Principles of Operating Systems

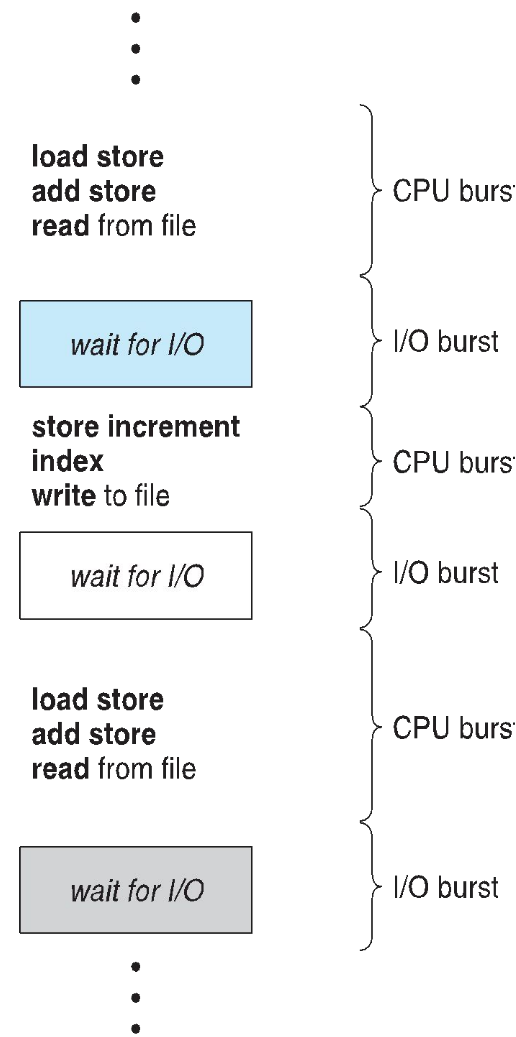
Lecture 3 - CPU Scheduling

Ardalan Amiri Sani (ardalan@uci.edu)

[*lecture slides contains some content adapted from previous slides by Prof. Nalini Venkatasubramanian, and course text slides © Silberschatz*]

Motivation

- CPU-I/O Burst Cycle
  - Process execution consists of a cycle of CPU execution and I/O wait.



Schedulers

- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (can be slow)
  - The long-term scheduler controls the **degree of multiprogramming**

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (microseconds/milliseconds) ⇒ (must be fast)

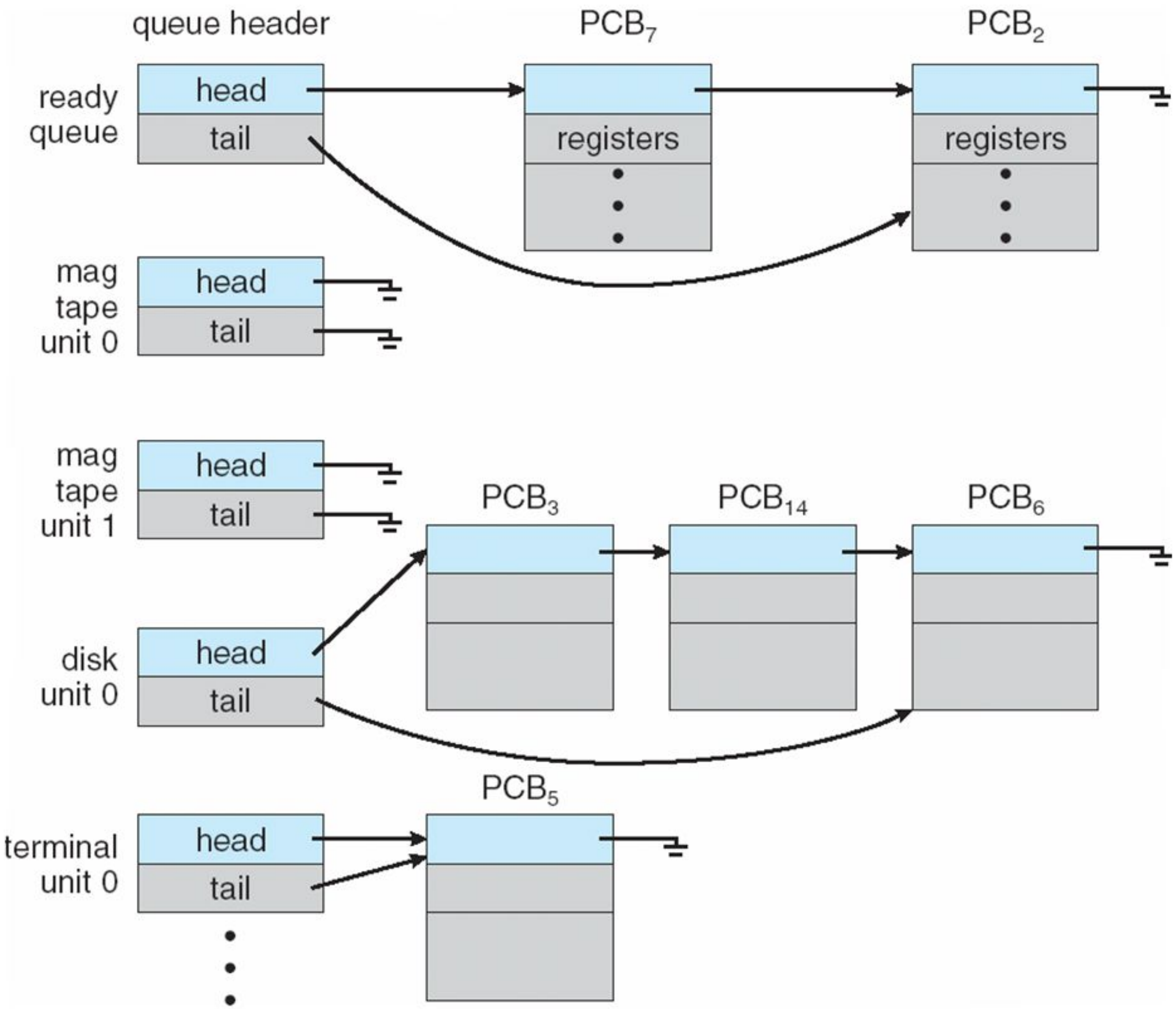
Schedulers

- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations; has short CPU bursts
  - **CPU-bound process** – spends more time doing computations; has long CPU bursts
- Long-term scheduler strives for good ***process mix***

