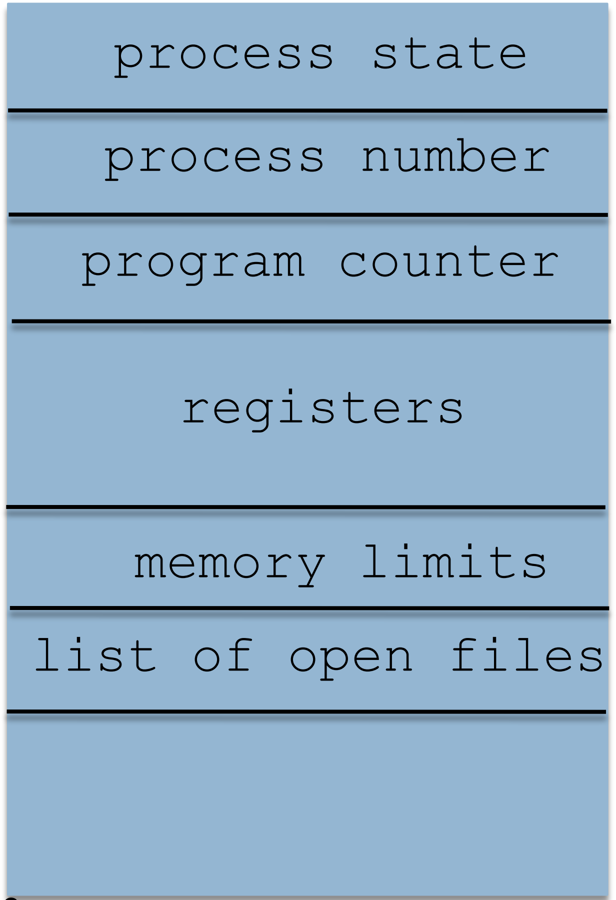


As CPUs get faster …

- Processes tend to get more I/O bound
  - CPUs are improving faster than disks
  - Generally speaking, “computation is free, moving data is expensive”
- Scheduling of I/O bound processes is essential for performance
  - Mostly about “slow” I/O such as disks, network, etc.

When CPU is idle, OS selects one of the processes in the ready queue to execute

- Records in the ready queue are **process control blocks** (PCB)



- **Implemented** as:
  - FIFO queue
  - Priority queue
  - Tree
  - Linked list

The Process Control Block (PCB)

