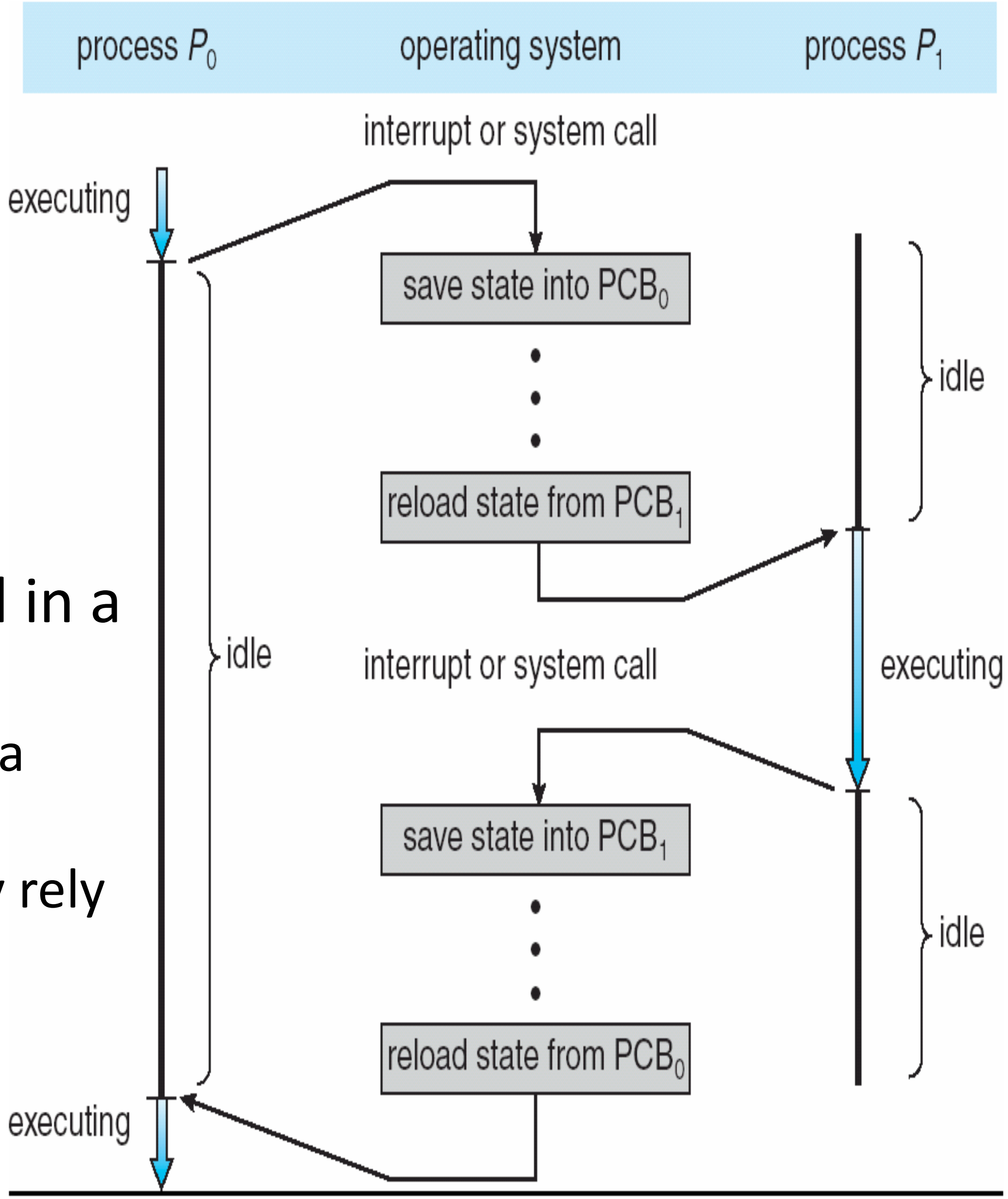
Agenda

- Assignment 1 is out, due September 29!
- Today's lecture
  - Role of the Scheduler and Scheduling Criteria
  - CPU Scheduling Algorithms
    - FCFS
    - Shortest Job First
    - RR
    - Priority Scheduling
    - Multilevel queues
- Reading: Sections 5.1-5.3
- Next Week: Threads and Multiprogramming chapters 3,4

Review

- Not all processes can run at the same time. Why?
- Processes run based on a schedule
- Those that are ready to run are stored in a ***Ready queue***
    - Abstract data structure implemented as a Queue, priority queue, etc…
    - Scheduler and scheduling algorithm may rely on more than one queue