CPU Scheduler

- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



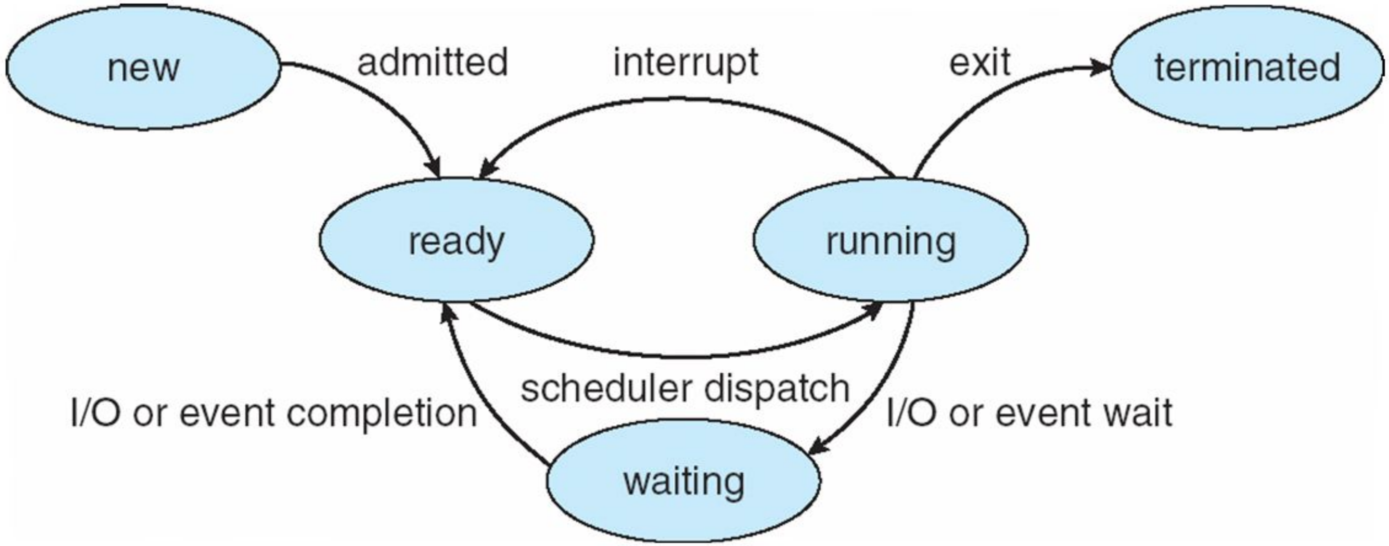
Different types of CPU Scheduling

- Non-preemptive Scheduling
  - Once CPU has been allocated to a process, the process keeps the CPU until
    - Process exits OR
    - Process switches to waiting state
- Preemptive Scheduling
  - Process can be interrupted and must release the CPU.

CPU scheduling decisions

- CPU scheduling decisions may take place when a process:
    a. switches from running state to waiting state
    b. switches from running state to ready state
    c. switches from waiting to ready
    d. terminates
    e. is admitted
- Scheduling under (a) and (d) is non-preemptive.
- All other scheduling is preemptive.

CPU scheduling decisions



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.
    - Dispatcher must be fast.

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 from its entry time.

Scheduling Criteria (cont.)

- Waiting time
  - amount of time a process has been waiting in the ready queue.
- Response Time (in a time-sharing environment)
  - amount of time it takes from when a request was submitted until the first response (and NOT the final output) is produced.

Optimization Criteria

- Maximize CPU Utilization
- Maximize Throughput
- Minimize Turnaround time
- Minimize Waiting time
- Minimize response time

First-Come, First-Served (FCFS) Scheduling

- Policy: Process that requests the CPU *FIRST* is allocated the CPU *FIRST.*
  - FCFS is a non-preemptive algorithm.
- Implementation - using FIFO queues
  - incoming process is added to the tail of the queue.
  - Process selected for execution is taken from head of queue.
- Gantt Charts are used to visualize schedules.

First-Come, First-Served (FCFS) Scheduling

- Example

Process	Burst Time
P1	24
P2	3
P3	3

Gantt Chart for Schedule

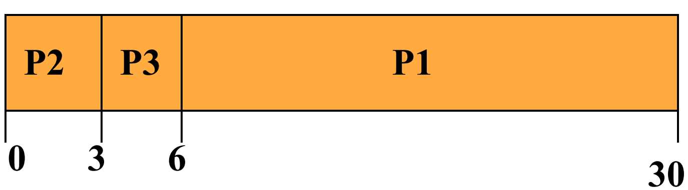
- Suppose all processes arrived at around time 0 in the following order:
  - P1, P2, P3
- Waiting time
  - P1 = 0;
  - P2 = 24;
  - P3 = 27;
- Average waiting time
  - (0+24+27)/3 = 17

FCFS Scheduling (cont.)

