

I.-C. Lin, Assistant Professor. Textbook: Operating System  
Concepts 8ed

# CHAPTER 5: PROCESS SCHEDULING



# Chapter 5: Process Scheduling



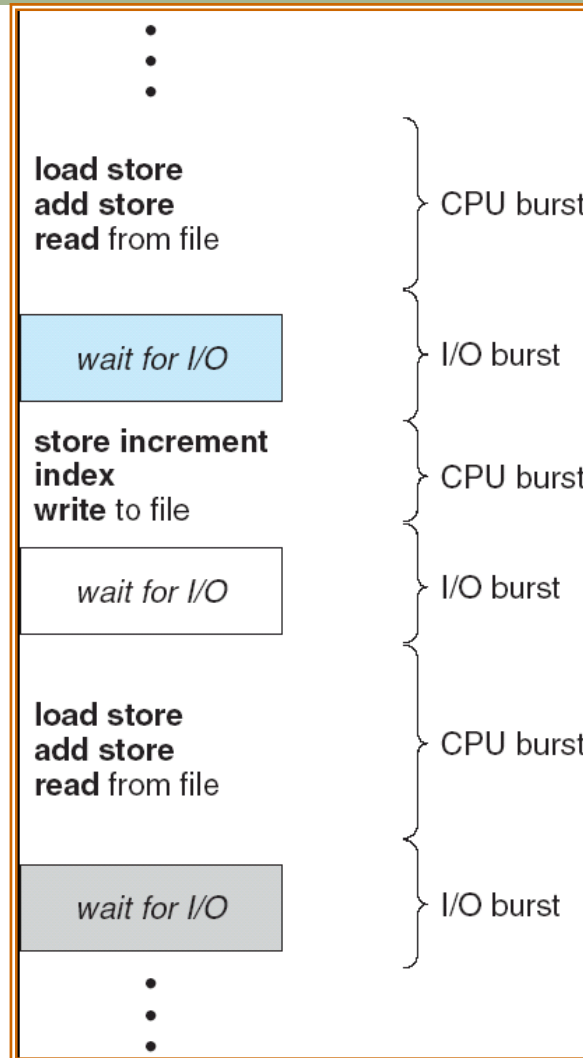
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Thread Scheduling
- Operating Systems Examples
- Algorithm Evaluation

# Basic Concepts

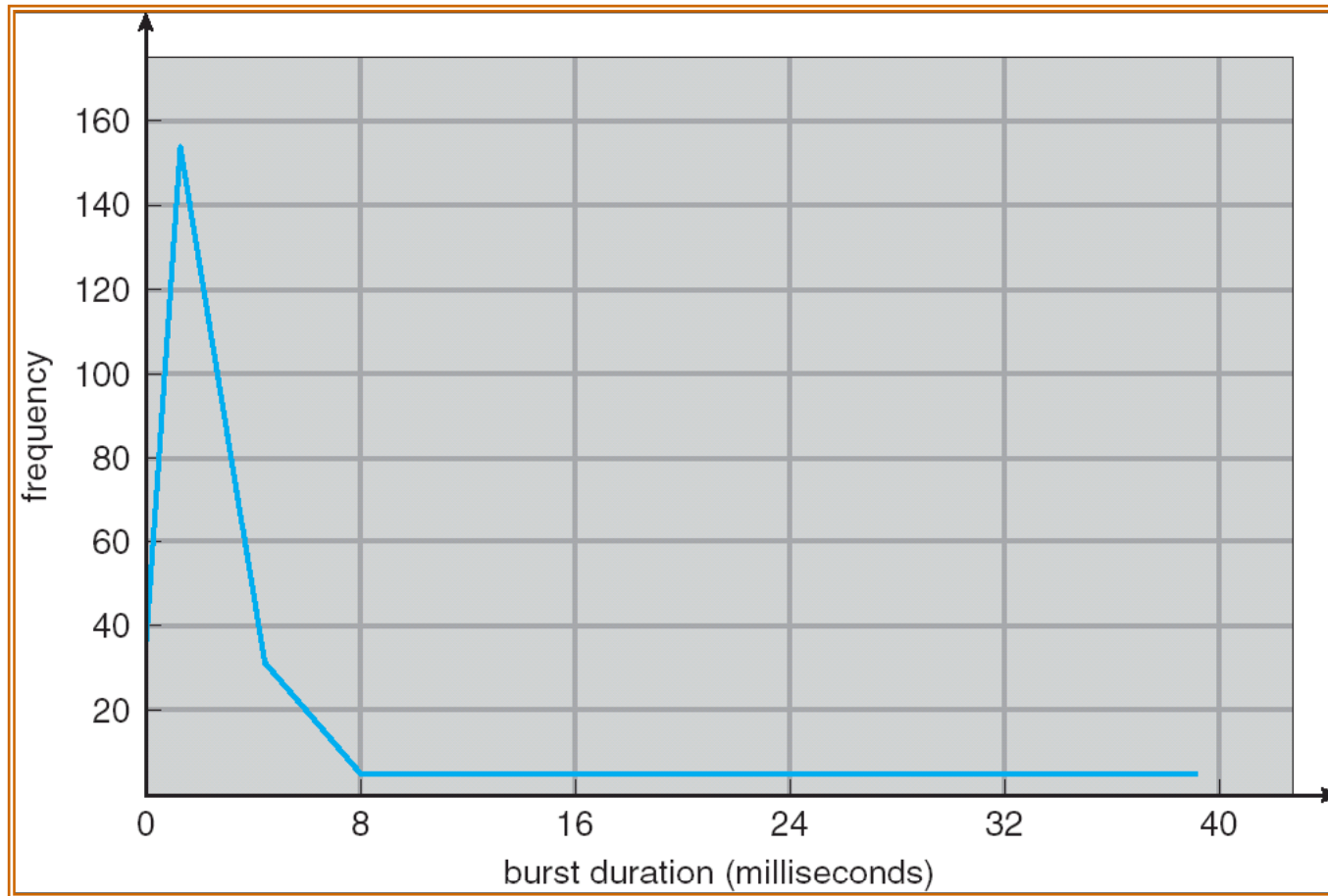


- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait
- CPU burst distribution

# Alternating Sequence of CPU And I/O Bursts



# Histogram of CPU-burst Times



# 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)



# Optimization Criteria



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

# First-Come, First-Served (FCFS) Scheduling

| <u>Process</u> | <u>Burst Time</u> |
|----------------|-------------------|
| $P_1$          | 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:  $(0 + 24 + 27)/3 = 17$

# 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

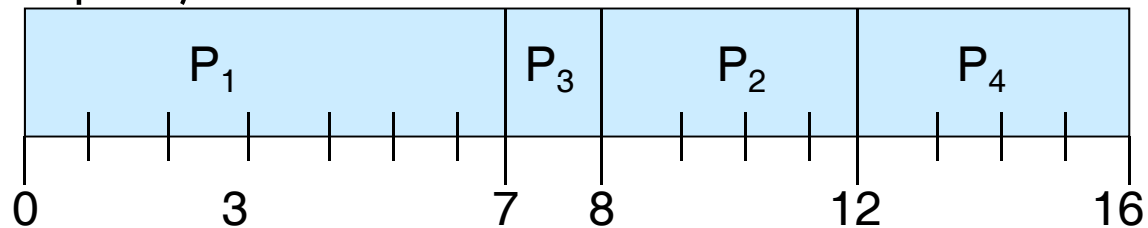
# Shortest-Job-First (SJR) Scheduling

- Use these lengths to schedule the process with the shortest time.
  
