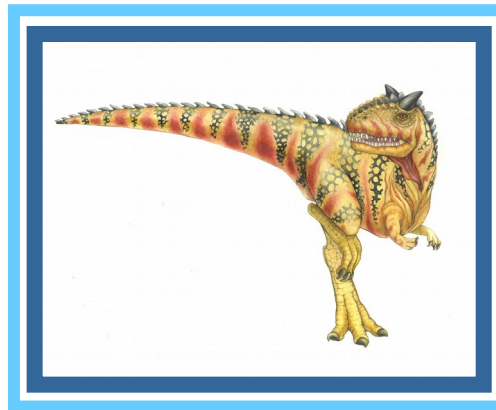# Chapter 5b: Advanced CPU Scheduling

Chapter 5b: CPU Scheduling

- Thread Scheduling

- Multiple-Processor Scheduling

- Real-Time CPU Scheduling

- Operating Systems Examples

- Algorithm Evaluation

Objectives

- Fill

Thread Scheduling

- Distinction between user-level and kernel-level threads

- When threads supported, threads scheduled, not processes

- 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

  - Typically done via priority set by programmer

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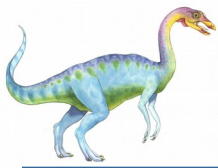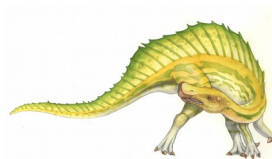
Pthread Scheduling

- 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
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

Pthread Scheduling API

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
    int i, scope;
    pthread_t tid[NUM THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }
```

Pthread Scheduling API

```
      /* set the scheduling algorithm to PCS or SCS */

      pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

      /* 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)
{

    /* do some work ... */

    pthread_exit(0);

