Chapter 6: CPU Scheduling

Chapter 6: CPU Scheduling

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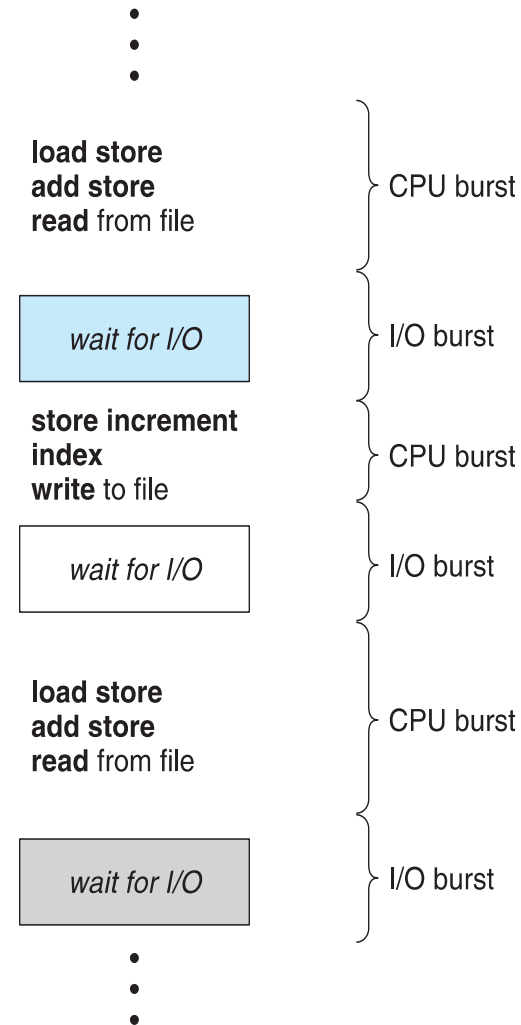- Algorithm Evaluation

Objectives

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems

- To describe various CPU-scheduling algorithms

- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

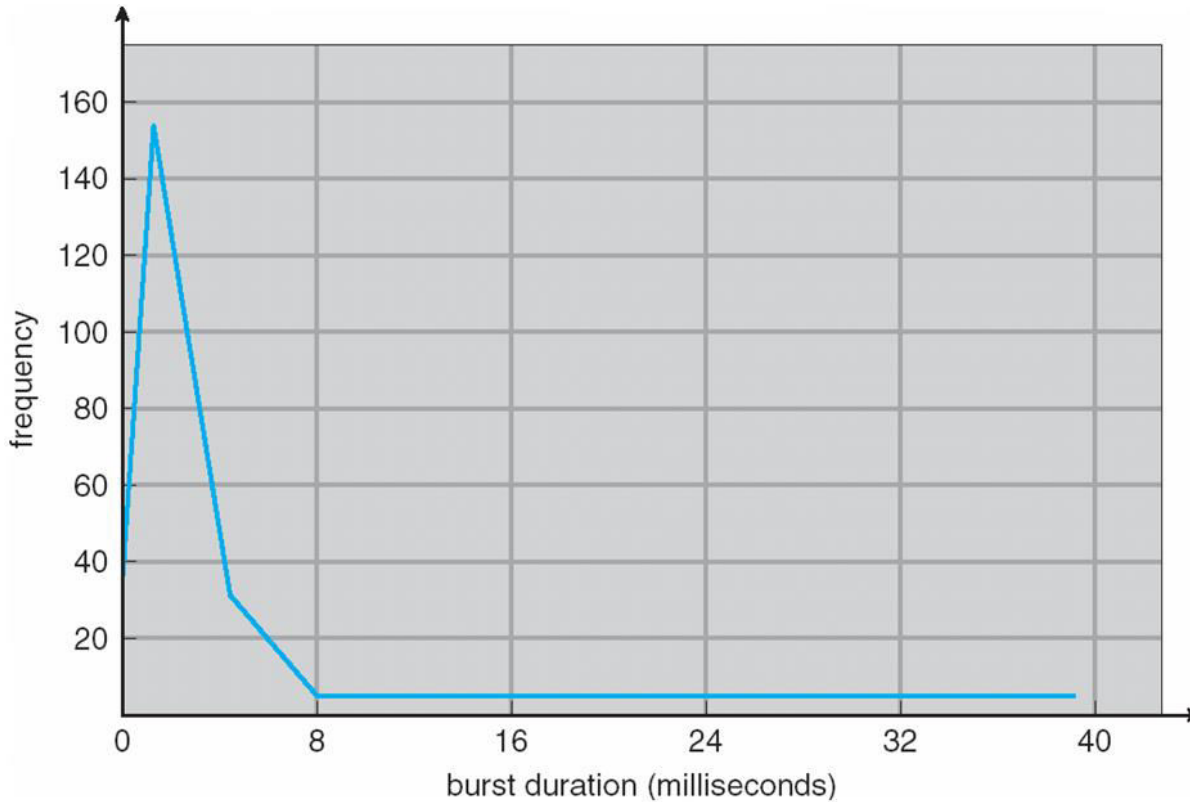- To examine the scheduling algorithms of several operating systems

