CS370 Operating Systems

Colorado State University
Yashwant K Malaiya

Fall 2016 Lecture 12

Slides based on

- **Text by Silberschatz, Galvin, Gagne**
- **Various sources**

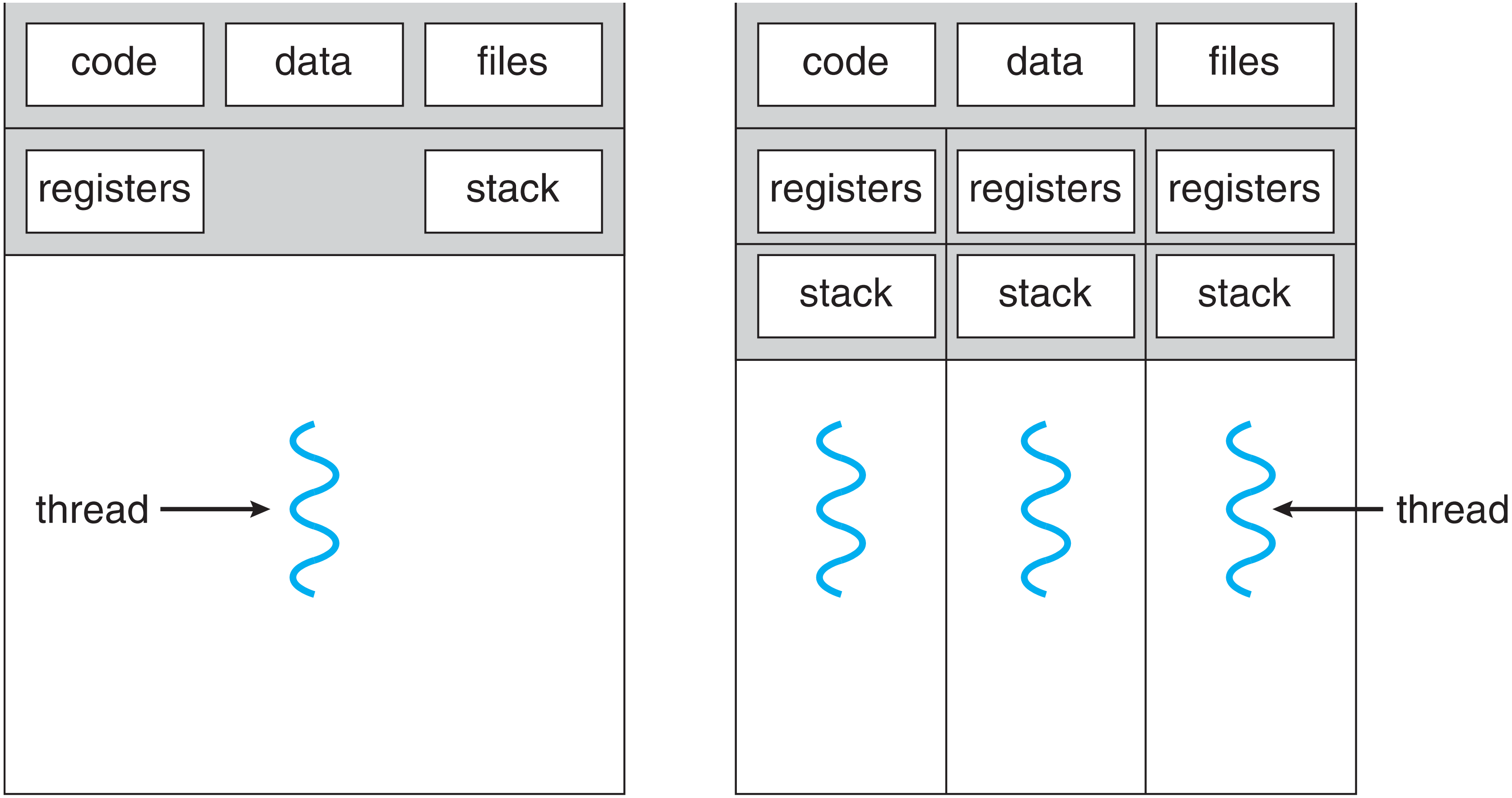
CPU Scheduling: Objectives

- CPU scheduling, the basis for multiprogrammed operating systems
- CPU-scheduling algorithms
- Evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- Scheduling algorithms of several operating systems