}
```

Multiprocessor CPU Scheduling

■ CPU scheduling is more complex when multiple CPUs are available

■ We assume that we are dealing with **homogeneous processors** within a multiprocessor system. This is, all processors are identical

■ Two approaches to multiprocessor scheduling

- **Asymmetric multiprocessing** – all scheduling decisions, I/O processing, and other system activities are handled by a single processor—the master server. The other processors execute only user code

- **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
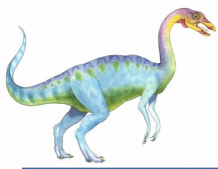
Multicore Processors

- SMP systems allow several threads to run concurrently by providing multiple physical processors.

- Multicore processor hardware places multiple computing cores on the same physical chip.

  - Each core maintains its architectural state and thus appears to the operating system to be a separate CPU.

- SMP systems that use multicore processors are faster and consume less power than systems in which each processor has its own physical chip.

- Multicore processors complicate scheduling issues

Memory Stall in Multicore Systems

- When a processor accesses memory, it spends a significant amount of time waiting for the data to become available. This situation, known as a memory stall

- Stall occurs because modern processors operate at much faster speeds than memory.
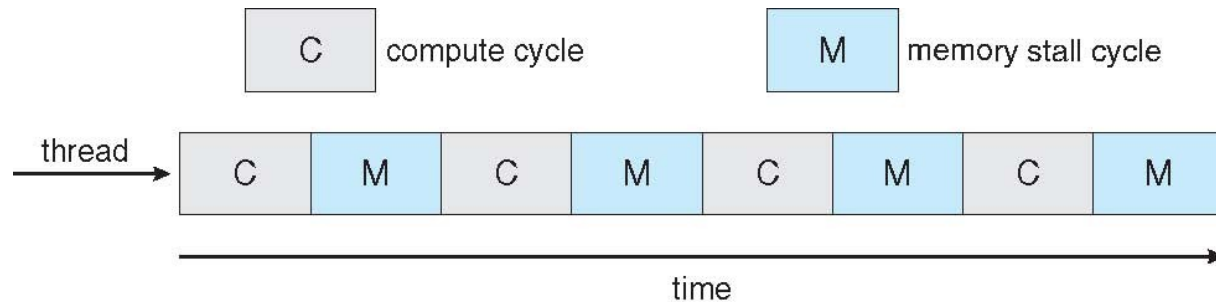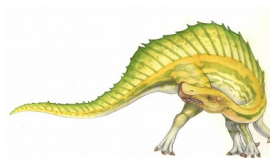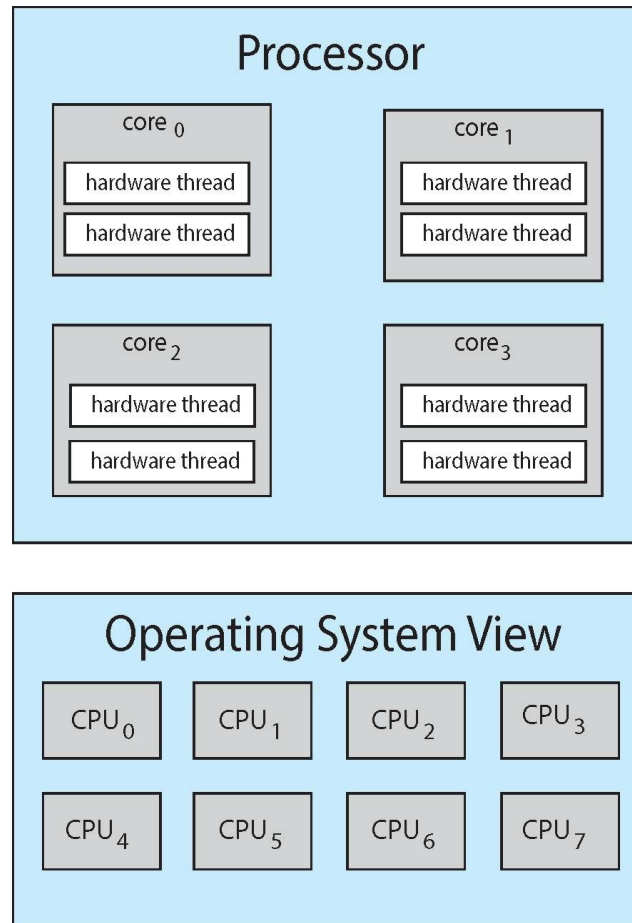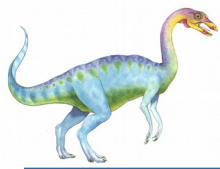
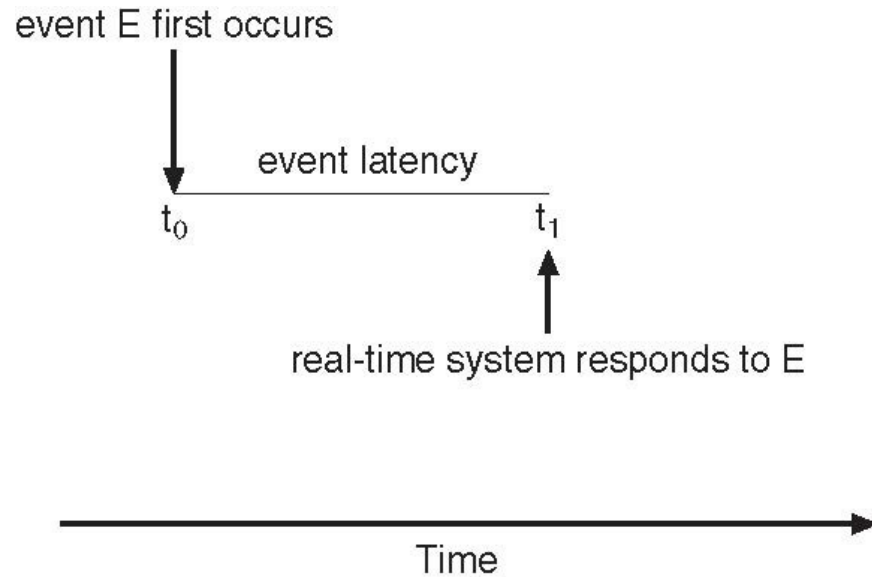# Multithreaded Multicore System
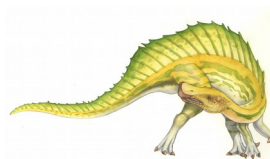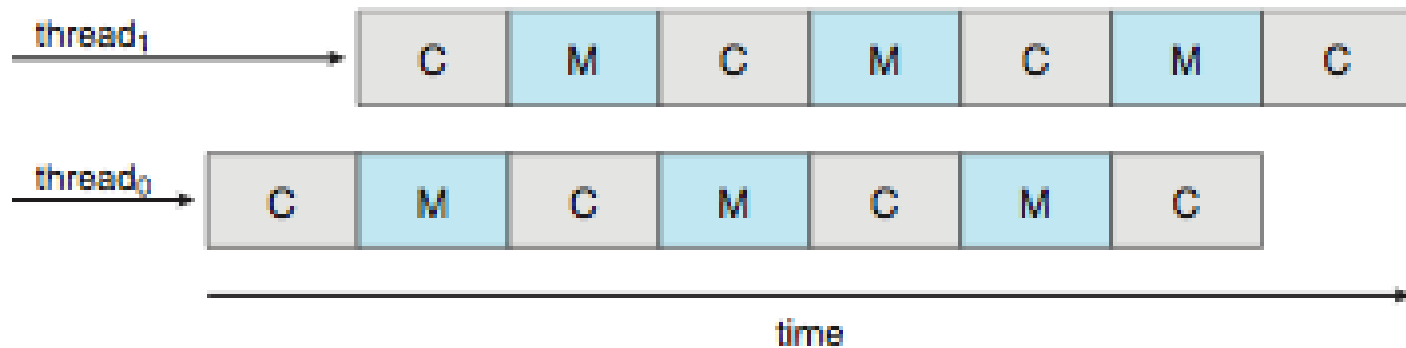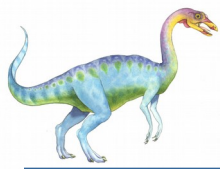
Figure 5.14

Figure 5.13
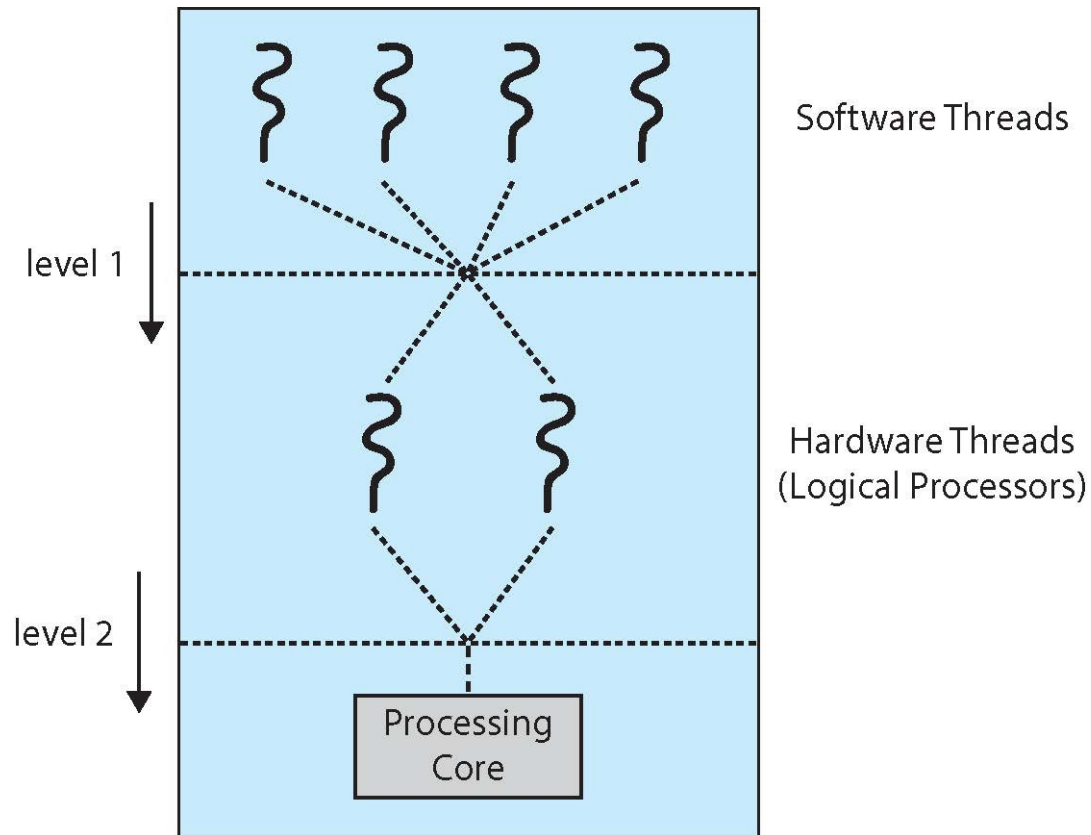
NUMA and CPU Scheduling

computer
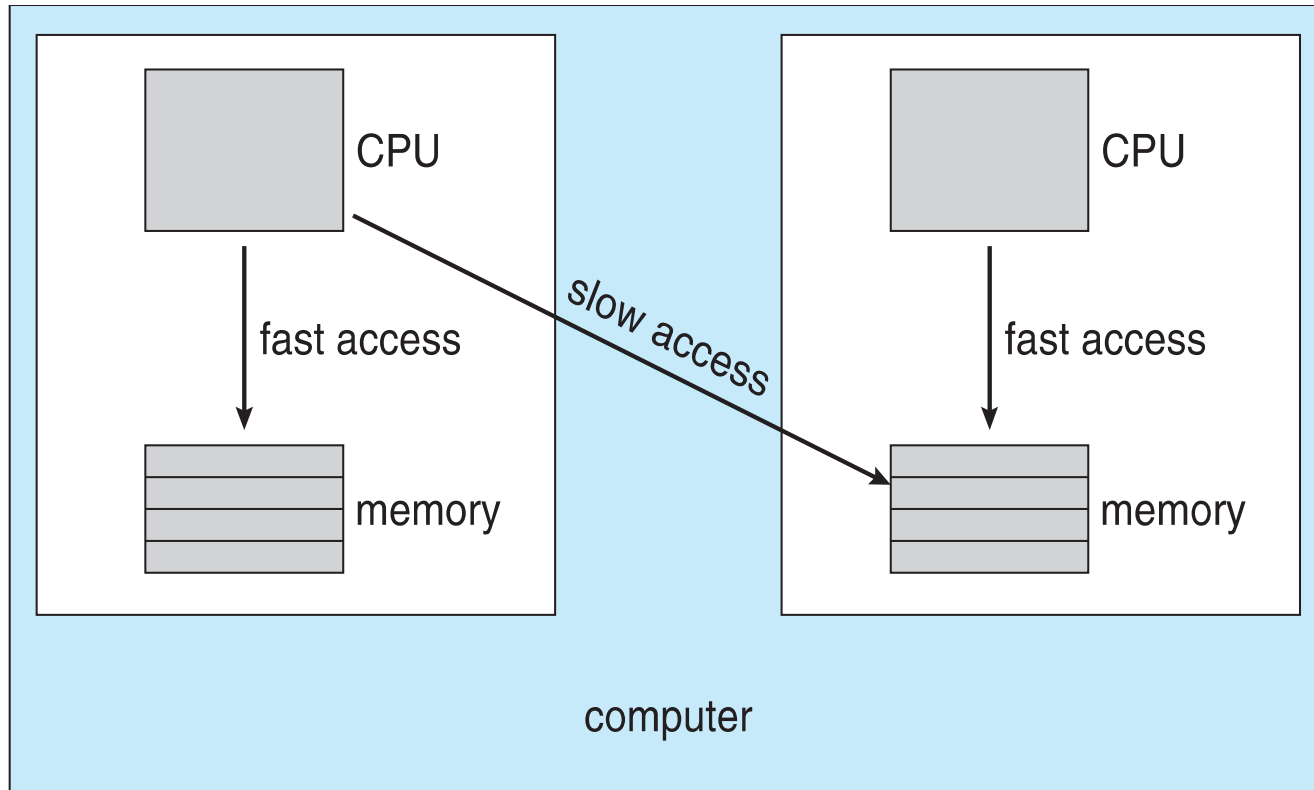
Note that memory-placement algorithms can also consider affinity

Figure 5.17

event E first occurs

event latency

t_0 t_1

real-time system responds to E

Time

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
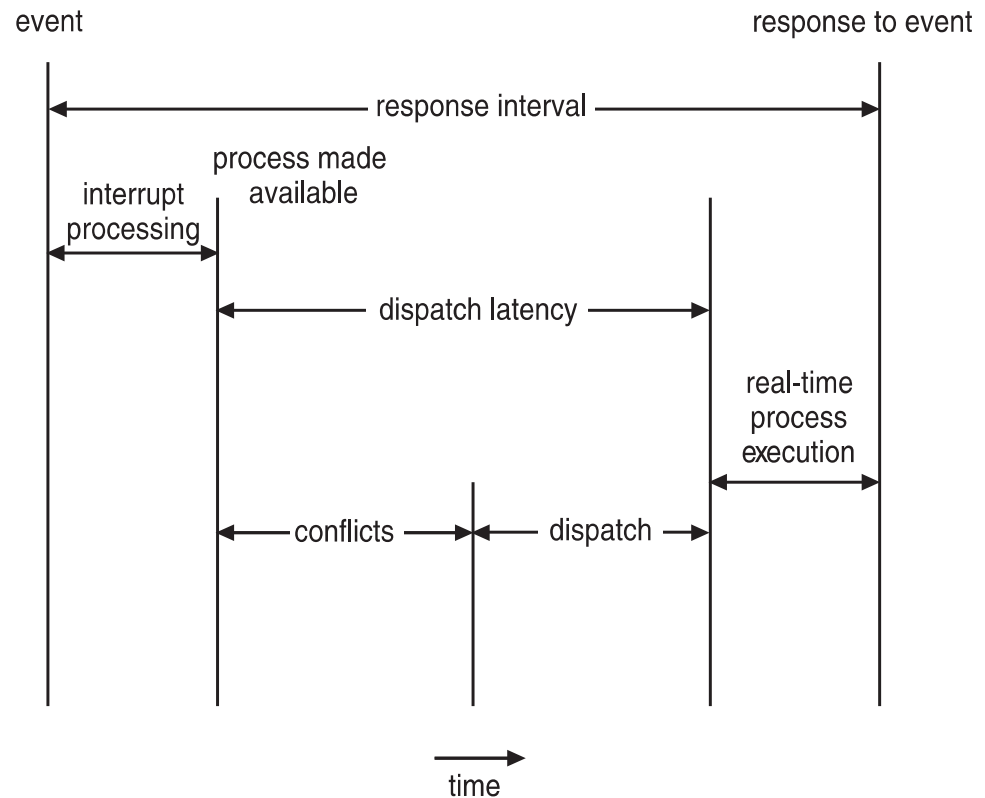
Real-Time CPU Scheduling

- Can present obvious challenges

- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled

- **Hard real-time systems** – task must be serviced by its deadline

- Two types of latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
  2. Dispatch latency – time for schedule to take current process off CPU and switch to another
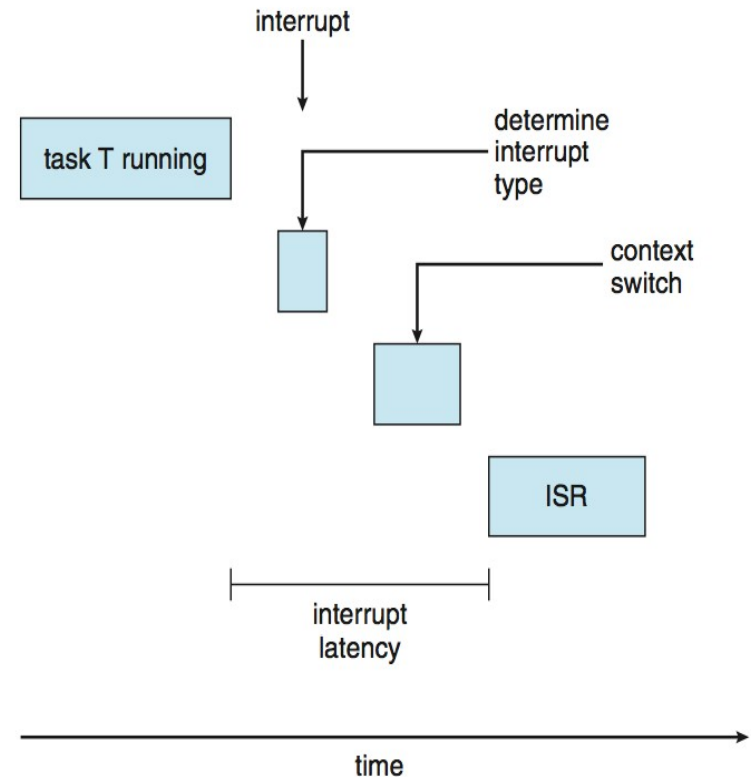
Real-Time CPU Scheduling (Cont.)

- Conflict phase of dispatch latency:

  1. Preemption of any process running in kernel mode

  2. Release by low-priority process of resources needed by high-priority processes

Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time t, deadline d, period p
  - $0 \leq t \leq d \leq p$
  - **Rate** of periodic task is $1/p$
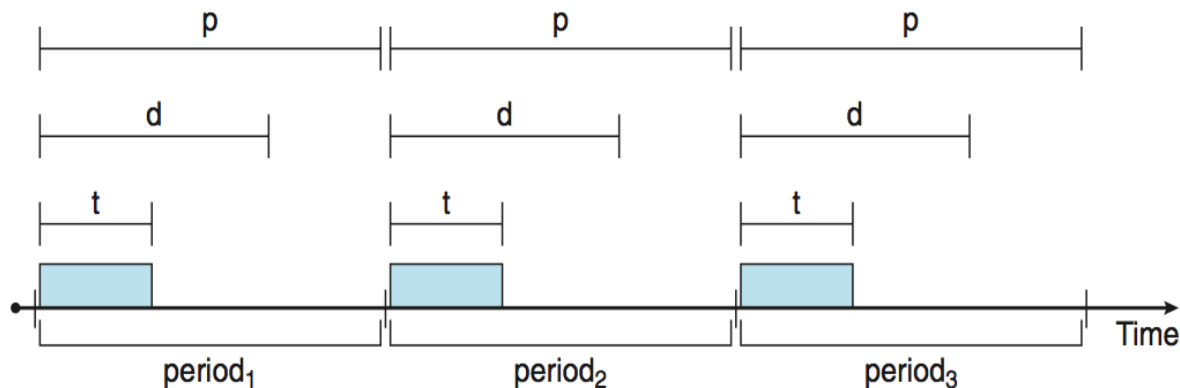
Virtualization and Scheduling

- Virtualization software schedules multiple guests onto CPU(s)

- Each guest doing its own scheduling

  - Not knowing it doesn't own the CPUs

  - Can result in poor response time

  - Can effect time-of-day clocks in guests

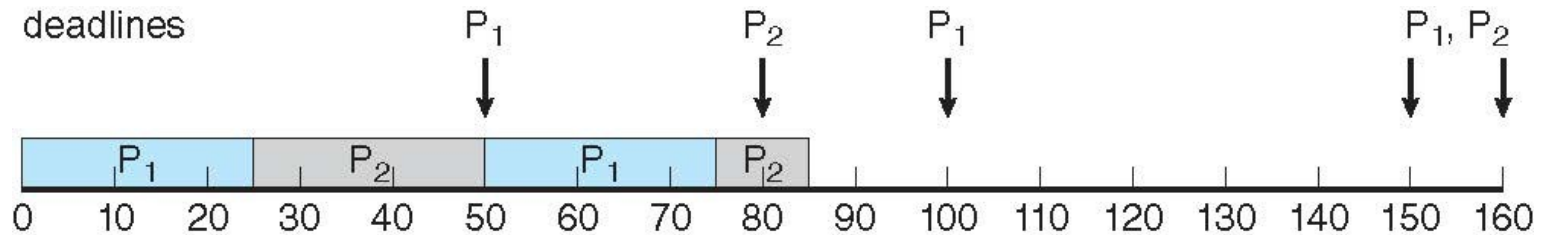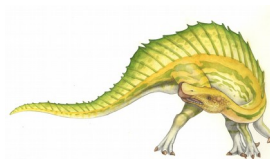- Can undo good scheduling algorithm efforts of guests
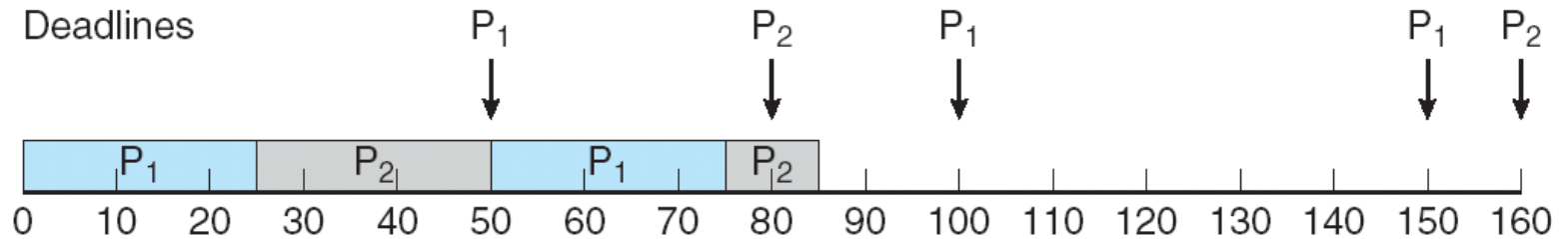
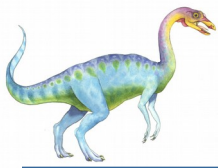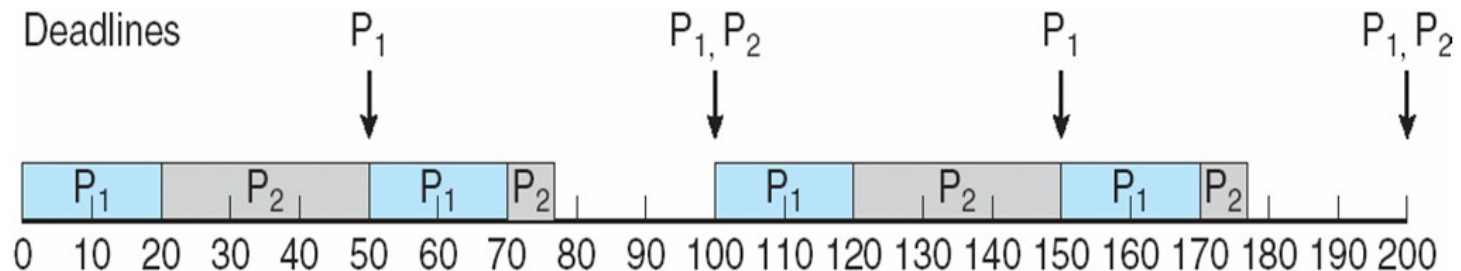# Figure 5.21
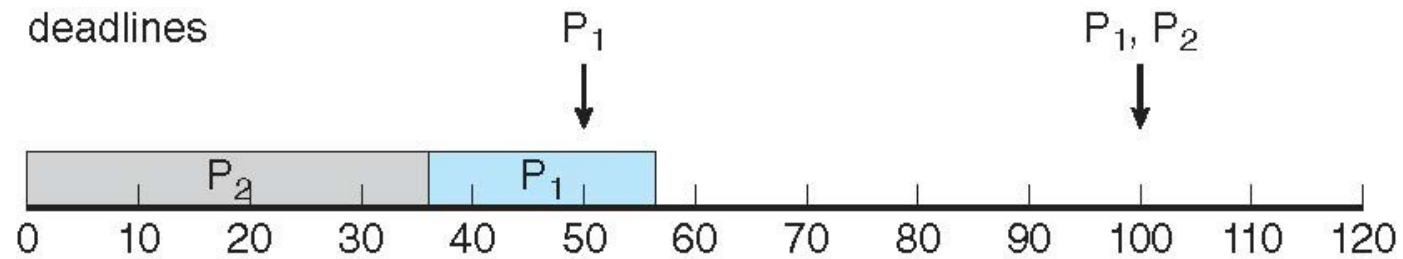
Figure 5.22
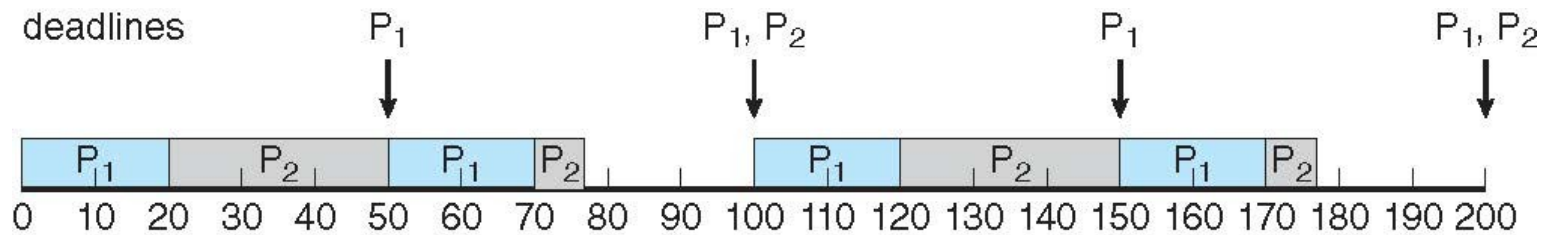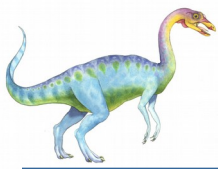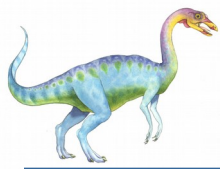
Figure 5.23
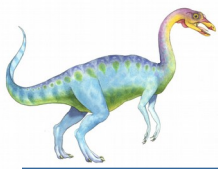
Missed Deadlines with Rate Monotonic Scheduling

Rate Montonic Scheduling

- A priority is assigned based on the inverse of its period

- Shorter periods = higher priority;

- Longer periods = lower priority

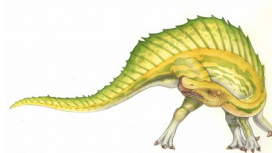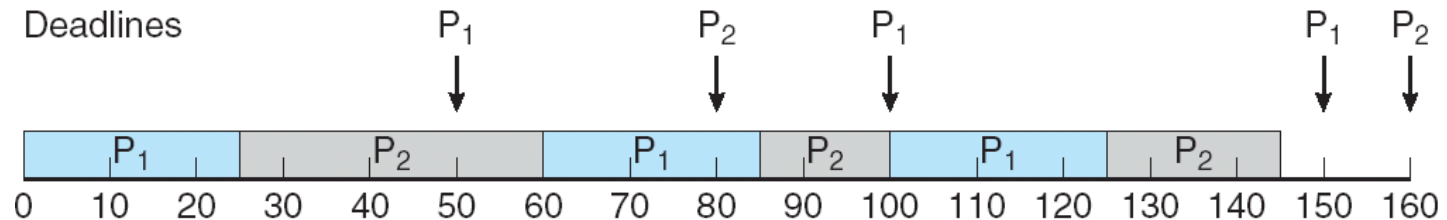- P_1 is assigned a higher priority than P_2.

Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:

  the earlier the deadline, the higher the priority;

  the later the deadline, the lower the priority

Proportional Share Scheduling

- T shares are allocated among all processes in the system

- An application receives N shares where $N < T$

- This ensures each application will receive N / T of the total processor time
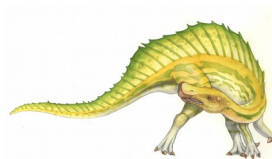
POSIX Real-Time Scheduling

- The POSIX.1b standard

- API provides functions for managing real-time threads

- Defines two scheduling classes for real-time threads:

- SCHED_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority

- SCHED_RR - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority

- Defines two functions for getting and setting scheduling policy:

- `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`

- `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

POSIX Real-Time Scheduling API

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
    int i, policy;
    pthread_t_tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* get the current scheduling policy */
    if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
        fprintf(stderr, "Unable to get policy.\n");
    else {
        if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
        else if (policy == SCHED_RR) printf("SCHED_RR\n");
        else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
    }
```

```
    /* set the scheduling policy - FIFO, RR, or OTHER */
    if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)

        fprintf(stderr, "Unable to set policy.\n");

    /* 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)
{

    /* do some work ... */

    pthread_exit(0);

}
```
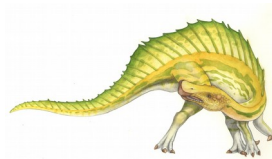
Operating System Examples

