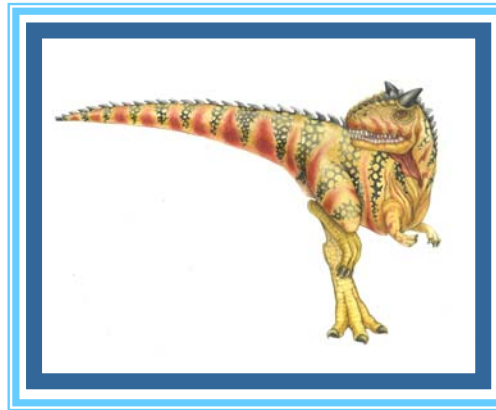# Chapter 5:  CPU Scheduling

# Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor 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
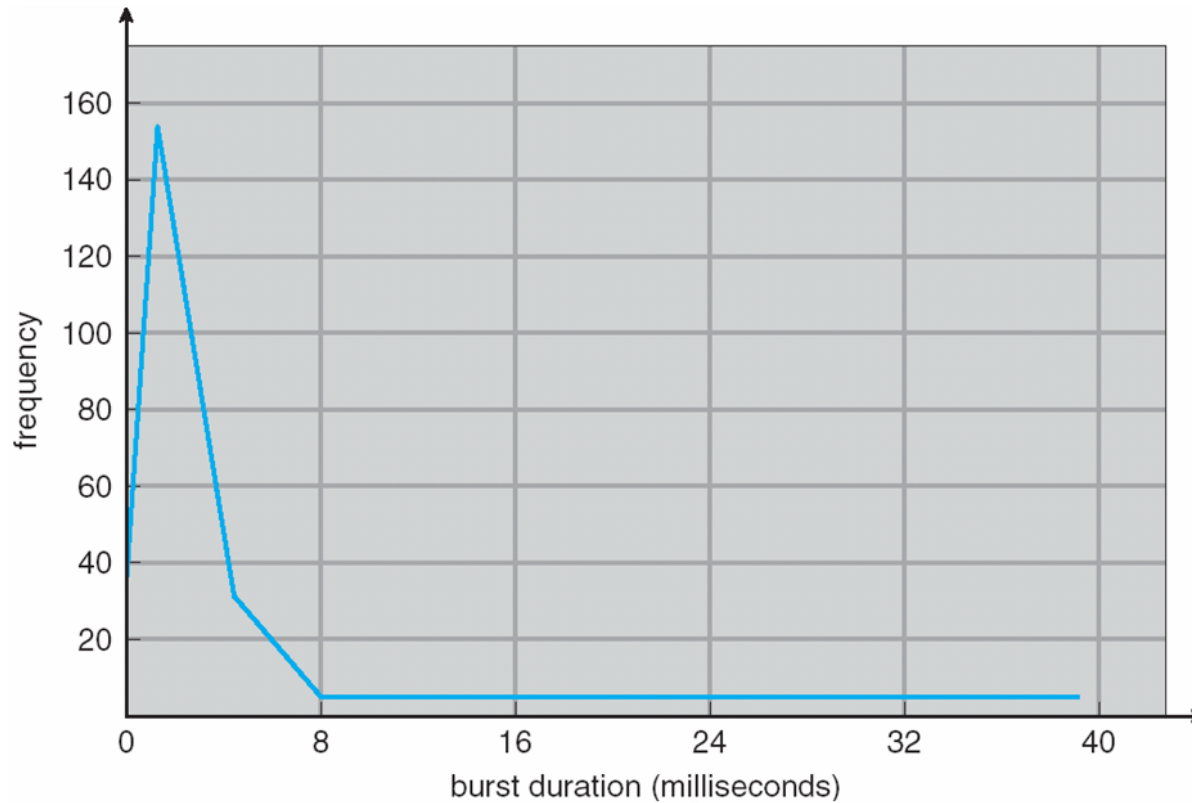
# Basic Concepts

- Maximum CPU utilization obtained with multiprogramming

- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait
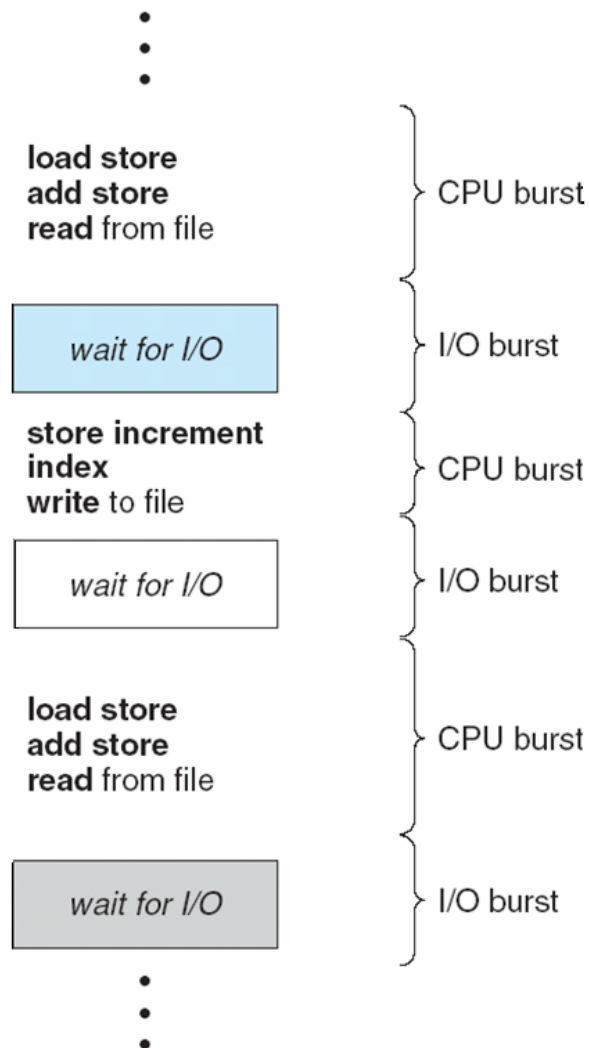
- **CPU burst** distribution

