Chapter 6: CPU Scheduling

adapted from Silberschatz, Galvin, Gagne

Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Operating Systems Examples

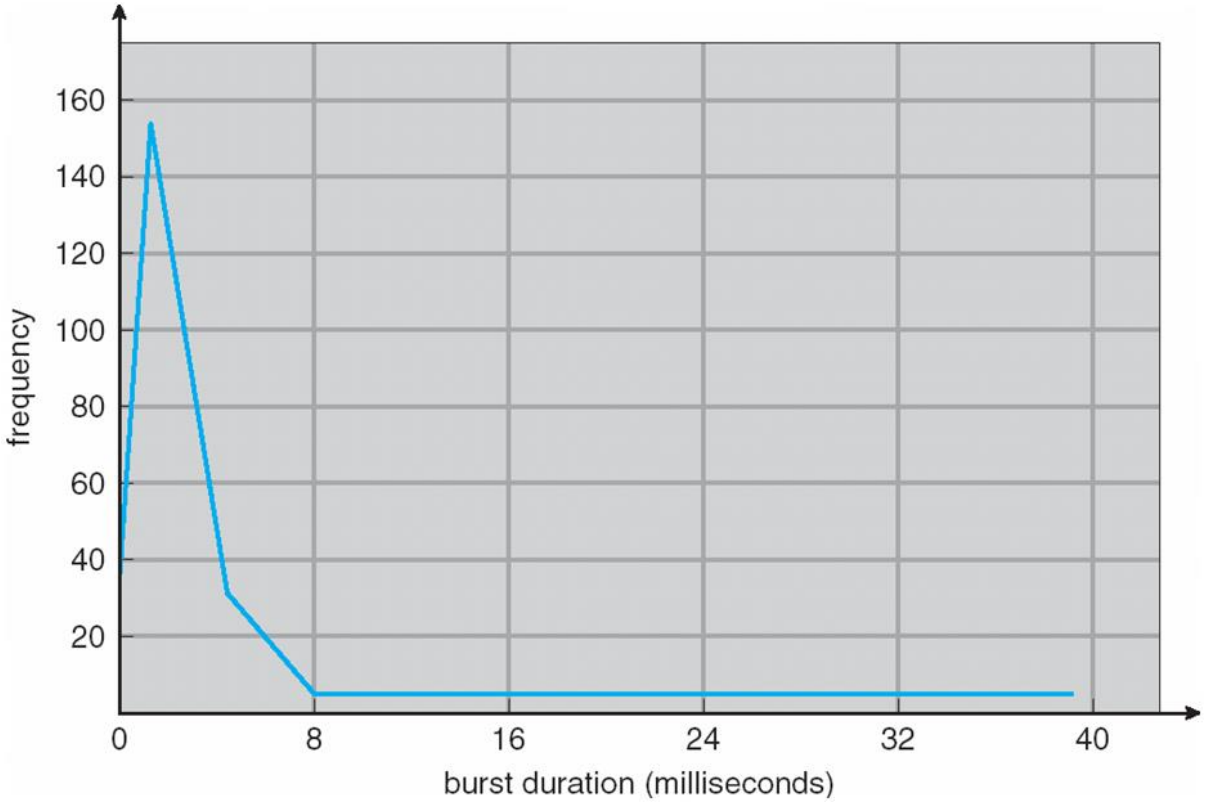
Objectives

- To introduce CPU scheduling, which is the basis for multi-task operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