- Linux scheduling

- Windows scheduling

- Solaris scheduling

Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm

- Version 2.5 moved to constant order $O(1)$ scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
  - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger q
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
    ‣ Two priority arrays (active, expired)
    ‣ Tasks indexed by priority
    ‣ When no more active, arrays are exchanged
  - Worked well, but poor response times for interactive processes
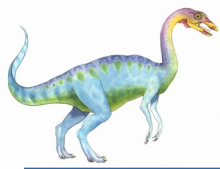
Linux Scheduling in Version 2.6.23 +

- ***Completely Fair Scheduler*** (CFS)

- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - 2 scheduling classes included, others can be added
    1. default
    2. real-time

- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases

- CFS scheduler maintains per task **virtual run time** in variable `vruntime`
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time

- To decide next task to run, scheduler picks task with lowest virtual run time

CFS Performance

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:
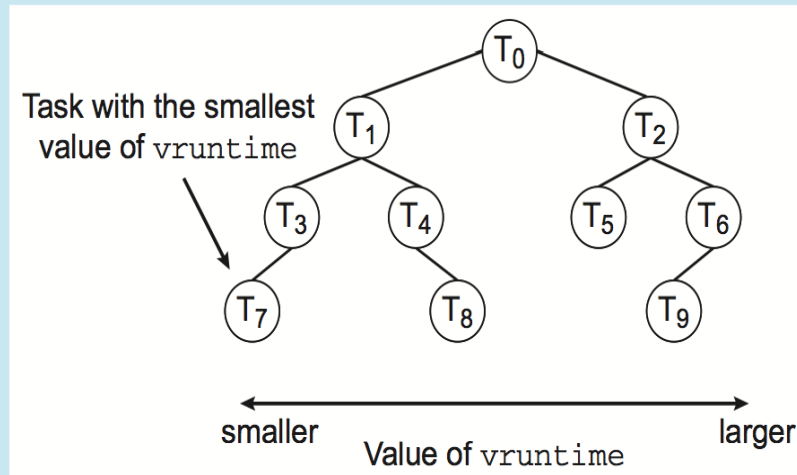
When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139

Real-Time	Normal
0 99	100 139

Higher ←———————————————————————→ Lower
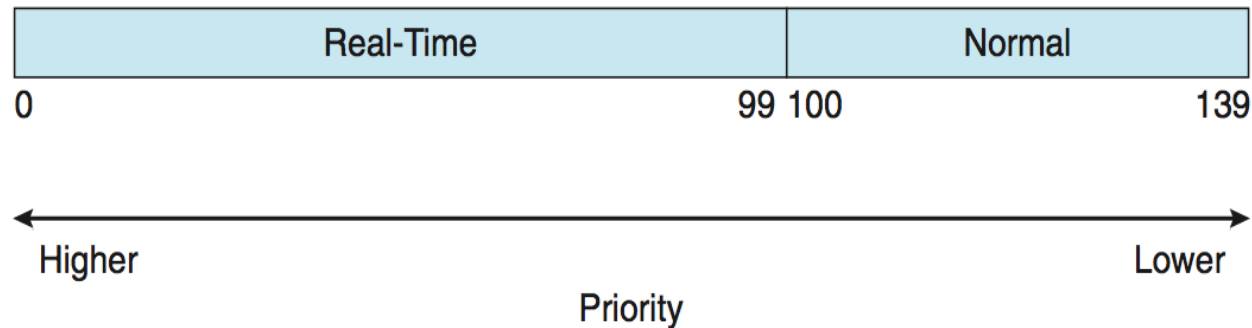
Priority

Figure 5.25
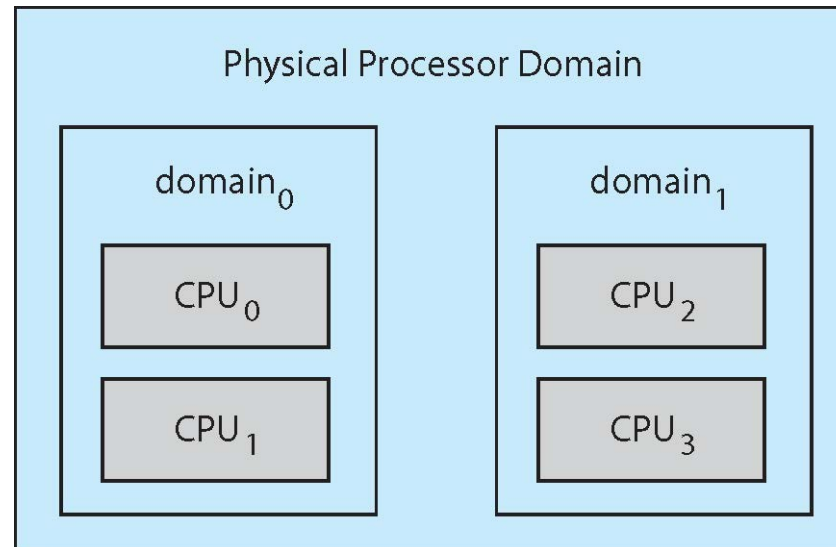
Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**

Windows Priority Classes

- Win32 API identifies several priority classes to which a process can belong
    - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
    - All are variable except REALTIME
- A thread within a given priority class has a relative priority
    - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base

Windows Priority Classes (Cont.)

- If wait occurs, priority boosted depending on what was waited for

- Foreground window given 3x priority boost

- Windows 7 added **user-mode scheduling** (**UMS**)

  - Applications create and manage threads independent of kernel

  - For large number of threads, much more efficient

  - UMS schedulers come from programming language libraries like C++ **Concurrent Runtime** (ConcRT) framework

Windows 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

Solaris

- Priority-based scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - Loadable table configurable by sysadmin

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
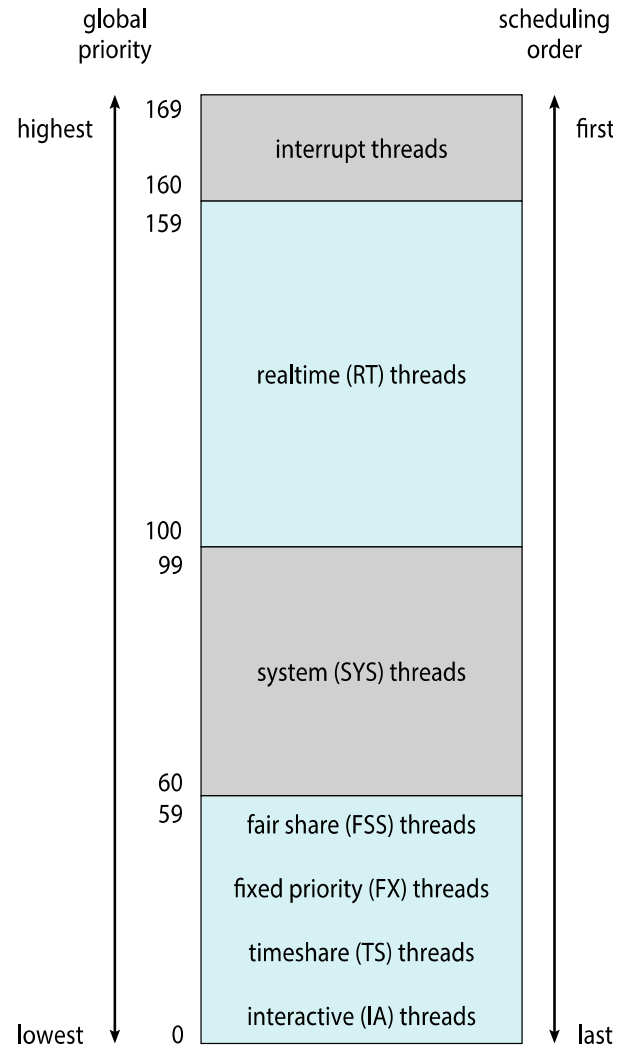
Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority

  - Thread with highest priority runs next

  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread

  - Multiple threads at same priority selected via RR

Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?