# Histogram of CPU-burst Times

```
                    •
                    •
                    •

        load store
        add  store         ⎫ CPU burst
        read from file      ⎭

       ┌──────────────┐
       │ wait for I/O │      ⎫ I/O burst
       └──────────────┘      ⎭

        store increment
        index              ⎫ CPU burst
        write to file       ⎭

       ┌──────────────┐
       │ wait for I/O │      ⎫ I/O burst
       └──────────────┘      ⎭

        load store
        add  store         ⎫ CPU burst
        read from file      ⎭

       ┌──────────────┐
       │ wait for I/O │      ⎫ I/O burst
       └──────────────┘      ⎭

                    •
                    •
                    •
```

# 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** – 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

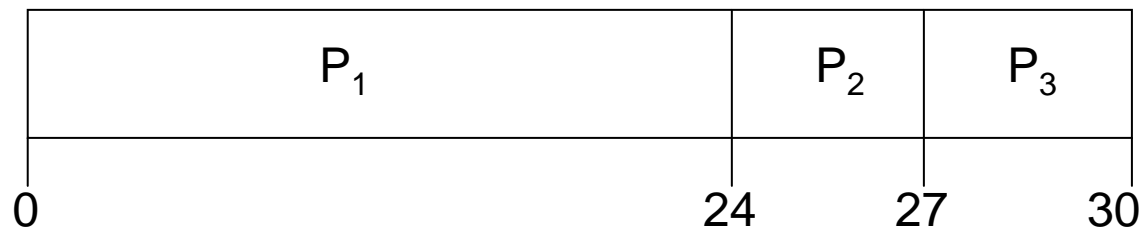- Min waiting time

- 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

# 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
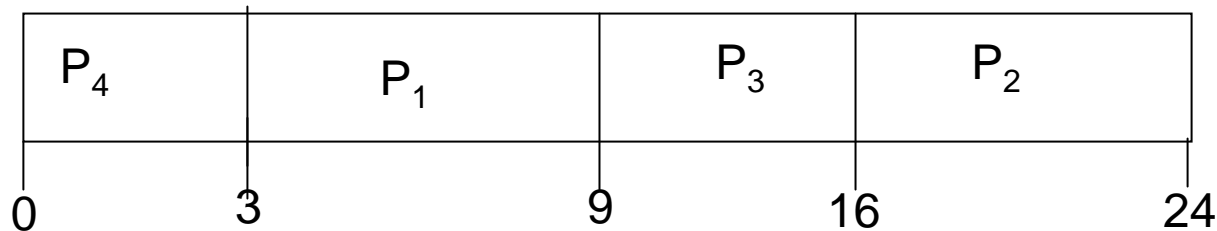  - The difficulty is knowing the length of the next CPU request.

# Example of SJF

| Process | 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 |

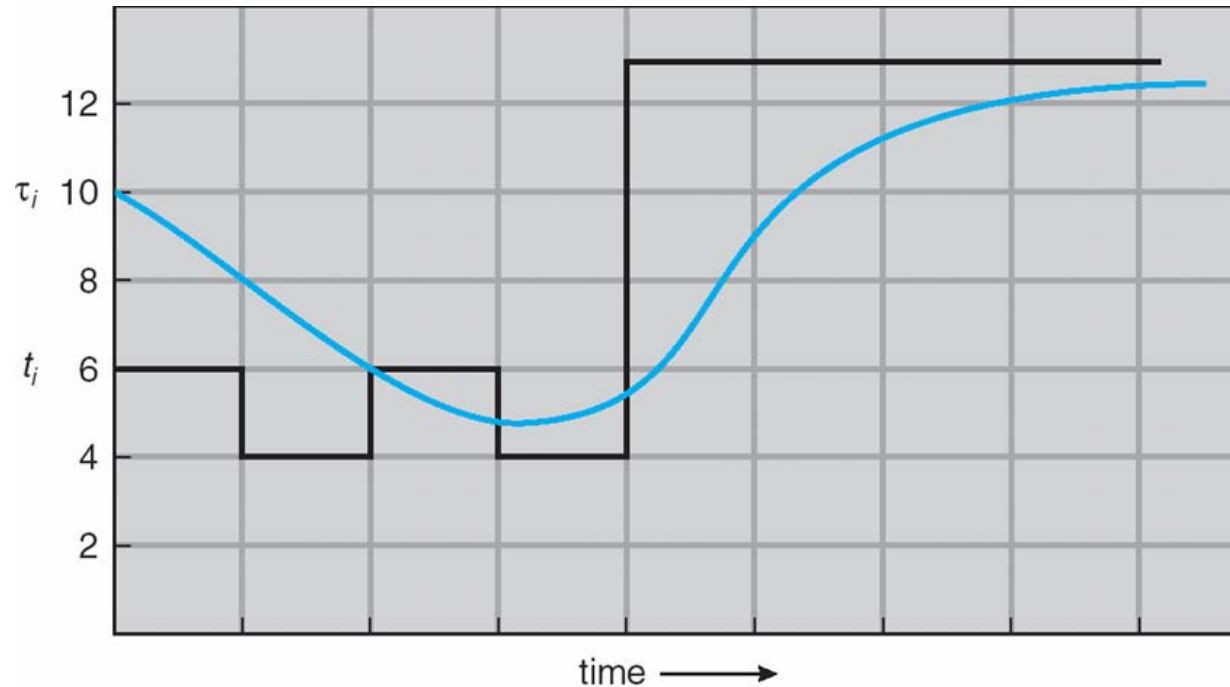- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