FAQ

- How can multiple threads execute concurrently, when programs execute linearly?
- How can a process execute on multiple cores?
- Thread-local storage (TLS) When is it needed?
  Ex: when using a thread pool, each transaction has a thread and a transaction identifier is needed. Syntax - C11, and Java use thread_local keyword to declare.
- Unix signals vs interrupts: Signals are a limited form of inter-process communication. Interrupts are often initiated by hardware.
- Foreground vs background process? A foreground process has access to the terminal (standard input and output)
- Daemon? Background service processes

Review



single-threaded process

multithreaded process

Colorado State University

Review

```
int main(int argc, char *argv[])
{
    /* get the default attributes */
    pthread_attr_init(&attr);

    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);

    /* now wait for the thread to exit */
    pthread_join(tid, NULL);

    printf("sum = %d\n",sum);
}
```

```
/**
 * Thread will begin control in this function
 */
void *runner(void *param)
{
int i, upper = atoi(param);
sum = 0;
    if (upper > 0) {
    for (i = 1; i <= upper; i++)
    sum += i;
}
pthread_exit(0);
}
```

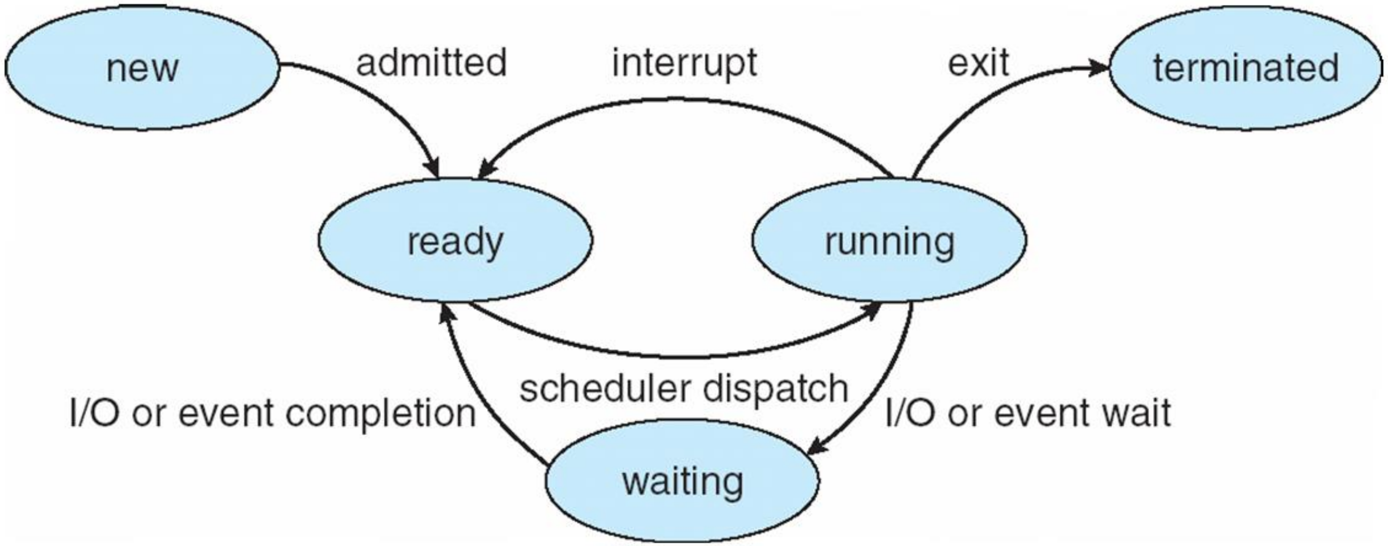
Question: Differences among calling a function, starting a child, creating a thread?