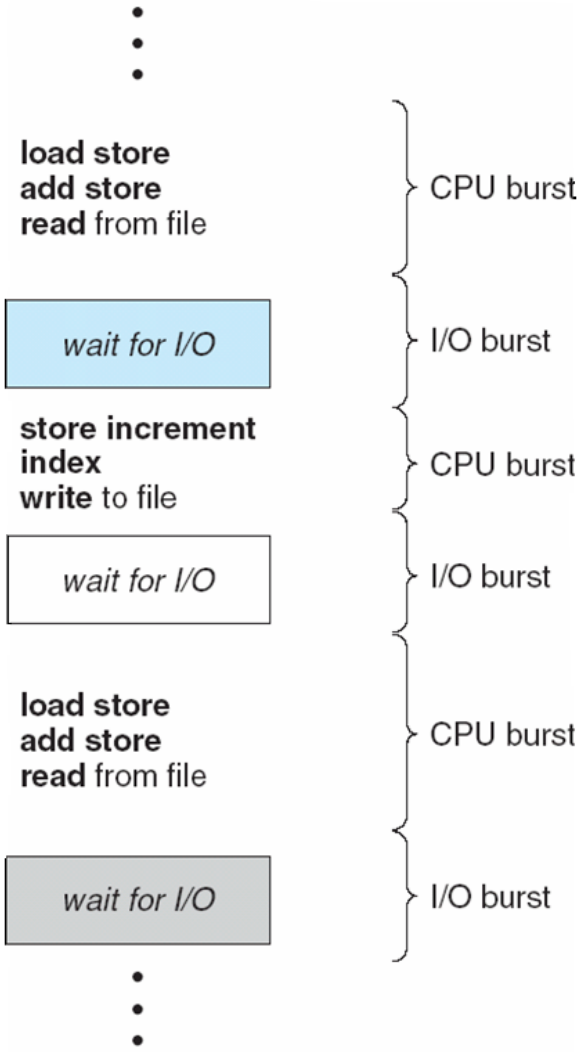
Basic Concepts

- 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** distribution

Histogram of CPU-burst Times



Alternating Sequence of CPU And I/O Bursts



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
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**

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** – the interval from the time of submission of a process to the time of completion. Sum of the time waiting to get into memory, waiting in the ready queue, executing on the CPU and doing I/O.
- **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, which excludes the time it takes to output the response (output device dependent).

Scheduling Algorithm Optimization Criteria

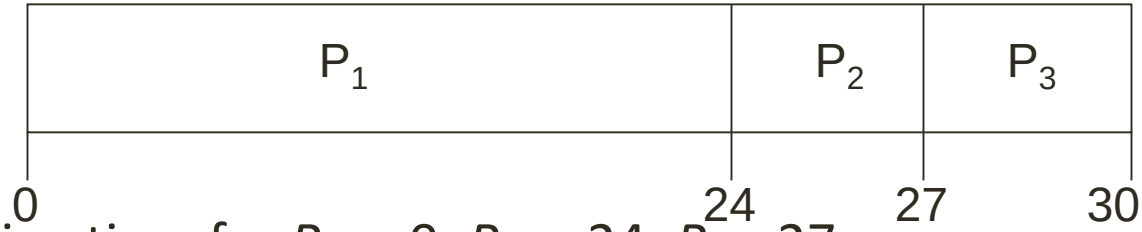
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

- In most cases, optimize the average measure
- In some cases, optimize the minimum or maximum value rather than average
  - Example, to guarantee all users get good service, minimize the maximum 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:



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

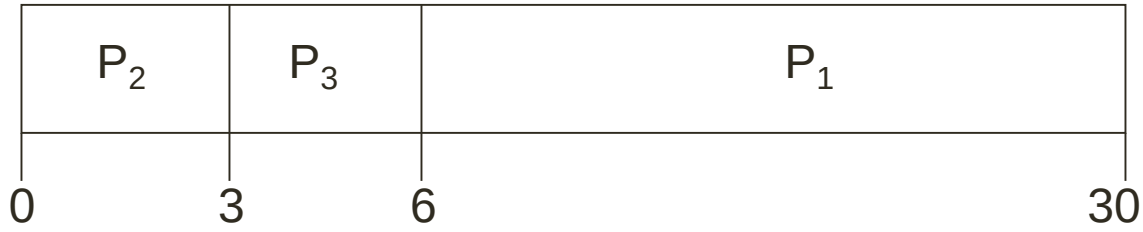
- Gantt Chart: a bar chart that illustrates a particular schedule, include the start and finish times of each of the participating process

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
- *Convoy effect* short process behind long process

Convoy effect

- Convoy effect is slowing down of the whole operating system because of few slow processes.
- It's usually used in context of vehicle traffics. Basically, a convoy affected region is the one where the slow moving traffic slows down the speed of all the vehicles and makes the whole process lengthier.

