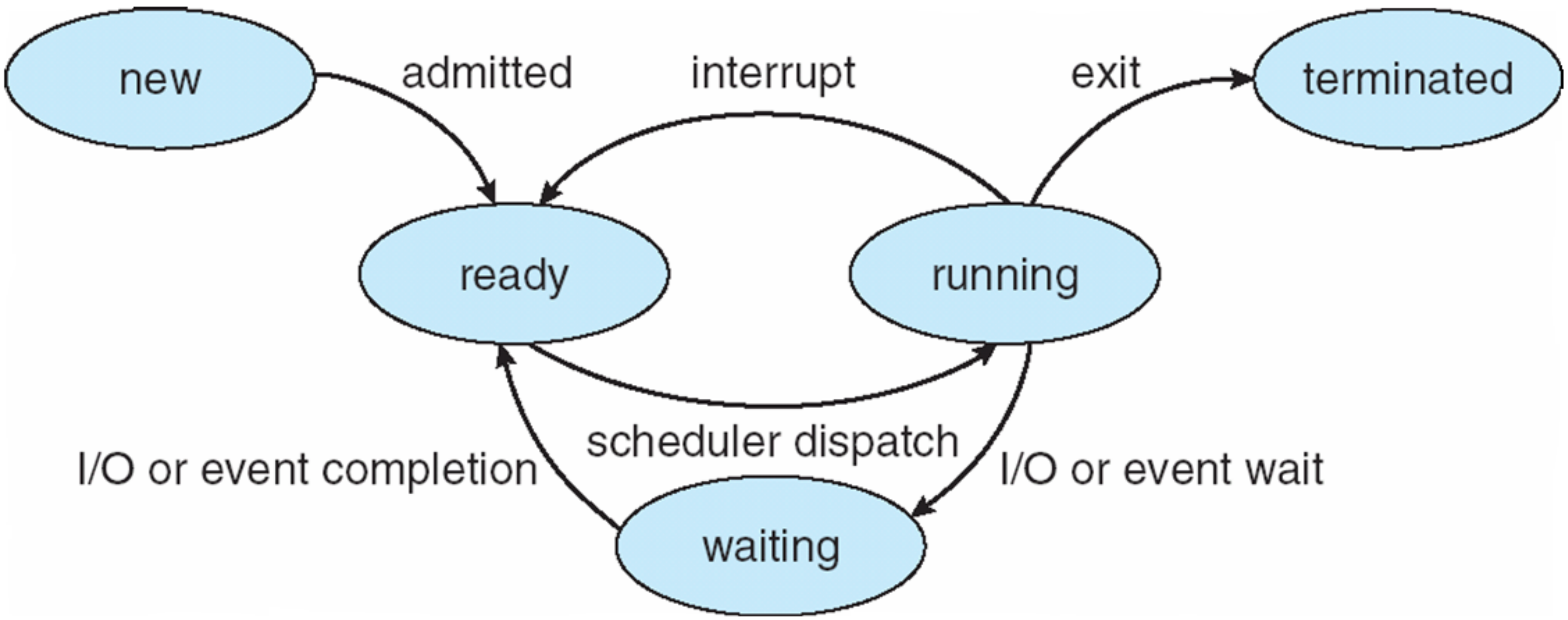
CSCI 3753
Operating Systems

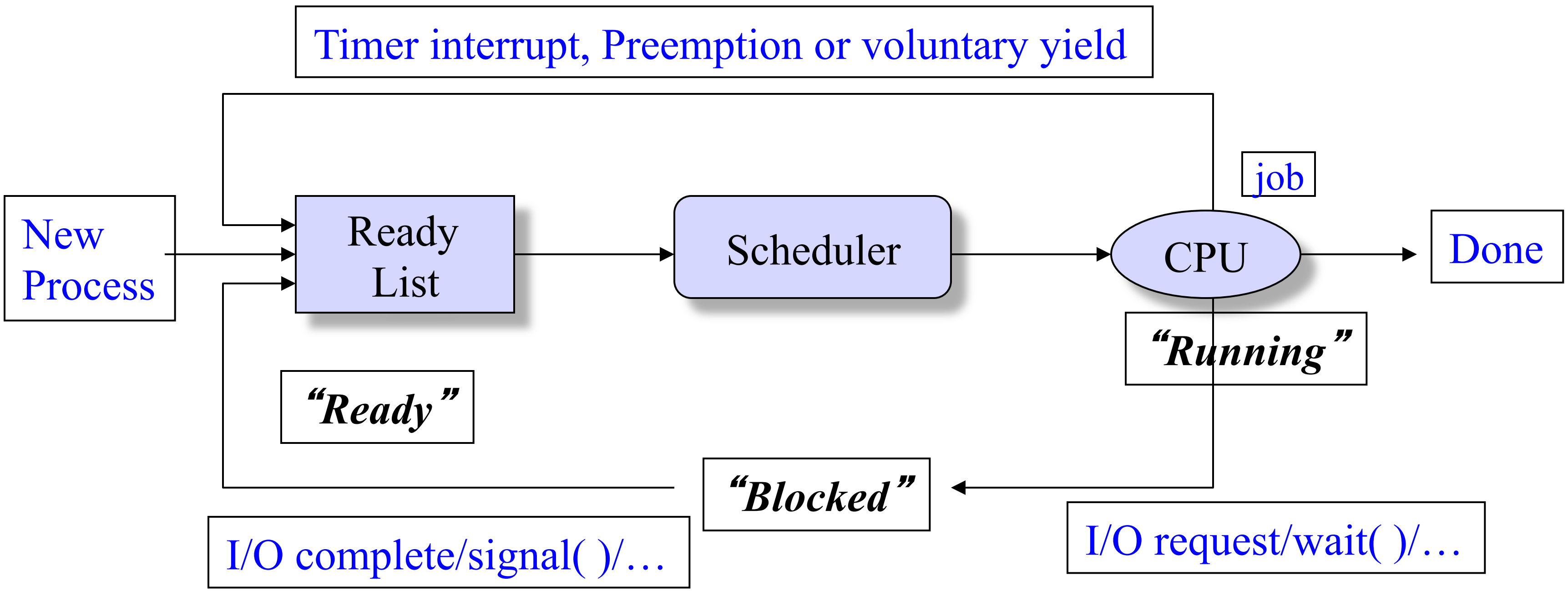
CPU Scheduling

Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder
Last Update: 02/15/13

Diagram of Process State



Also called “blocked” state



Process scheduling problem: When more than one processes are in Ready list, how does OS decide which process to run next?

- Carried out by short term scheduler or CPU scheduler

Scheduling Definitions

- *execution time* $E(P_i)$ = the time on the CPU required to fully execute process i
- *wait time* $W(P_i)$ = sum of the times process i spends in the ready state
- *turnaround time* $T(P_i)$ = the time from 1^{st} entry of process i into the system to its final exit from the system (exits last run state)
- *response time* $R(P_i)$ = the time from 1^{st} entry of process i into the ready queue to its 1^{st} scheduling on the CPU (1^{st} run state)
  - Some processes can generate early results, so if they get some CPU time quickly, they can start producing output sooner. A quick response time from the scheduler benefits such processes.

- *CPU utilization*: Percentage of time the CPU is busy
- *Throughput*: # of processes completed per time unit

Scheduling Goals

- Maximize CPU utilization: 40% to 90%
- Maximize throughput
- Minimize average or peak turnaround time
- Minimize average or peak waiting time
- Minimize average or peak response time
- Maximize fairness
- Meet deadlines or delay guarantees
- Ensure priorities are adhered to

Some of these goals are contradictory.
Any scheduling algorithm that favors
one class of jobs hurts another class of jobs.

Preemptive vs Non-preemptive Scheduling

- Non-preemptive scheduling
  - A running process keeps the CPU until terminating or switching to waiting state
  - A long-running CPU-bound process can prevent other processes from getting the CPU
- Preemptive scheduling
  - A running process may be forced to give up CPU to move to the Ready state
  - Relies on timer interrupts
  - Can result in race conditions among processes

Scheduling Analysis

- We analyze various scheduling policies to see how efficiently they perform with respect to metrics like:
  - Wait time, turnaround time, response time, etc.
- Some algorithms will be optimal in certain metrics
- To simplify analysis assume:
  - No blocking I/O. Focus only on scheduling processes/tasks that have provided their execution times
  - Processes execute until completion, unless otherwise noted, e.g round robin.