- Example

Process	Burst Time
P1	24
P2	3
P3	3

Gantt Chart for Schedule



- Suppose the arrival order for the processes is
  - P2, P3, P1
- Waiting time
  - P1 = 6; P2 = 0; P3 = 3;
- Average waiting time
  - (6+0+3)/3 = 3 , better..
- *Convoy Effect*:
  - short processes waiting behind long process, e.g., one CPU bound process, many I/O bound processes.

Shortest-Job-First (SJF) Scheduling

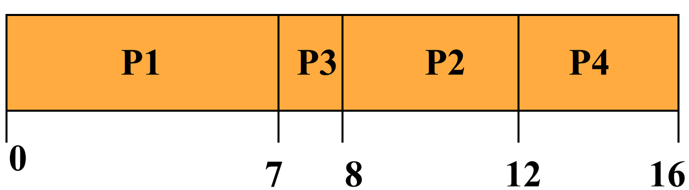
- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Two Schemes:
  - Scheme 1: Non-preemptive
    - Once CPU is given to the process it cannot be preempted until it completes its CPU burst.
  - Scheme 2: Preemptive
    - If a new CPU process arrives with CPU burst length less than remaining time of current executing process, preempt. Also called Shortest-Remaining-Time-First (SRTF).
  - 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

Non-Preemptive SJF Scheduling

- Example

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

Gantt Chart for Schedule

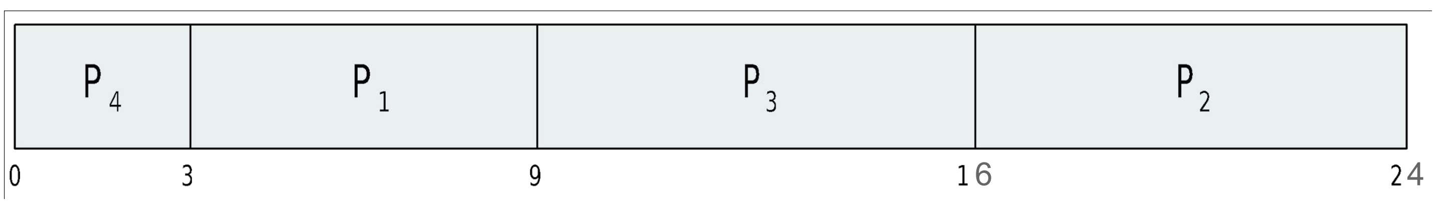


Average waiting time = (0+6+3+7)/4 = 4

Non-Preemptive SJF Scheduling

Process	Arrival Time	Burst Time
P_1	0	6
P_2	2	8
P_3	5	7
P_4	0	3

- SJF Gantt chart



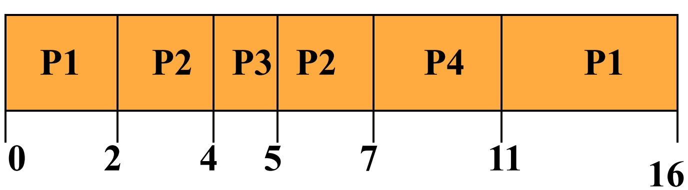
- Average waiting time = ((3-0) + (16-2) + (9-5) + 0) / 4 = 5.25
- Average turnaround time = ((9-0) + (24-2) + (16-5) + (3-0))/4 = 11.25

Preemptive SJF Scheduling (SRTF)

- Example

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

Gantt Chart for Schedule

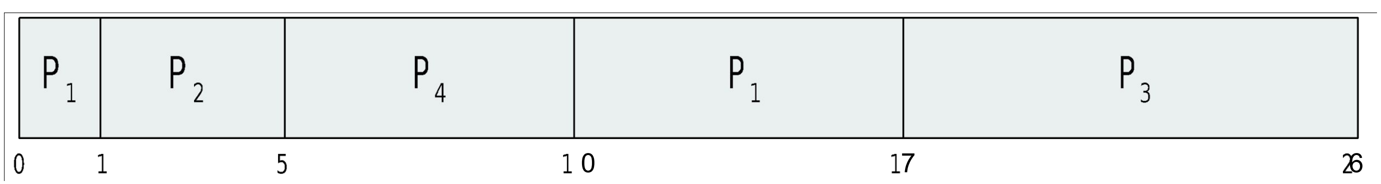


Average waiting time =
(9+1+0+2)/4 = 3

Preemptive SJF Scheduling (SRTF)