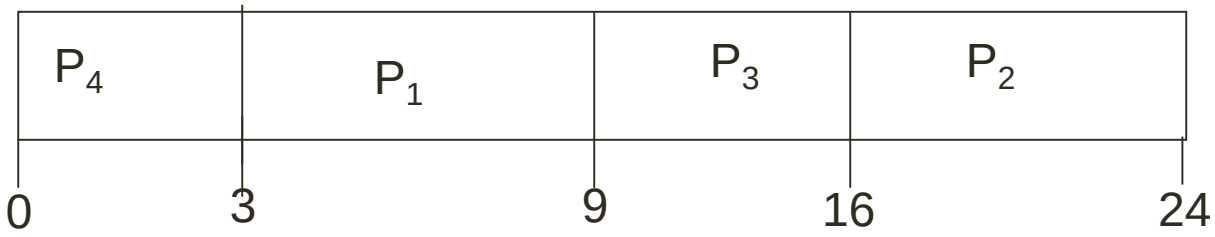
Shortest-Job-First (SJF) Scheduling

- 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

Example of SJF

Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4	5.0	3

- SJF scheduling chart



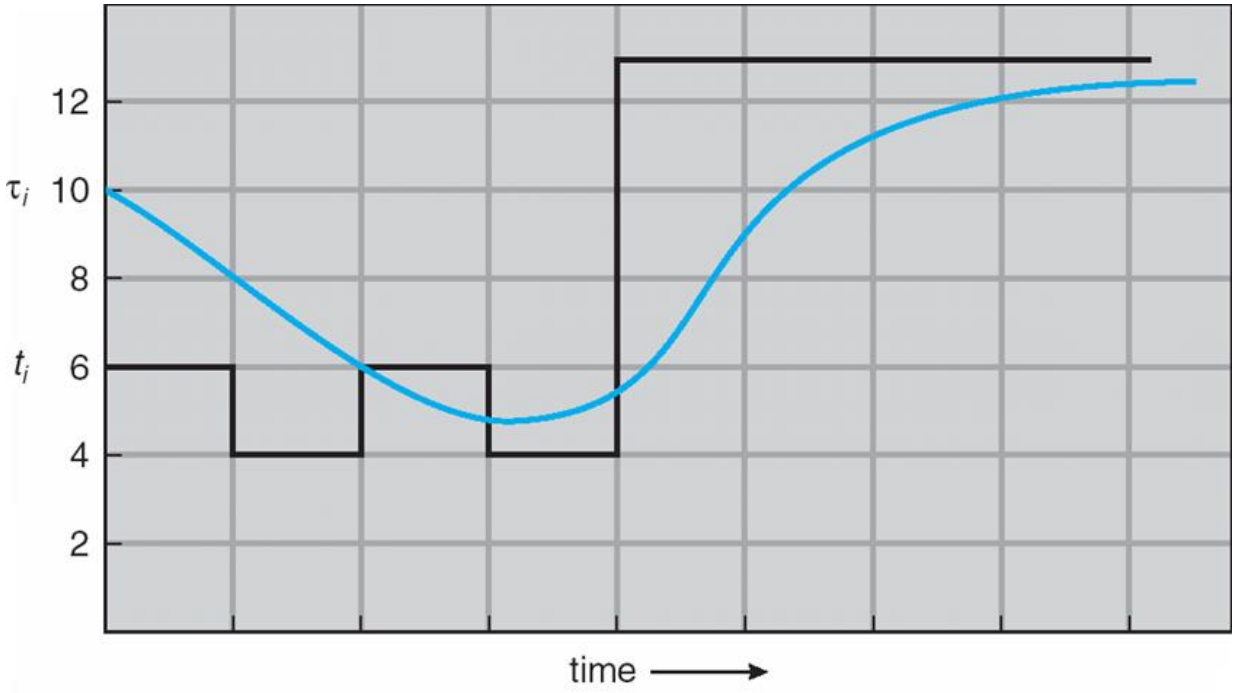
- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Determining Length of Next CPU Burst

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

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

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 ...