- Determine criteria, then evaluate algorithms

- **Deterministic modeling**

  - Type of **analytic evaluation**

  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload

- Consider 5 processes arriving at time 0:

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12
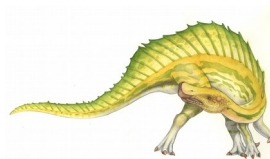
Deterministic Evaluation

- For each algorithm, calculate minimum average waiting time

- Simple and fast, but requires exact numbers for input, applies only to those inputs

  - FCS is 28ms:

P_1	P_2	P_3	P_4	P_5

  0 10   39 42   49   61

  - Non-preemptive SFJ is 13ms:

P_3	P_4	P_1	P_5	P_2

  0 3   10   20   32   61

  - RR is 23ms:

P_1	P_2	P_3	P_4	P_5	P_2	P_5	P_2

  0   10   20 23   30   40   50 52   61
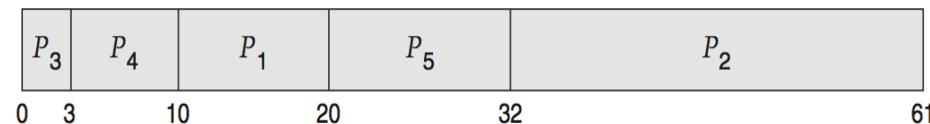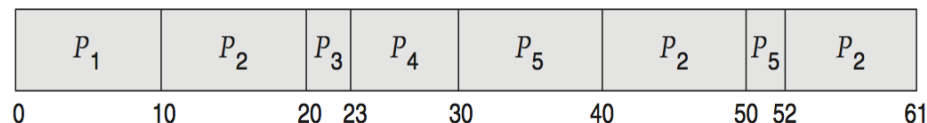
Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically

  - Commonly exponential, and described by mean

  - Computes average throughput, utilization, waiting time, etc

- Computer system described as network of servers, each with queue of waiting processes

  - Knowing arrival rates and service rates

  - Computes utilization, average queue length, average wait time, etc
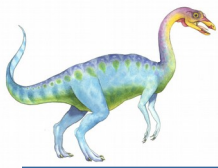
Little's Formula

- *n* = average queue length

- *W* = average waiting time in queue

- *λ* = average arrival rate into queue

- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:

    n = *λ* x *W*

    - Valid for any scheduling algorithm and arrival distribution

- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds
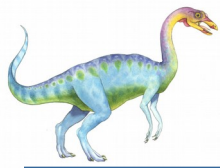
Simulations

- Queueing models limited

- **Simulations** more accurate

  - Programmed model of computer system

  - Clock is a variable

  - Gather statistics indicating algorithm performance

  - Data to drive simulation gathered via

    ‣ Random number generator according to probabilities

    ‣ Distributions defined mathematically or empirically

    ‣ Trace tapes record sequences of real events in real systems

Evaluation of CPU Schedulers by Simulation
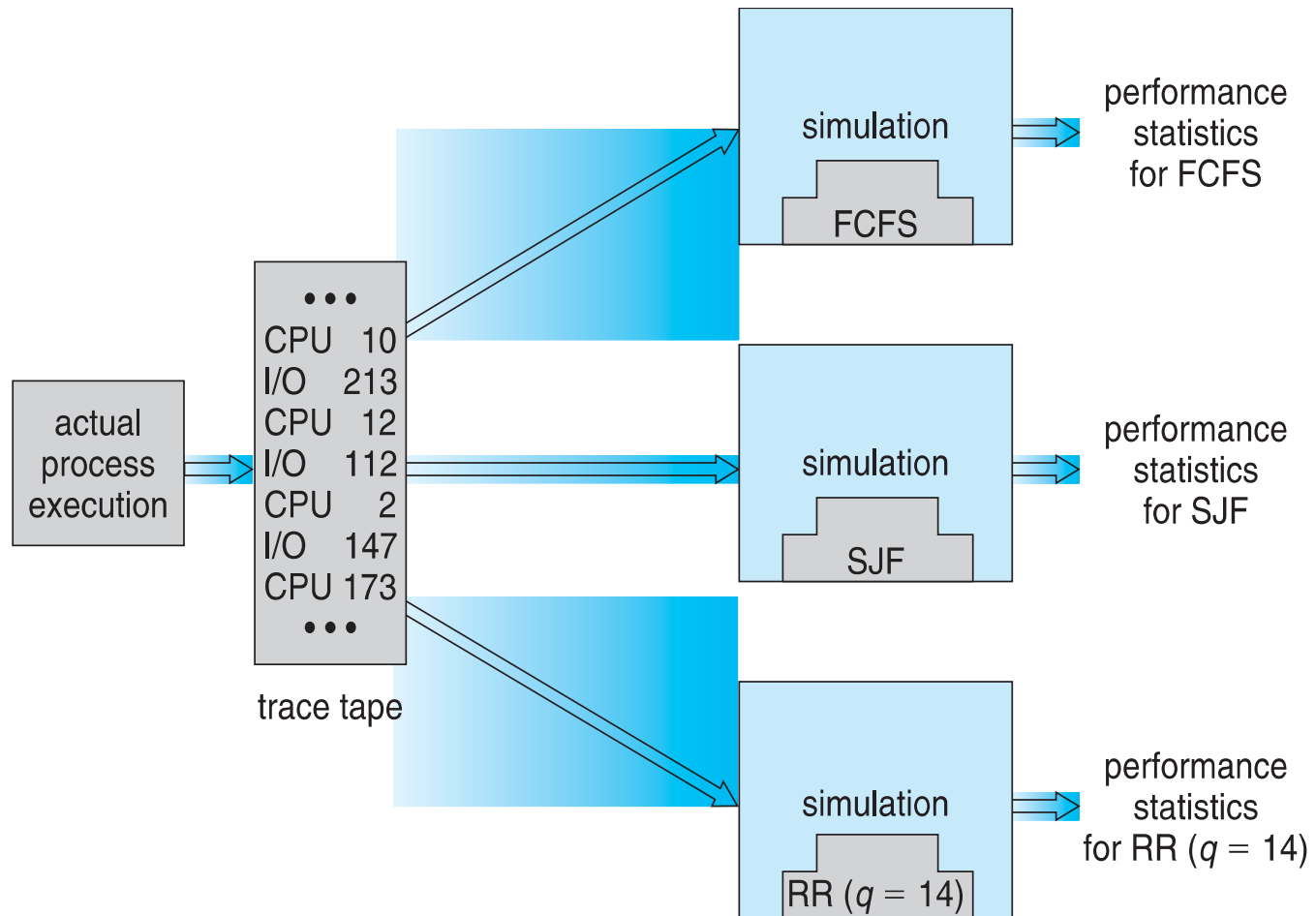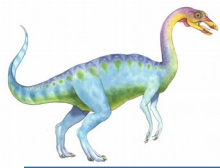
Implementation

- Even simulations have limited accuracy

- Just implement new scheduler and test in real systems

    - High cost, high risk

    - Environments vary

- Most flexible schedulers can be modified per-site or per-system

- Or APIs to modify priorities

- But again environments vary

End of Chapter 5b

Figure 5.01


```
              ⋮

load store
add store          ⎫
read from file     ⎬  CPU burst
                   ⎭

┌──────────────┐   ⎫
│ 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
└──────────────┘   ⎭

              ⋮
```
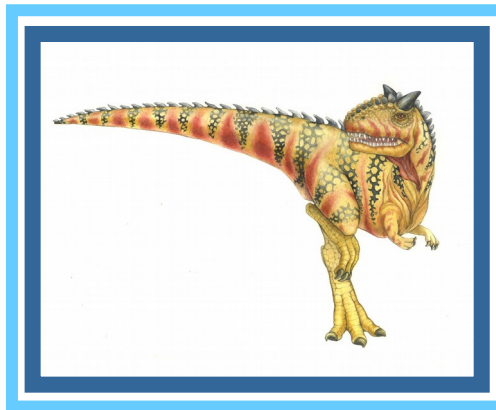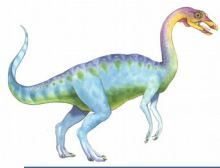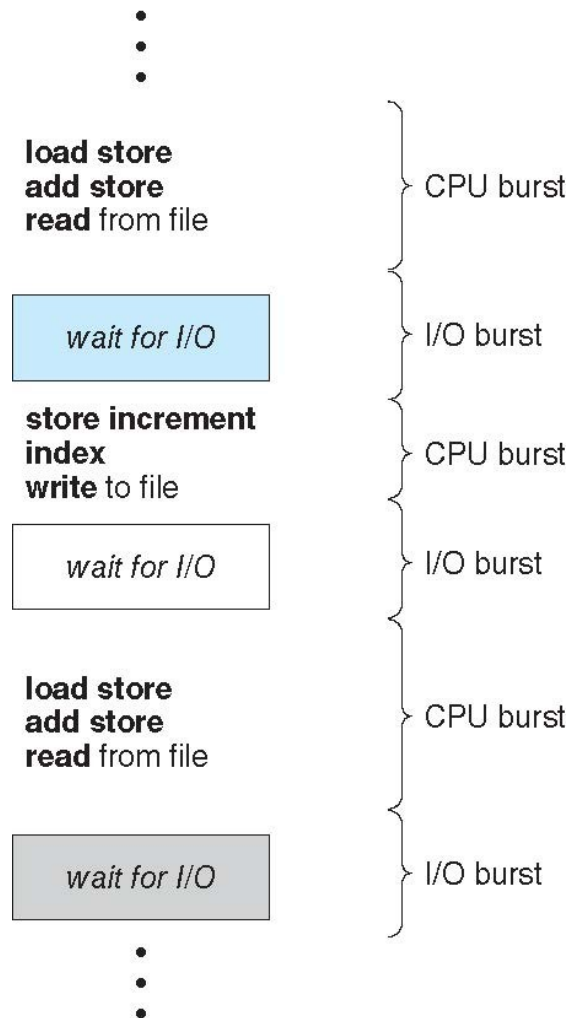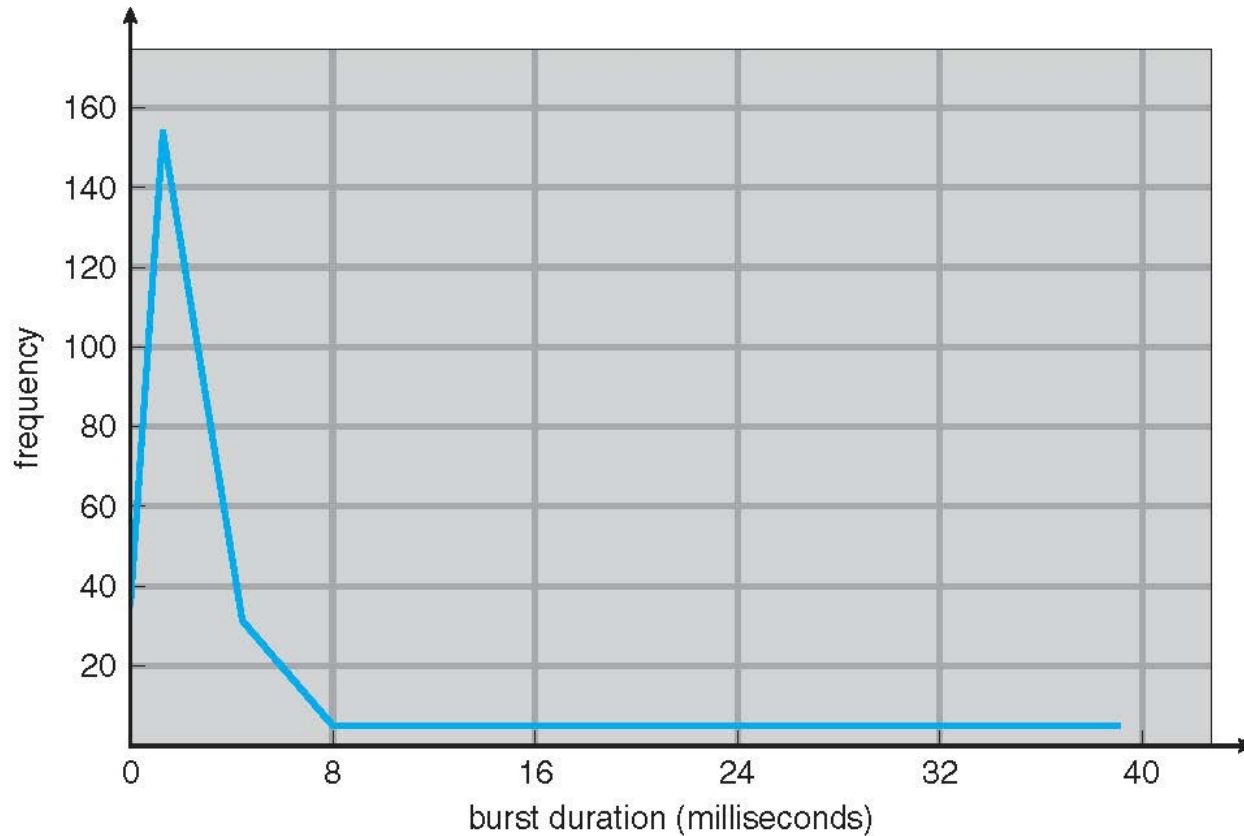
Figure 5.02
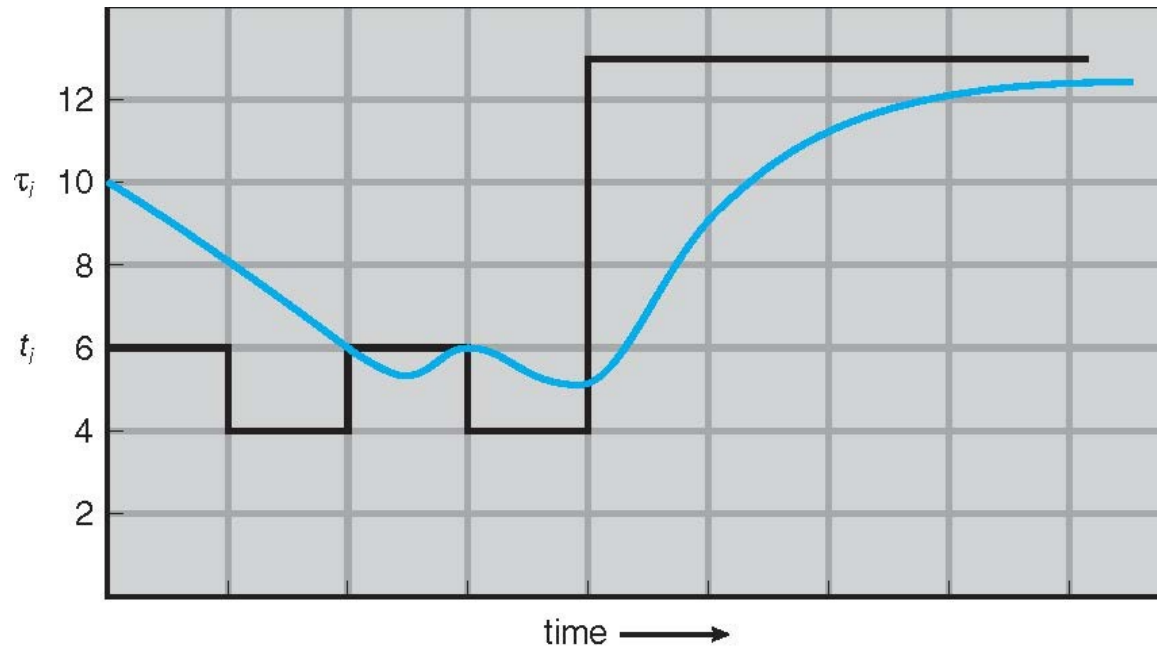
Figure 5.03

CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...
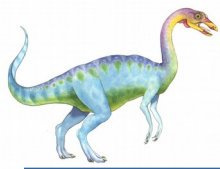
Figure 5.04
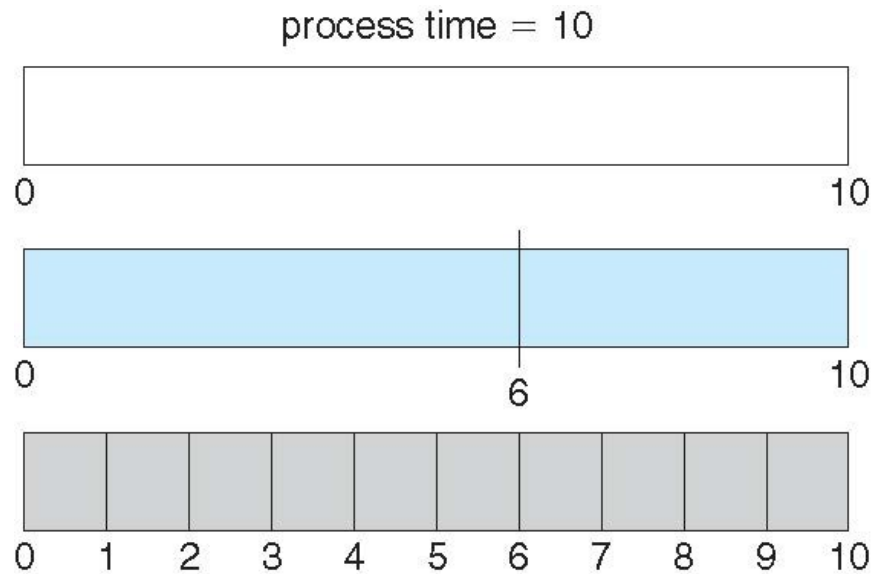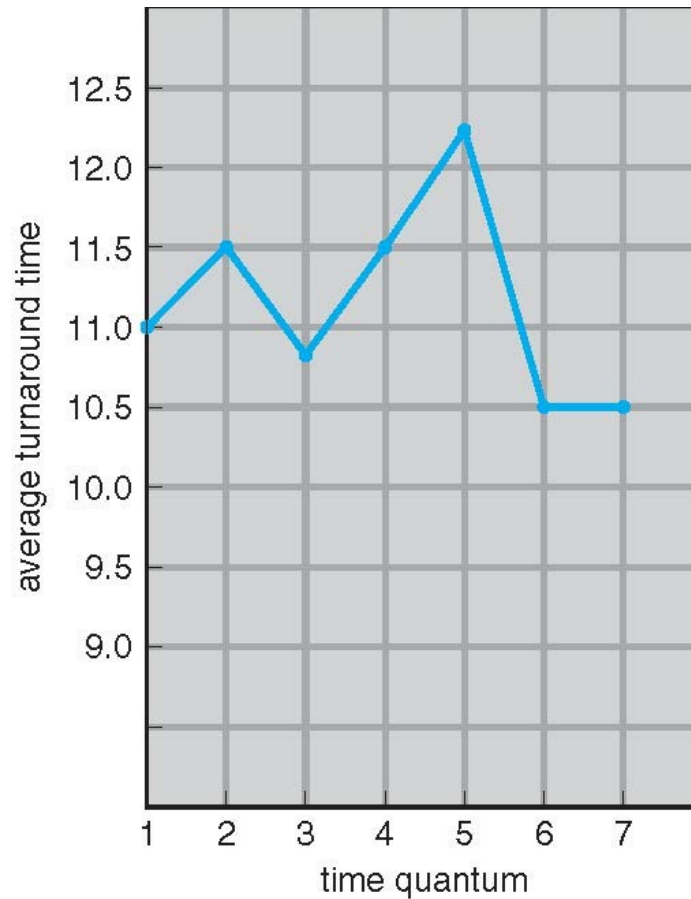
Figure 5.05

process	time
P_1	6
P_2	3
P_3	1
P_4	7
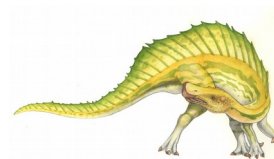
Figure 5.06
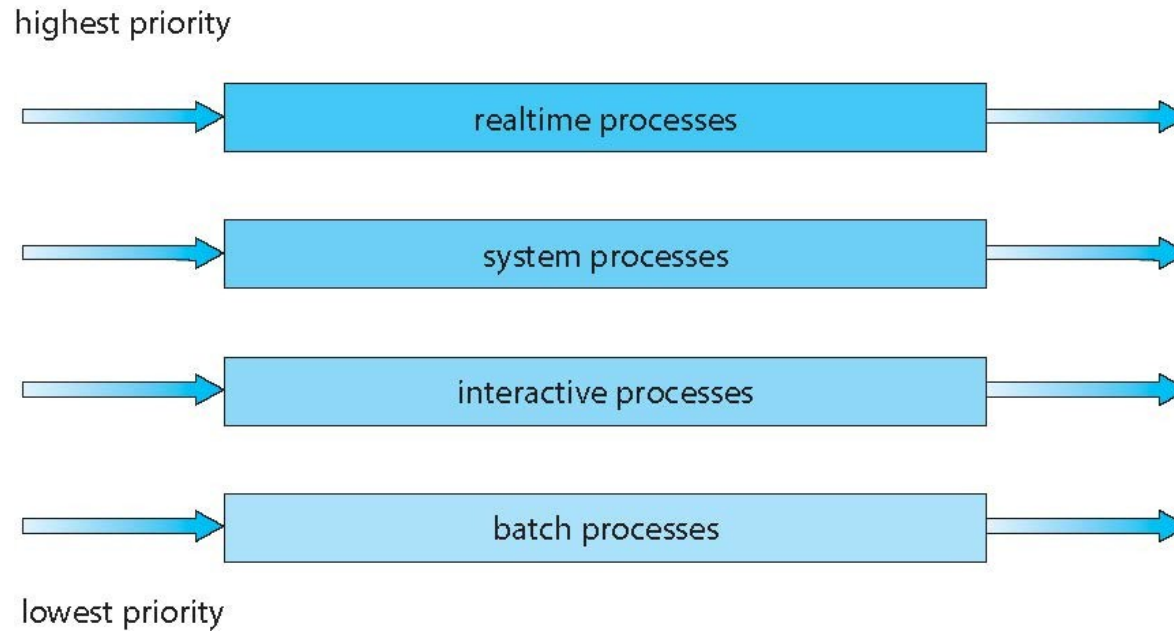
Figure 5.07
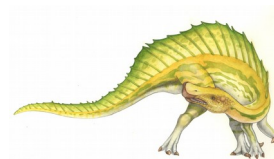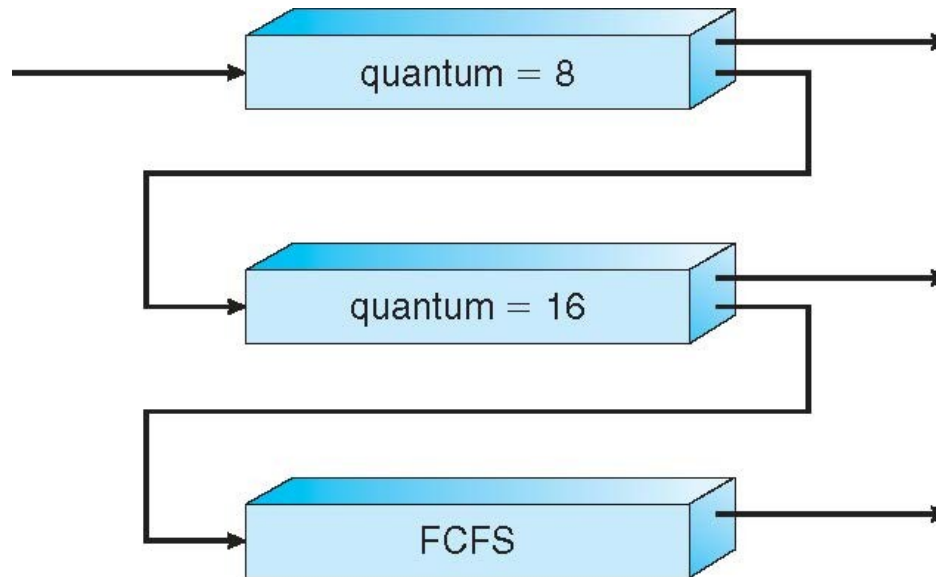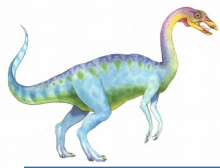
Figure 5.09