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

⋮

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

⋮

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**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider 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

- **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)

Scheduling Algorithm Optimization Criteria

- Max CPU utilization

- Max throughput

- Min turnaround time

- Min waiting time

- Min response time

First- Come, First-Served (FCFS) Scheduling

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
The Gantt Chart for the schedule is:

P_1		P_2	P_3
0	24	27	30

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

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2, P_3, P_1$$

☐ The Gantt chart for the schedule is:

P_2	P_3	P_1
0 3	6	30

☐ 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

    ☐ Consider one CPU-bound and 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

- 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
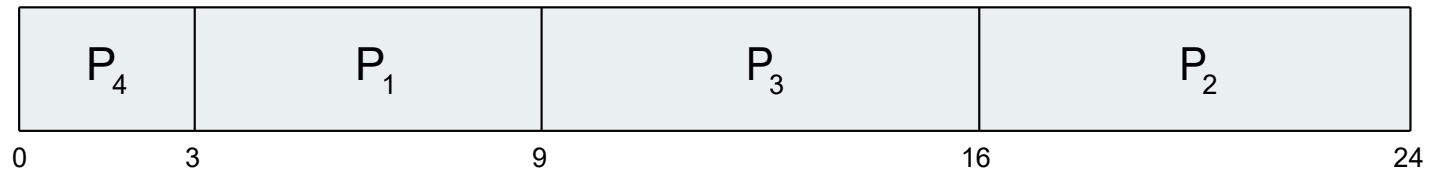
    - Could ask the user