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart

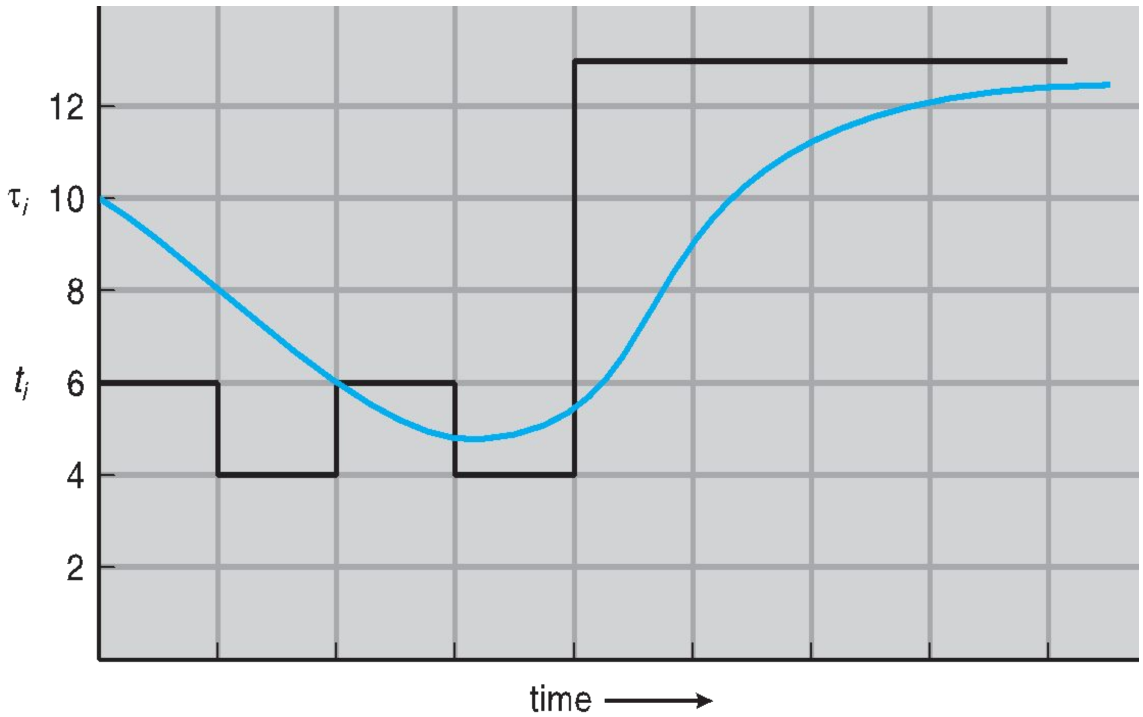


- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec
- Average turnaround time = ((17-0) + (5-1) + (26-2) + (10-3))/4 = 13 msec

Determining Length of Next CPU Burst

- One can only estimate the length of burst.
- Use the length of previous CPU bursts and perform exponential averaging.
    - t_n = actual length of nth burst
    - τ_{n+1} = predicted value for the next CPU burst
    - $\alpha = 0,\ 0 \le \alpha \le 1$
    - Define
        - $\tau_{n+1} = \alpha\, t_n + (1- \alpha)\, \tau_n$

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	. . .

Exponential Averaging(cont.)

- $\alpha = 0$
  - $T_{n+1} = T_n$;   Recent history does not count
- $\alpha = 1$
  - $T_{n+1} = t_n$; Only the actual last CPU burst counts.
- Expanding the formula:
  - $T_{n+1} = \alpha t_n + (1-\alpha)\, \alpha t_{n-1} + \ldots + (1-\alpha)^j\, \alpha t_{n-j} + \ldots + (1-\alpha)^{(n+1)}\, T_0$
    - Each successive term has less weight than its predecessor.
- Commonly, α is set to 0.5

Priority Scheduling

- A priority value (integer) is associated with each process. Can be based on
  - Cost to user
  - Importance to user
  - Aging
  - %CPU time used in last X hours.
- CPU is allocated to the process with the highest priority.
  - Preemptive
  - Nonpreemptive

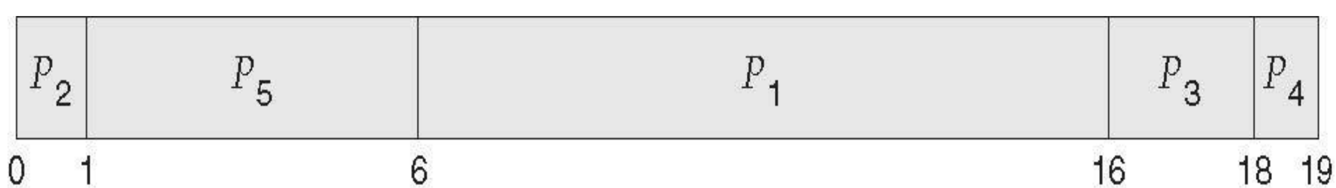
Priority Scheduling (cont.)

- SJF is a priority scheme where the 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.

Priority Scheduling - Non-preemptive

ProcessA	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Assume all arrival times to be 0 when not specified
- Priority scheduling Gantt Chart

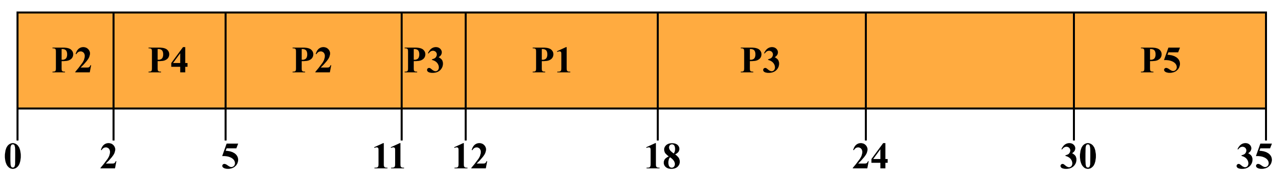


- Average waiting time = (6 + 0 + 16 + 18 + 1)/5 = 8.2 msec

Priority Scheduling - Preemptive

ProcessA	Burst Time	Priority	Arrival Time
P_1	6	3	12
P_2	8	2	0
P_3	7	4	4
P_4	3	1	2
P_5	5	5	30

- Gantt Chart



- Average waiting time = (0+3+(7+6)+0+0)/5 = 16/5 = 3.2 msec
- Average turnaround time = (6 + 11 + 20 + 3 + 5)/5 = 45/5 = 9 msec
- Average response time (assuming immediate response by a process when executed) = (0 + 0 + 7 + 0 + 0) / 5 = 1.4 msec
- CPU utilization = 29 / 35 = 0.83 = 83%
- Throughput = 5 / 35 = 0.14 #process/msec

Project note

- Asks to implement priority scheduling
  - Preemptive or non-preemptive?