Queues

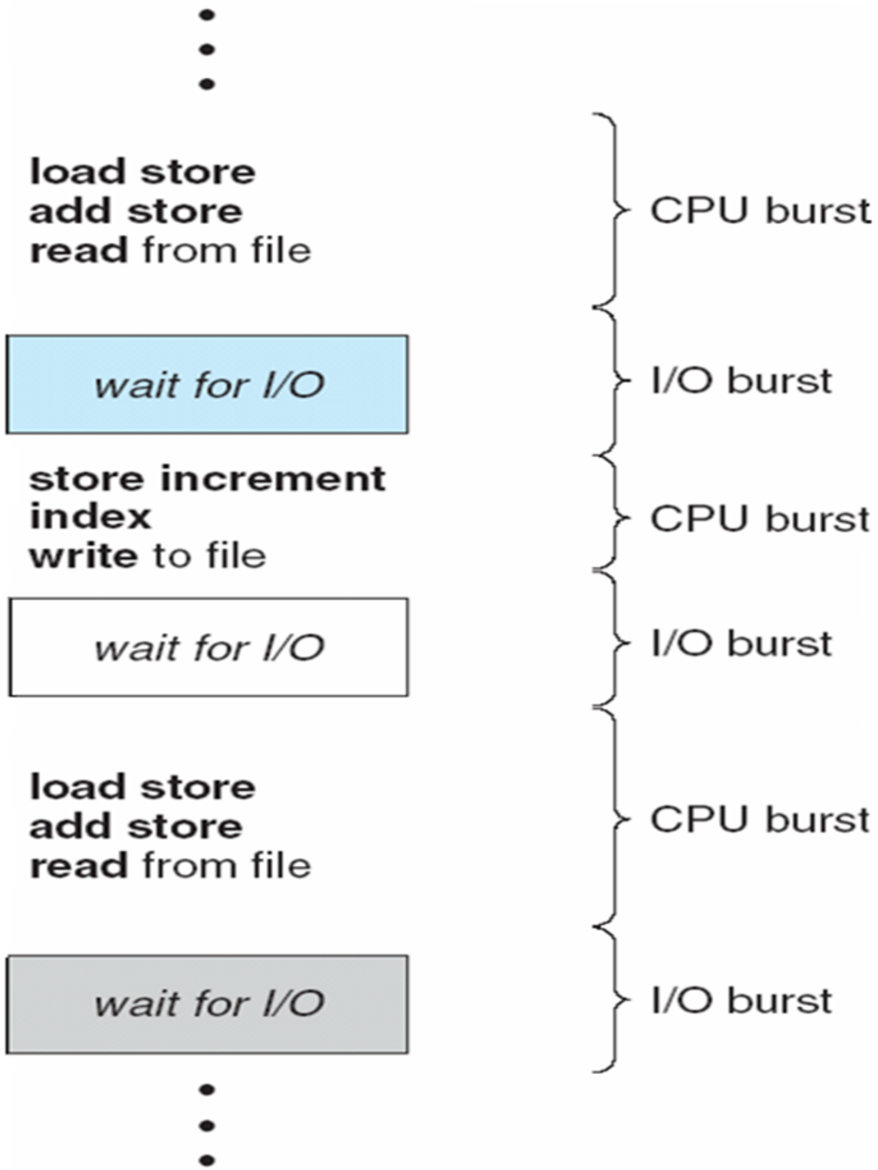
- Used to schedule processes
- While a process is waiting in the queue, it is in the waiting state
    - **Job** queue – set of all processes
    - **Ready** queue – processes in main memory, ready to execute
    - **Device** queues – processes waiting for an I/O device
- Processes move among various queue creating a cycle