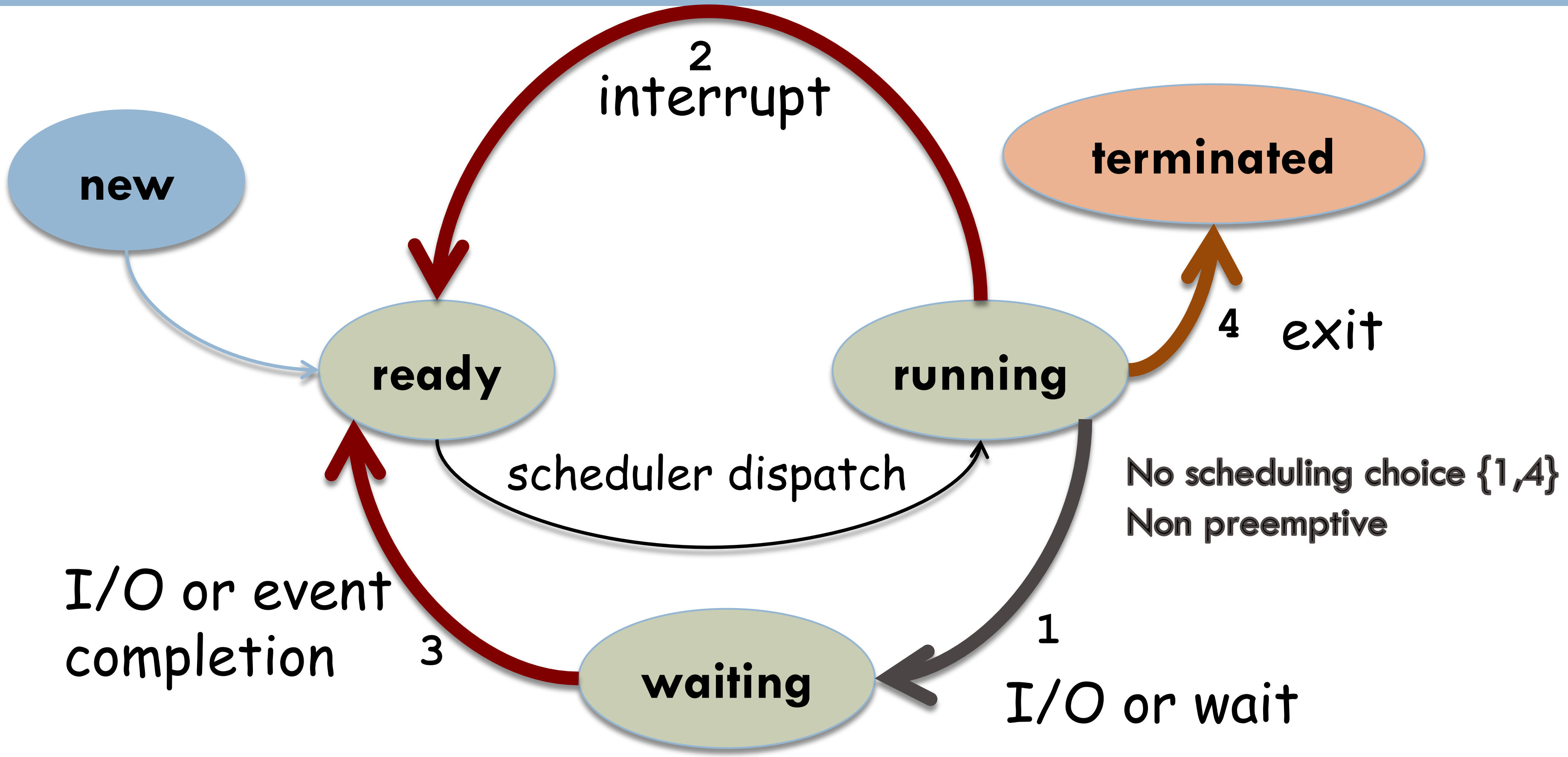
- When a process is not running,
  - The kernel maintains the hardware execution state of a process within the PCB
    - Program counter, stack pointer, registers, etc.
- When a process is being context-switched away from the CPU
  - The hardware state is transferred into the PCB

The Process Control Block (PCB) is a data structure with several fields

- Includes process ID, execution state, program counter, registers, priority, accounting information, etc.

- In Linux:
  - Kernel stores the list of tasks in a circular doubly linked list called the **task list**
  - Each element in the task list is a process descriptor of the type struct `task_struct`, which is defined in `<linux/sched.h>`
  - Relatively large data structure: 1.7 KB on a 32-bit machine with ~100 fields

CPU scheduling takes places under the following circumstances



Nonpreemptive or cooperative sheduling

- Process **keeps** CPU *until it relinquishes* it when:
  1. It terminates
  2. It switches to the waiting state
- Sometimes the *only* method on certain hardware platforms
  - E.g. when they don't have a hardware timer
- Used by initial versions of OS
  - Windows: Windows 3.x
  - Mac OS

Preemptive scheduling

- Pick a process and let it run for a **maximum of some fixed time**
- If it is still running at the end of time interval?
  - **Suspend** it ..
- Pick another process to run

Preemptive scheduling: Requirements

- A **clock interrupt** at the end of the time interval to give control of CPU back to the scheduler
- If no hardware timer is available?
  - Nonpremptive scheduling is the only option

Preemptive scheduling impacts …

- Concurrency management
- Design of the OS
- Interrupt processing

Preemptive scheduling incurs some costs: Manage concurrency

- Access to **shared data**
  - Processes **A** and **B** share data
  - Process **A** is updating *when* it is **preempted** to let Process **B** run
  - Process **B** tries to read data, which is now in an *inconsistent* state

Preemptive scheduling incurs some costs: Affects the design of the OS

- System call processing
  - Kernel may be changing kernel data structure (I/O queue)
- Process preempted in the middle AND
  - Kernel needs to read/modify same structure?
- SOLUTION: **Before** context switch
  - Wait for system call to complete OR
  - I/O blocking to occur