# Determining Length of Next CPU Burst

- Can only estimate the length

- 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. $\tau_{n+1} = $ predicted value for the next CPU burst
  3. $\alpha, 0 \leq \alpha \leq 1$
  4. 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" ($\tau_i$) | 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 $\alpha$ 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 $\equiv$ highest priority)
  - Preemptive
  - Nonpreemptive

- SJF is a priority scheduling where priority is the predicted next CPU burst time

- Problem $\equiv$ **Starvation** – low priority processes may never execute

- Solution $\equiv$ **Aging** – as time progresses increase the priority of the process

# 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 $\Rightarrow$ 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
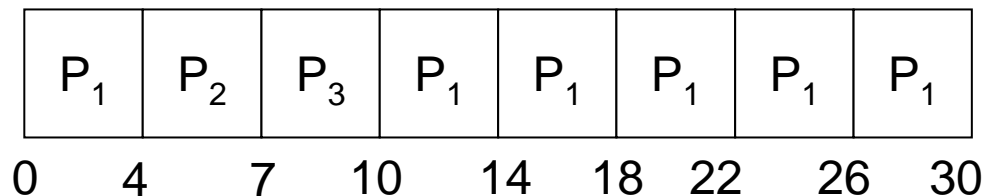
- Typically, higher average turnaround than SJF, but better *response*
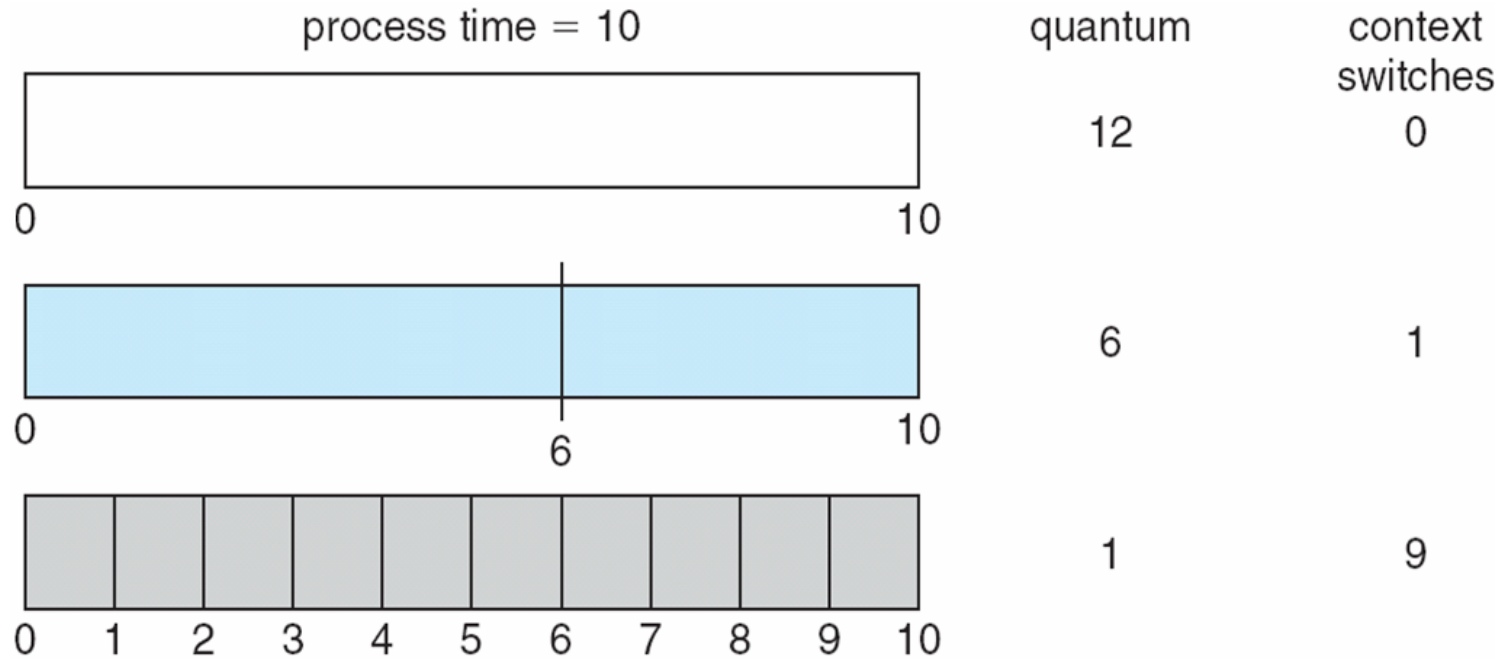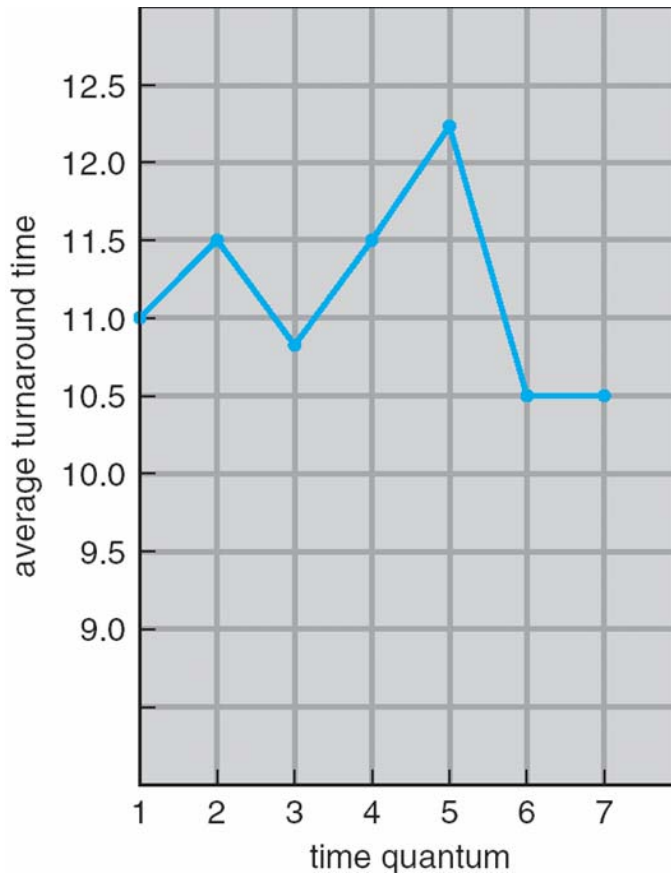
| process | time |
|---------|------|
| $P_1$ | 6 |
| $P_2$ | 3 |
| $P_3$ | 1 |
| $P_4$ | 7 |