FCFS Scheduling

- First Come First Serve: order of arrival dictates order of scheduling
  - Non-preemptive, processes execute until completion
- If processes arrived in order P1, P2, P3 before time 0, then *Gantt chart* of CPU service time is:

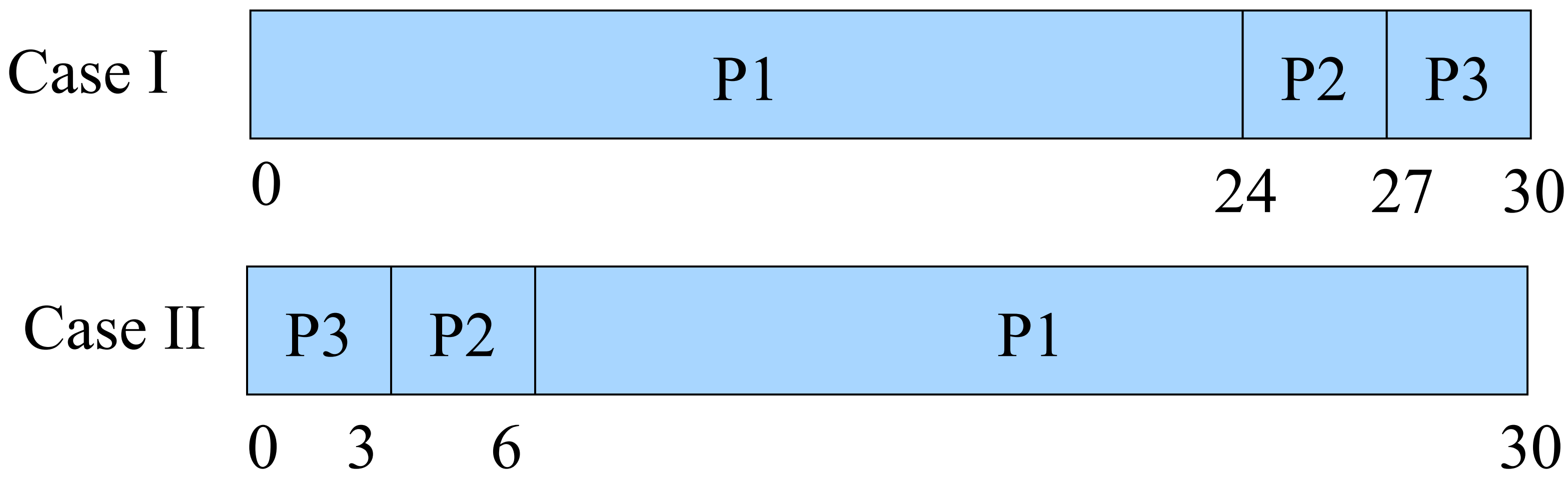
Process	CPU Service Time
P1	24
P2	3
P3	3

FCFS Scheduling (2)

- If processes arrive in reverse order P3, P2, P1 around time 0, then Gantt chart of CPU service time is:

Process	CPU Service Time
P1	24
P2	3
P3	3

FCFS Scheduling (3)



- Case I: average wait time is (0+24+27)/3 = 17 seconds
- Case II: average wait time is (0+3+6)/3 = 3 seconds
- FCFS wait times are generally not minimal - vary a lot if order of arrival changed, which is especially true if the process service times vary a lot (are spread out)
- Case I: average turnaround time is (24+27+30)/3 = 27 seconds
- Case II: average turnaround time is (3+6+30)/3 = 13 seconds
- A lot of variation in turnaround time too.

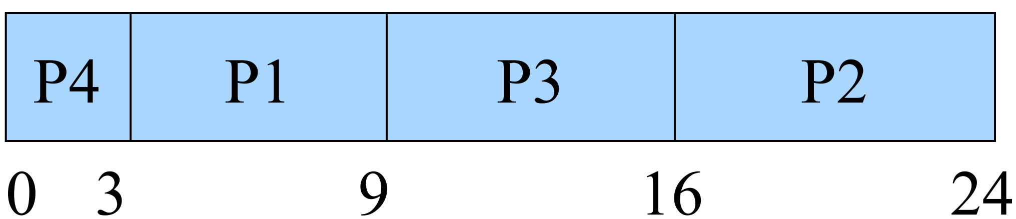
Shortest Job First (SJF) Scheduling

- Choose the process/thread with the lowest execution time
  - gives priority to shortest or briefest processes
  - minimizes the average wait time
    - Intuition: moving a long process before a short one increases the wait time of short processes a lot.
    - Conversely, moving long process to the end decreases wait time seen by short processes
- SJF minimizes the average wait time out of all possible scheduling policies.
- Problem: must know run times in advance unlike FCFS
  - Predict using exponential averages … (see textbook)