The CPU- I/O Burst Cycle

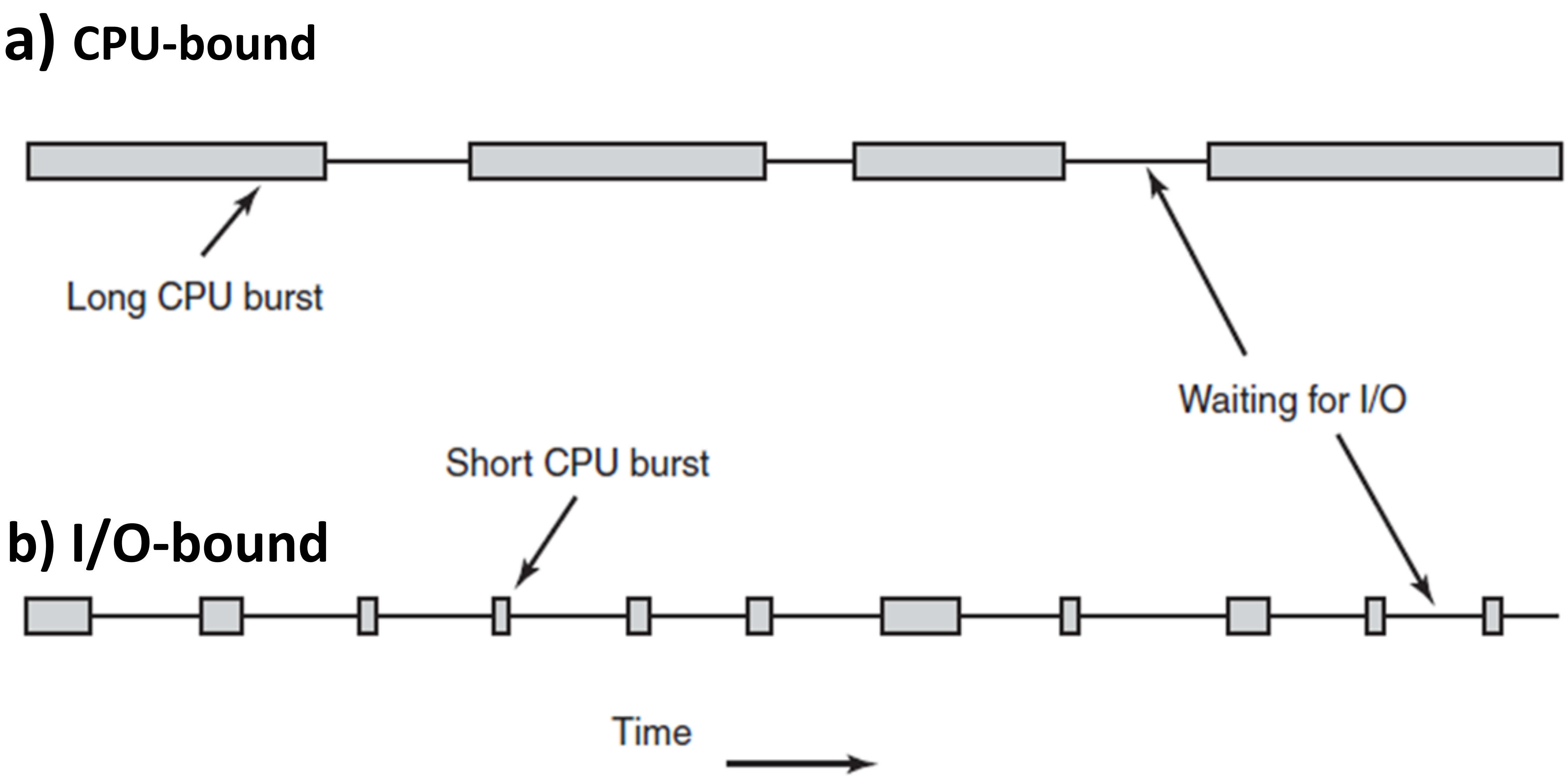
Most processes have a very common execution pattern

CPU burst : perform computation on the CPU
- Execute instructions
- Access memory
- Does not require OS services

I/O burst : performs an I/O operation
- Access a file, network, or other device
- Make a system call to kernel
- Kernel initiates request to hardware
- Hardware is typically slow compared to CPU
- Process waits until hardware request completes
- Process is blocked until request completes



Process Behavior



CPU Scheduling

- During an I/O burst, a process is blocked and is not using the CPU
- We want to maximize the use of the CPU and let it run another process
- Questions:
  - Which process do we choose? There may be many ready processes
  - What happens when waiting process becomes unblocked
  - What if a process does not perform any I/O?
- These issues are addressed by the scheduler

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **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 output (for time-sharing environment)