Colorado State University

Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Diagram of Process State

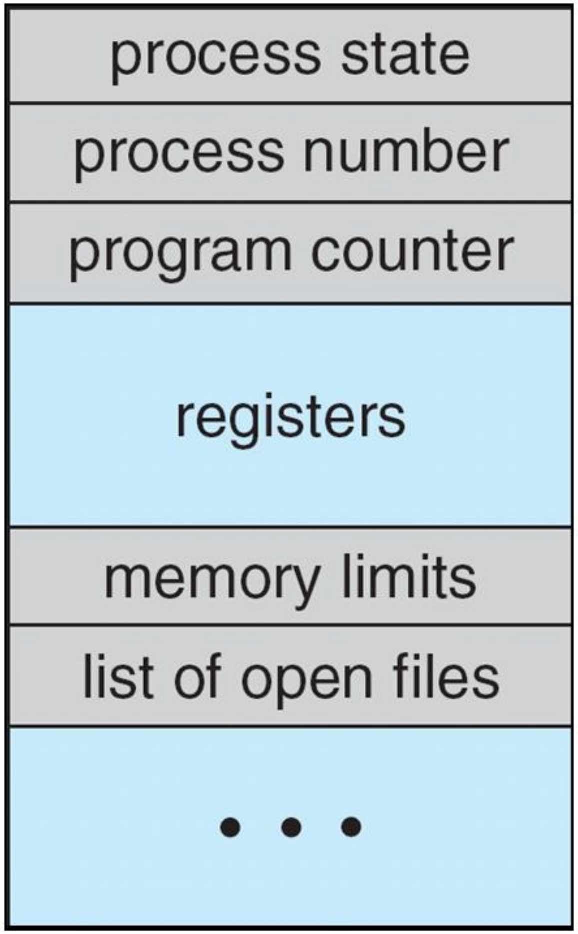


Ready to Running: scheduled by scheduler
Running to Ready: scheduler picks another process, back in ready queue
Running to Waiting (Blocked) : process blocks for input/output
Waiting to Ready: Input available

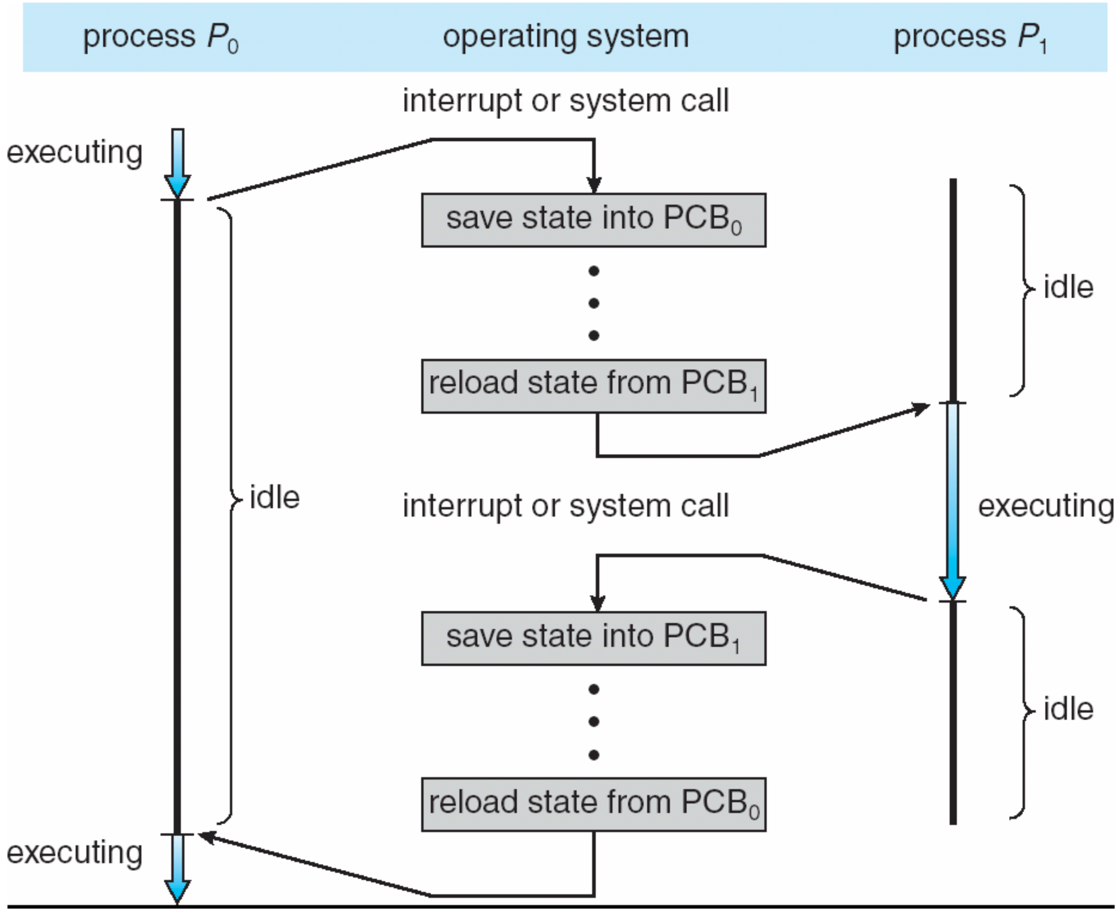
Process Control Block (PCB)