Preemptive scheduling incurs some costs: Interrupt processing

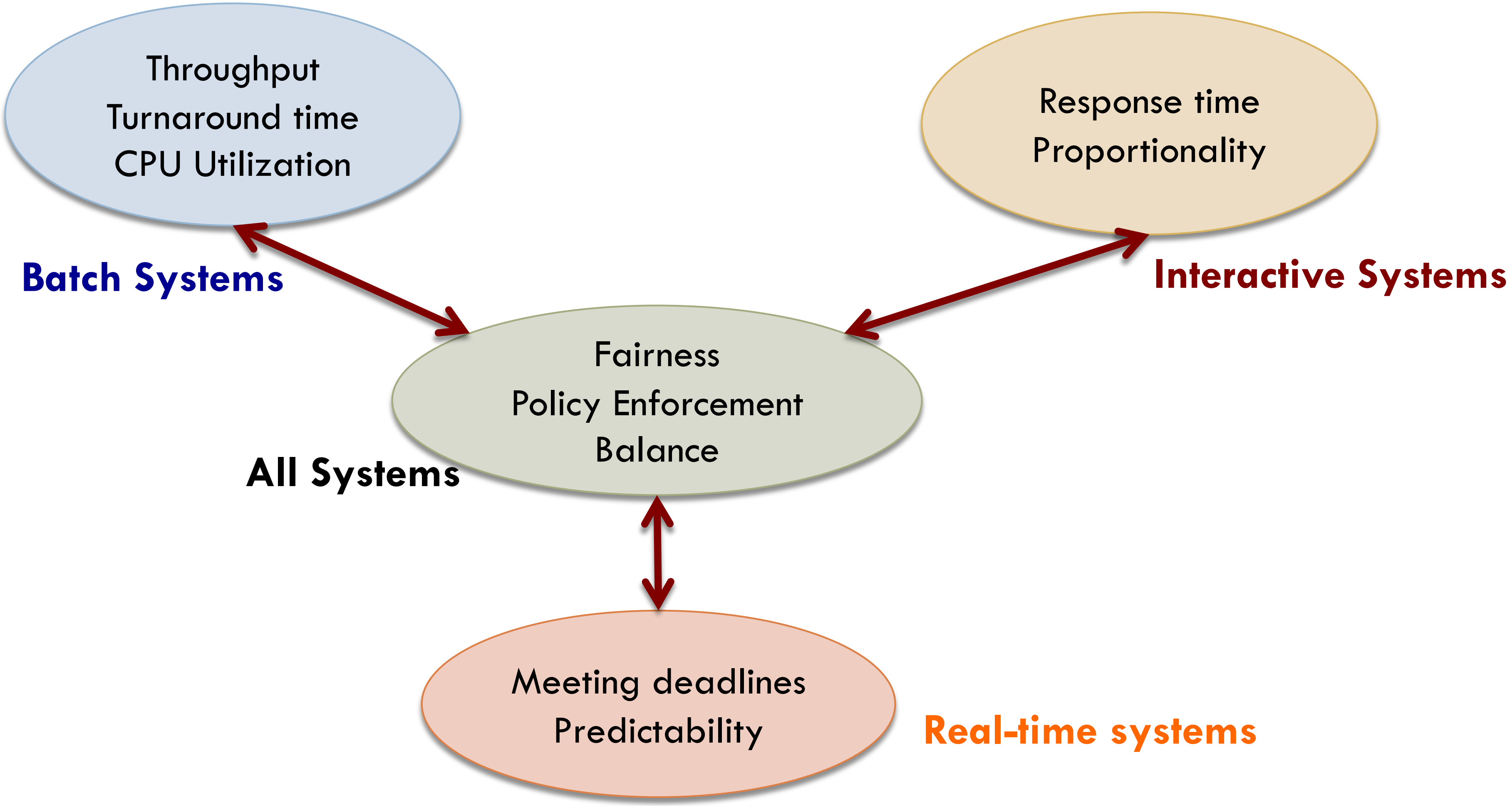
- Interrupts can occur at **any** time
  - Cannot always be ignored by kernel
    - Consequences: Inputs lost or outputs overwritten
- Guard code affected by interrupts from simultaneous use:
  - Disable interrupts during entry
  - Enable interrupts at exit
  - CAVEAT: Should not be done often, and critical section must contain few instructions