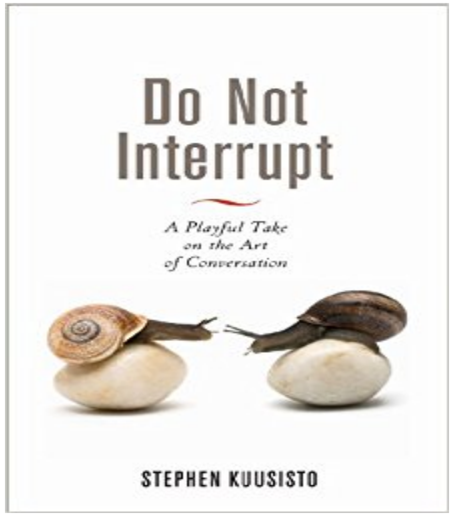
CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them

- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state (I/O or timer interrupt)
  3. Switches from waiting to ready
  4. Terminates

- The type of scheduling algorithm determines when a scheduler runs:
  - Scheduling under 1 and 4 is **non-preemptive (cooperative)**
  - All other scheduling is **preemptive**

Non-preemptive Scheduling



- Scheduling decisions occur when:
  - A running process blocks
  - A running process terminates
- Processes are permitted to run until they request a kernel service

  E.g., perform I/O
- Characteristics
  - Efficient (no unnecessary overhead)
  - Simple to implement (simple kernel structure)
  - Predictable, works well for real-time systems
  - Allows processes to monopolize CPU (happened on older OSes)

Preemptive Scheduling

…for interrupting…

- Processes may be switched (preempted)
  - I/O interrupt
    - Completion of I/O operation
    - Process moves from waiting to ready queue
    - Process moves from waiting to running
  - Timer interrupt
    - Preprogrammed times (10-100 ms)
    - Quantum: the max amount of time that a process can remain on the CPU
- Characteristics
  - Timer servicing adds extra overhead
  - Harder to implement, any interrupt can result in a process switch
  - Less predictable
  - Safer in multiuser systems

First-Come, First-Served (FCFS)

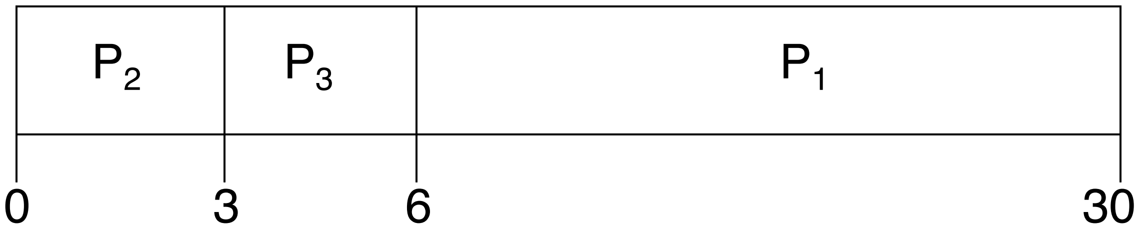
- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
- The Gantt Chart for the schedule is:

Process	Burst Time
P1	24
P2	3
P3	3

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

First-Come, First-Served (FCFS)

- Suppose that the processes arrive in the order: P_2 , P_3 , P_1
- The Gantt Chart for the schedule is:



Process	Burst Time
P1	24
P2	3
P3	3

- 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
- Convoy effect short process behind long process

FCFS

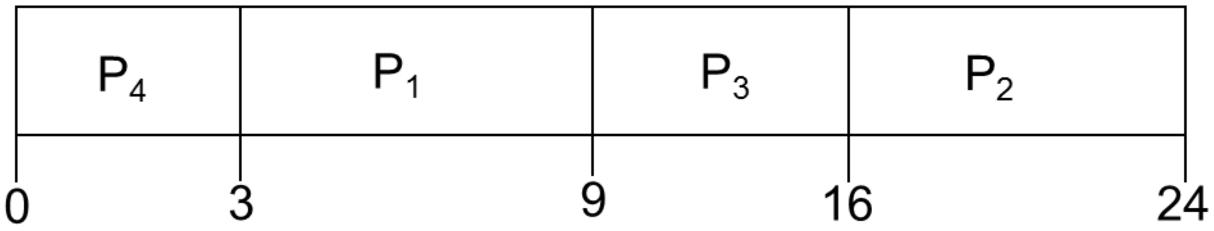
- Pros:
  - Simple FIFO queue needed for implementation
  - Processes at head of queue gets to run next
  - Processes continue to run until it requests a service from the kernel
- Cons:
  - Average waiting time can be long
  - Non-pre-emptive, poor for multi-user systems
- But, can be enhanced when used with pre-emption (Roud Robin)

Shortest-Job-First (SJF)