Place Holder

Figure 5.10

Figure 5.11

Figure 5.14

slow access

fast access

fast access

CPU

CPU

memory

memory

computer
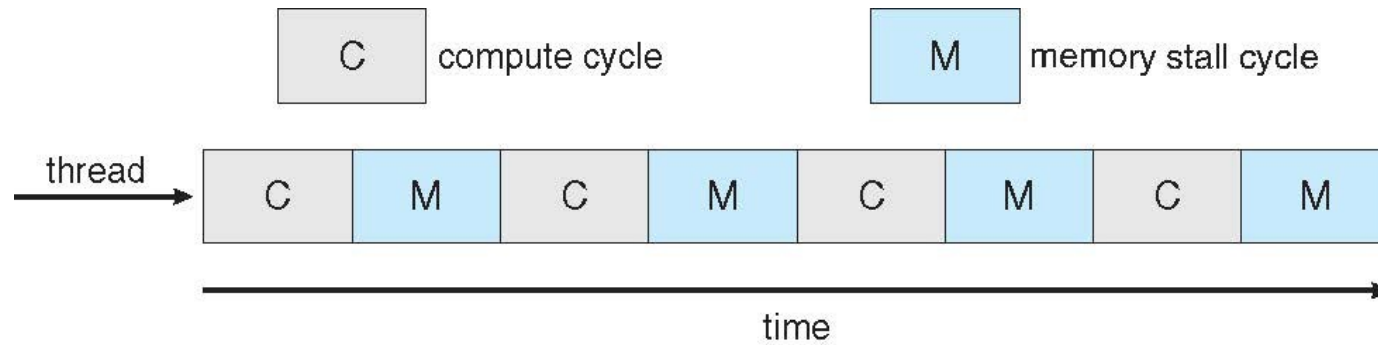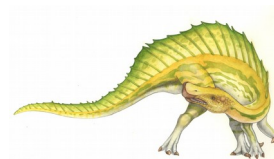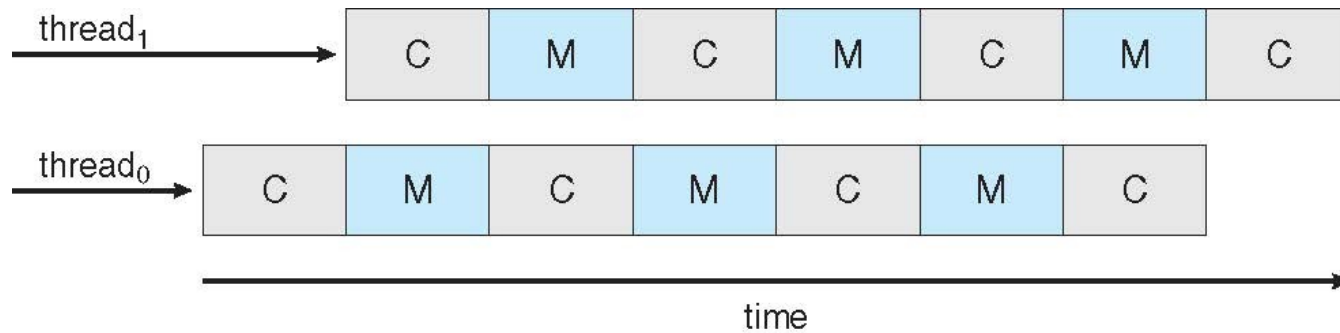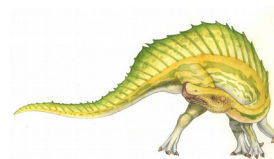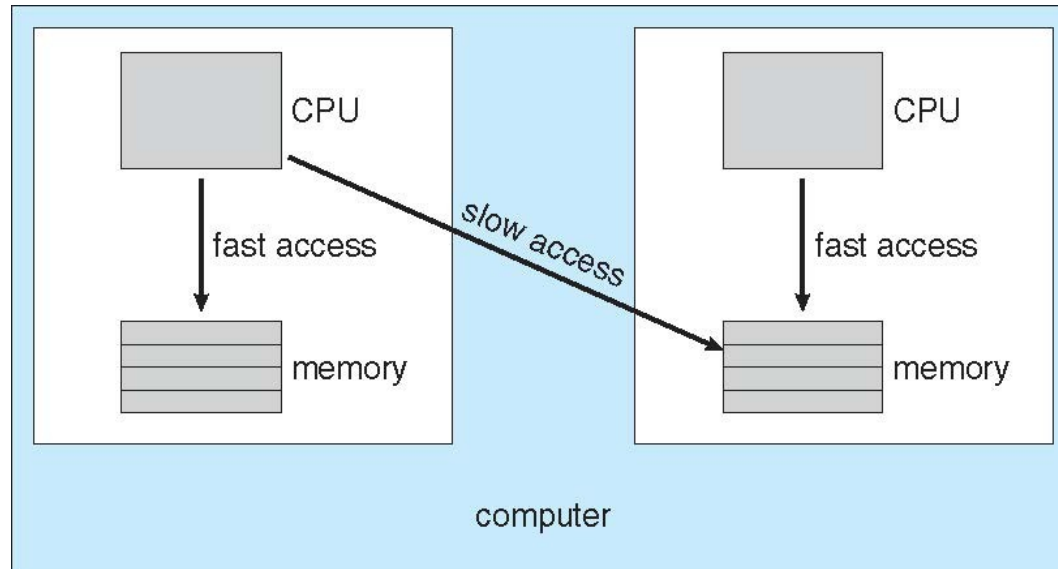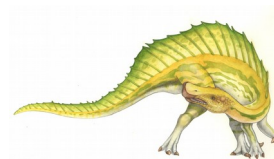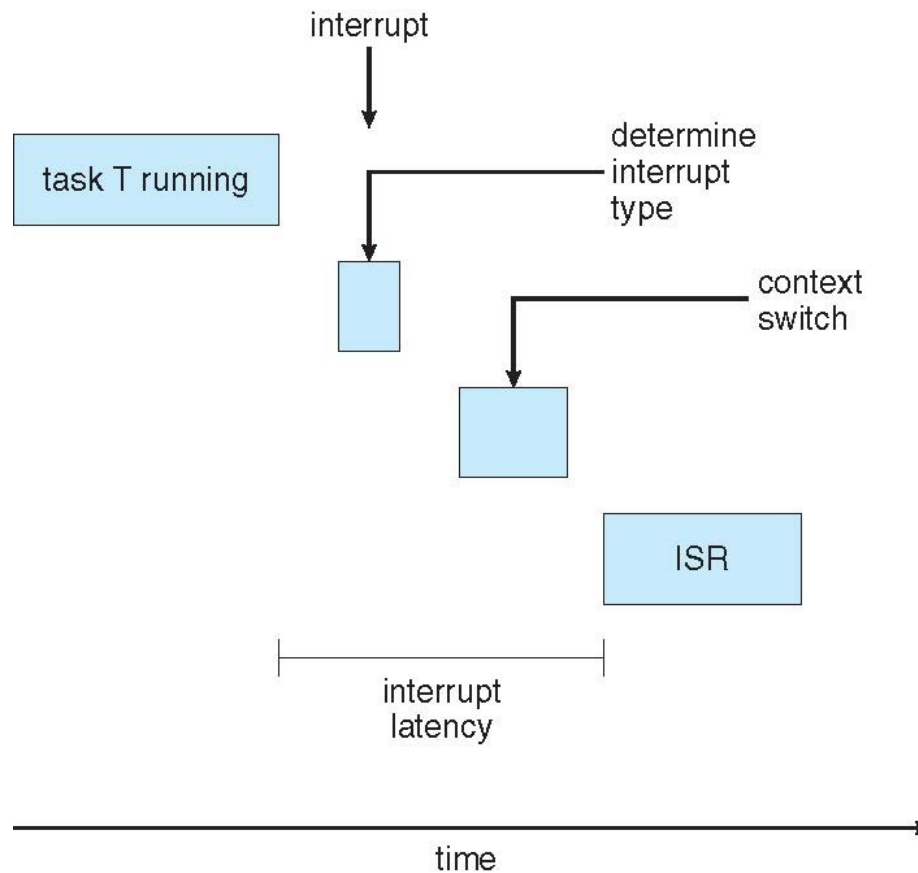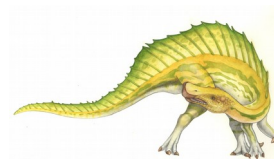
Figure 5.16
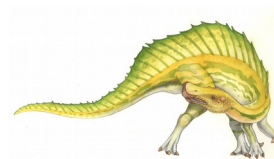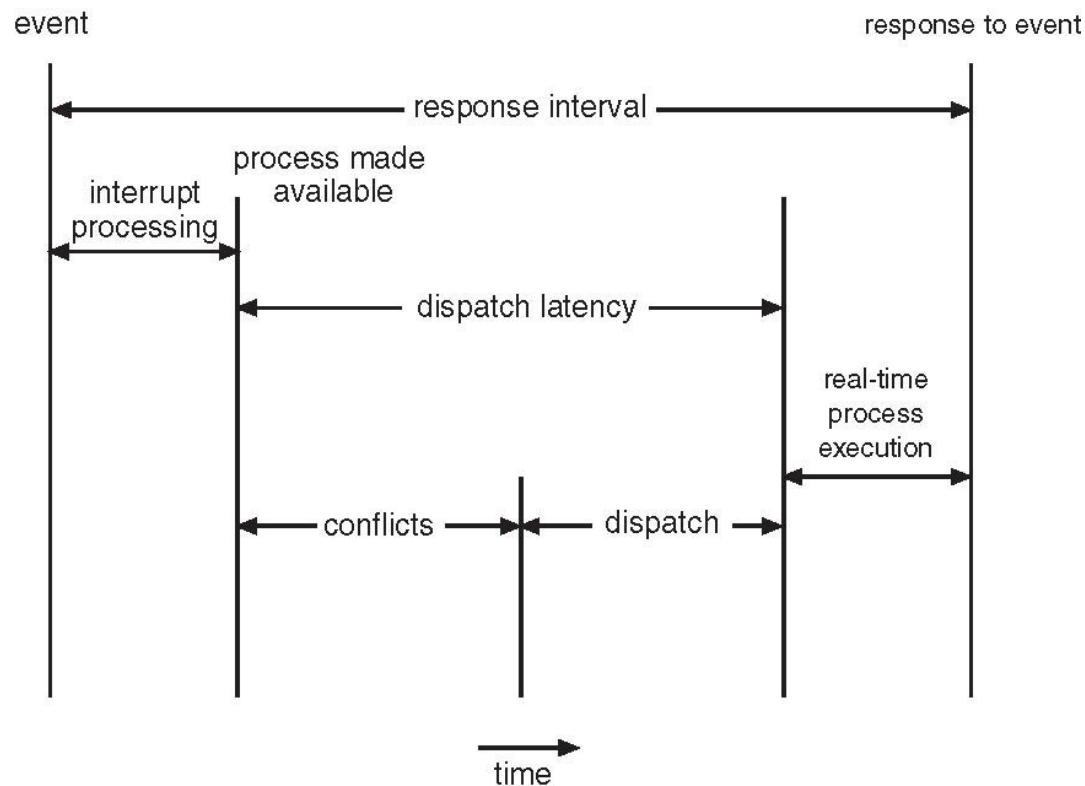
Figure 5.17

Figure 5.18

Figure 5.22

Figure 5.24

Real-Time	Normal
0 99	100 139

Higher ←——————————————————————→ Lower

Priority

Figure 5.25

Physical Processor Domain

domain$_0$

CPU$_0$

CPU$_1$

domain$_1$

CPU$_2$

CPU$_3$
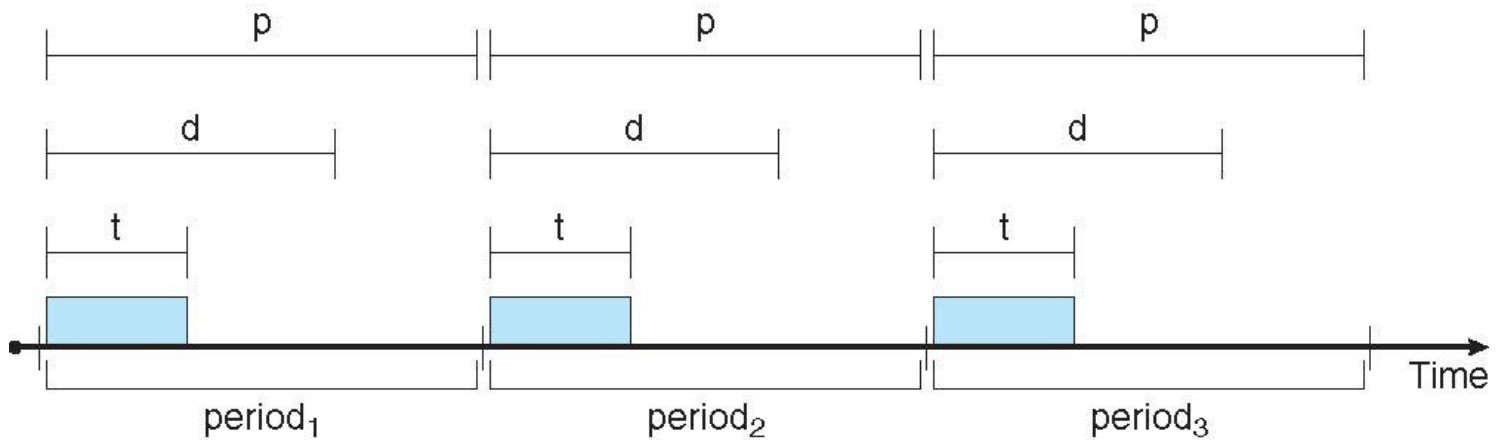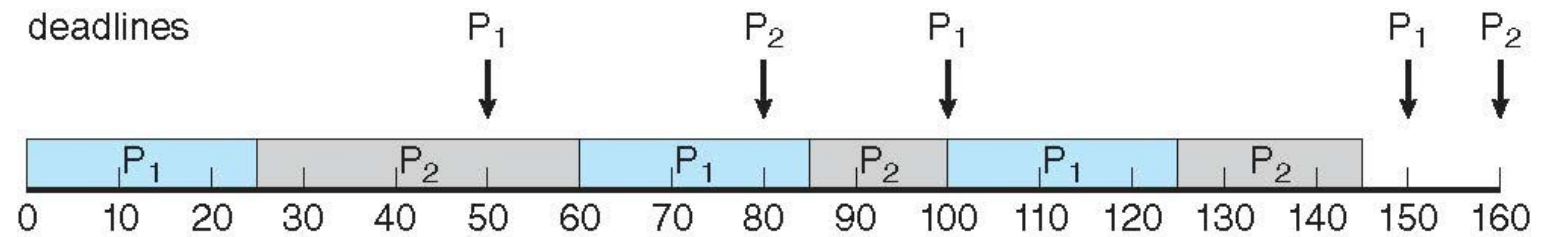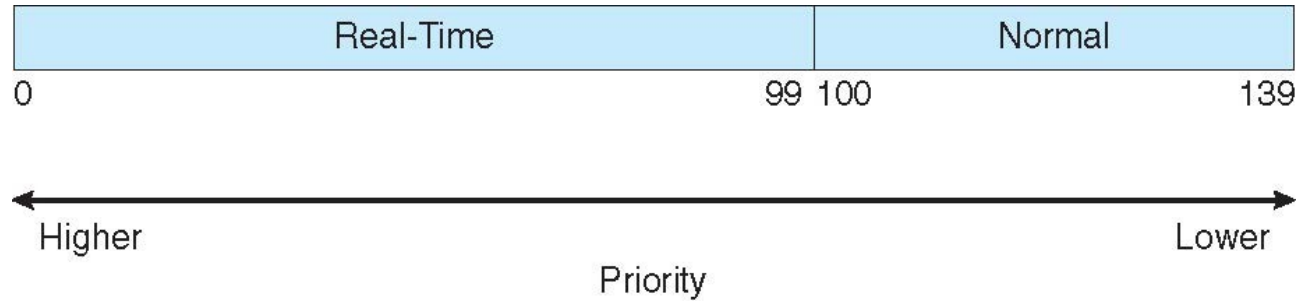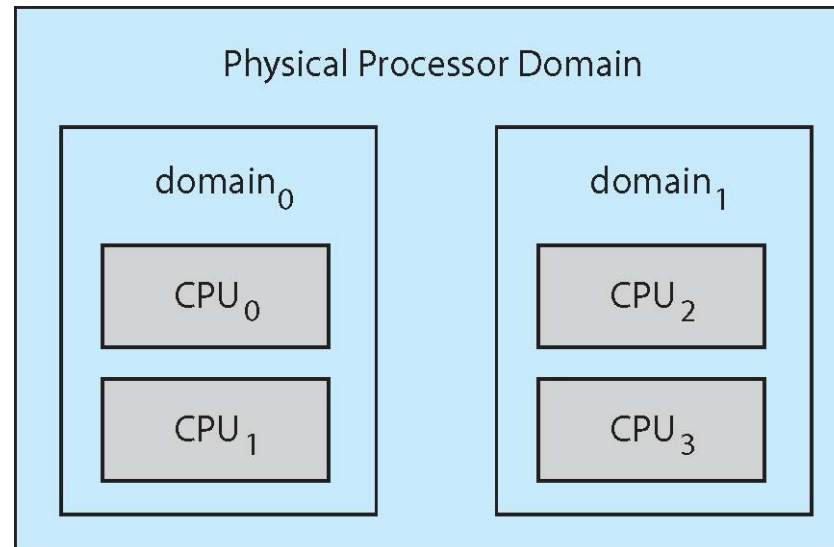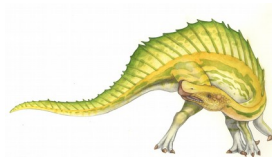
Figure 5.26