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

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)
  - Nonpreemptive: newly arrived process simply put into the queue
  - Preemptive: if the priority of the newly arrived process is higher than priority of the currently running process---preempt the CPU
- SJF is a priority scheduling where priority is the predicted next CPU burst time

Priority Scheduling (Cont)

- Problem ≡ **Starvation** – low priority processes may never execute
  - Rumors has it that when they shut down the IBM 7094 at MIT in 1973, they found a low priority process that had been submitted in 1967 and had not yet been run.
- Solution ≡ **Aging** – as time progresses increase the priority of the process
  - Example: priority range from 127 (low) to 0 (high)
  - Increase priority of a waiting process by 1 every 15 minutes
  - 32 hours to reach priority 0 from 127

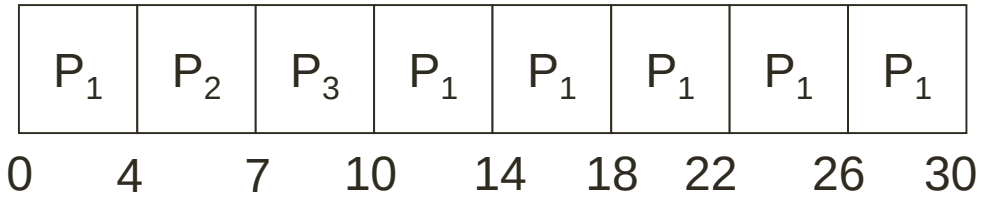
Round Robin (RR)

- 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.
- 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.
- Performance
  - q large $\Rightarrow$ FCFS
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high
  - Rule of thumb: 80% CPU bursts should be shorter than quantum

Example of RR with Time Quantum = 4

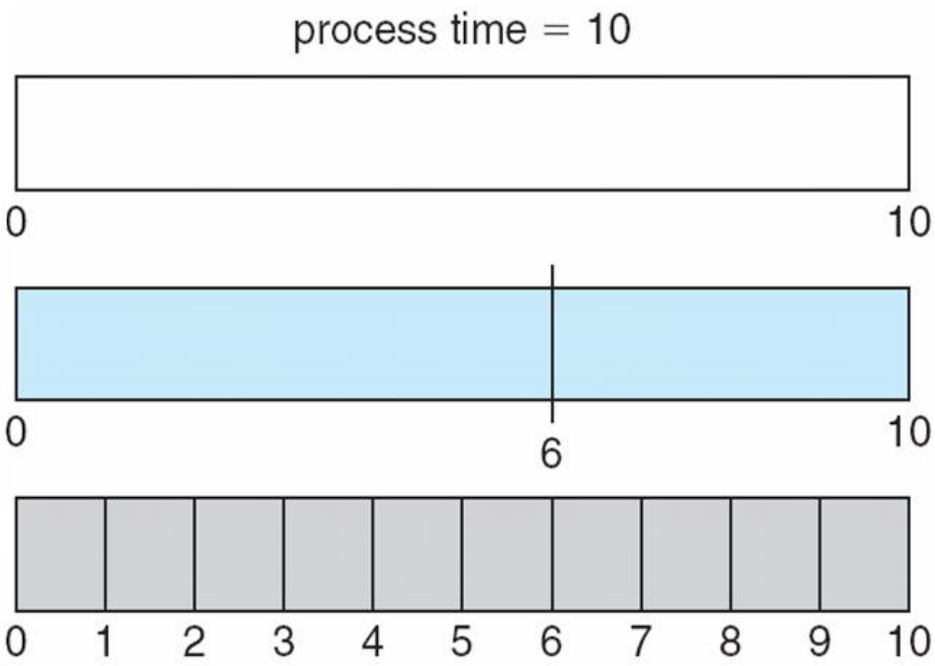
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