- Associate with each process the length of its next CPU burst.
- Use these lengths to schedule the process with the shortest time
- Pros:
  - Simple priority queue needed for implementation
  - Processes at head of queue gets to run next
  - Process' priority is based on its CPU burst length
  - Processes continue to run until it requests a service from the kernel
  - Optimal – gives minimum average waiting time for a given set of processes
- Cons:
  - Hard to implement
  - Non-pre-emptive, poor for multi-user systems
  - The difficulty is knowing the length of the next CPU request

SJF Example

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0)/4

 = 7

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Determining Length of the Next CPU Burst

- Problem: don’t know the length of the process’ next CPU burst
- Can only estimate the length based on history
- Can be done by using the length of previous CPU bursts, using exponential averaging

Exponential Moving Average

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

- τ_{n+1} = predicted value for the next CPU burst
- τ_n = Where 0 < actual length of the nth CPU burst
- $\alpha < 1$
- The larger the α the more weight the most recent term has.

- Note: For a regular average $\alpha = n^{-1}$
- Why is this called the exponential moving average?
- If we expand τ_n

$$\tau_n = \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2\alpha t_{n-2} \ldots = \sum_{i=0}^{n} (1-\alpha)^i \alpha t_{n-i}$$

- Each previous term is weighed exponentially less than the next
- Typical value of $\alpha = 1/2$

Exponential Averaging Examples

- 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

Priority Scheduling

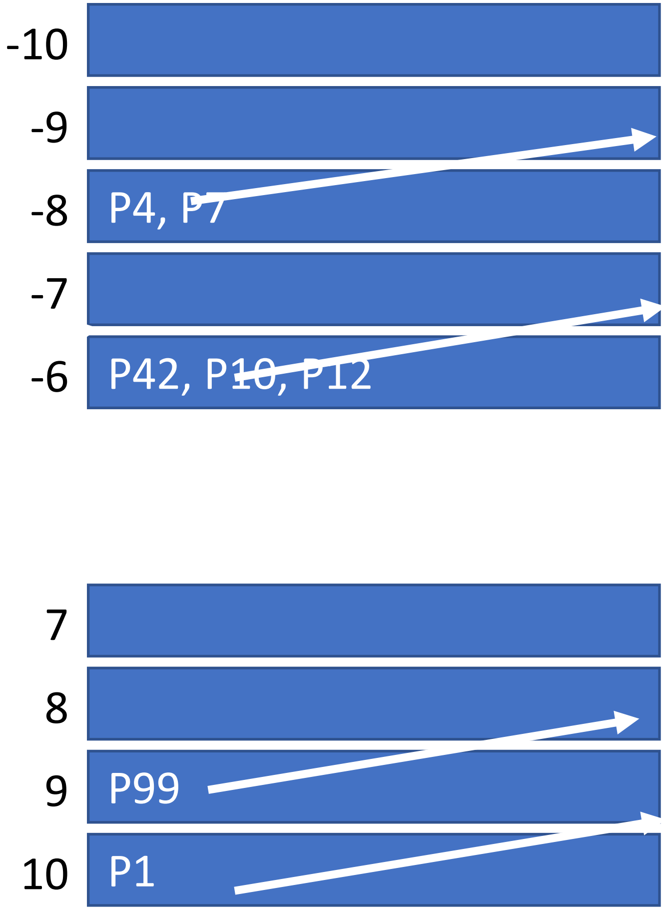
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Implementing Priority Scheduling

- Problem: Standard Priority Queue are expensive
  - O(log n) access time
  - Tree based or heap based
  - Inserting / removing is “complex”
- Idea: Use multiple queues
  - Priorities are fixed -10… 10
  - Use one queue per priority
  - Check each queue for next process