Information associated with each process
(also called **task control block**)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



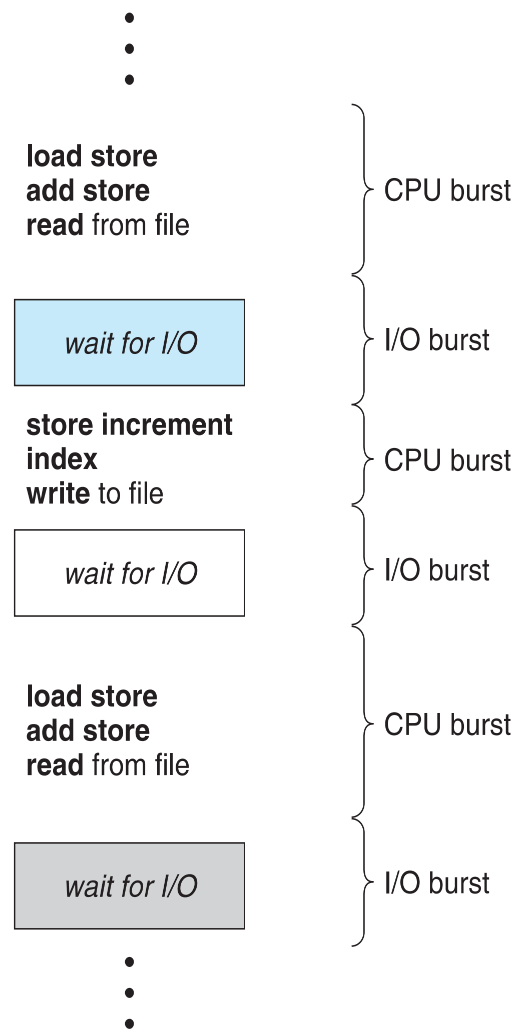
CPU Switch From Process to Process



Colorado State University

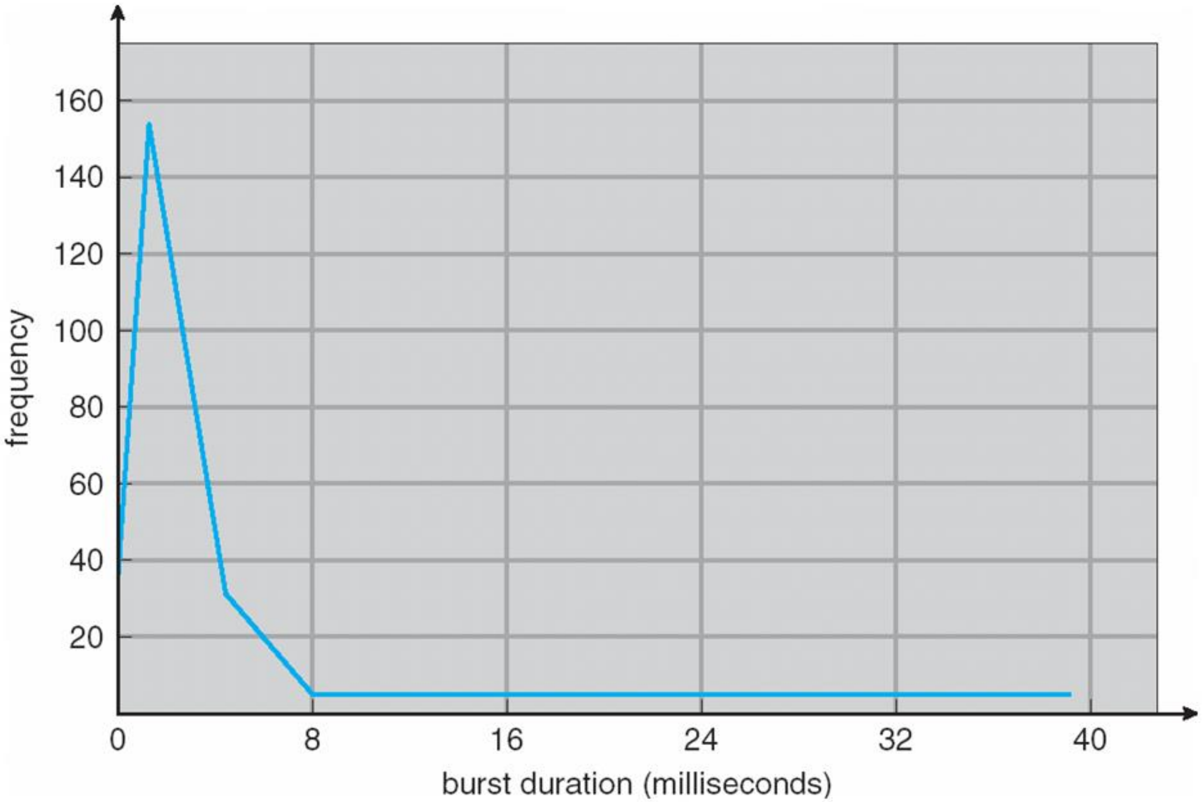
Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



Colorado State University

Histogram of CPU-burst Times



CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive. These need to be considered**
  - access to shared data by multiple processes