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart

P$_4$	P$_1$	P$_3$	P$_2$

0   3   9   16   24
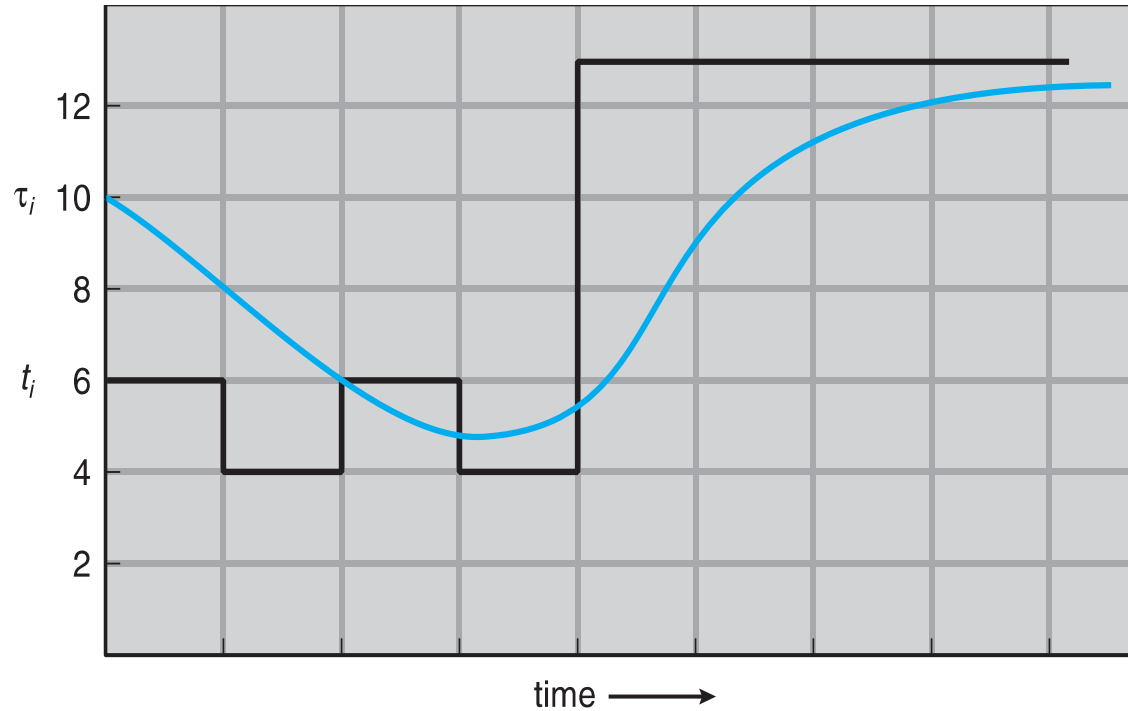
- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one

  - Then pick process with shortest predicted next CPU burst

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

  1. t_n = actual length of nth CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. $\alpha, 0 \le \alpha \le 1$
  4. Define : $\tau_{n=1} = \alpha\, t_n + (1 - \alpha)\tau_n.$

- Commonly, α set to ½

- Preemptive version called **shortest-remaining-time-first**

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

Examples of Exponential Averaging

- $\alpha = 0$
    - $\tau_{n+1} = \tau_n$
    - Recent history does not count

- $\alpha = 1$
    - $\tau_{n+1} = \alpha\, t_n$
    - 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
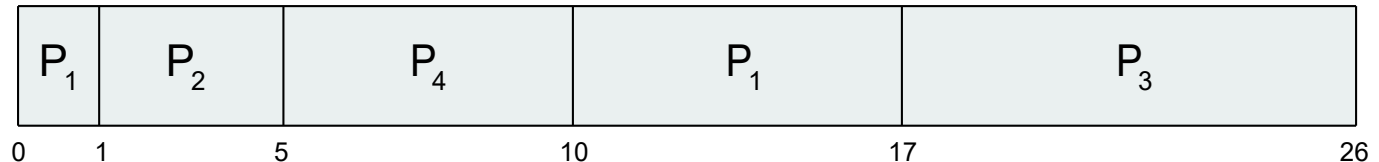
Example of Shortest-remaining-time-first

☐ Now we add the concepts of varying arrival times and preemption to the analysis

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

P_1	P_2	P_4	P_1	P_3
0 1	5		10	17 26

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

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 priority scheduling where priority is the inverse of predicted next CPU burst time

- Problem ≡ **Starvation** – low priority processes may never execute

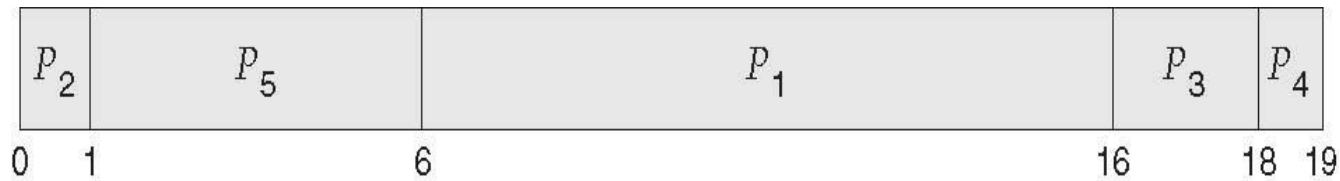- Solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

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

- Priority scheduling Gantt Chart

- Average waiting time = 8.2 msec

Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** q), 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.