Project note

- Asks to implement priority scheduling
    - Preemptive or non-preemptive?
        - Preemptive
- New concept: ***priority donation***
    - A high-priority thread donates its priority to a low-priority thread
    - Why?

Project note

- Asks to implement priority scheduling
  - Preemptive or non-preemptive?
    - Preemptive
- New concept: ***priority donation***
  - A high-priority thread donates its priority to a low-priority thread
  - Why?
    - To address ***priority inversion***, which can happen if a higher priority thread needs to wait for a lower priority thread to release a resource, e.g., a lock.

Round Robin (RR)

- 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.
- *n* processes, time quantum = *q*
  - Each process gets 1/*n* CPU time in chunks of at most *q* time units at a time.
  - No process waits more than (*n*-1)*q* time units before it can run (note that the overall wait time can be higher).
  - Performance
    - Time slice *q* too large - ?

Round Robin (RR)

- 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.
- n processes, time quantum = q
  - Each process gets $1/n$ CPU time in chunks of at most q time units at a time.
  - No process waits more than $(n-1)q$ time units before it can run (note that the overall wait time can be higher).
  - Performance
    - Time slice q too large - FIFO behavior
    - Time slice q too small - ?

Round Robin (RR)

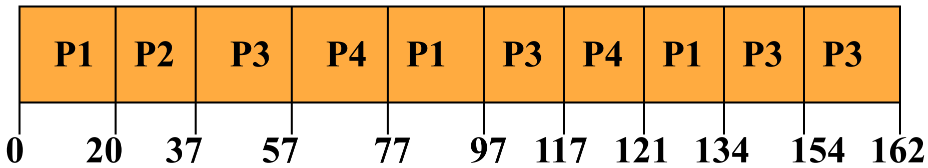
- Each process gets a small unit of CPU time
  - Time quantum usually 10-100 milliseconds.
  - After this time has elapsed, the process is preempted and added to the end of the ready queue.
- *n* processes, time quantum = *q*
  - Each process gets 1/*n* CPU time in chunks of at most *q* time units at a time.
  - No process waits more than (*n*-1)*q* time units before it can run (note that the overall wait time can be higher).
  - Performance
    - Time slice *q* too large - FIFO behavior
    - Time slice *q* too small - Overhead of context switch is too expensive.
    - Heuristic - 70-80% of CPU bursts within timeslice