The dispatcher is invoked during **every** process switch

- **Gives control** of CPU to process selected by the scheduler
- Operations performed:
  - Switch context
  - Switch to user mode
  - Restart program at the right location
- Dispatch latency
  - Time to stop one process and start another

SCHEDULING CRITERIA

Scheduling Algorithms: Goals



CPU Utilization

- Difference between elapsed time and idle time
- Average over a period of time
  - Meaningful only within a context

Scheduling Criteria: Choice of scheduling algorithm may favor one over another

- **CPU Utilization**: Keep CPU as busy as possible? For example:
  - 40% for lightly loaded system
  - 90% for heavily loaded system
- **Throughput**: Number of completed processes per time unit? For example:
  - Long processes: 1/hour
  - Short processes: 10/second

Scheduling Criteria: Choice of scheduling algorithm may favor one over another

- Turnaround time
  - $t_{completion} - t_{submission}$
- **Waiting** time
  - Total time spent waiting in the ready queue
- Response time
  - Time to start responding
  - $t_{first\_response} - t_{submission}$
  - Generally *limited* by speed of output device

What are we trying to achieve?

- Objective is to *maximize* the **average** measure

- Sometimes averages are not enough
  - Desirable to optimize minimum & maximum values
    - For good service put a ceiling on maximum response time
  - **Minimize the variance** instead of the average
    - *Predictability* more important
    - *High variability*, but faster on average, not desirable

Scheduling Algorithms

- **Decides** which process in the ready queue is allocated the CPU
- Could be preemptive or nonpreemptive
- Optimize ***measure*** of interest
- We will use **Gantt charts** to illustrate *schedules*
  - Bar chart with start and finish times for processes

First Come, First Served Scheduling (FCFS)

First-Come, First-Served Scheduling (FCFS)

- Process requesting CPU first, gets it first
- Managed with a FIFO queue
  - When process **enters** ready queue?
    - PCB is tacked to the **tail** of the queue
  - When CPU is **free**?
    - It is allocated to process at the **head** of the queue
- Simple to write and understand

Average waiting times in FCFS

Process	Burst Time
P1	24
P2	3
P3	3

`Wait time = (0 + 24 + 27)/3 = 17`

`Wait time = (6 + 0 + 3)/3 = 3`

Disadvantages of the FCFS scheme (1)

- Once a process gets the CPU, it keeps it
  - Till it terminates or does I/O
  - Unsuitable for time-sharing systems
- Average waiting time is generally not minimal
  - **Varies substantially** if CPU burst times vary greatly

Disadvantages of the FCFS scheme (2)

- Poor performance in certain situations
  - 1 CPU-bound process and many I/O-bound processes
  - **Convoy effect**: Smaller processes wait for the one big process to get off the CPU

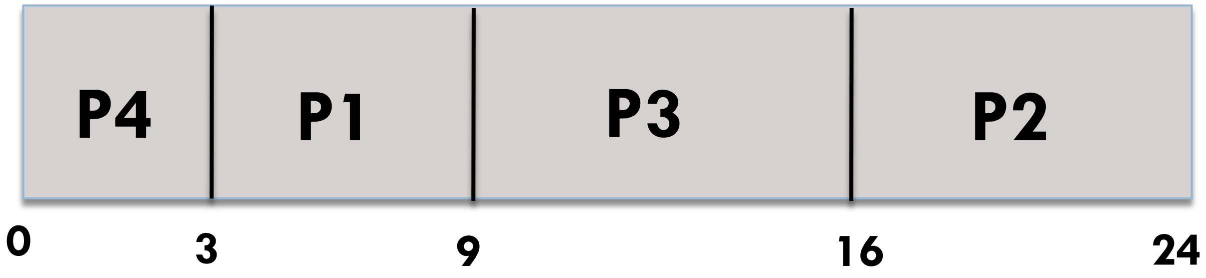
SHORTEST JOB FIRST (SJF)