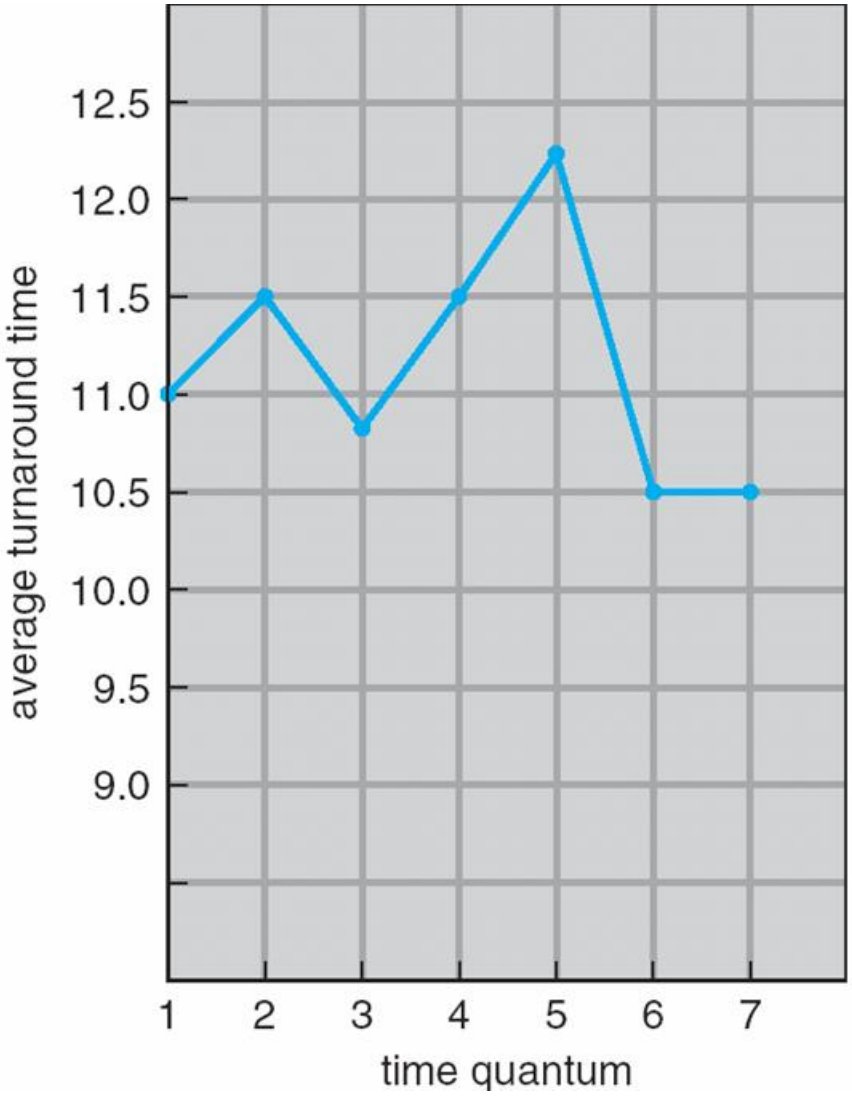
- Typically, higher average turnaround than SJF, but better *response*

Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies With The Time Quantum

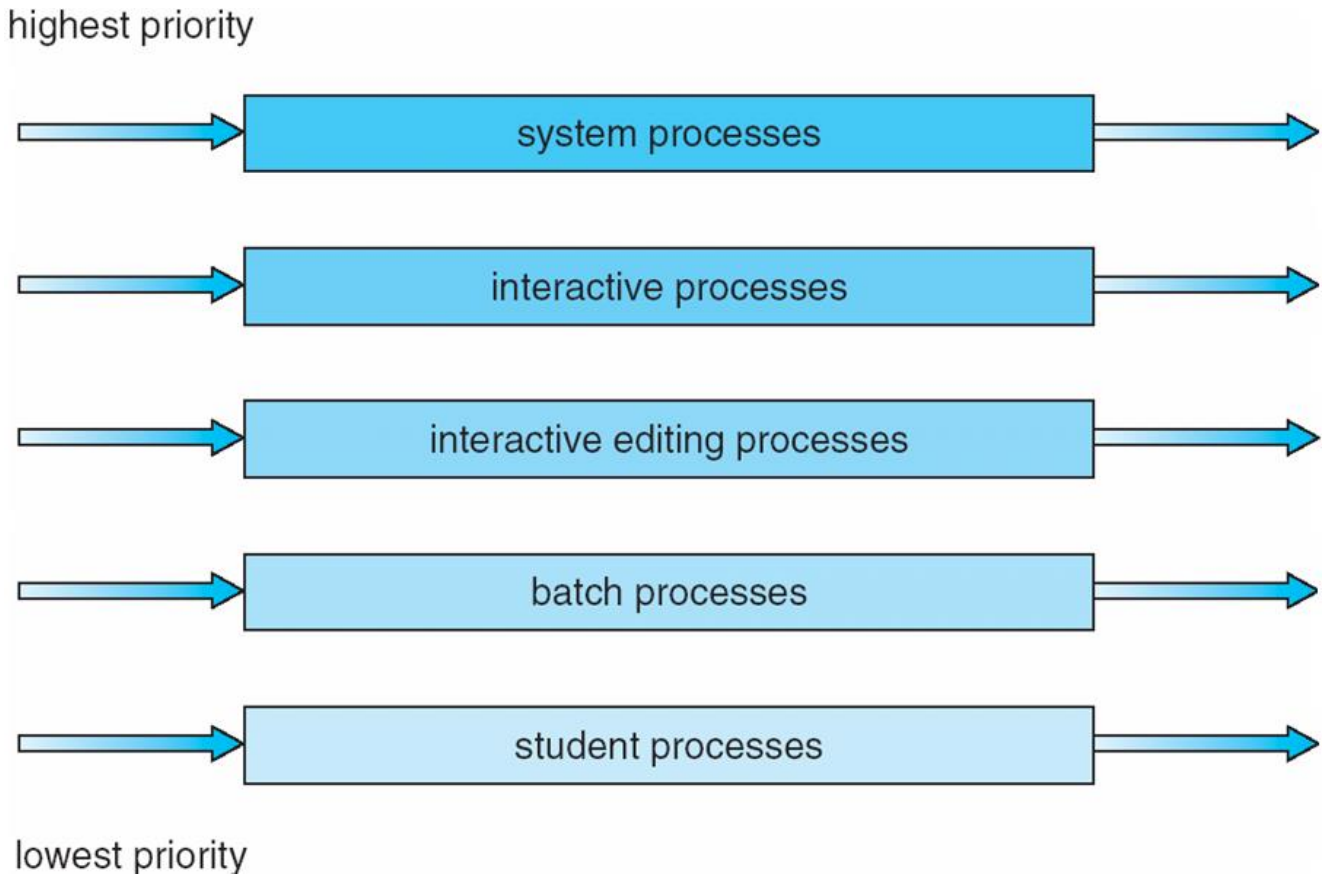


process	time
P_1	6
P_2	3
P_3	1
P_4	7

Multilevel Queue

- 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; (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