  - preemption while in kernel mode
  - interrupts occurring during crucial OS activities

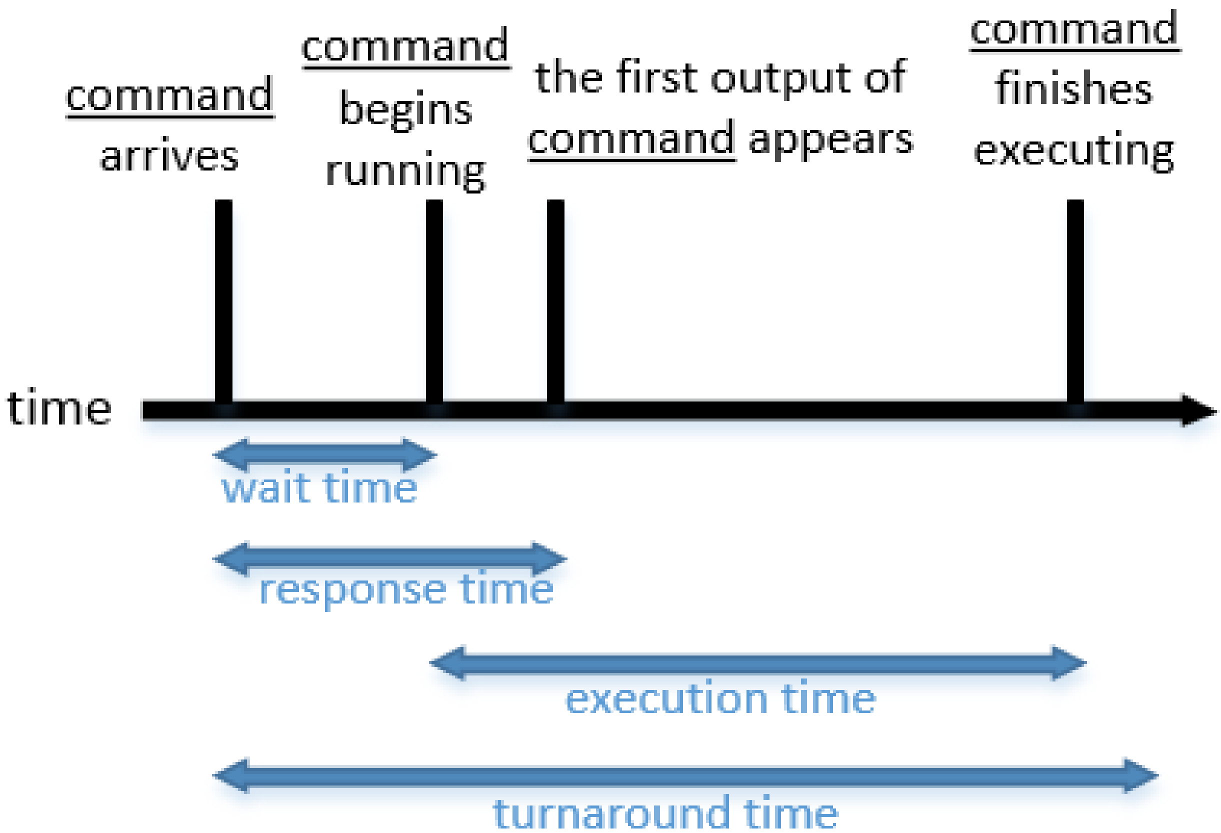
Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible: Maximize
- **Throughput** – # of processes that complete their execution per time unit: Maximize
- **Turnaround time** –time to execute a process from submission to completion: Minimize
- **Waiting time** – amount of time a process has been waiting in the ready queue: Minimize
- **Response time** –time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment): Minimize