Shortest Job First (SJF) scheduling algorithm

- When CPU is available it is assigned to process with **smallest CPU burst**
- Moving a short process before a long process?
  - Reduction in waiting time for short process
    GREATER THAN
    Increase in waiting time for long process
- Gives us **minimum average waiting time** for a **set** of processes that arrived *simultaneously*
  - Provably Optimal

Depiction of SJF in action

Process	Burst Time
P1	6
P2	8
P3	7
P4	3



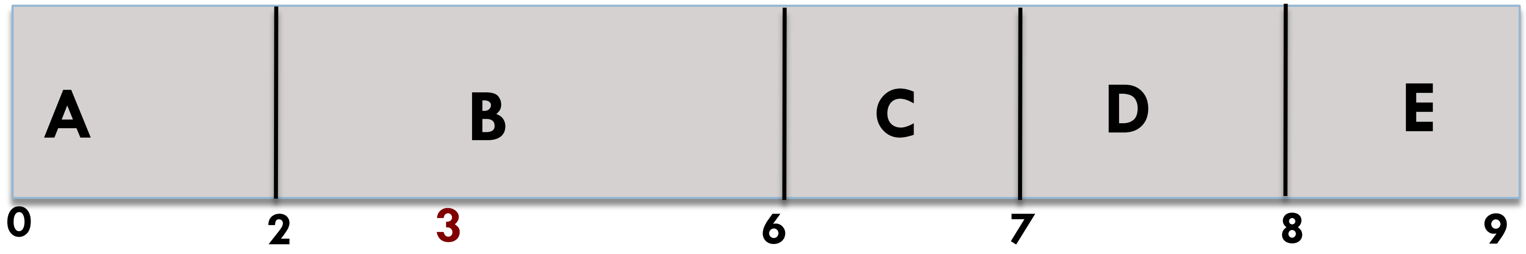
`Wait time = (3 + 16 + 9 + 0)/4 = 7`

SJF is optimal ONLY when ALL the jobs are available simultaneously

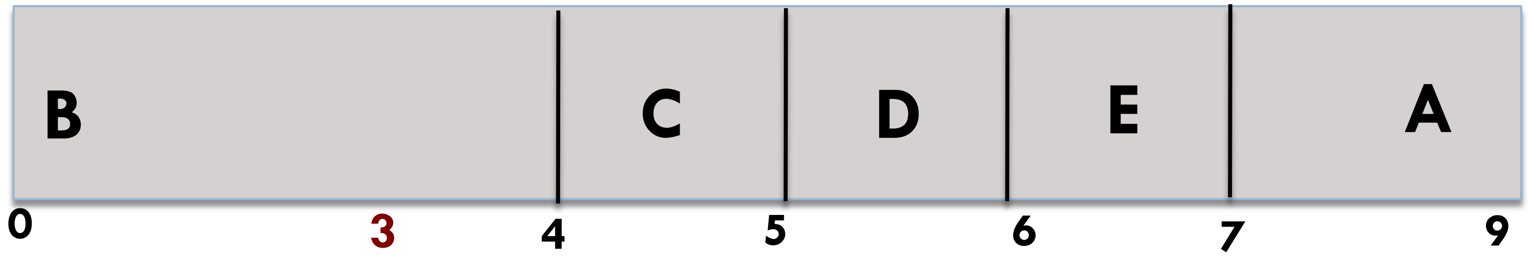
- Consider 5 processes **A**, **B**, **C**, **D** and **E**
  - Run times are: 2, 4, 1, 1, 1
  - Arrival times are: 0,0, 3, 3, 3

- SJF will run jobs: **A**, **B**, **C**, **D** and **E**
  - Average wait time: (0 + 2 + 3 + 4 + 5)/5 = 2.8
  - **But** if you run **B**, **C**, **D**, **E** and **A** ?
    - Average wait time: (7 + 0 + 1 + 2 +3)/5 = 2.6!