# 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

highest priority
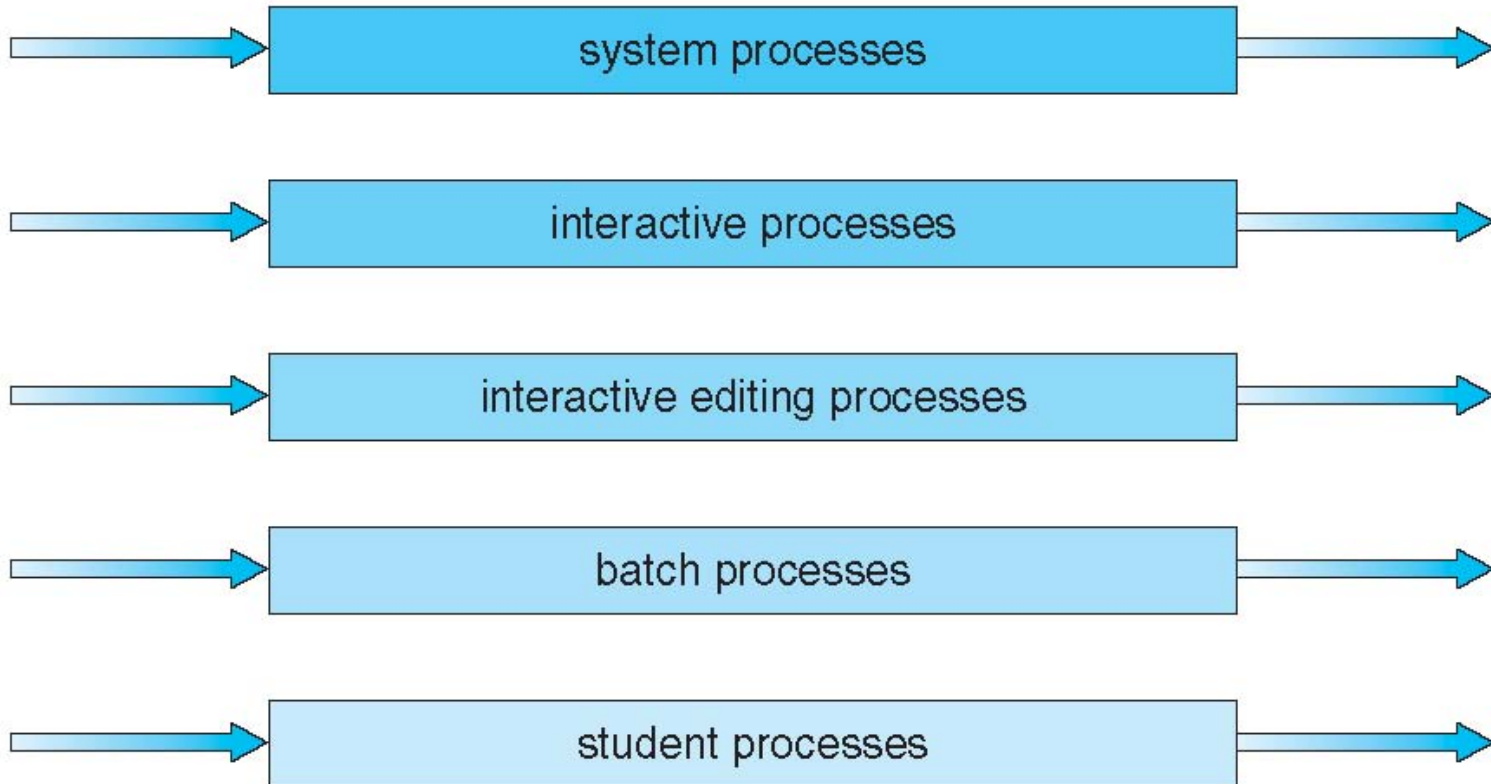


lowest priority

# 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 $Q_2$.

# Multilevel Feedback Queues
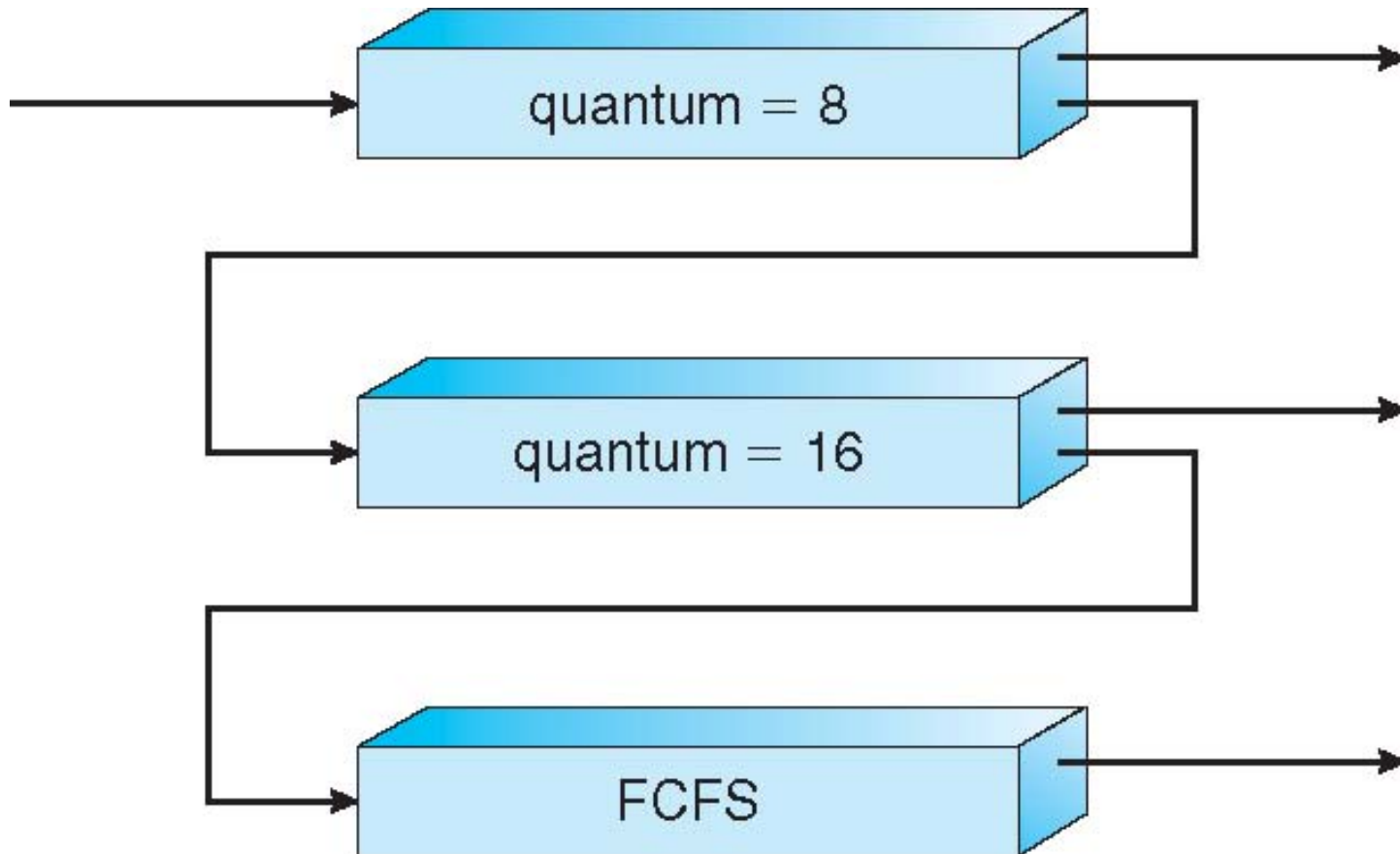
# Thread Scheduling

- 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

# Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD SCOPE PROCESS schedules threads using PCS scheduling
  - PTHREAD SCOPE SYSTEM schedules threads using SCS scheduling.

# Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[])
{
     int i;
    pthread t tid[NUM THREADS];
    pthread attr t attr;
    /* get the default attributes */
    pthread attr init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread attr setscope(&attr, PTHREAD SCOPE SYSTEM);
    /* set the scheduling policy - FIFO, RT, or OTHER */
    pthread attr setschedpolicy(&attr, SCHED OTHER);
    /* create the threads */
    for (i = 0; i < NUM THREADS; i++)
            pthread create(&tid[i],&attr,runner,NULL);
```

# Pthread Scheduling API

```
        /* now join on each thread */
        for (i = 0; i < NUM THREADS; i++)
                pthread join(tid[i], NULL);
}
 /* Each thread will begin control in this function */
void *runner(void *param)
{
    printf("I am a thread\n");
    pthread exit(0);
}
```
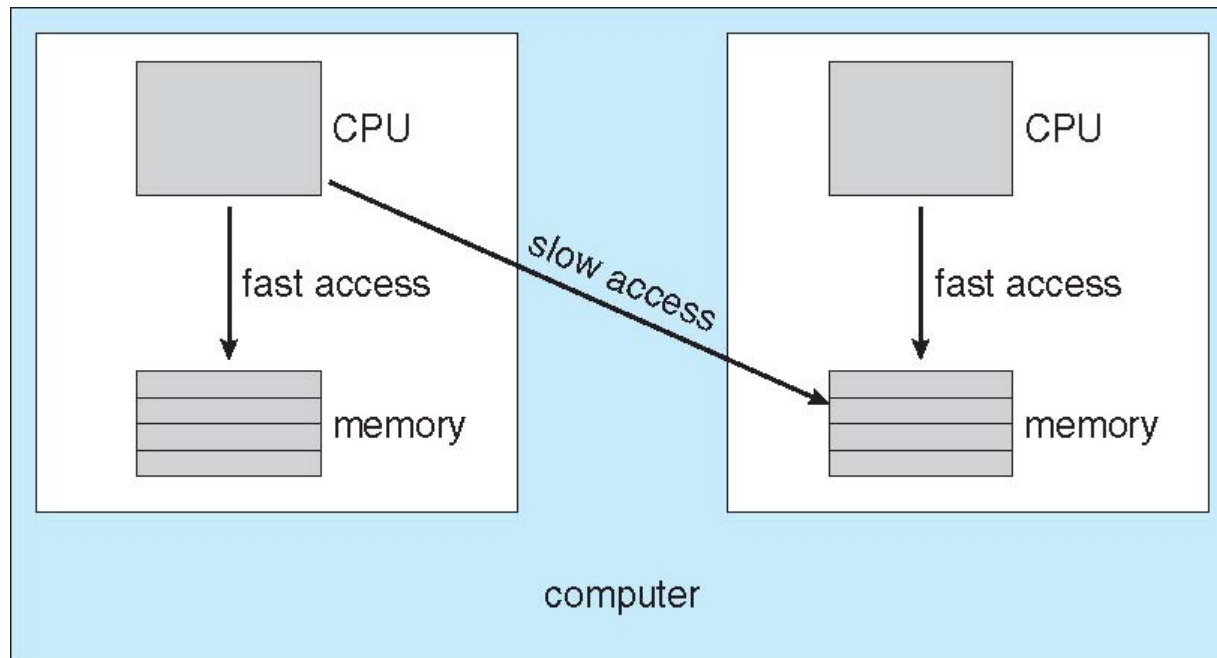
# Multiple-Processor Scheduling

- 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