- Two schemes:
  - ▣ Nonpreemptive
    - once CPU given to the process it cannot be preempted until completes its CPU burst
  - ▣ Preemptive [Shortest-Remaining-Time-First (SRTF)]
    - if a new process arrives with CPU burst length less than remaining time of current executing process, preempt.
  
- SJF is **optimal** – gives *minimum average waiting time* for a given set of processes

# Example of Non-Preemptive SJF

| <u>Process</u> | <u>Arrival Time</u> | <u>Burst Time</u> |
|----------------|---------------------|-------------------|
| $P_1$          | 0.0                 | 7                 |
| $P_2$          | 2.0                 | 4                 |
| $P_3$          | 4.0                 | 1                 |
| $P_4$          | 5.0                 | 4                 |

- SJF (non-preemptive)

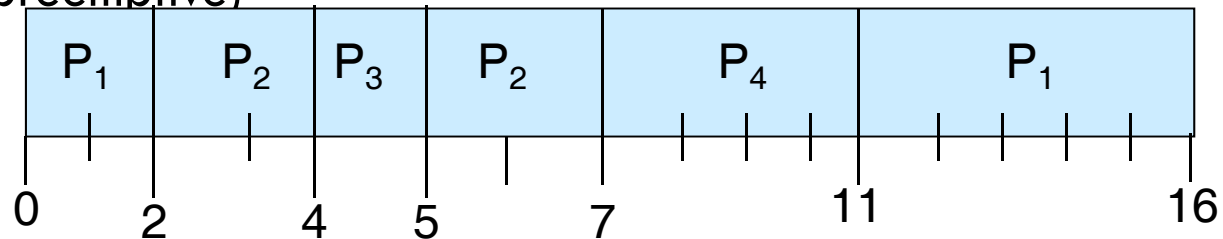


- Average waiting time =  $(0 + 6 + 3 + 7)/4 = 4$

# Example of Preemptive SJF

| <u>Process</u> | <u>Arrival Time</u> | <u>Burst Time</u> |
|----------------|---------------------|-------------------|
| $P_1$          | 0.0                 | 7                 |
| $P_2$          | 2.0                 | 4                 |
| $P_3$          | 4.0                 | 1                 |
| $P_4$          | 5.0                 | 4                 |

- SJF (preemptive)

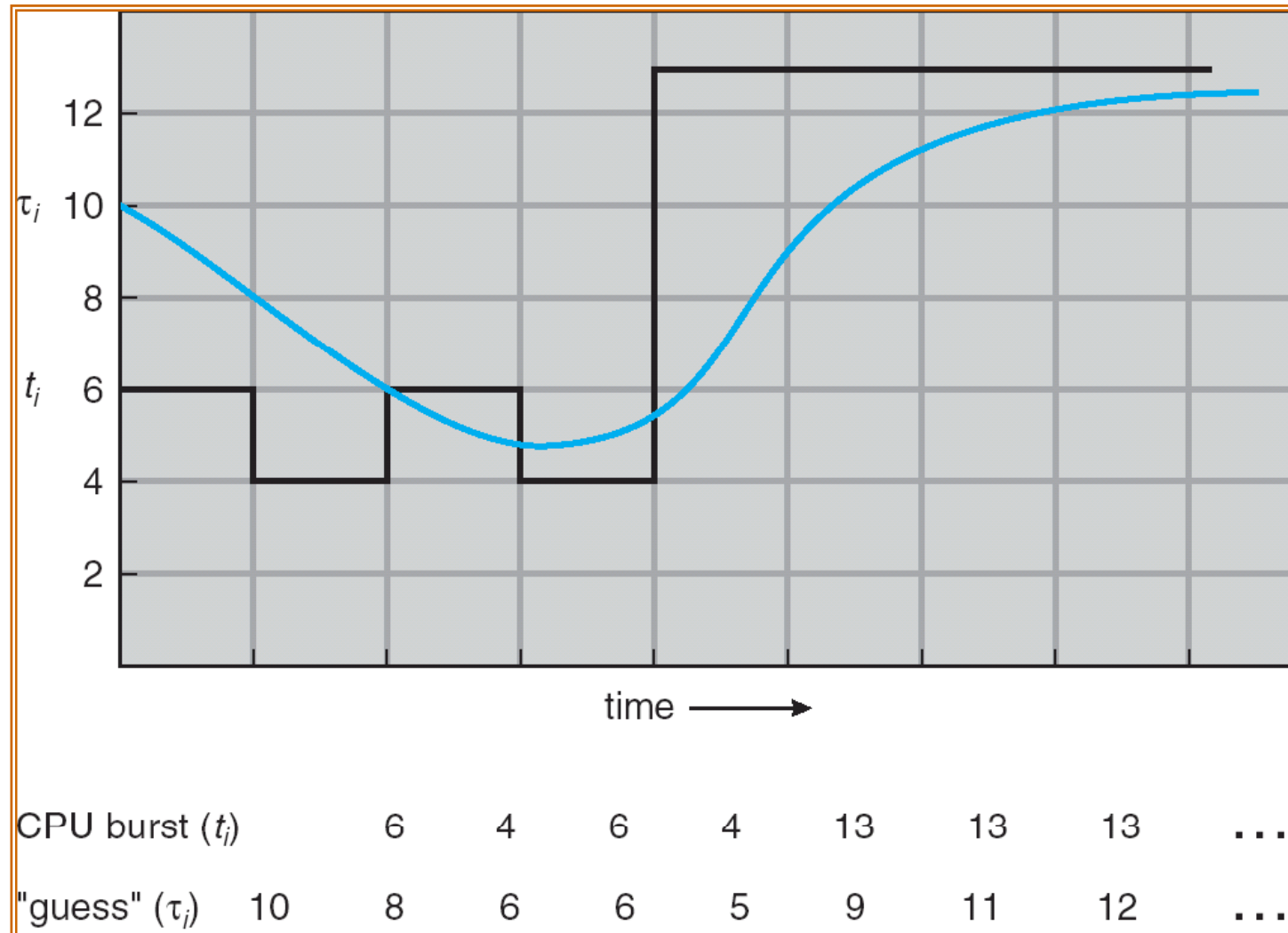


- Average waiting time =  $(9 + 1 + 0 + 2)/4 = 3$

# 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  $n^{\text{th}}$  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