Visualizing the different runs of A, B, C, D and E



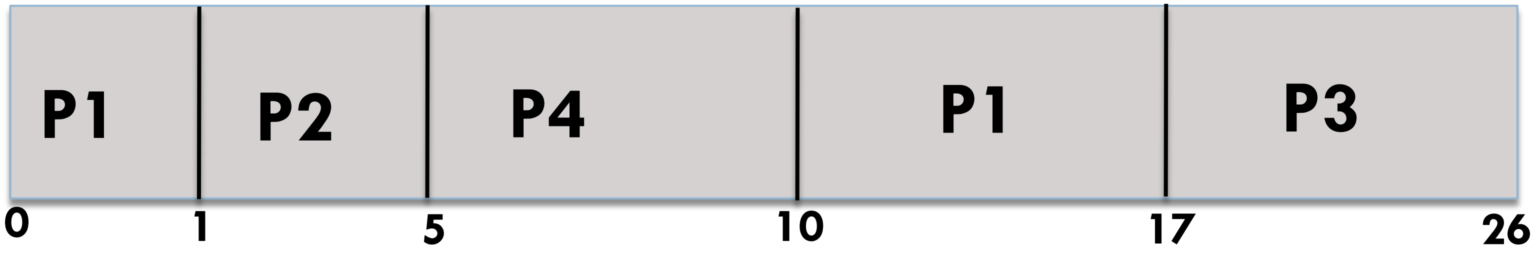
Average wait time: (0 + 2 + 3 + 4 + 5)/5 = 2.8



Average wait time: (7 + 0 + 1 + 2 +3)/5 = 2.6

Preemptive SJF

- A new process arrives in the ready queue
  - If it is shorter than the currently executing process
    - Preemptive SJF will preempt the current process



Process	Arrival	Burst
P1	0	8
P2	1	4
P3	2	9
P4	3	5

```
Wait time =
[(10-1) + (1-1) + (17-2) + (5-3)]/4
= 26/4 = 6.5
```

Use of SJF in long term schedulers

- Length of the process time limit
  - Used as CPU burst estimate
- Motivate users to accurately estimate time limit
  - Lower value will give faster response times
  - Too low a value?
    - Time limit exceeded error
    - Requires resubmission!

The SJF algorithm and short term schedulers

- **No way to know** the length of the next CPU burst
- So try to **predict** it
- Processes scheduled *based on predicted* CPU bursts

Prediction of CPU bursts: Make estimates based on past behavior

- t_n : Length of the n^{th} CPU burst
- τ_n : Estimate for the n^{th} CPU burst
- α : Controls weight of recent and past history
- $\tau_{n+1} = \alpha t_n + (1-\alpha)\,\tau_n$

- Burst is predicted as an exponential average of the measured lengths of previous CPU bursts

α controls the relative weight of recent and past history

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

- Value of t_n contains our most recent information, while τ_n stores the past history
- $\tau_{n+1} = \alpha t_n + (1-\alpha)\,\alpha t_{n-1} + \ldots + (1-\alpha)^j\,\alpha t_{n-j} + \ldots + (1-\alpha)^{n+1}\,\alpha\tau_0$

- α is less than 1, $(1-\alpha)$ is also less than one
  - **Each successive term has less weight than its predecessor**

The choice of α in our predictive equation

- $\tau_{n+1} = \alpha t_n + (1-\alpha)\, \tau_n$
- If $\alpha=0$, $\tau_{n+1} = \tau_n$
  - Current conditions are transient
- If $\alpha=1$, $\tau_{n+1} = t_n$
  - Only most recent bursts matter
  - History is assumed to be old and irrelevant

The choice of α in our predictive equation

- If $\alpha = 1/2$
  - Recent history and past history are **equally weighted**

- With $\alpha = ½$; successive estimates of τ

  $t_0/2 \quad t_0/4 + t_1/2 \quad t_0/8 + t_1/4 + t_2/2 \quad t_0/16 + t_1/8 + t_2/4 + t_3/2$

  - By the 3^{rd} estimate, weight of t_0 has dropped to 1/8.

The contents of this slide-set are based on the following references

- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9th edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330.* [Chapter 6]
- *Andrew S Tanenbaum. Modern Operating Systems. 4th Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620.* [Chapter 2]