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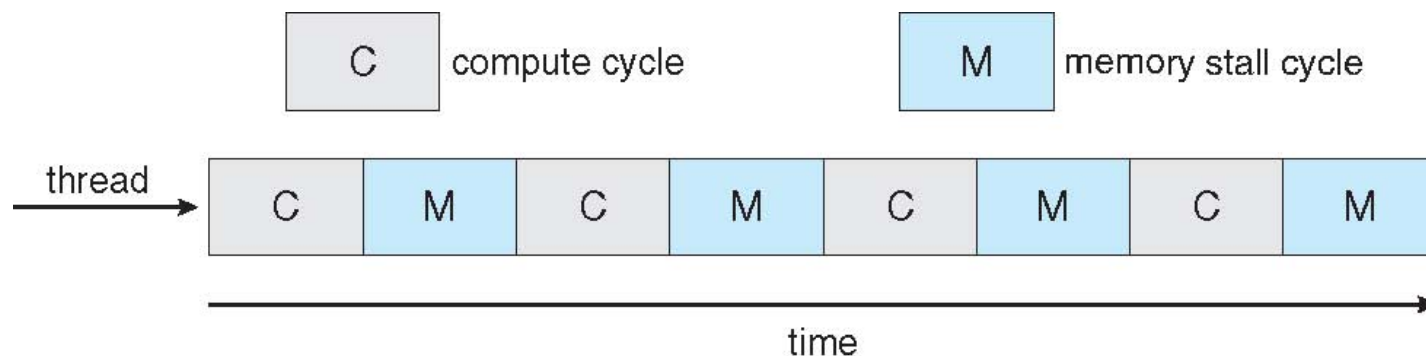  - **hard affinity**

# 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

# Multithreaded Multicore System

# Operating System Examples

- Solaris scheduling
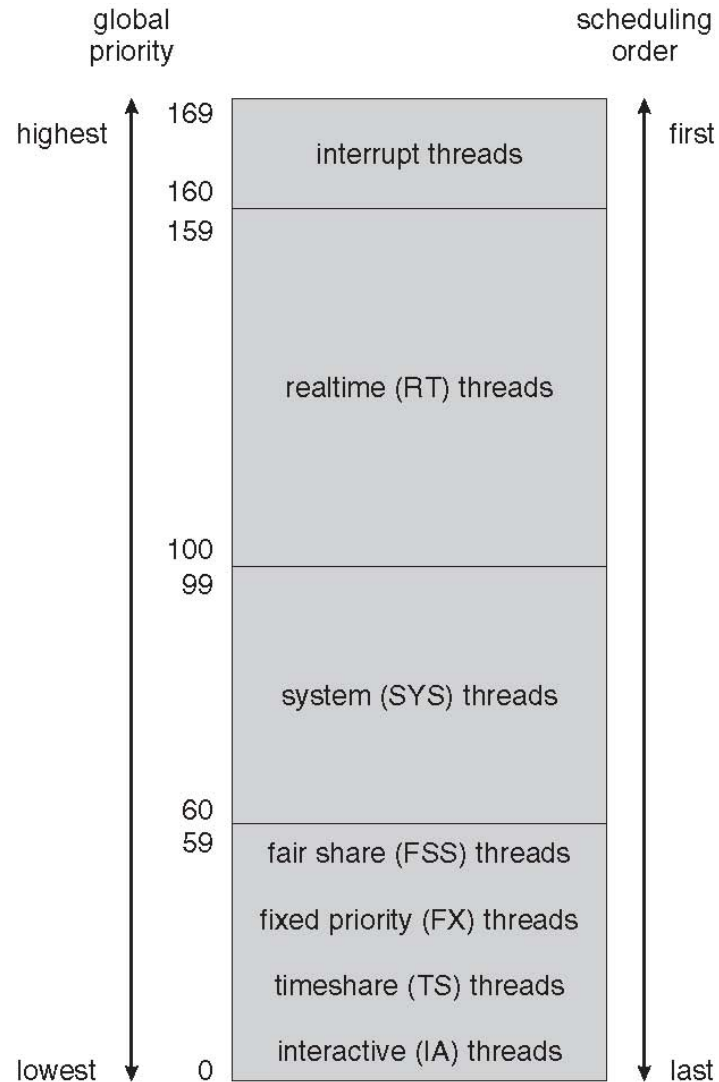
- Windows XP scheduling

- Linux scheduling

# Solaris Dispatch Table

| priority | time quantum | time quantum expired | return from sleep |
|---|---|---|---|
| 0 | 200 | 0 | 50 |
| 5 | 200 | 0 | 50 |
| 10 | 160 | 0 | 51 |
| 15 | 160 | 5 | 51 |
| 20 | 120 | 10 | 52 |
| 25 | 120 | 15 | 52 |
| 30 | 80 | 20 | 53 |
| 35 | 80 | 25 | 54 |
| 40 | 40 | 30 | 55 |
| 45 | 40 | 35 | 56 |
| 50 | 40 | 40 | 58 |
| 55 | 40 | 45 | 58 |
| 59 | 20 | 49 | 59 |

# Solaris Scheduling

# Windows XP Priorities

| | real-time | high | above normal | normal | below normal | idle priority |
|---|---|---|---|---|---|---|
| time-critical | 31 | 15 | 15 | 15 | 15 | 15 |
| highest | 26 | 15 | 12 | 10 | 8 | 6 |
| above normal | 25 | 14 | 11 | 9 | 7 | 5 |