    E.g. P4 is the next process to run
- Aging

  Process at head of each queue is promoted to next queue each quantum



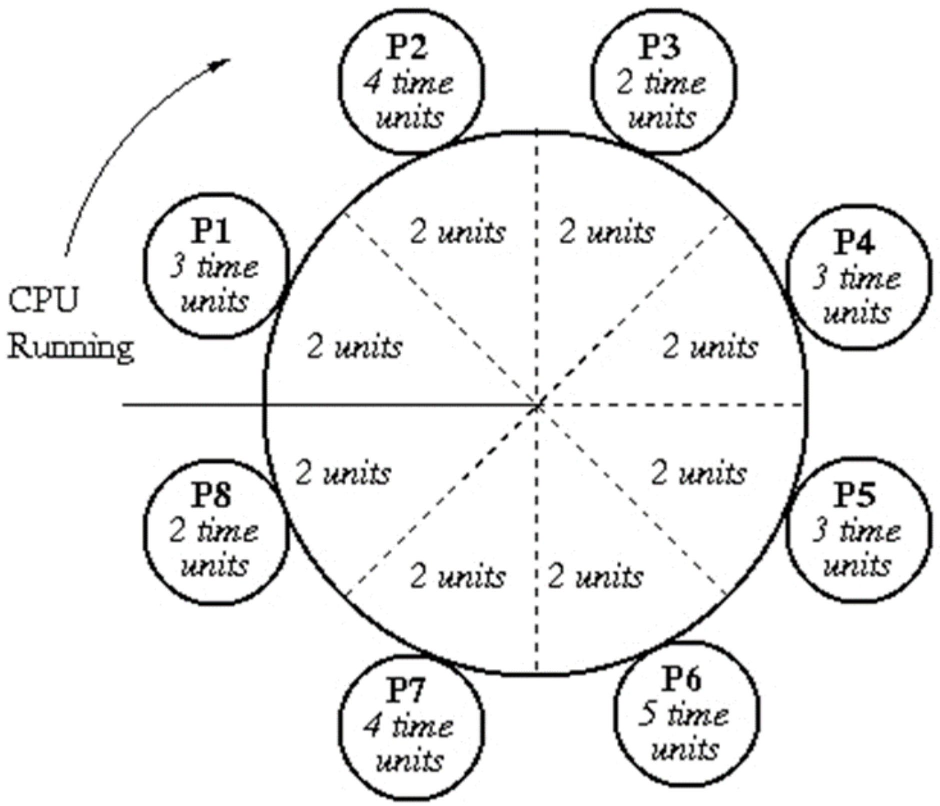
Round Robin Scheduling

- Basic Idea:
  - Single FIFO ready queue of all processes
  - Process at head of queue gets to run next
  - Process continues to run until it
    - Blocks on I/O
    - Terminates
    - Uses up it’s quantum of execution (timer interrupt goes off)
- Characteristics
  - Simple implementation
  - Indiscriminate between I/O bound and CPU bound processes
  - High waiting time
  - Convoy effect

Round Robin Scheduling

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

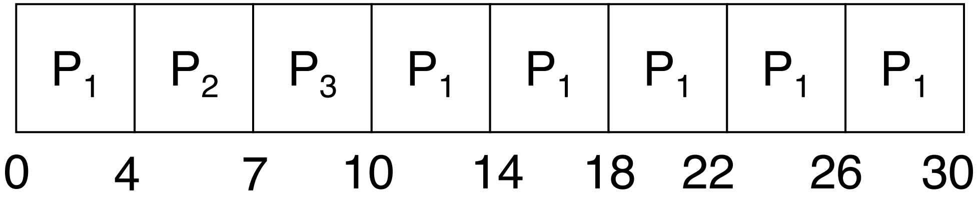
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.



RR Example

Process	Burst Time
P1	24
P2	3
P3	3

- The Gantt chart is:



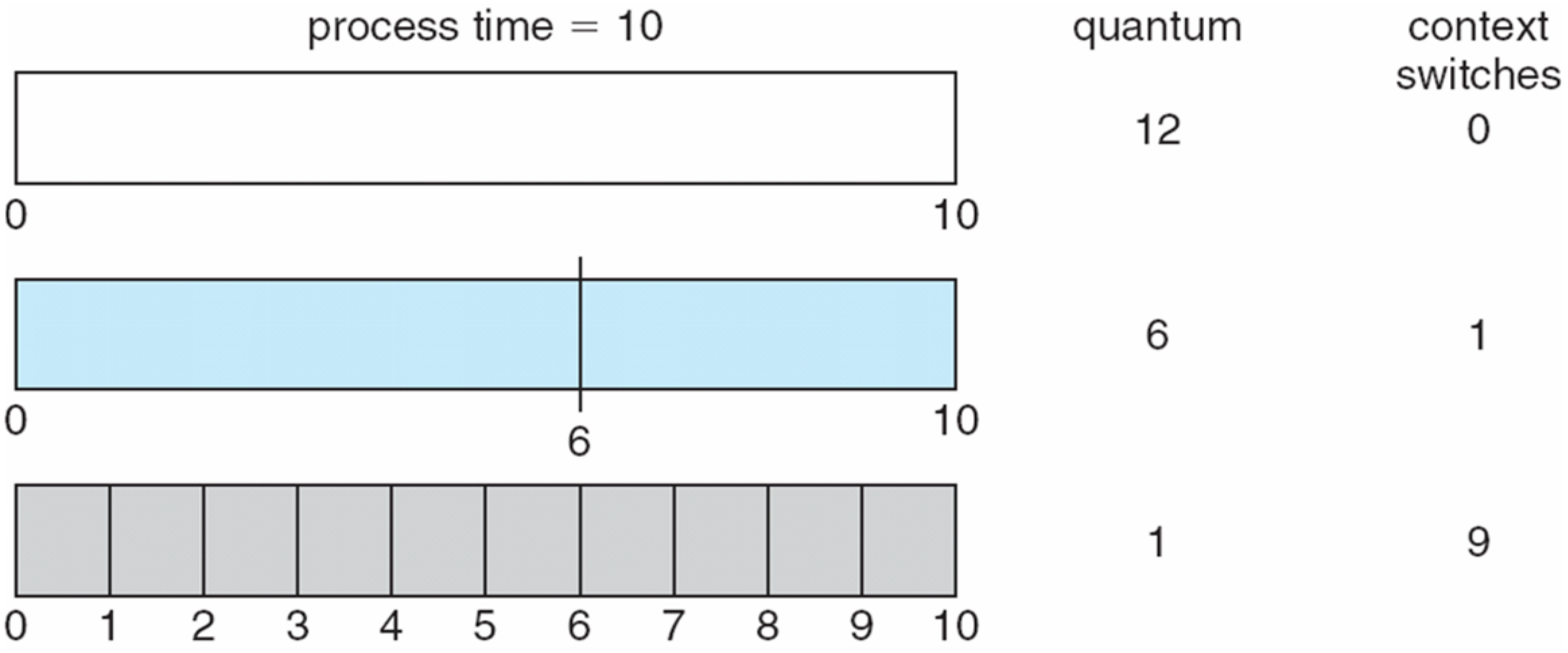
- Typically, higher average turnaround than SJF, but better response

Time Quantum and Context

Performance

q large $\Rightarrow$ FIFO

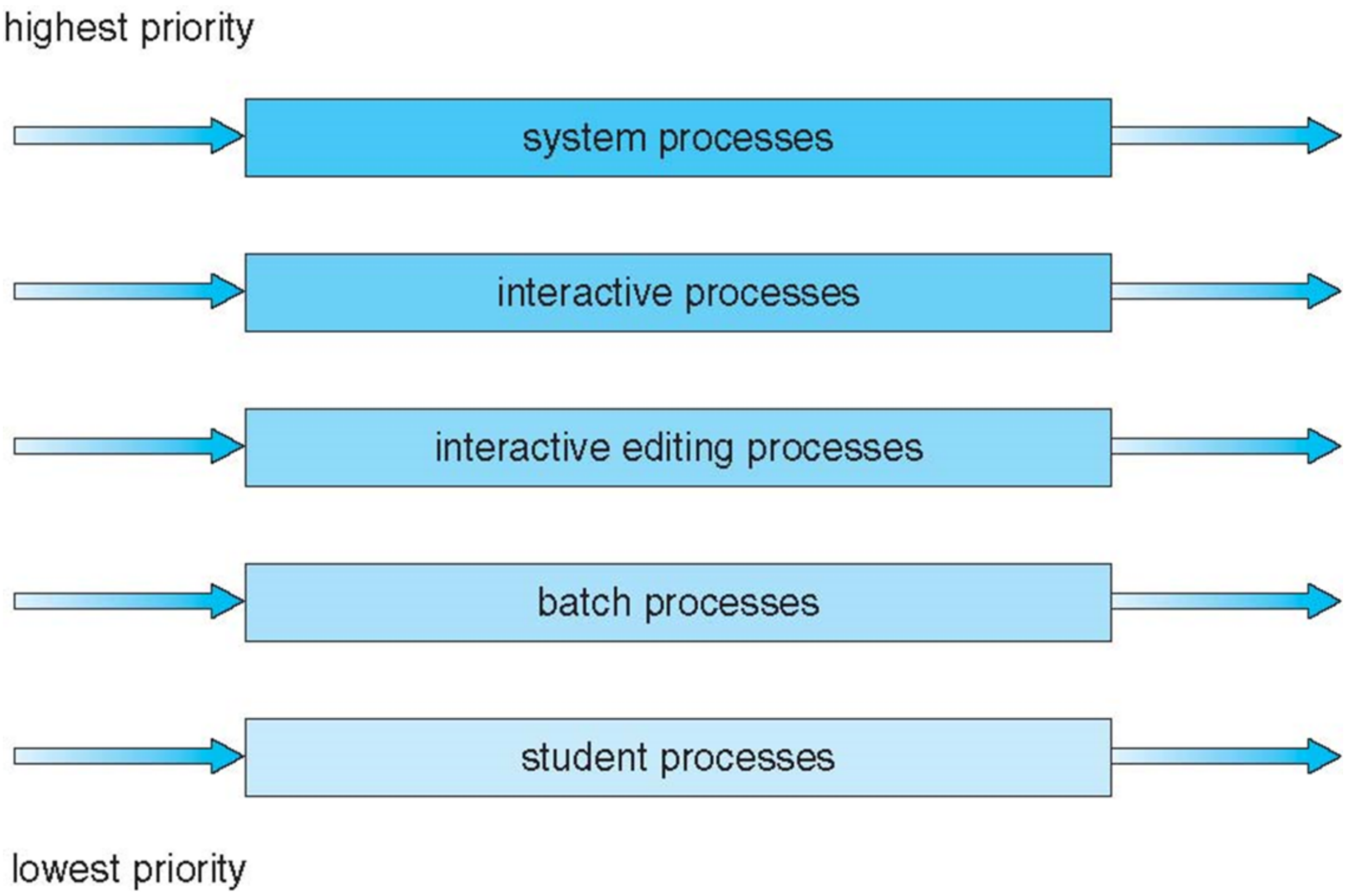
q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high