- Timer interrupts every quantum to schedule next process

- Performance

  - q large $\Rightarrow$ FIFO

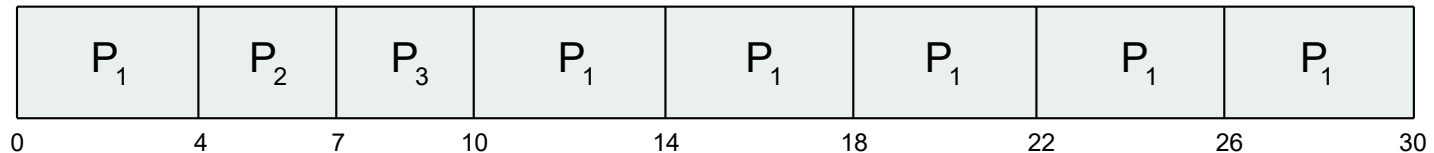 - q small $\Rightarrow q$ must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

☐ The Gantt chart is:

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0   4   7   10   14   18   22   26   30

☐ Typically, higher average turnaround than SJF, but better *response*

☐ q should be large compared to context switch time
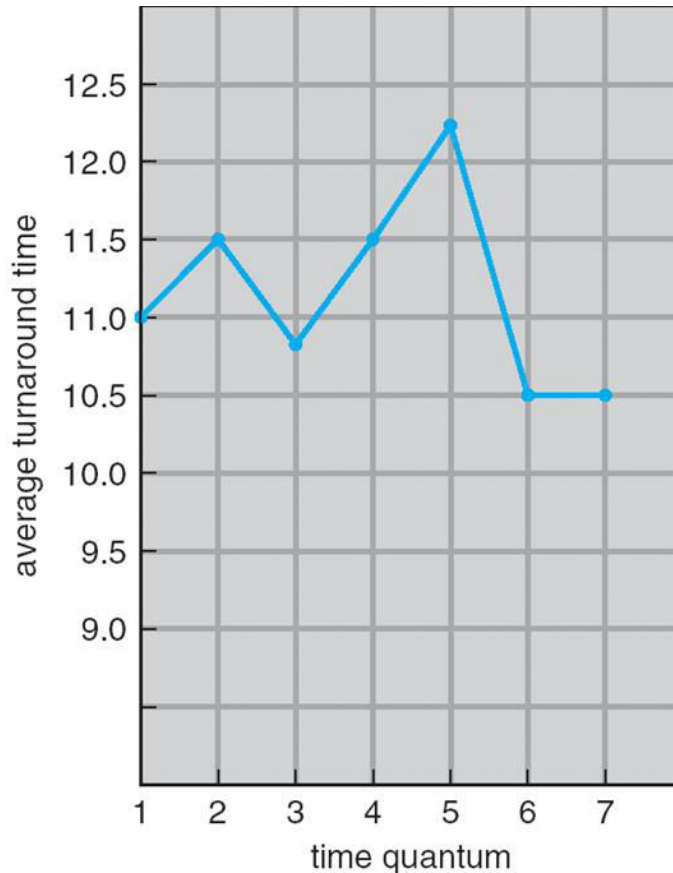
☐ q usually 10ms to 100ms, context switch < 10 usec