Round Robin Example

- Time Quantum = 20

Process	Burst Time
P1	53
P2	17
P3	68
P4	24

Gantt Chart for Schedule

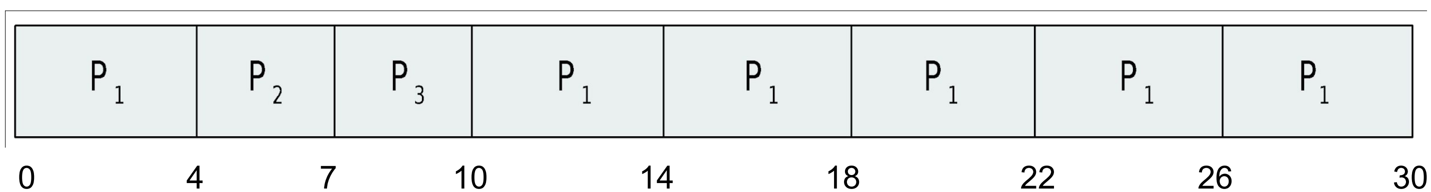


Typically, higher average turnaround time than SRTF, but better response time

Round Robin Example

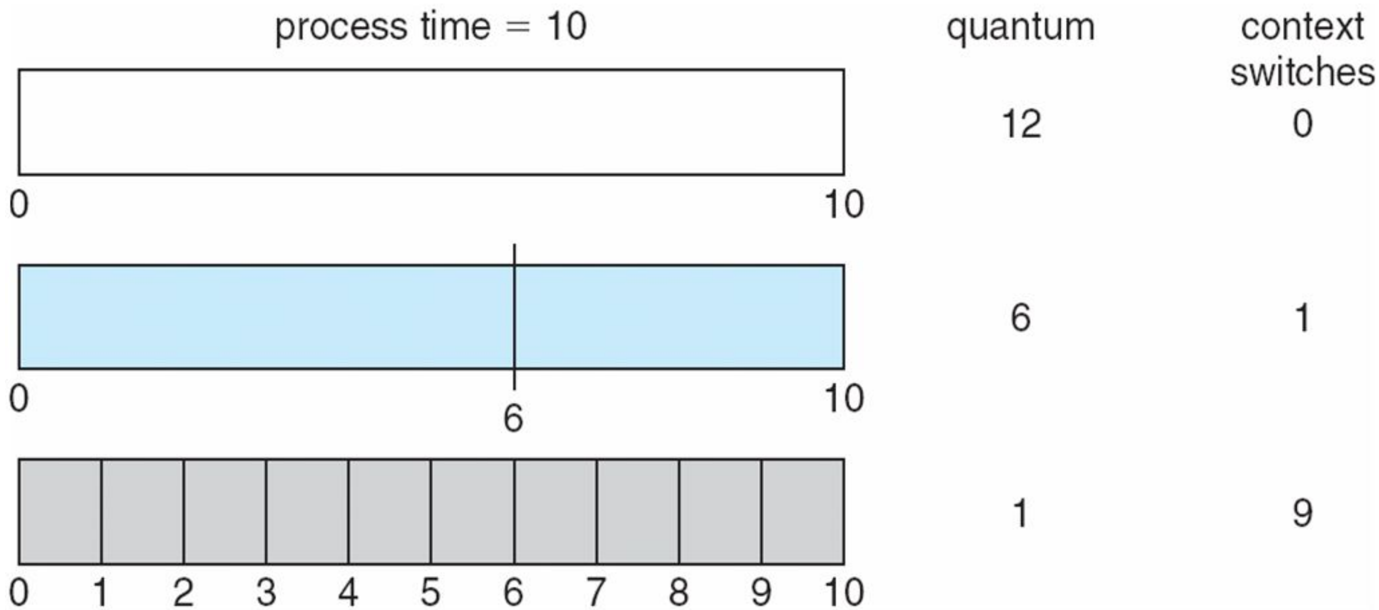
Process	Burst Time
P_1	24
P_2	3
P_3	3

- Assume all arrive at time 0 in the following order: P_1, P_2, P_3
- The Gantt chart is (quantum = 4):

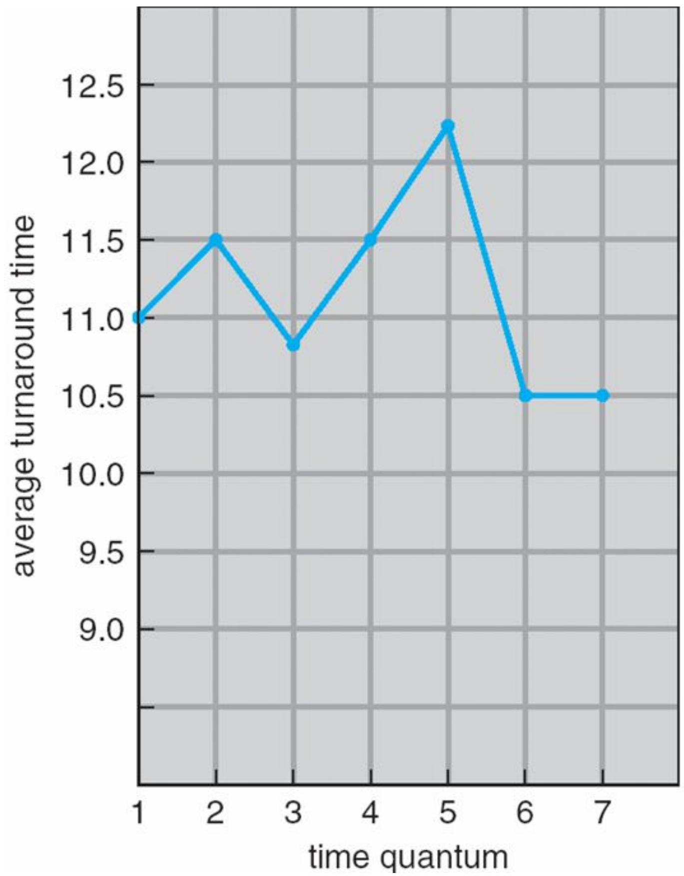


- Average waiting time = (6 + 4 + 7)/3 = 5.67
- Average turnaround time = (30 + 7 + 10)/3 = 15.67

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



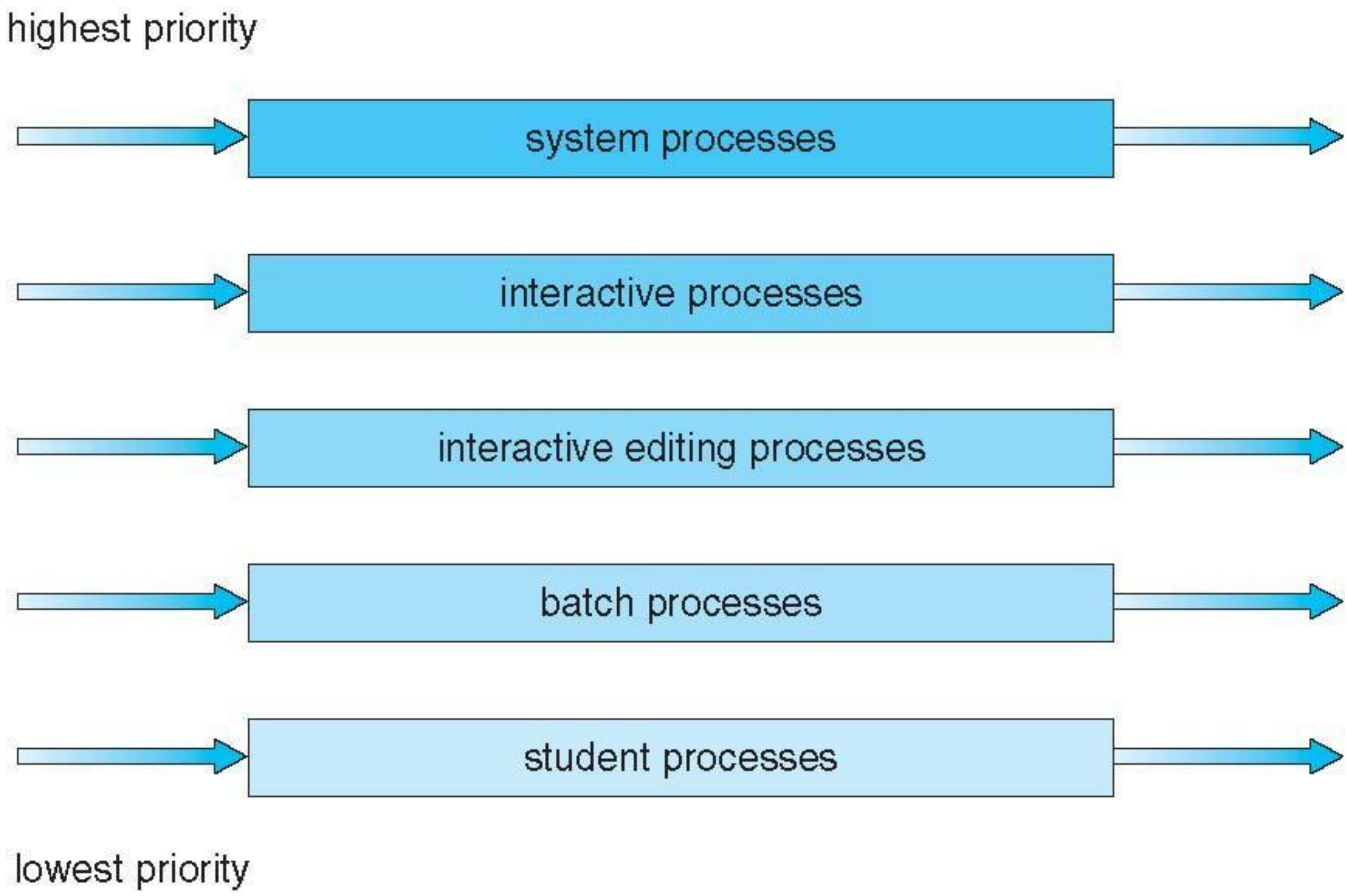
process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts
should be shorter than q