Terms *for a single process*



UCLA

Scheduling Algorithms

- **We will now examine several major scheduling approaches**
- **Decides** which process in the ready queue is allocated the CPU
- Could be preemptive or nonpreemptive
  - preemptive: remove in middle of execution
- Optimize ***measure*** of interest
  - We will use **Gantt charts** to illustrate *schedules*
  - Bar chart with start and finish times for processes

Nonpreemptive vs Preemptive sheduling

- **Nonpreemptive:** Process keeps CPU until it relinquishes it when
  - **It terminates**
  - **It switches to the waiting state**
  - Used by initial versions of OSs like Windows 3.x
- **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 and pick another process to run
- A **clock interrupt** at the end of the time interval to give control back of CPU back to scheduler

Scheduling Algorithms

Algorithms

- First- Come, First-Served (FCFS)
- Shortest-Job-First (SJF)
  - Shortest-remaining-time-first
- Priority Scheduling
- Round Robin (RR) with time quantum
- Multilevel Queue
  - Multilevel Feedback Queue

Comparing Performance

- Average waiting time etc.

First- Come, First-Served (FCFS) Scheduling

- 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

First- Come, First-Served (FCFS) Scheduling

Henry Gantt, 1910s

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3 *but almost the same time*.
  The Gantt Chart for the schedule is:

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17
- Throughput: 3/30 = 0.1 per unit

FCFS Scheduling (Cont.)

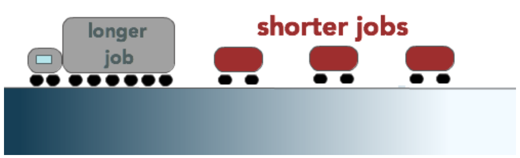
Suppose that the processes arrive in the order:

$$P_2\ ,\ P_3\ ,\ P_1$$

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6;\ P_2 = 0,\ P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- But note -Throughput: 3/30 = 0.1 per unit
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

The Convoy Effect, visualized



Shortest-Job-First (SJF) Scheduling

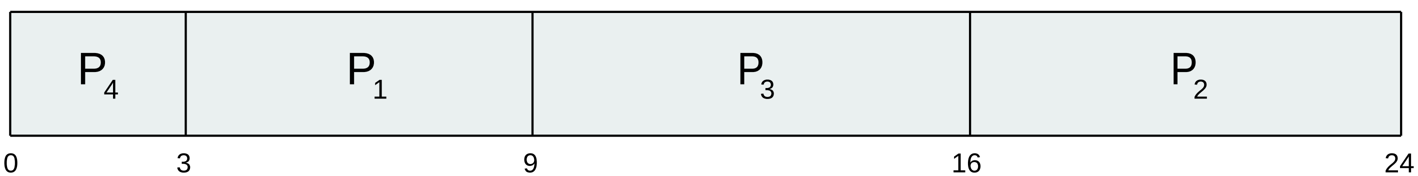
- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- Reduction in waiting time for short process *GREATER THAN* Increase in waiting time for long process
- SJF is optimal – gives **minimum average waiting time** for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