# 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:

$$\begin{aligned}\tau_{n+1} = & \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots \\ & + (1 - \alpha)^i \alpha t_{n-i} + \dots \\ & + (1 - \alpha)^{n+1} \tau_0\end{aligned}$$

- 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

# Priority Scheduling (cont.)



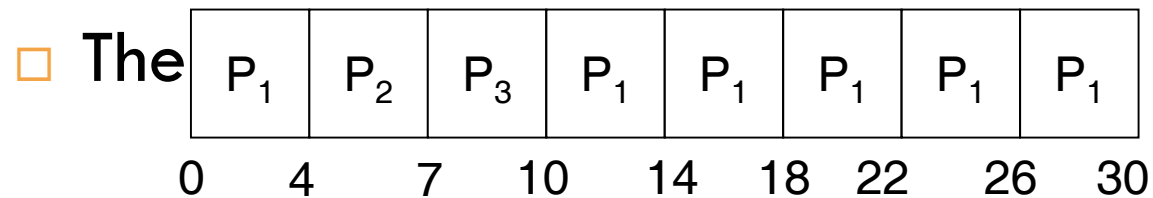
- Problem  $\equiv$  Starvation
  - ▣ low priority processes may never execute
- Solution  $\equiv$  Aging
  - ▣ as time progresses increase the priority of the process

# 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

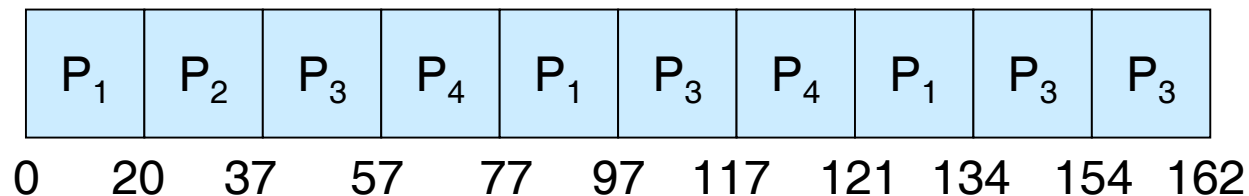
| <u>Process</u> | <u>Burst Time</u> |
|----------------|-------------------|
| $P_1$          | 24                |
| $P_2$          | 3                 |
| $P_3$          | 3                 |



# Example of RR with Time Quantum = 20

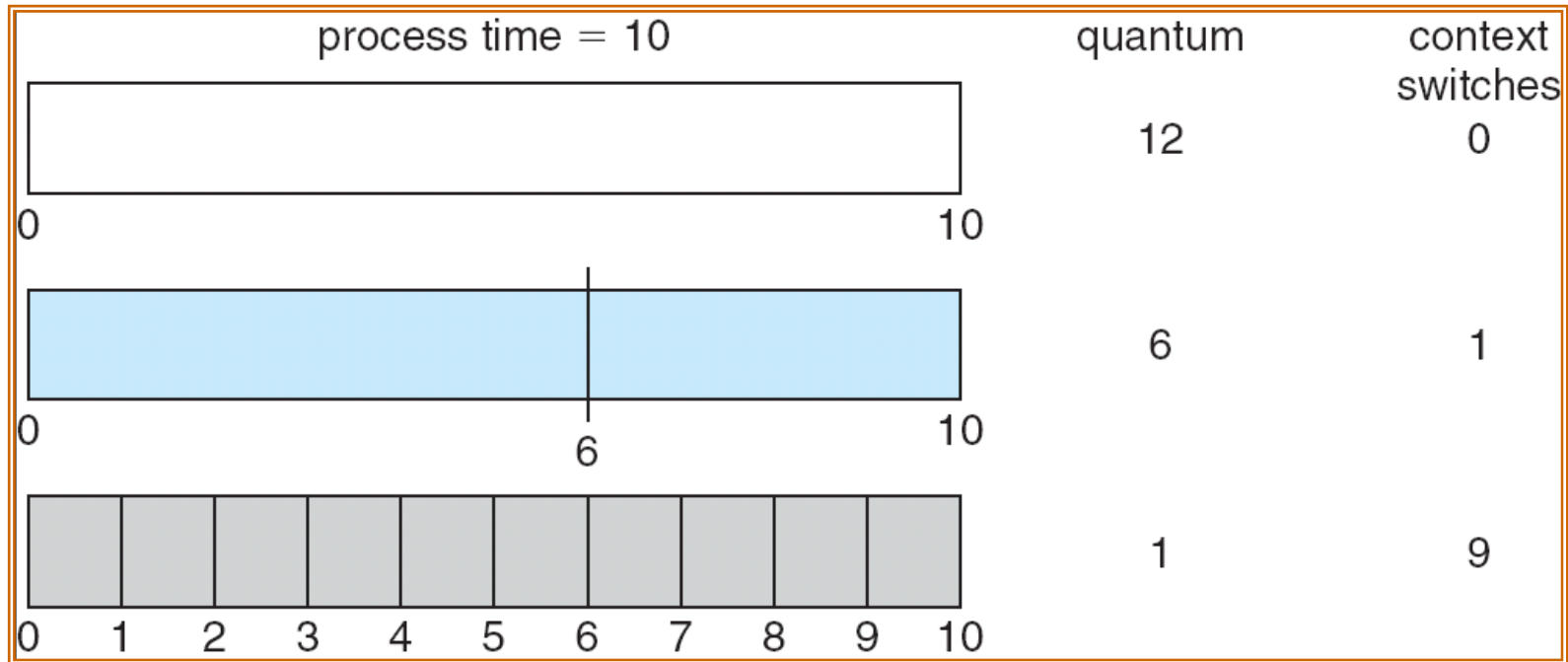
| <u>Process</u> | <u>Burst Time</u> |
|----------------|-------------------|
| $P_1$          | 53                |
| $P_2$          | 17                |
| $P_3$          | 68                |
| $P_4$          | 24                |

- The Gantt chart is:

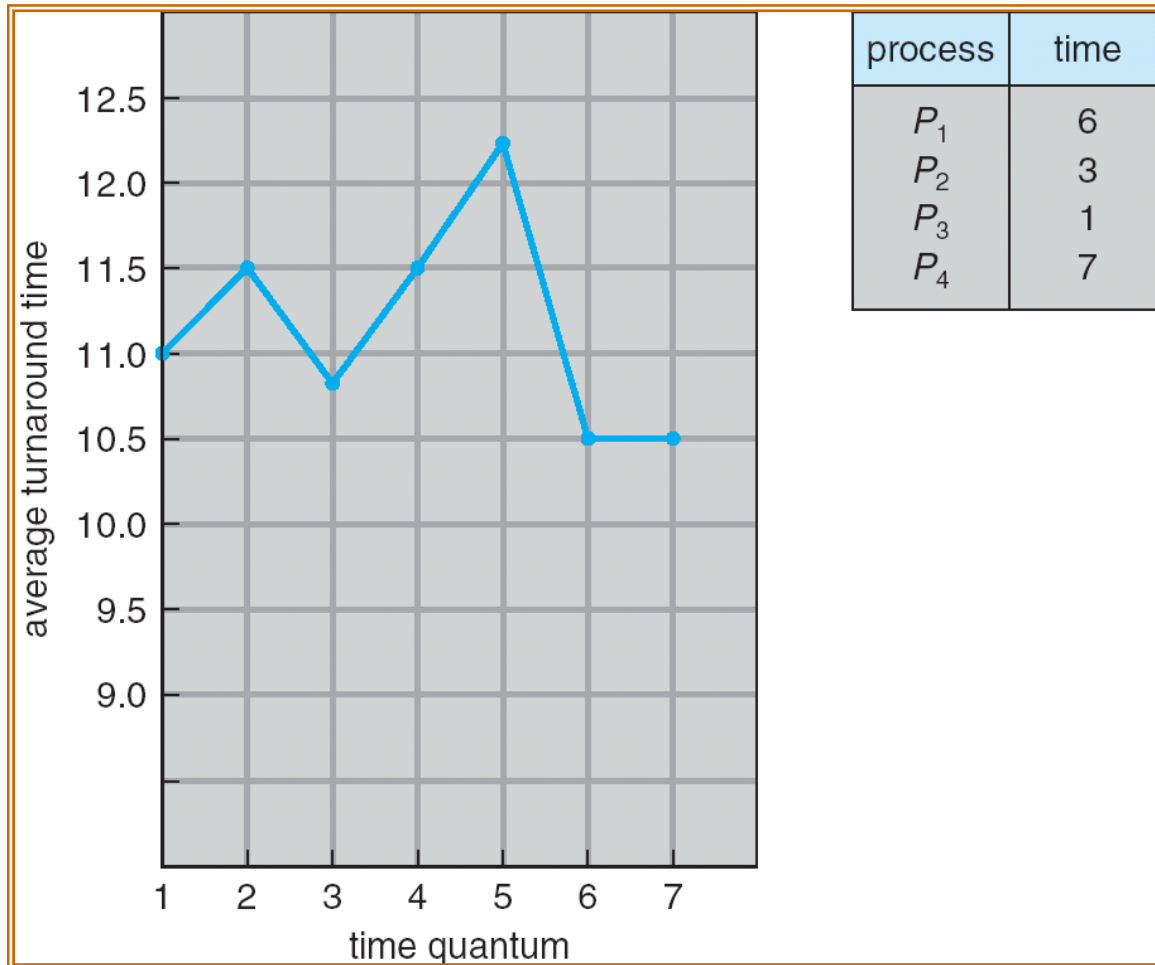


- Typically, higher average turnaround than SJF, but better *response*

# Time Quantum and Context Switch Time



# Turnaround Time Varies With The Time Quantum



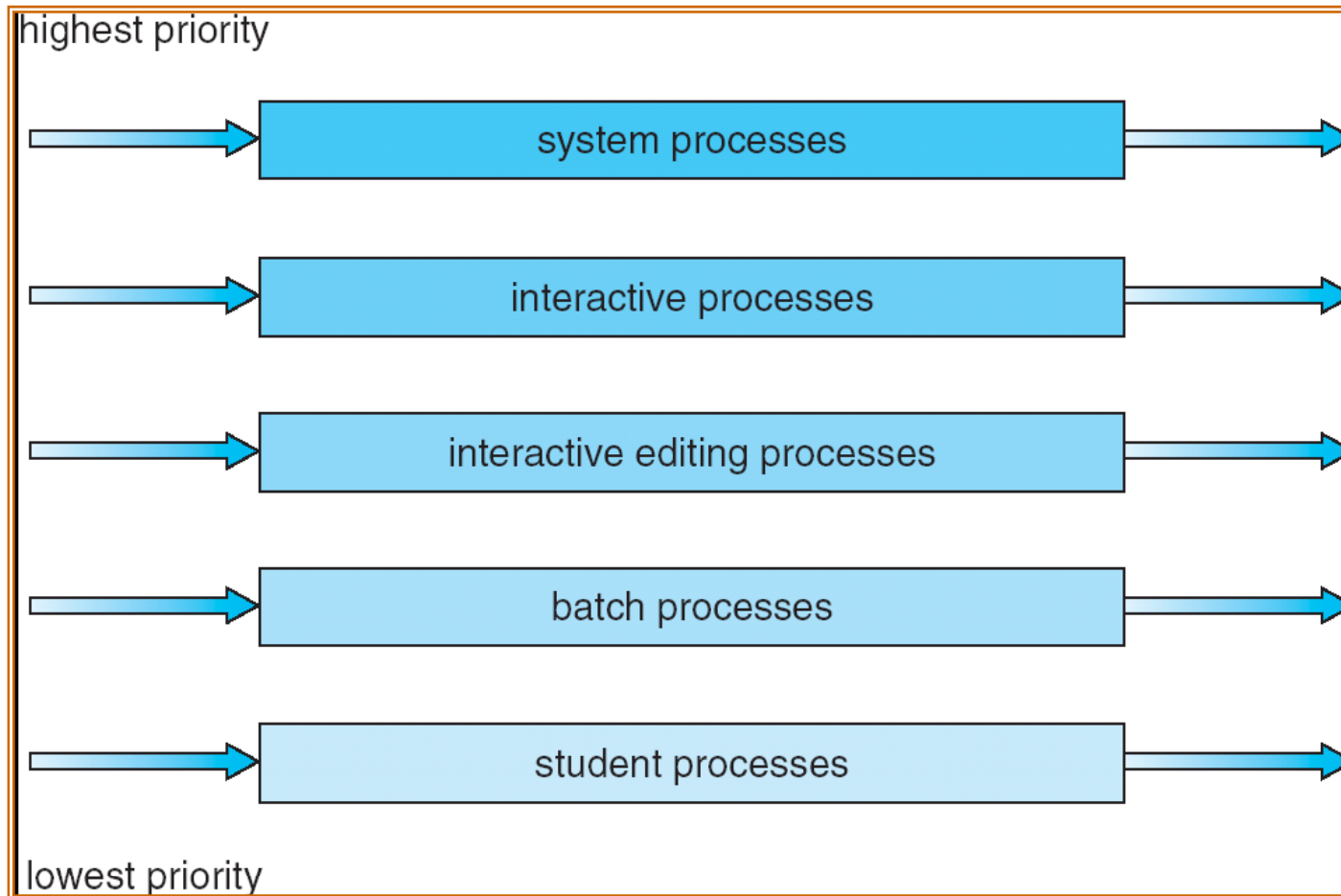
| process | time |
|---------|------|
| $P_1$   | 6    |
| $P_2$   | 3    |
| $P_3$   | 1    |
| $P_4$   | 7    |