Multilevel Queue

- Ready Queue partitioned into separate queues
  - Example: system processes, foreground (interactive), background (batch), student processes….
- Each queue has its own scheduling algorithm
  - Example: foreground (RR), background (FCFS)
- Processes assigned to one queue permanently.
- Scheduling must be done between the queues
  - Fixed priority - serve all from foreground, then from background. Possibility of starvation.
  - Time slice - Each queue gets some CPU time that it schedules - e.g. 80% foreground (RR), 20% background (FCFS)

Multilevel Queue scheduling

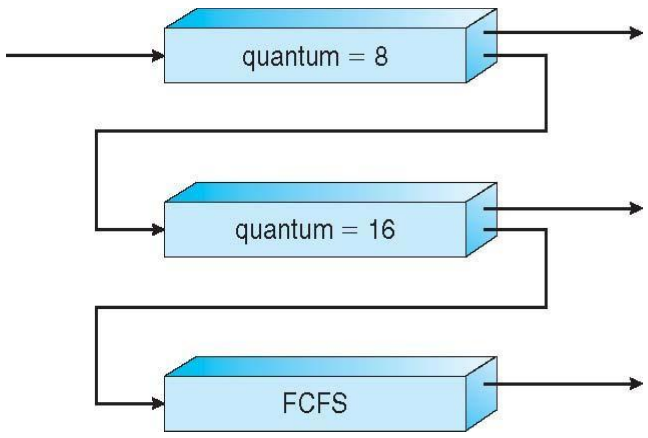


Multilevel Feedback Queue

- A process can *move* between the queues.
  - Aging can be implemented this way.
- Parameters for a multilevel feedback queue scheduler:
  - number of queues.
  - scheduling algorithm for each queue and between queues.
  - 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 -
  - Q0 - RR with time quantum 8 milliseconds
  - Q1 - RR with time quantum 16 milliseconds
  - Q2 - FCFS
- Scheduling
  - New job enters Q0 - When it gains CPU, it receives 8 milliseconds. If job does not finish, move it to Q1.
  - At Q1, when job gains CPU, it receives 16 more milliseconds. If job does not complete, it is moved to queue Q2.



Project note: multilevel feedback queue in 4.4BSD

- 64 priories, hence 64 queues
- The scheduler selects a process from the highest priority queue that is not empty
- Priority calculated based on a “nice” value and the recent CPU time usage
  - Higher nice means lower priority
  - More recent CPU time means lower priority
- No priority donation
- Priorities re-calculated every once in a while
- Each queue uses round-robin for its own scheduling

Source: pintos documents: https://www.ics.uci.edu/~ardalan/courses/os/pintos/pintos_8.html#SEC141

Thread Scheduling

- When threads supported, threads scheduled, not processes
- The CPU scheduler schedules the kernel threads.