process time = 10

quantum	context switches
12	0
6	1
1	9
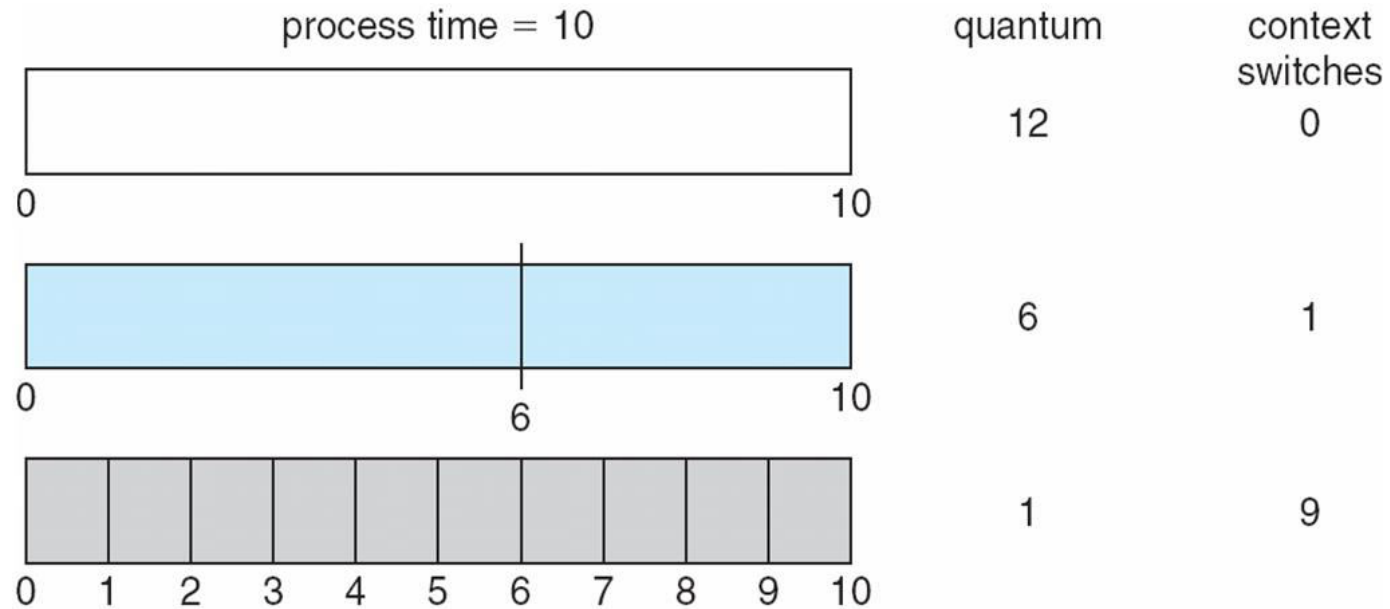
Turnaround Time Varies With The Time Quantum

| 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 is partitioned into separate queues, eg:
    - **foreground** (interactive)
    - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
    - foreground – RR
    - background – FCFS
- Scheduling must be done between the queues:
    - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
    - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
    - 20% to background in FCFS