# 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 Queues

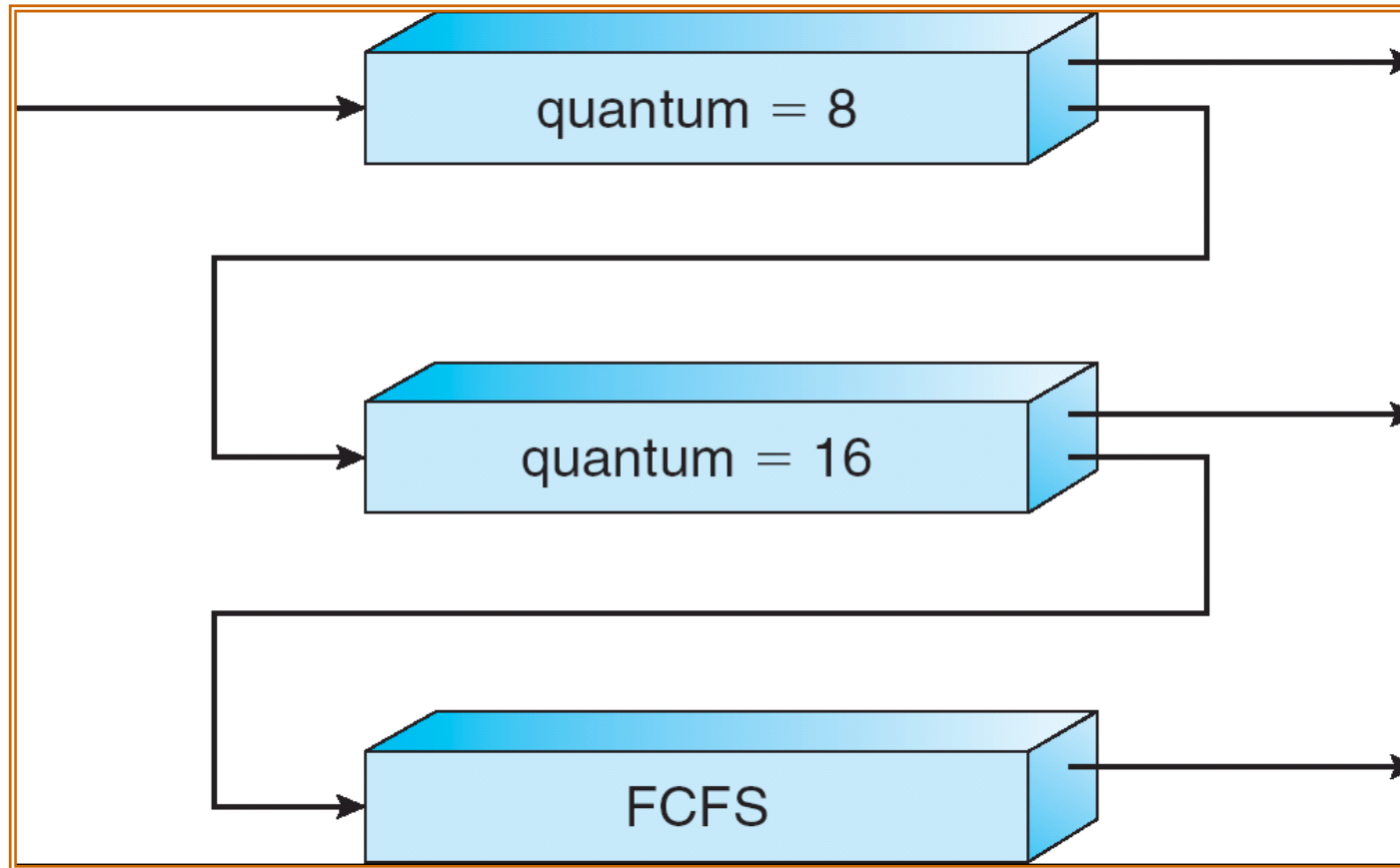
- Three queues:

- $Q_0$  – 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

```
#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_t init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread_attr_t setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* set the scheduling policy - FIFO, RT, or OTHER */
    pthread_attr_t setschedpolicy(&attr, SCHED_OTHER);
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);
}
```

# 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



# Thread Scheduling



- Local Scheduling

- ▣ How the threads library decides which thread to put onto an available LWP
- ▣ **Process-contention-scope (PCS)**

- Global Scheduling

- ▣ How the kernel decides which kernel thread to run next
- ▣ **System-contention-scope (SCS)**

# 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_t init(&attr);
    /* set the scheduling algorithm to PROCESS or
    SYSTEM */
    pthread_attr_t setscope(&attr, PTHREAD SCOPE
    SYSTEM);
    /* set the scheduling policy - FIFO, RT, or OTHER */
    pthread_attr_t setschedpolicy(&attr, SCHED OTHER);
    /* create the threads */
    for (i = 0; i < NUM THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);

    /* 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
- *Load sharing*
- *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

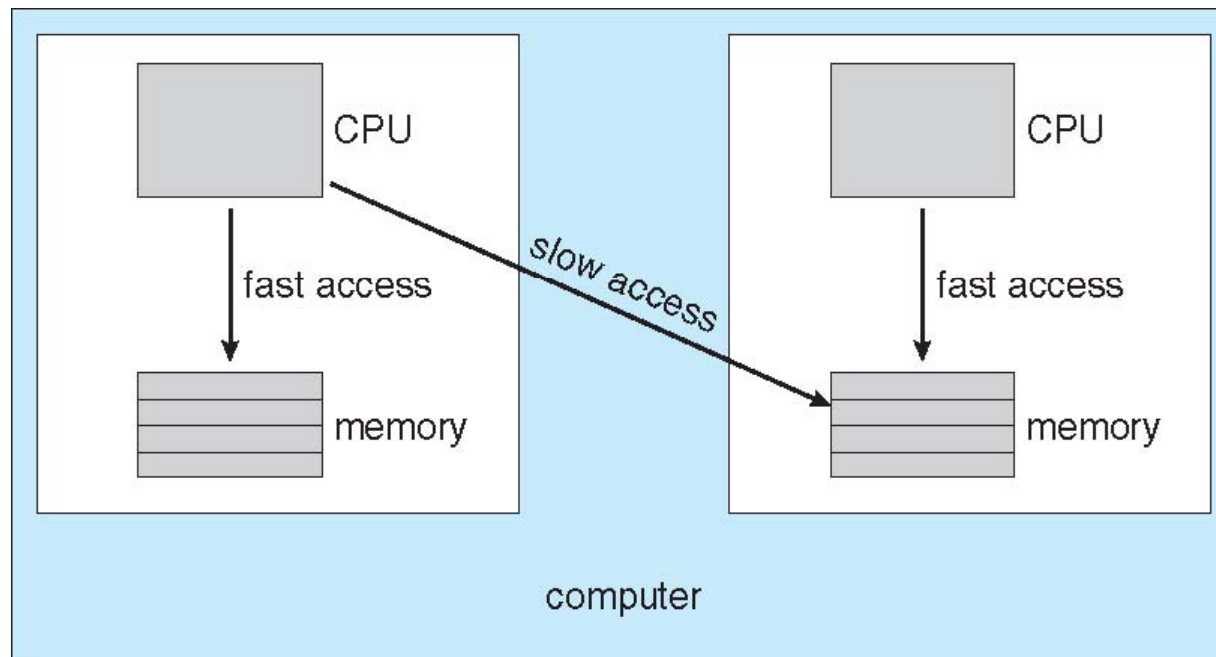
# Multiple-Processor Scheduling



- Processor Affinity
  - Avoid migration of processes from one processor to another.
  - Soft affinity (mostly, but not guaranteed)
  - hard affinity (no migration)