Multilevel Queue Scheduling

- Ready queue is partitioned into separate queues:
  - Foreground (interactive)
  - Background (batch)
- Each queue has its own scheduling algorithm
  - Foreground – RR
  - Background – FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling → Possibility of **starvation**
  - Time slice - each queue gets a certain amount of CPU time which it can schedule amongst its processes

Multilevel Queue Scheduling



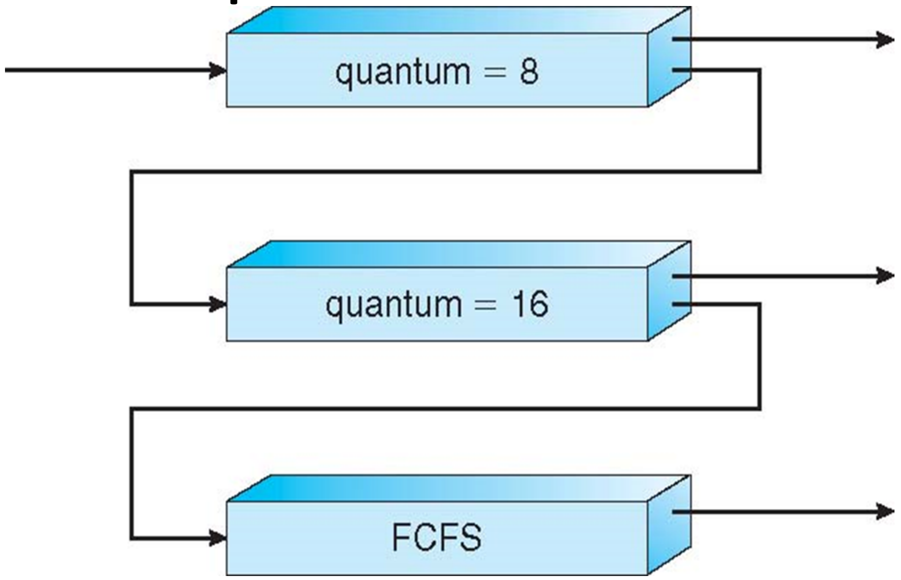
Multilevel Queue Scheduling

- Characteristics
  - All process get to run
  - More important processes can be run more often
  - If we assign processes correctly, we get low waiting times
  - Used in many systems today
- **Problem**: priorities must be determined a priori
- System cannot adjust to what the process does

Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Multilevel Feedback Queue Example



- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served RR. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
  - At Q_1 job is again served RR and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

What’s Next?

- Now that we have a better idea on how the OS schedules processes, we have more questions to address:
- What to do with these processes?
- Why is having multiple processes useful?
- How to make sure processes work together without interfering with each other’s resources?