Multilevel Queue Scheduling

highest priority

→ system processes →

→ interactive processes →

→ interactive editing processes →

→ batch processes →

→ student processes →
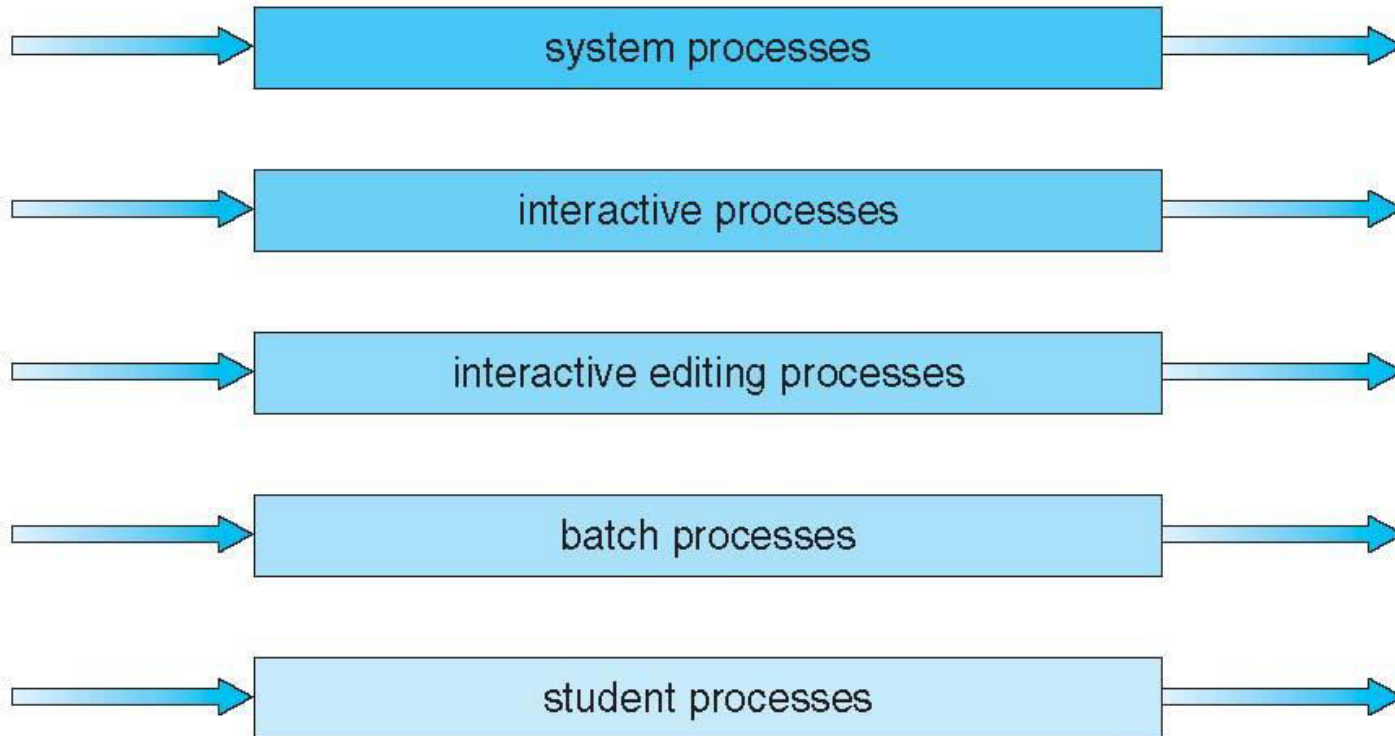
lowest priority

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

Example of Multilevel Feedback Queue

- 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 FCFS
    - 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 FCFS and receives 16 additional milliseconds
   - If it still does not complete, it is preempted and moved to queue Q_2
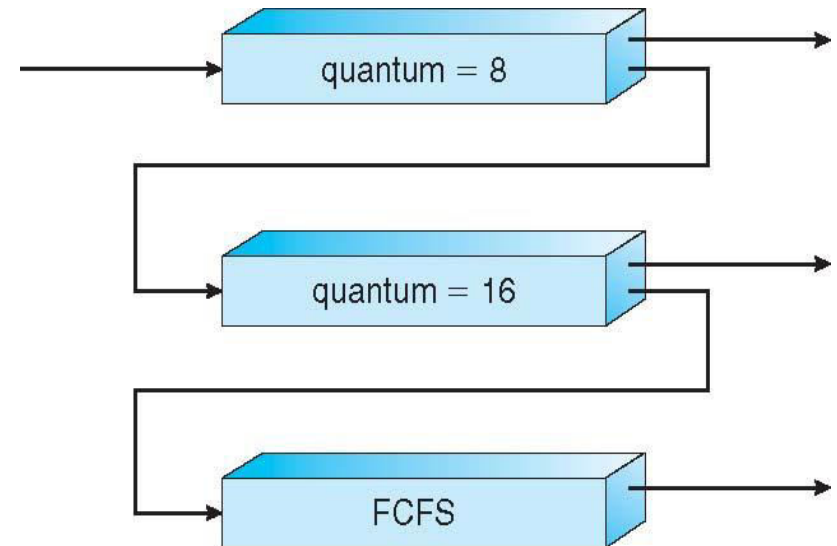
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available