| normal | 24 | 13 | 10 | 8 | 6 | 4 |
| below normal | 23 | 12 | 9 | 7 | 5 | 3 |
| lowest | 22 | 11 | 8 | 6 | 4 | 2 |
| idle | 16 | 1 | 1 | 1 | 1 | 1 |

# Linux Scheduling

- Constant order $O(1)$ scheduling time

- Two priority ranges: time-sharing and real-time

- **Real-time** range from 0 to 99 and **nice** value from 100 to 140
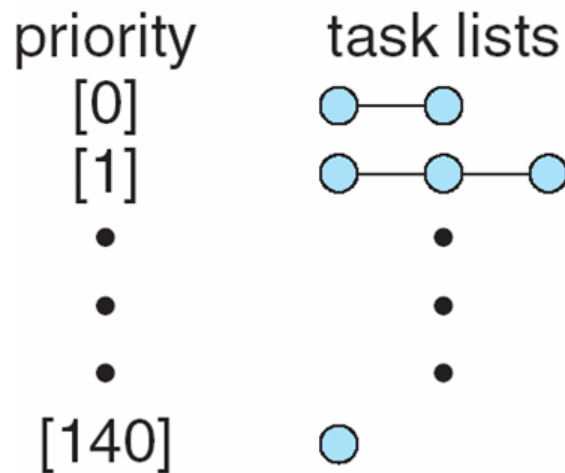
- (figure 5.15)

# Priorities and Time-slice length

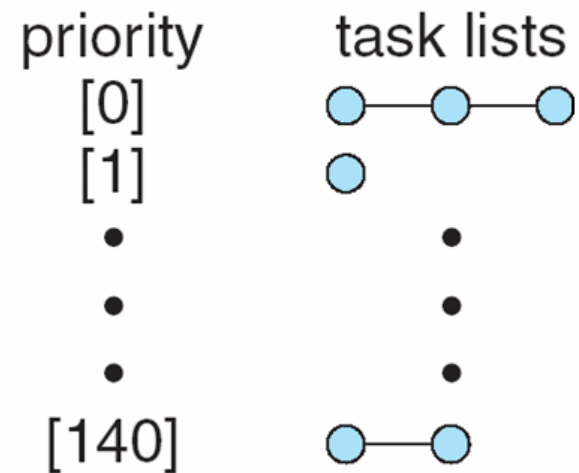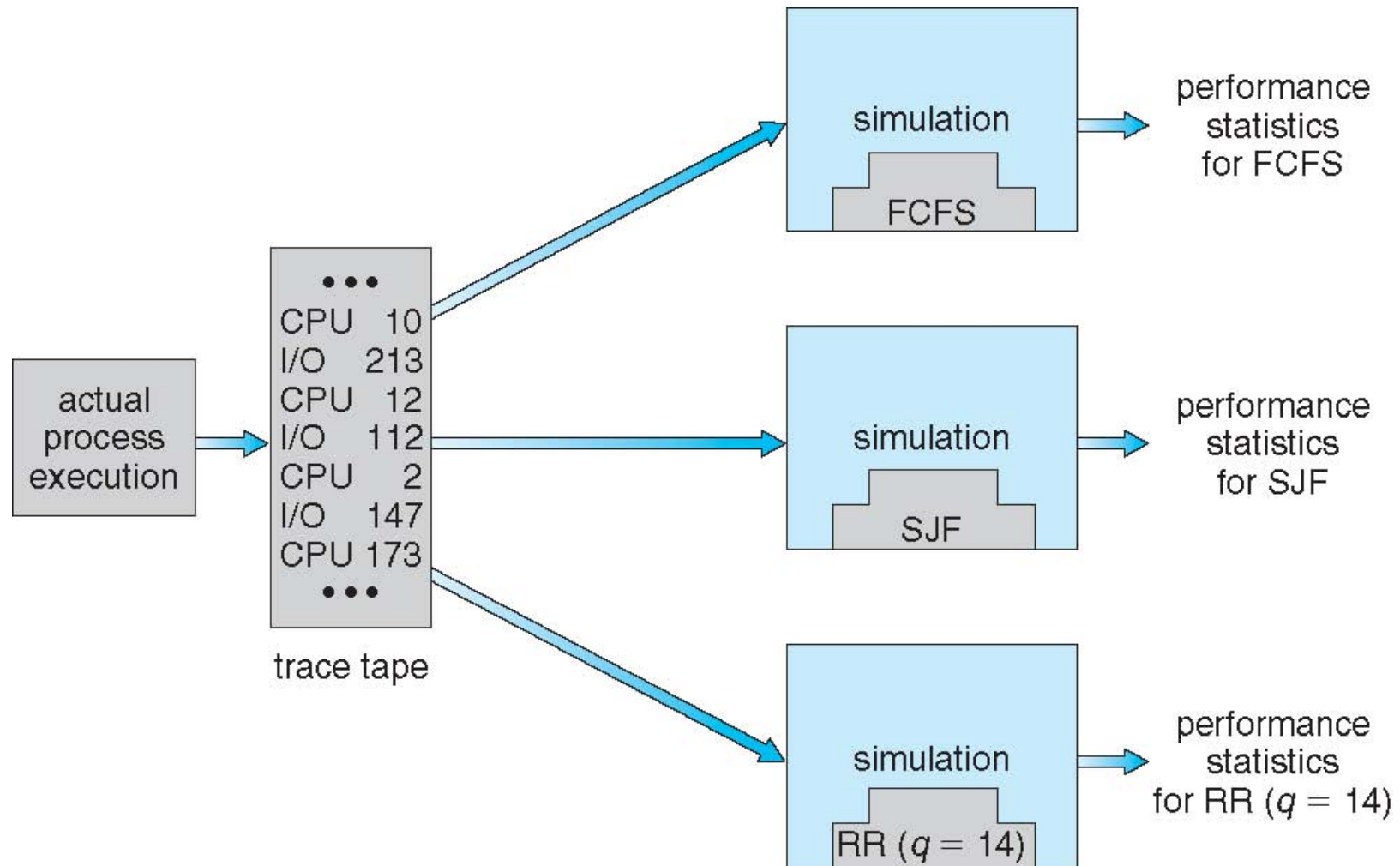| numeric priority | relative priority | | time quantum |
|---|---|---|---|
| 0 | highest | real-time tasks | 200 ms |
| • | | | |
| • | | | |
| • | | | |
| 99 | | | |
| 100 | | other tasks | |
| • | | | |
| • | | | |
| • | | | |
| 140 | lowest | | 10 ms |

# List of Tasks Indexed According to Priorities

# Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload

- Queueing models

- Implementation

# Evaluation of CPU Schedulers by Simulation

# End of Chapter 5