Shortest Job First Scheduling

- In this example, P1 through P4 are in ready queue at time 0:
  - *can prove SJF minimizes wait time* - out of 24 possibilities of ordering P1 through P4, the SJF ordering has the lowest average wait time

Process	CPU Execution Time
P1	6
P2	8
P3	7
P4	3



average wait time
$= (0+3+9+16)/4$
$= 7$ seconds

Shortest Job First Scheduling

- Can be preemptive
  - i.e. when a new job arrives in the ready queue, if its execution time is less than the currently executing job's remaining execution time, then it can preempt the current job
    - Shortest remaining time first
  - For simplicity, we assumed in the preceding analysis that jobs ran to completion and no new jobs arrived until the current set had finished.
  - Compare to FCFS: a new process can't preempt earlier processes, because its order is later than the earlier processes

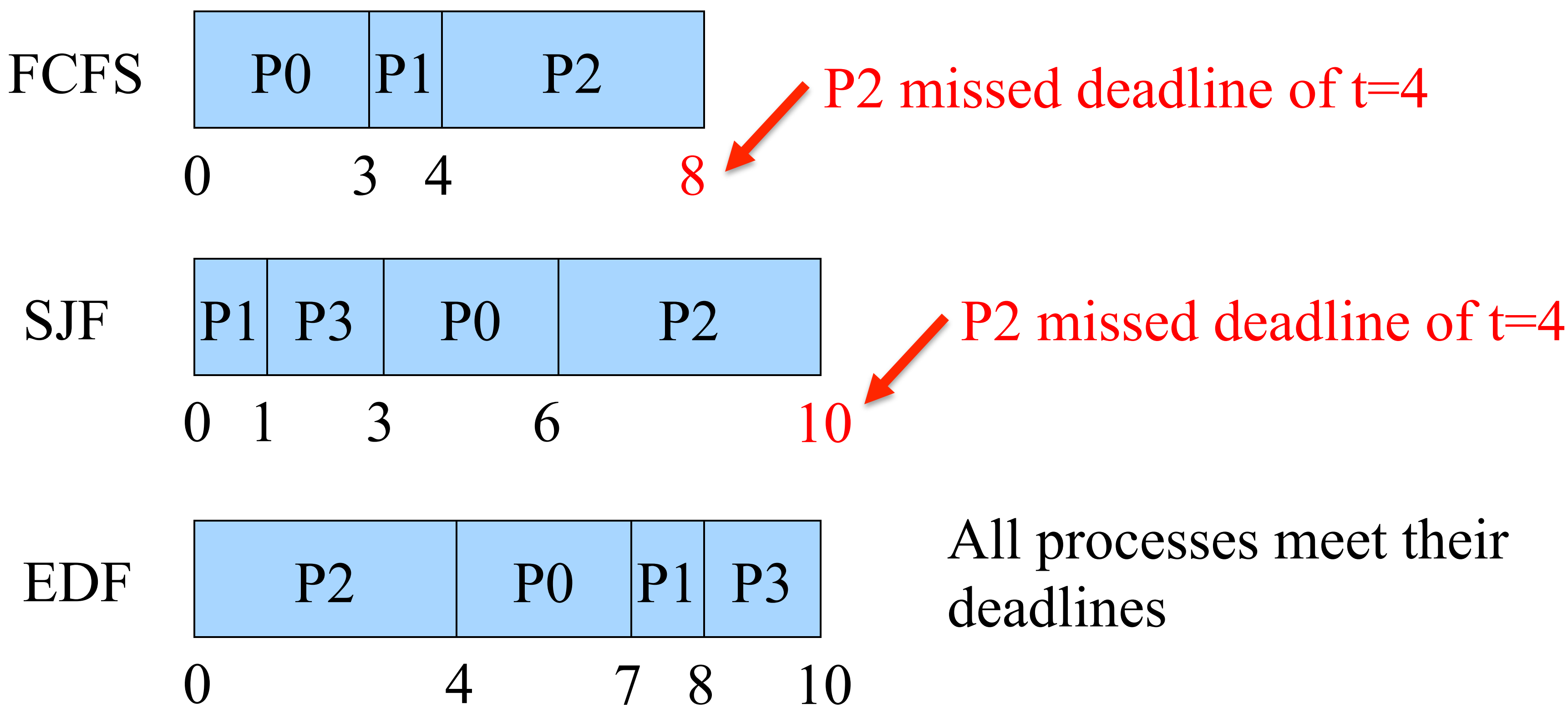
Deadline Scheduling

- Hard real time systems require that certain processes *must* complete execution within a certain time, or the system crashes
  - e.g. robots need a real time OS (RTOS) whose processes (actuating an arm/leg) must be scheduled by a certain deadline