- **Homogeneous processors** within a multiprocessor

- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing

- **Symmetric multiprocessing** (**SMP**) – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes

  - Currently, most common

- **Processor affinity** – process has affinity for processor on which it is currently running

  - **soft affinity**

  - **hard affinity**

  - Variations including **processor sets**

NUMA and CPU Scheduling
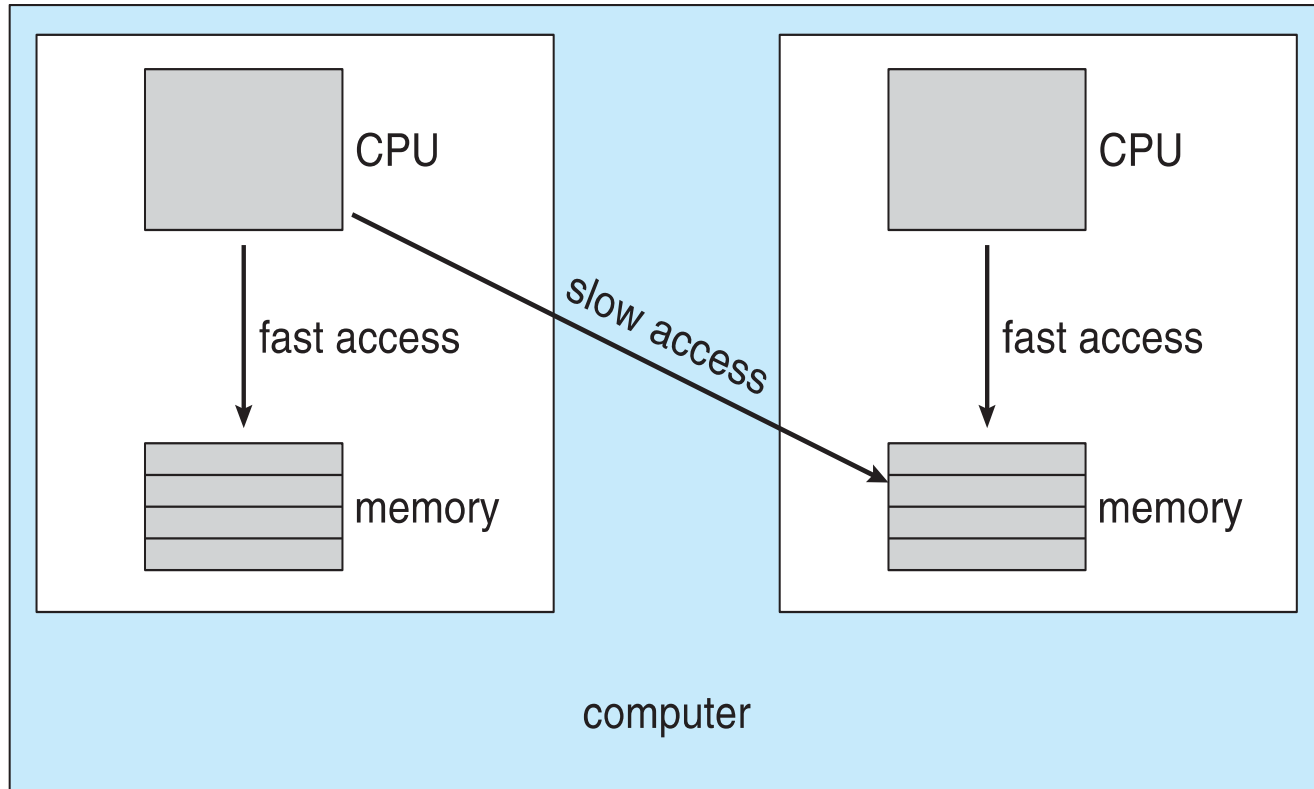
Note that memory-placement algorithms can also consider affinity

Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency

- **Load balancing** attempts to keep workload evenly distributed

- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs

- **Pull migration** – idle processors pulls waiting task from busy processor
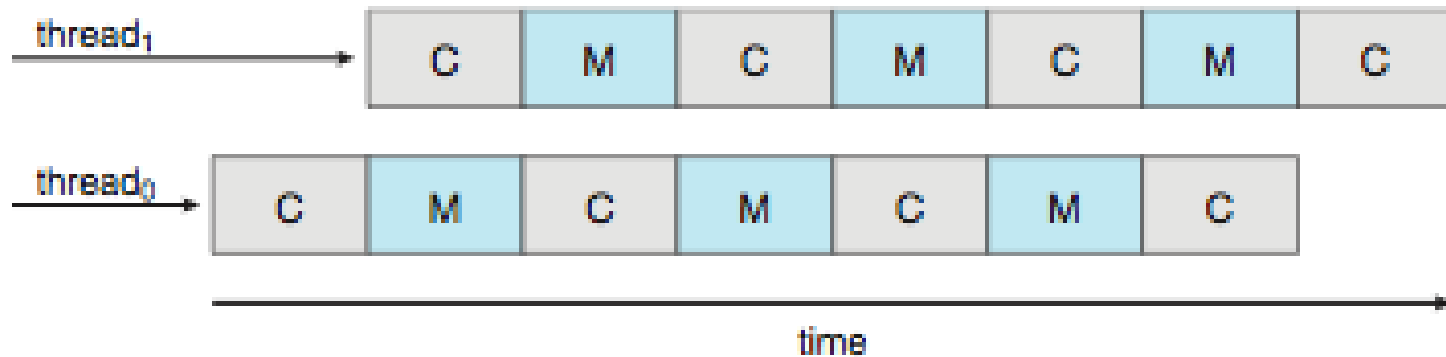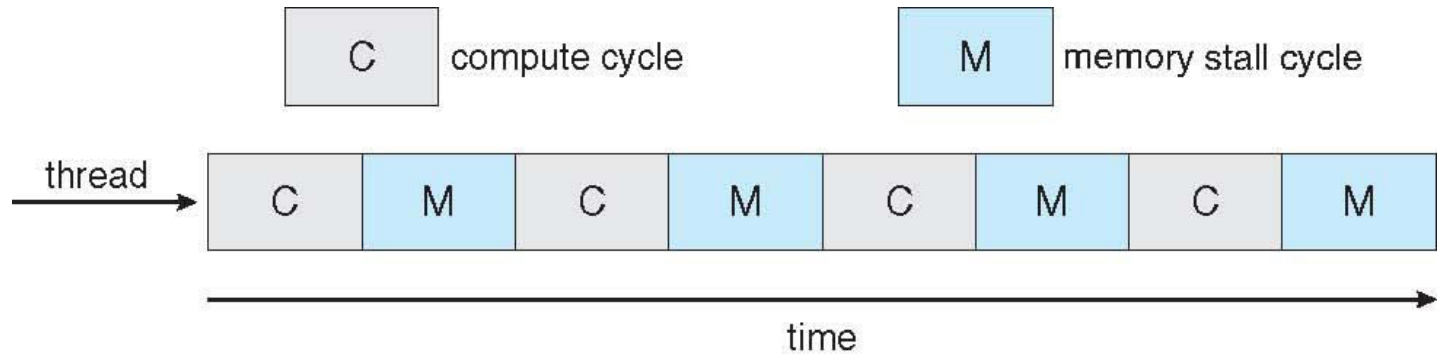
Multicore Processors

- Recent trend to place multiple processor cores on same physical chip

- Faster and consumes less power

- Multiple threads per core also growing

    - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

Multithreaded Multicore System

End of Chapter 6