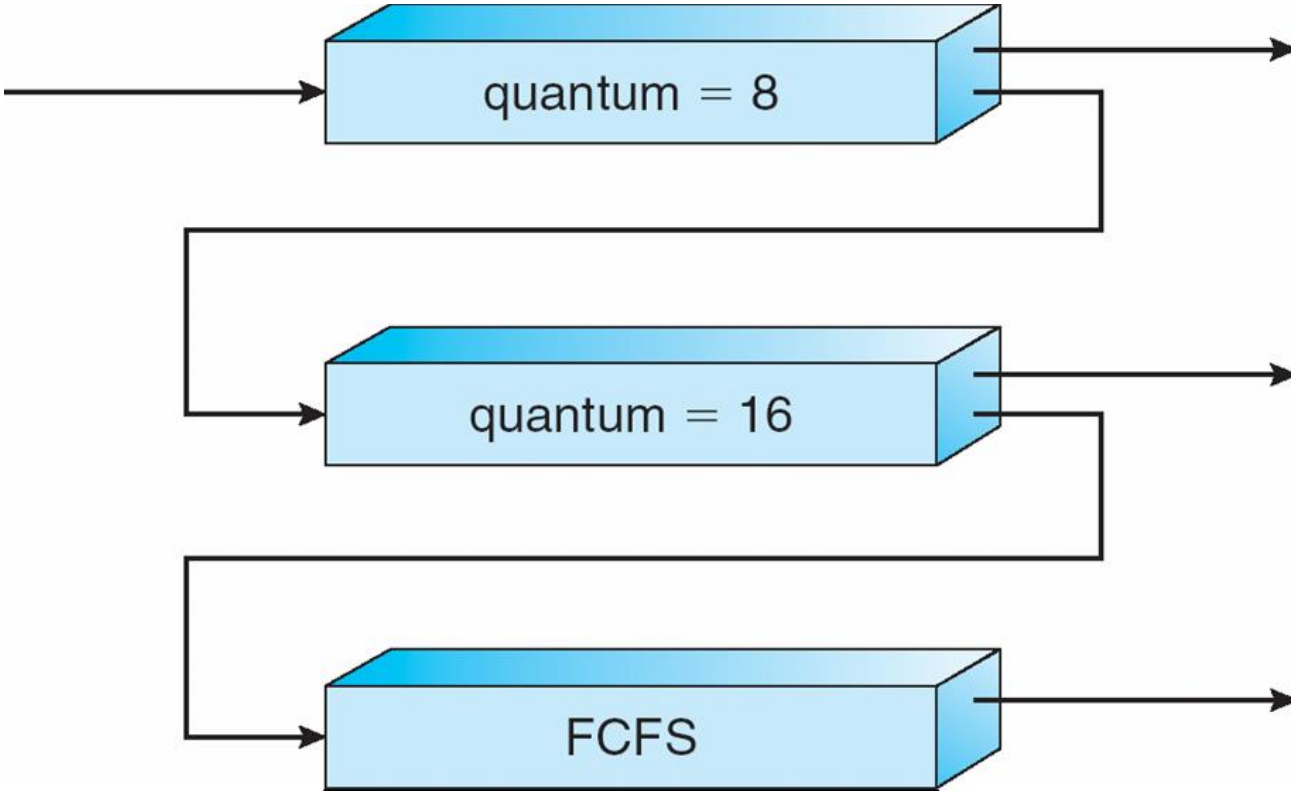
Multilevel Feedback Queue

- 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. 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 receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queues