Multiple-Processor Scheduling

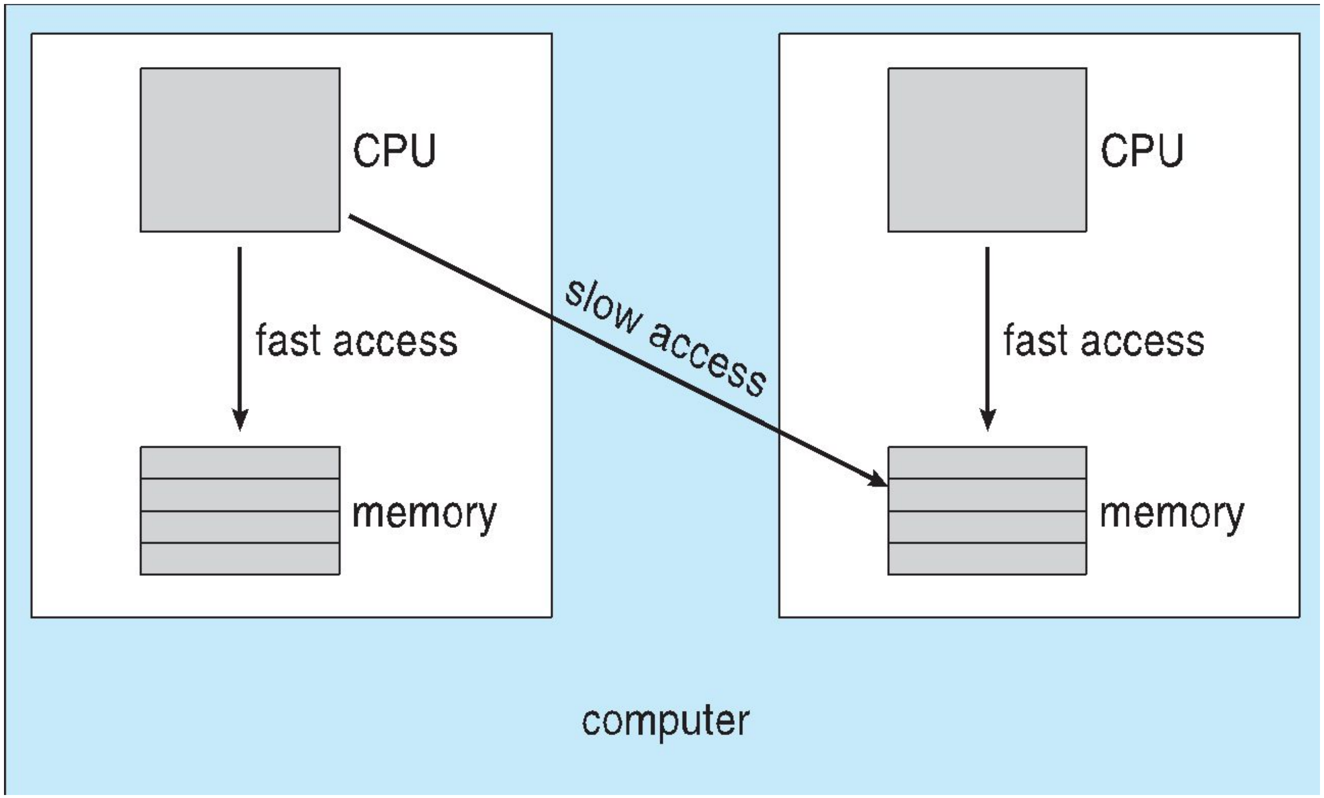
- CPU scheduling becomes more complex when multiple processors are available.

- Symmetric multiprocessing (SMP)
  - Self scheduled - each processor dispatches a job from ready queue
  - All processes in common ready queue, or each processor has its own private queue of ready processes
  - Homogeneous processors within multiprocessor
  - Currently, most common multiprocessor setup
  - Difficulties: access to shared data structure, making sure all processes are scheduled and that no process is scheduled by separate processors
- Asymmetric multiprocessing
  - One main processor schedules the other processors
  - only 1 processor accesses the system data structures, alleviating the need for data sharing

Multicore Processors

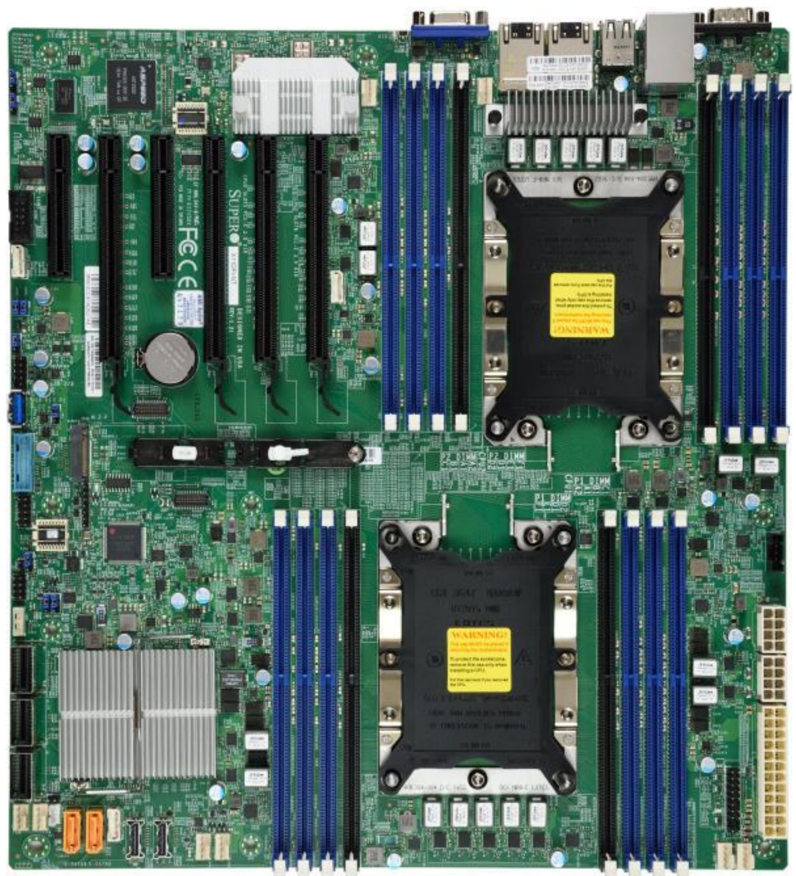
- Place multiple processor cores on same physical chip
- Faster and consumes less power

NUMA and CPU Scheduling: considers processor affinity



Note that memory-placement algorithms can also consider affinity

NUMA and CPU Scheduling: considers processor affinity



Supermicro X11DPi-N motherboard

Image source: https://www.supermicro.com/products/motherboard/Xeon/C620/X11DPi-N.cfm

Hyperthreading

- Multiple threads per core
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
- One CPU core looks like two cores to the operating system with hyperthreading

Hyperthreading