	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

Figure 5.27

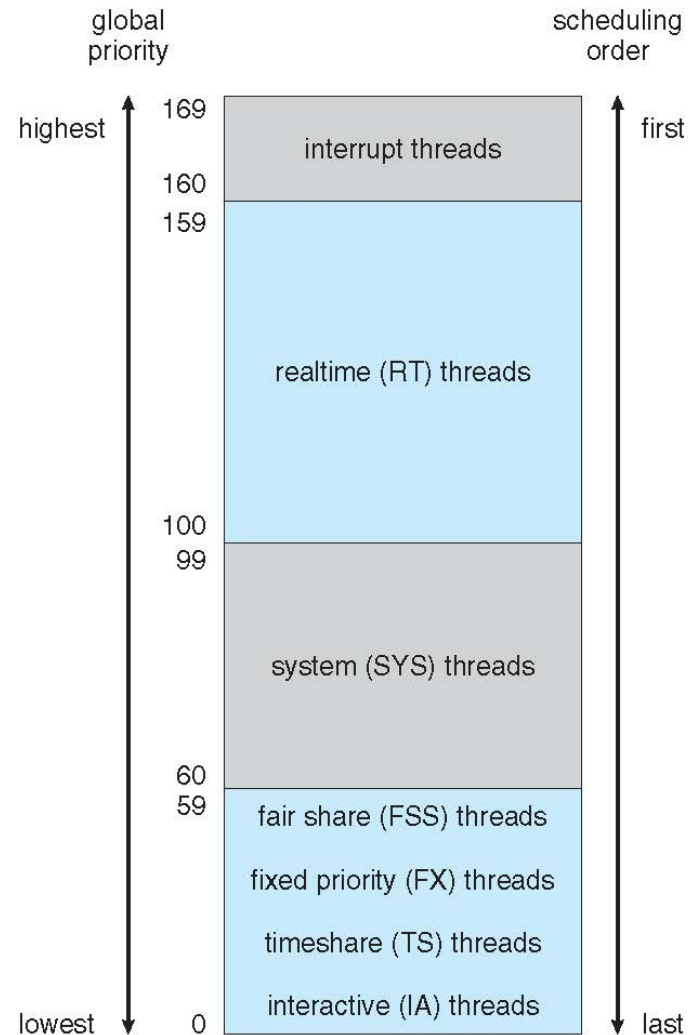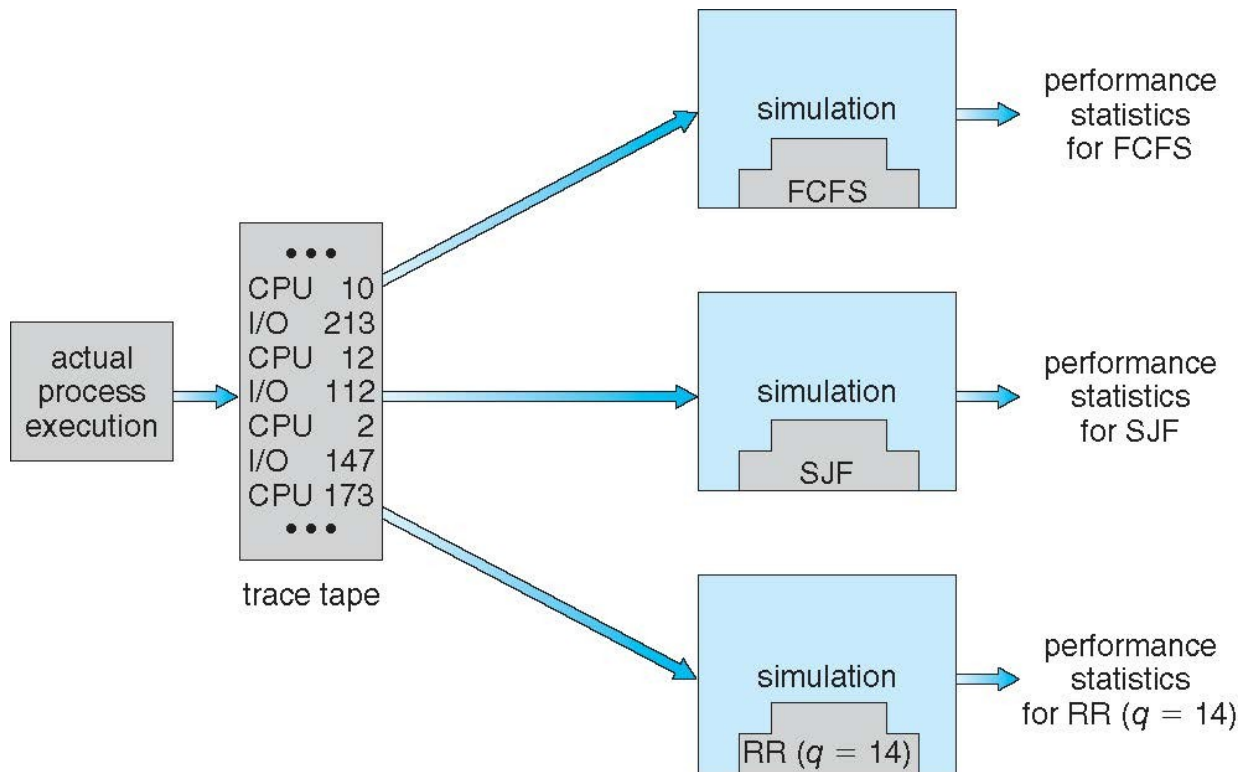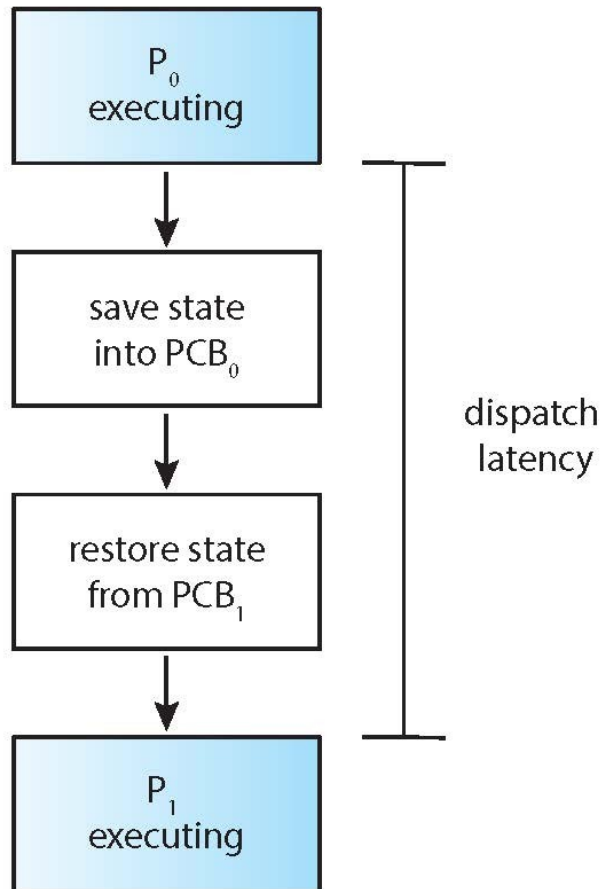| priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Figure 5.28