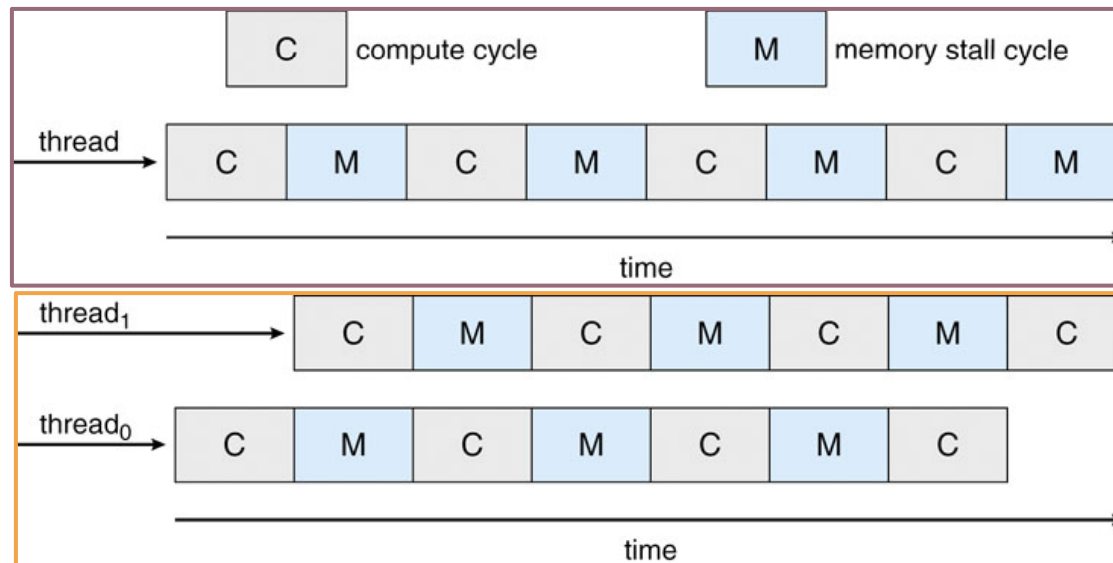
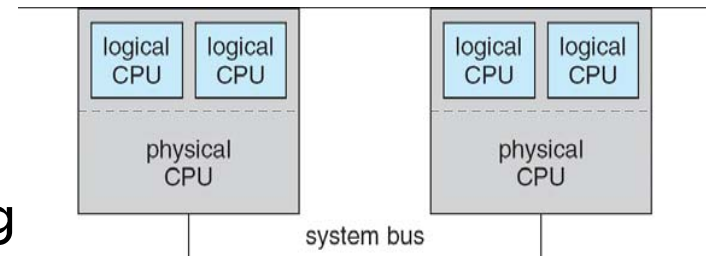
# NUMA and CPU Scheduling

- Non-uniform memory access (NUMA) also affect affinity of processor and memory.



# Multi-core and multi-threaded

- 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



# Real-Time Scheduling

- *Hard real-time systems*

- required to complete a critical task within a guaranteed amount of time
- Special-purpose software and hardware

- *Soft real-time computing*

- requires that critical processes receive priority over less fortunate ones
- Real-time processes
  - The priority must not degrade over time.
  - The dispatch latency must be small.



# Real-Time Scheduling (cont.)

- Most versions of UNIX
  - ▣ wait system call to complete.
  - ▣ Dispatch latency can be long.
  
- To allow system calls to be preemptible.
  - ▣ Insert preemption points
  - ▣ Make the entire kernel preemptible (Solaris 2)
    - Use synchronization mechanisms to protect kernel data structure.