Thread Scheduling

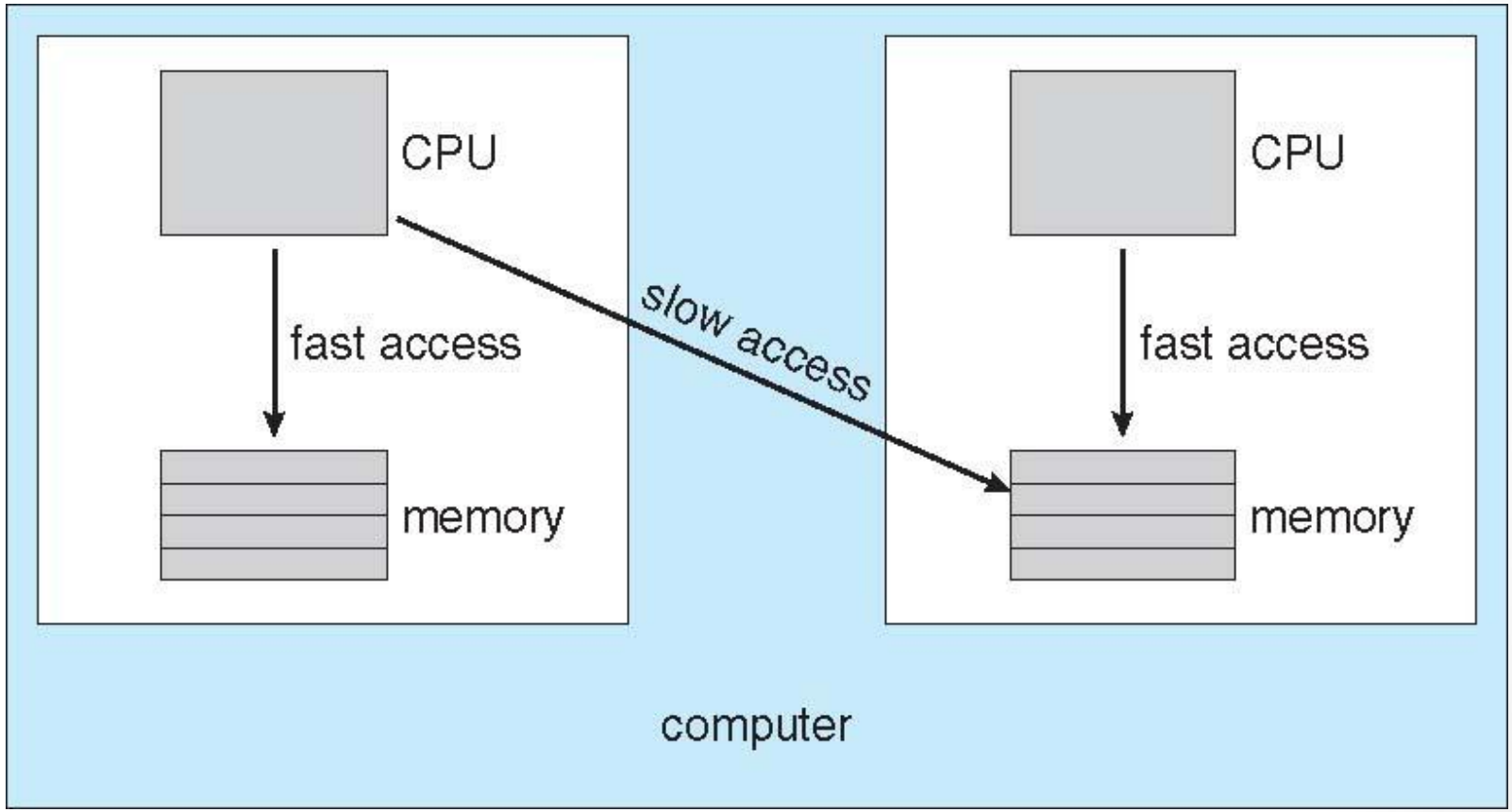
- Distinction between user-level and kernel-level threads
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process

- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **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
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**

NUMA and CPU Scheduling



Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consume less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

Linux Scheduling

- Before Linux kernel version 2.5, traditional UNIX scheduling, not adequately support SMP
- Linux kernel version 2.5, O(1) scheduler
  - Constant scheduling time regardless number of tasks
  - Better support for SMP
  - Poor response time for interactive processes
- After Linux kernel version 2.6.23, CFS-completely fair scheduler
  - Default scheduler now

Completely Fair Scheduler

- Scheduling class
  - Standard Linux kernel implements two scheduling classes
  - (1) Default scheduling class: CFS
  - (2) Real-time scheduling class
- Varying length scheduling quantum
  - Traditional UNIX scheduling uses 90ms fixed scheduling quantum
  - CFS assigns a proposition of CPU processing time to each task
- Nice value
  - -20 to +19, default nice is 0
  - Lower nice value indicates a higher relative priority
  - Task with lower nice value receives higher proportion of CPU time

Completely Fair Scheduler (Cont)

- Virtual run time
  - Each task has a per-task variable **vruntime**
  - Decay factor
    - Lower priority has higher rate of decay
    - Normal priority (nice = 0) virtual run time is identical to actual physical run time
    - A task with lower priority runs for 200 milliseconds, its **vruntime** will be higher than 200 milliseconds
    - A task with higher priority runs for 200 milliseconds, its **vruntime** will be lower than 200 milliseconds
- Lower virtual run time, higher priority
  - To decide which task to run next, scheduler chooses the task that has the smallest **vruntime** value
  - Higher priority can preempt lower priority

Completely Fair Scheduler (Cont)

- Example: Two tasks have the same nice value
- One task is I/O bound and the other is CPU bound
- **vruntime** of I/O bound will be shorter than **vruntime** of CPU bound
- I/O bound task will eventually have higher priority and preempt CPU-bound tasks whenever it is ready to run

End Chapter 6