Figure 5.29
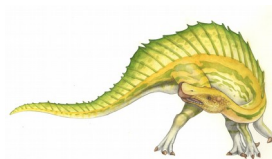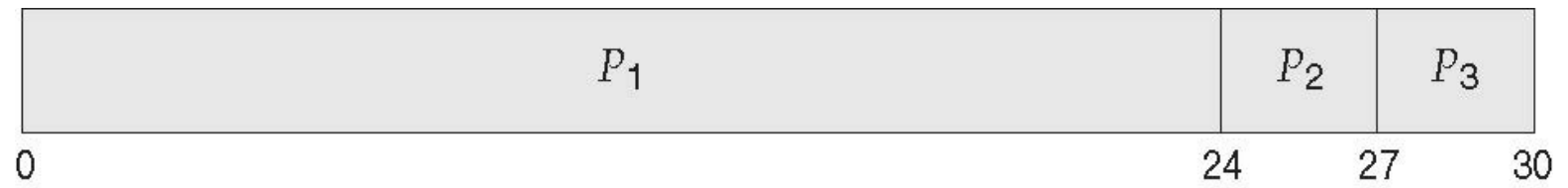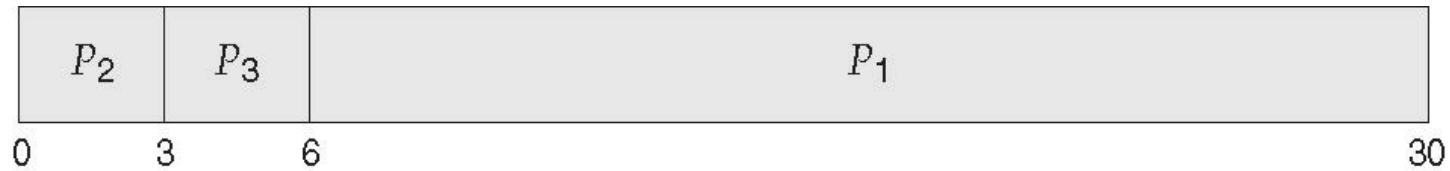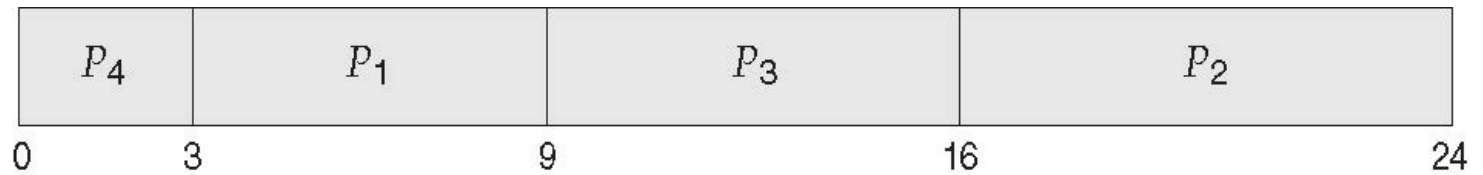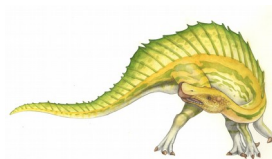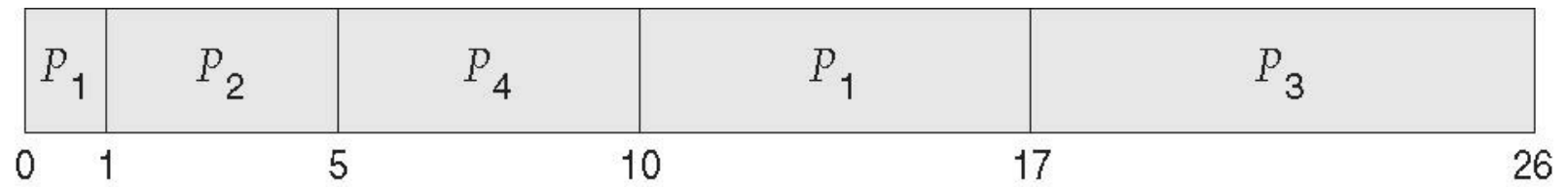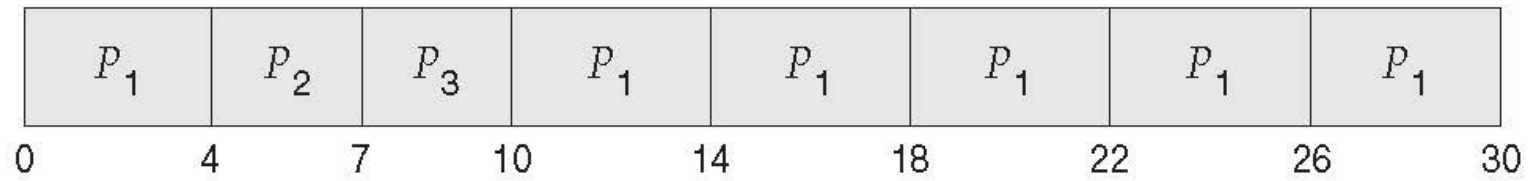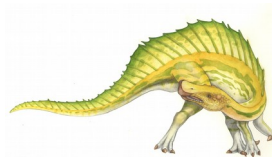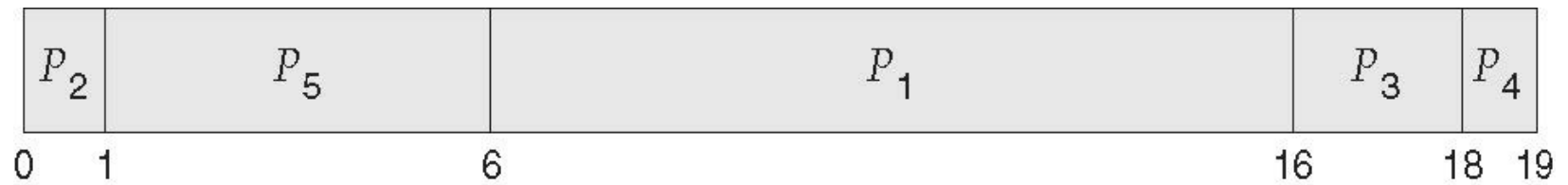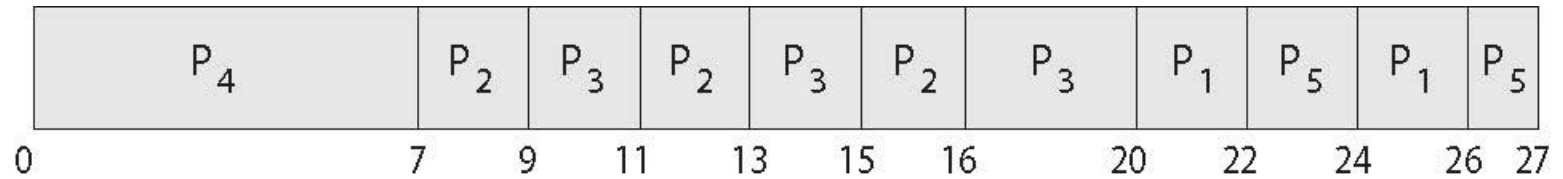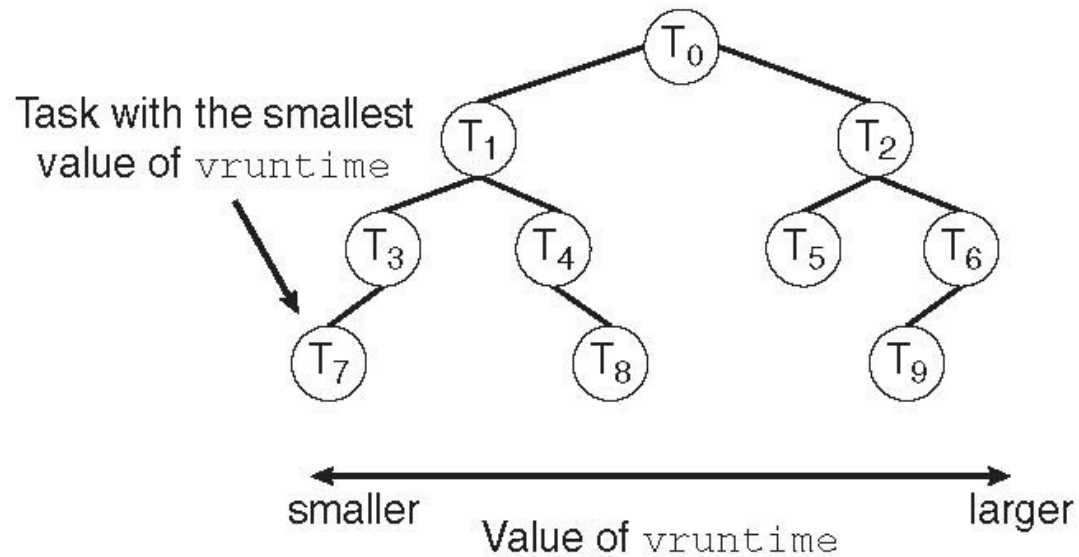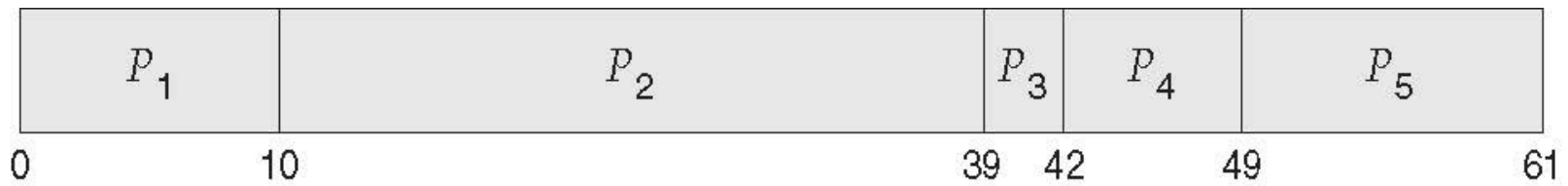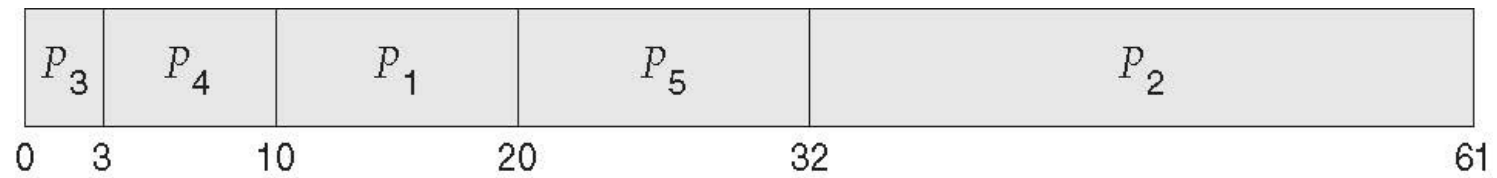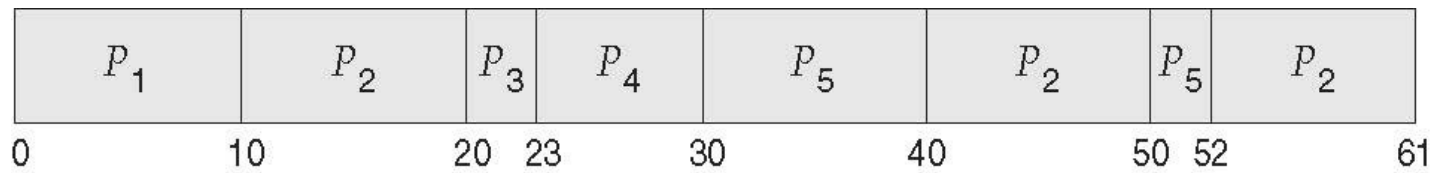
Figure Priority Queues

priority = 0 | T_0 | T_1 | T_2 | T_3 | T_4

priority = 1 | T_5 | T_6 | T_7

priority = 2 | T_8 | T_9 | T_{10} | T_{11}

priority = n | T_x | T_y | T_z
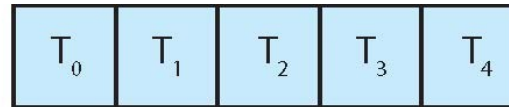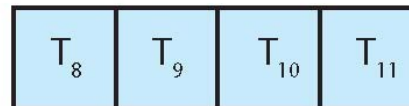
CPU Scheduler

- **CPU 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
  - 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

Preemptive and nonpreemptive Scheduling

- **Non-preemptive** – once a CPU is allocated to the process, the process keeps the CPU until it releases the CPU either when it terminates or it switches to the waiting state

- **Preemptive** -- a CPU can be taken away from a process at any time. Issues to consider in preemptive scheduling:

  - Consider access to shared data

  - Consider preemption while in kernel mode

  - Consider interrupts occurring during crucial OS activities

The role of the Dispatcher

First- Come, First-Served (FCFS) Scheduling

- Consider the following three processes and their burst time

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
- We use Gantt Chart to illustrate a particular schedule

P_1			P_2	P_3
0			24	27 30
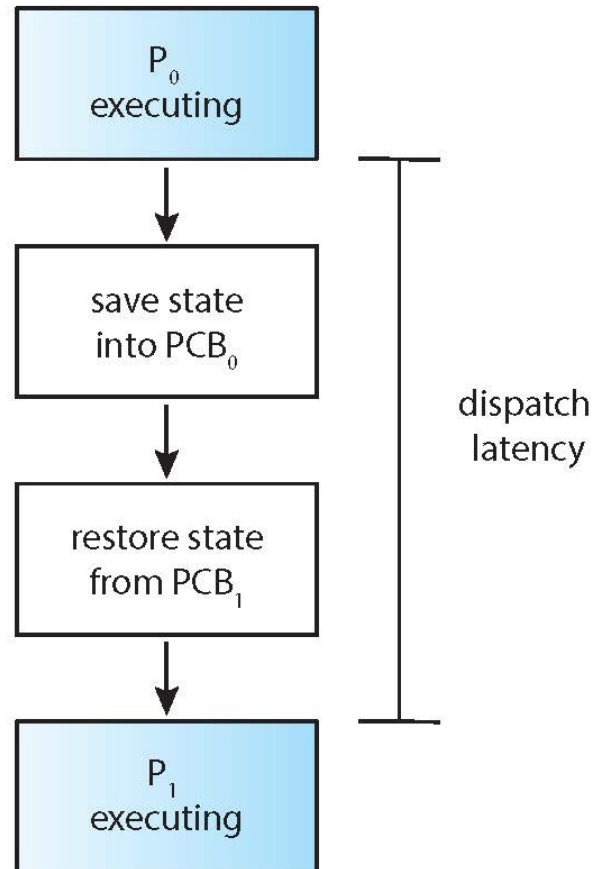
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: $(0 + 24 + 27)/3 = 17$

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
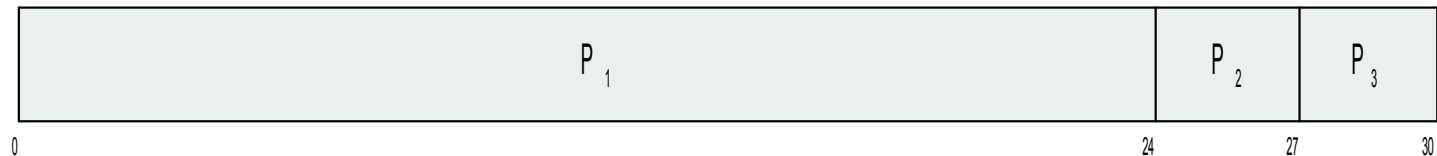
0   24   27   30

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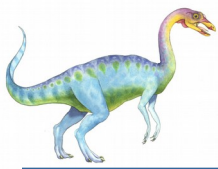
Example of Shortest-remaining-time-first

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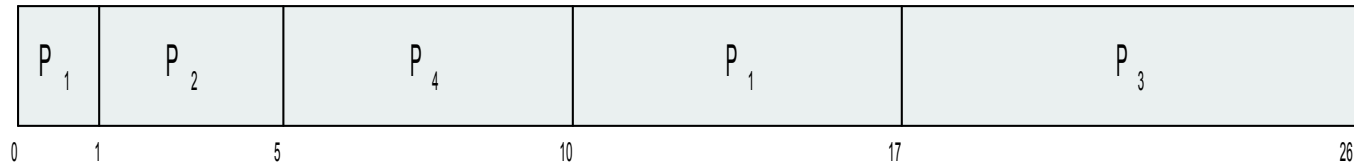
- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the CPU 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

Performance of RR Algorithm

- The performance of the RR algorithm depends on the size of the time quantum. At one extreme, if the time quantum is extremely

- large, the \RR\ policy is the same as the

- \FCFS\ policy. In contrast, if the time quantum is extremely small

- (say, 1 millisecond), the \RR\ approach can result in a large number

- of context switches. Assume, for example, that we have only one

- process of 10 time units. If the quantum is 12 time units, the process

- finishes in less than 1 time quantum, with no overhead. If the quantum

- is 6 time units, however, the process requires 2 quanta, resulting in

- a context switch. If the time quantum is 1 time unit, then nine

- context switches will occur, slowing the execution of the process

- accordingly (\Fref{fig:showing-smaller}).

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**

Memory Stall in Multicore Systems