Process	CPU Execution Time	Deadline from now
P0	3	7
P1	1	9
P2	4	4
P3	2	10

Deadline Scheduling

- *Earliest deadline first* (EDF) selects the process with the nearest/soonest deadline
  - This is the process that most urgently needs to be completed



Deadline Scheduling

- Even EDF may not be able to meet all deadlines:
  - In previous example, if P3's deadline was t=9, then EDF cannot meet P3's deadline
- Admission control policy
  - Check on entry to system whether a process's deadline can be met, given the current set of processes already in the ready queue and their deadlines
    - If all deadlines can be met with the new process, then admit it
    - Else, deny admission to this process if its deadline can't be met. Note FCFS or SJF had no notion of refusing admission

Deadline Scheduling

- Admission control used when scheduling policies try to provide different Qualities of Service (QOS)
  - It’s common in network-based QOS scheduling policies for routers – can’t admit a new source of packets if its QOS deadlines or guarantees cannot be met at a router
- EDF can be preemptive
  - A process that arrives with an earlier deadline can preempt one currently executing with a later deadline.

Round Robin Scheduling

- The CPU scheduler rotates among the processes in the ready queue, giving each a time slice
  - e.g. if there are 3 processes P1, P2, & P3, then the scheduler will keep rotating among the three: P1, P2, P3, P1, P2, P3, P1, …
  - treats the ready queue as a circular queue
  - useful for time sharing multitasking systems and therefore is a popular scheduling algorithm

Round Robin Scheduling

- Simple and fair, though wait times can be long
  - Fair: If there are n processes, each process gets 1/n of CPU
  - Simple: Don't need to know service times a priori
- RR is an example of preemptive scheduling – a process may be forced to relinquish CPU before it's done
  - This was not the case for FCFS, and only occurred when new processes arrived for SJF and EDF. Now, we allow timer-based interrupts.
- A process can finish before its time slice is up. The scheduler just selects the next process in the queue

Round Robin Scheduling

- Example: suppose we use a time slice of 4 ms, and ignoring context switch overhead
- Now P1 is time sliced out, and P2 and P3 are allowed to run sooner than FCFS
- average response time is fast at 3.66 ms
  - Compare to FCFS w/ long 1st process

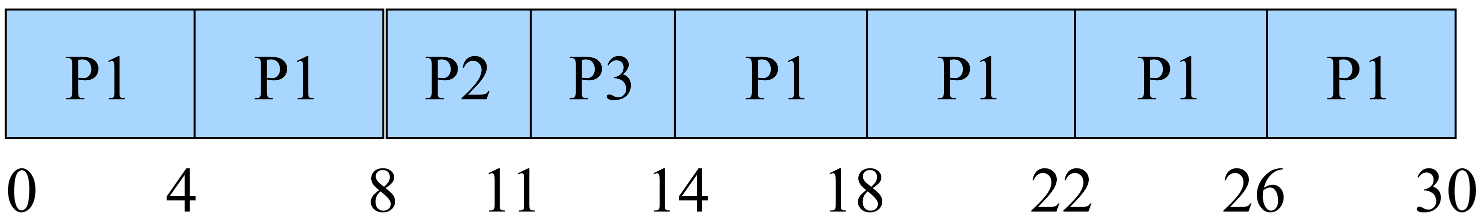
Process	CPU Execution Time (ms)
P1	24
P2	3
P3	3

P1	P2	P3	P1	P1	P1	P1	P1

0 4 7 10 14 18 22 26 30

Round Robin Scheduling

- Weighted Round Robin - each process is given some number of time slices, not just one per round
  - In previous example, could give P1 2 time slices, and P2 and P3 only 1 each round



Round Robin Scheduling

- Weighted Round Robin is a way to provide preferences or priorities even with preemptive time slicing
  - Example: If 3 processes all want a great deal of compute time, & OS gives P1 2 time slots per round, P2 1 time slot/round, and P3 4 time slots/round, then in steady state, P1 gets 2/7 of CPU bandwidth, P2 gets 1/7 of CPU, and P3 gets 4/7 of CPU
  - In general, if process P_i gets N_i slots per round, then the fraction α_i of the CPU bandwidth that process i gets in steady state in WRR is:

$$\alpha_\iota = \frac{N_i}{\sum_i N_i}$$