# Real-Time Scheduling (cont.)



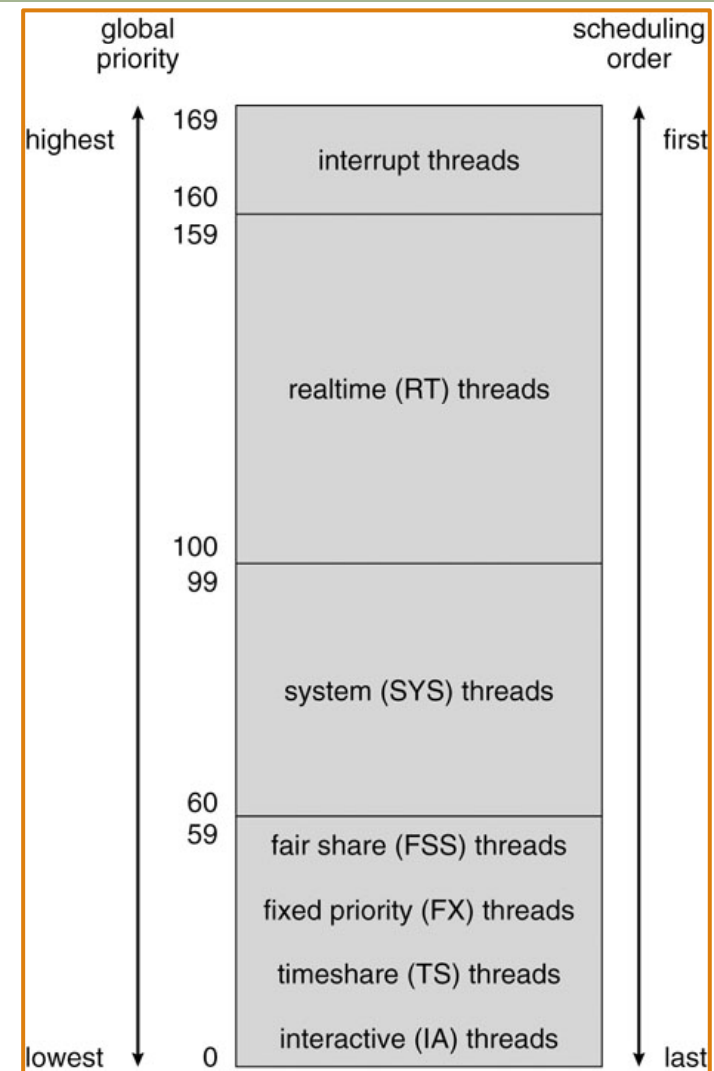
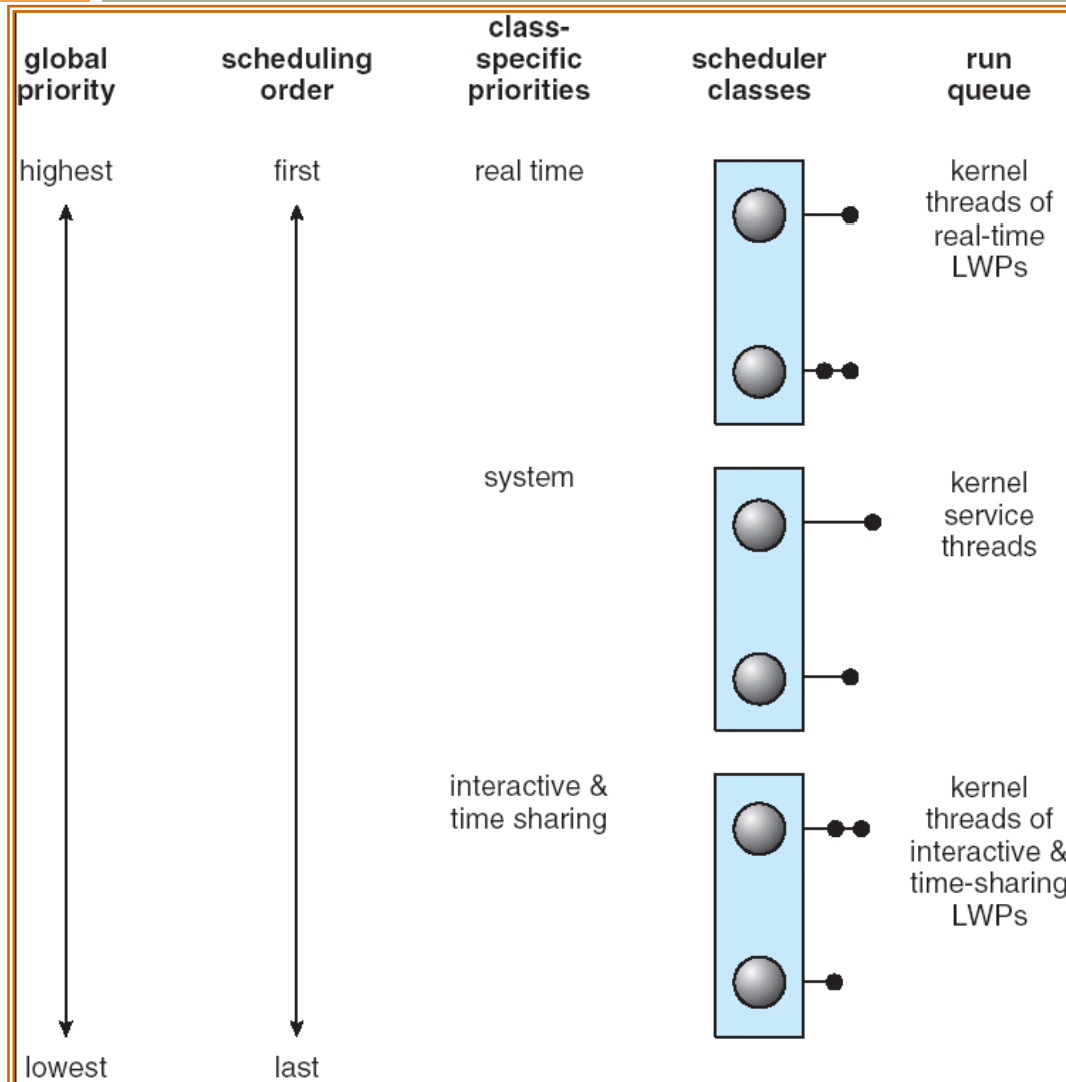
- What happen if the high-priority process needs to read or modify kernel data being access by lower-priority process?
- Problem: priority inversion
- Priority-inheritance protocol
  - ▣ The low-priority processes inherit the high priority until the resources is released.

# Operating System Examples



- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

# Solaris 2 Scheduling



# Solaris Dispatch Table for time-sharing and interactive threads

| 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                |

A higher priority value means higher priority.

# Windows XP

- A soft real-time OS.
- Priorities are divided into two classes:
  - ▣ The variable class: 1~15
  - ▣ The real-time class: 16~31
- The base priority + the relative priority
  - ▣ Low the thread priority when it is interrupted (time).
  - ▣ Boost the priority when a thread is released from “wait”.
  - ▣ Give good response time to interactive threads.
  - ▣ The foreground process: time slice X 3.

# 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

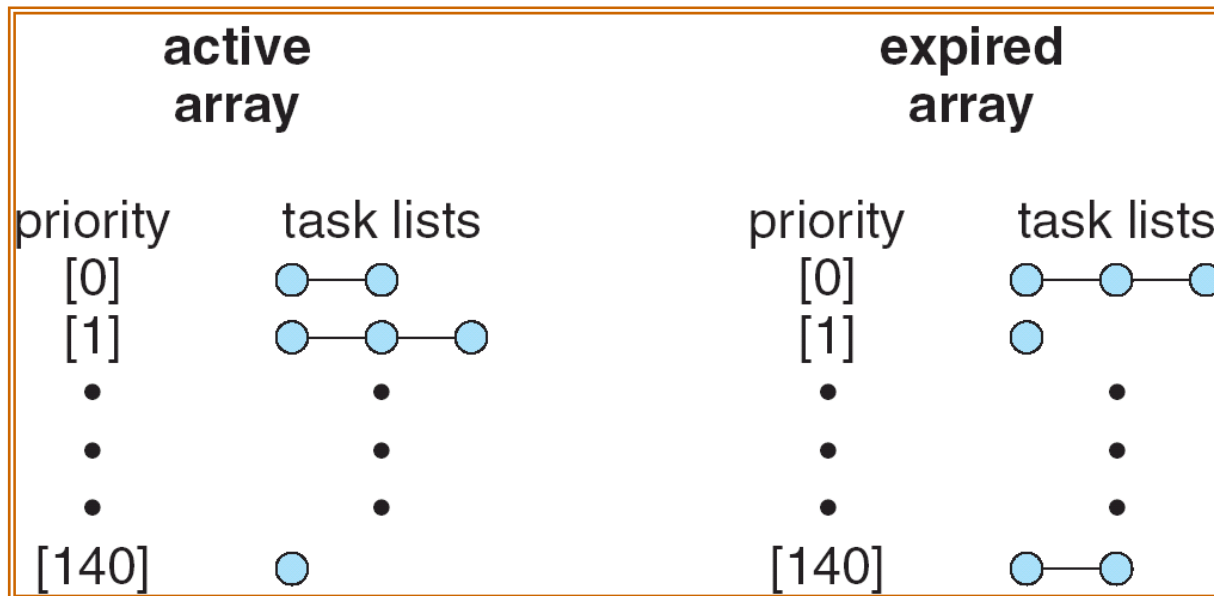
- Two algorithms: time-sharing and real-time
- Time-sharing
  - ▣ Prioritized credit-based – process with most credits is scheduled next
  - ▣ Credit subtracted when timer interrupt occurs
  - ▣ When credit = 0, another process chosen
  - ▣ When all runnable processes have credit = 0, recrediting occurs
    - Based on factors including priority and history
    - E.g. Credits = credits/2 + priority
- Real-time
  - ▣ Soft real-time
  - ▣ Posix.1 b compliant – two classes
    - FCFS and RR
    - Highest priority process always runs first



# The Relationship Between Priorities and Time-slice length

| <u>numeric priority</u> | <u>relative priority</u> |                 | <u>time quantum</u> |
|-------------------------|--------------------------|-----------------|---------------------|
| 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
  - ▣ Too specific
  
- Queueing models
  - ▣ The mathematics of complicated algorithms can be difficult to work with.
  - ▣ May not be accurate.