Colorado State University

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- All arrive at time 0.
- SJF scheduling chart



- Average waiting time for P_1,P_2,P_3,P_4 = (3 + 16 + 9 + 0) / 4 = 7

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the recent bursts
  - Then pick process with shortest predicted next CPU burst

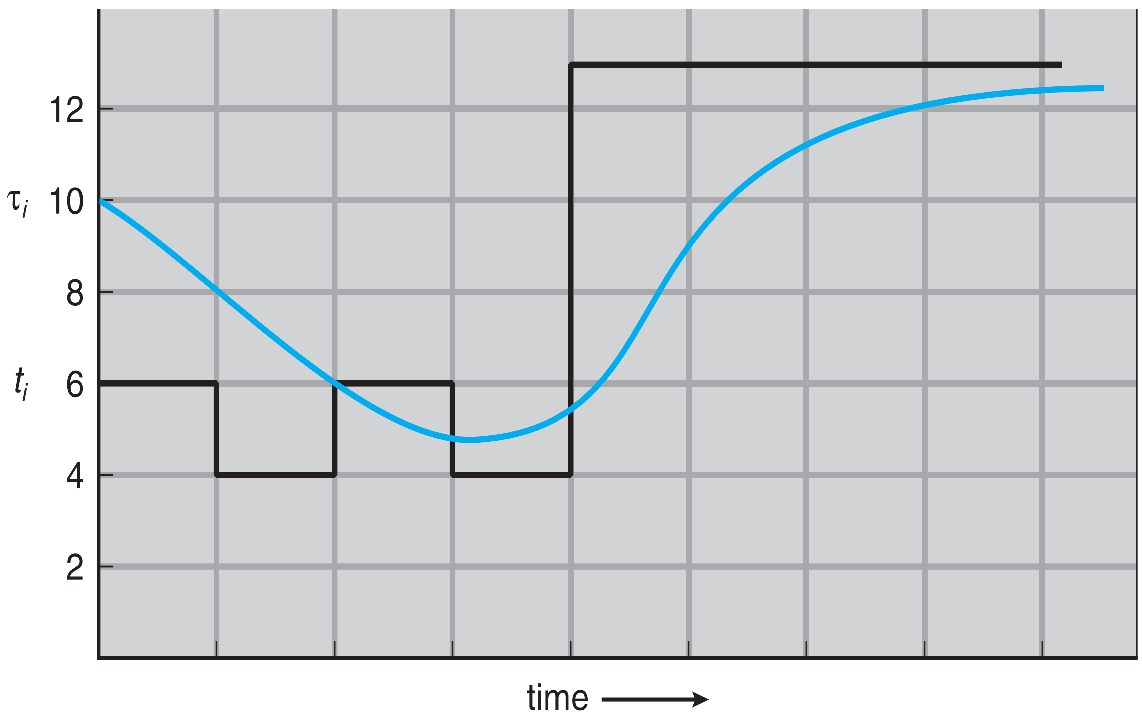
- Can be done by using the length of previous CPU bursts, using *exponential averaging*

$$
\begin{aligned}
&1.\ t_n = \text{actual length of } n^{th} \text{ CPU burst}\\
&2.\ \tau_{n+1} = \text{predicted value for the next CPU burst}\\
&3.\ \alpha, 0 \le \alpha \le 1\\
&4.\ \text{Define}: \quad \tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.
\end{aligned}
$$

- Commonly, α set to ½
- Preemptive version called **shortest-remaining-time-first**

Colorado State University

Prediction of the Length of the Next CPU Burst



CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Blue points: guess
Black points: actual
$\alpha = 0.5$

Ex:
0.5x6 +0.6x10 = 8

Examples of Exponential Averaging

- α =0
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- α =1
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots + (1 - \alpha)^j \alpha\, t_{n-j} + \ldots + (1 - \alpha)^{n+1} \tau_0$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Widely used for predicting stock-market etc

Colorado State University