# Algorithm Evaluation (cont.)



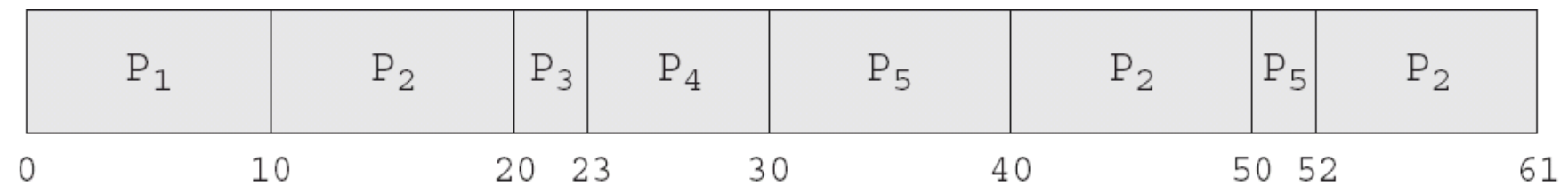
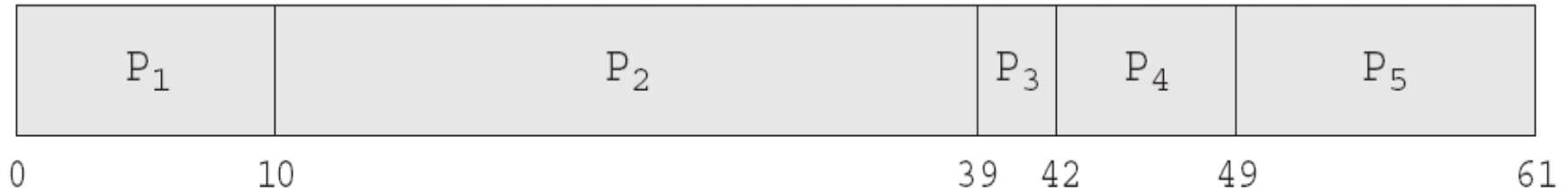
## □ Simulation

- According to probability distributions.
- Trace tapes: recording the sequence of actual events.
- Can be expensive

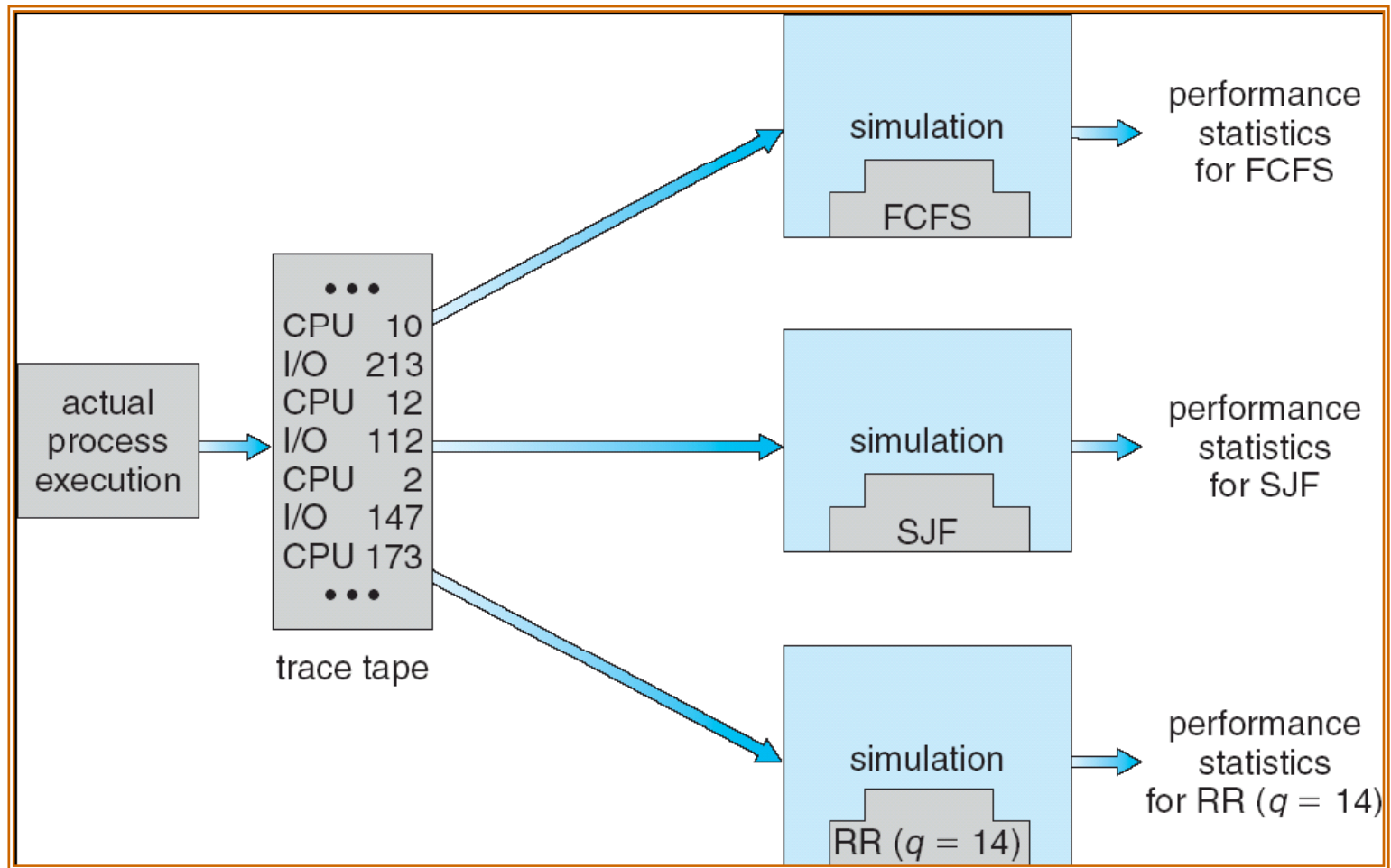
## □ Implementation

- Put the actual algorithm in the real system
- The cost is high
- The environment will change

# Deterministic modeling



# 5.15



END OF CHAPTER 5

