

# Lesson 5: CPU Scheduling

# CPU Scheduling

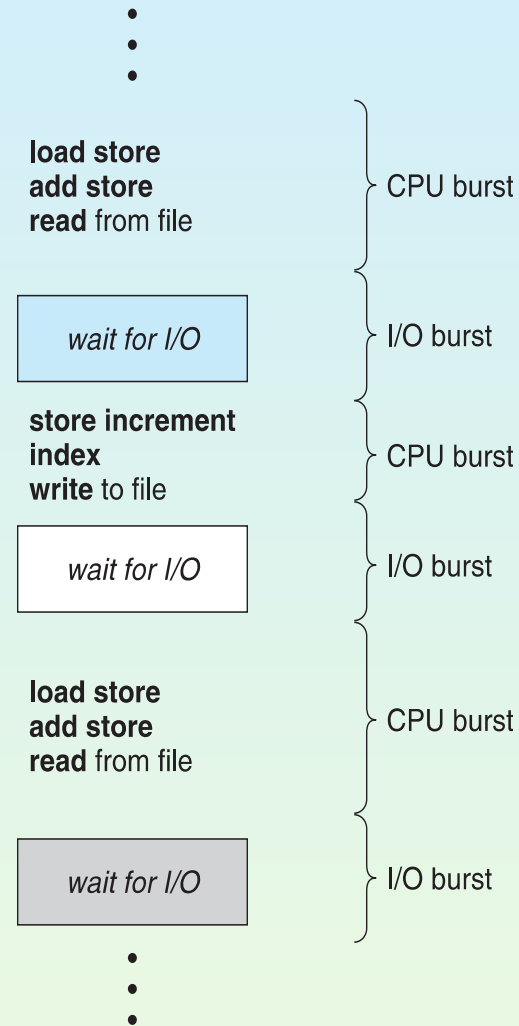
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

# Objectives

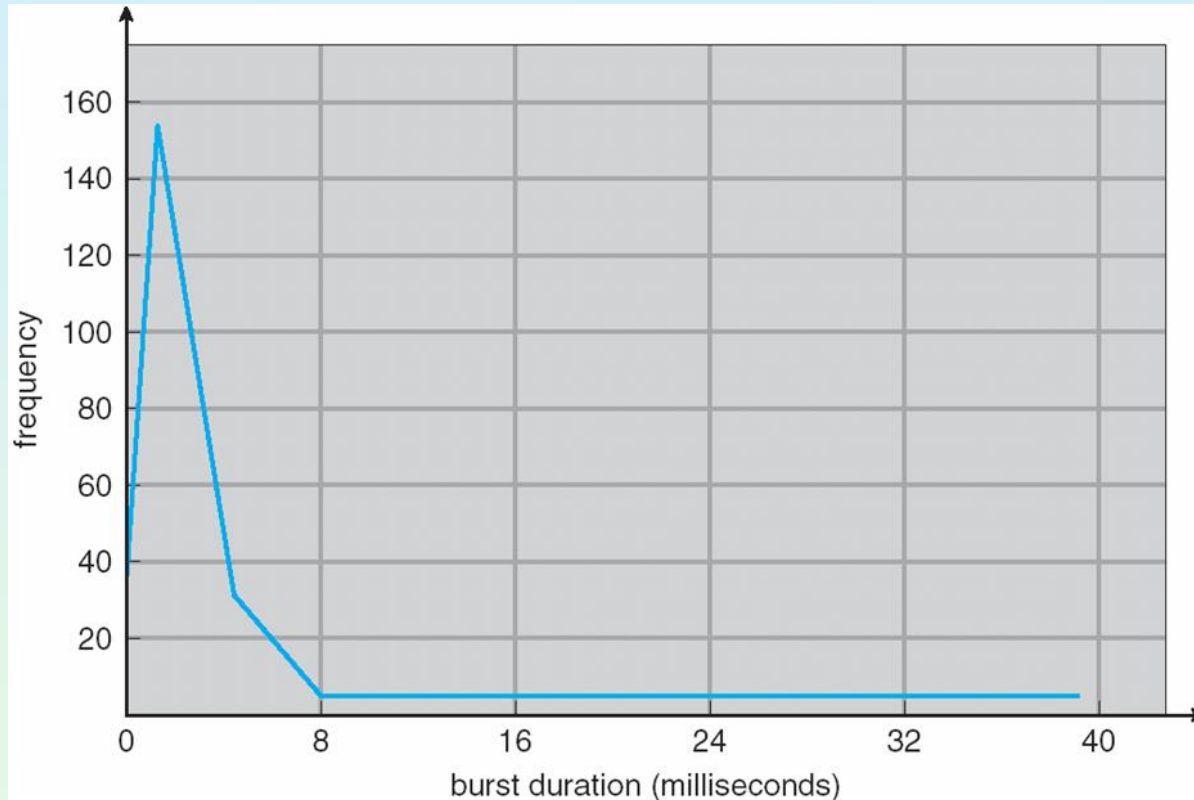
- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems

# Basic Concepts

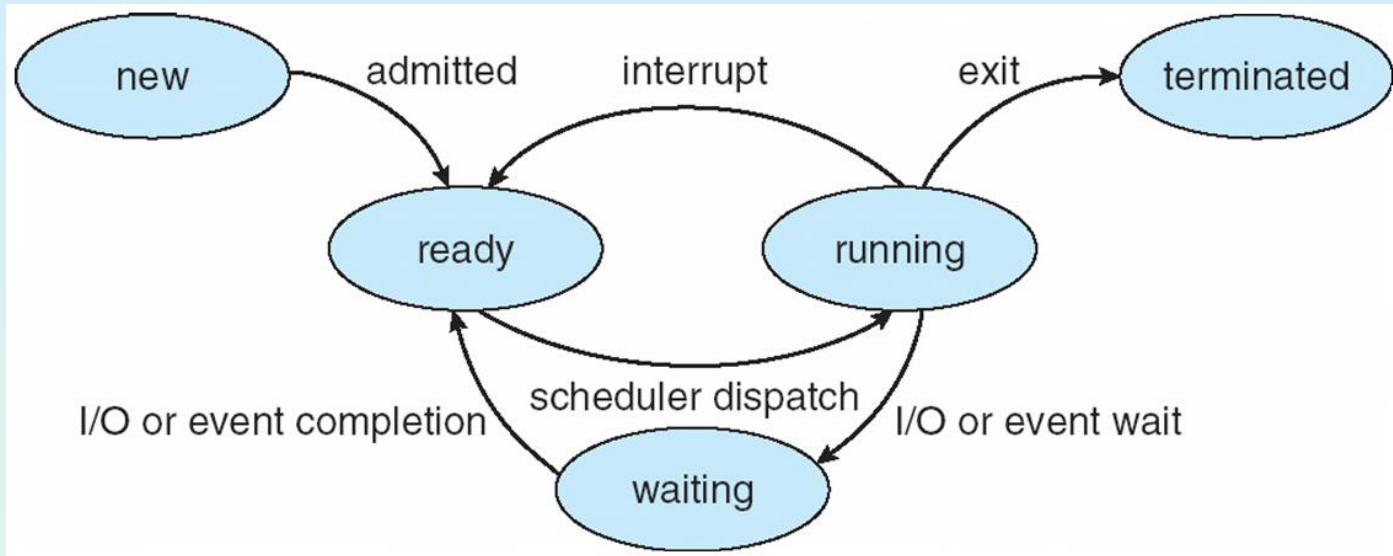
- Maximum CPU utilization obtained with multiprogramming
- CPU-I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



# Histogram of CPU-burst Times



# Recollect from Chap.3



# CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- For situations 1 and 4, **there is no choice** in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.
- 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

# Preempt

- **1** : to acquire (as land) by preemption
- **2** : to seize upon to the exclusion of others : take for oneself <the movement was then preempted by a lunatic fringe>
- **3** : to replace with something considered to be of greater value or priority : take precedence over <the program did not appear, having been preempted by a baseball game — Robert MacNeil>
- **4** : to gain a commanding or preeminent place in
- **5** : to prevent from happening or taking place : forestall, preclude
- **From Merriam-Webster Online Dictionary**



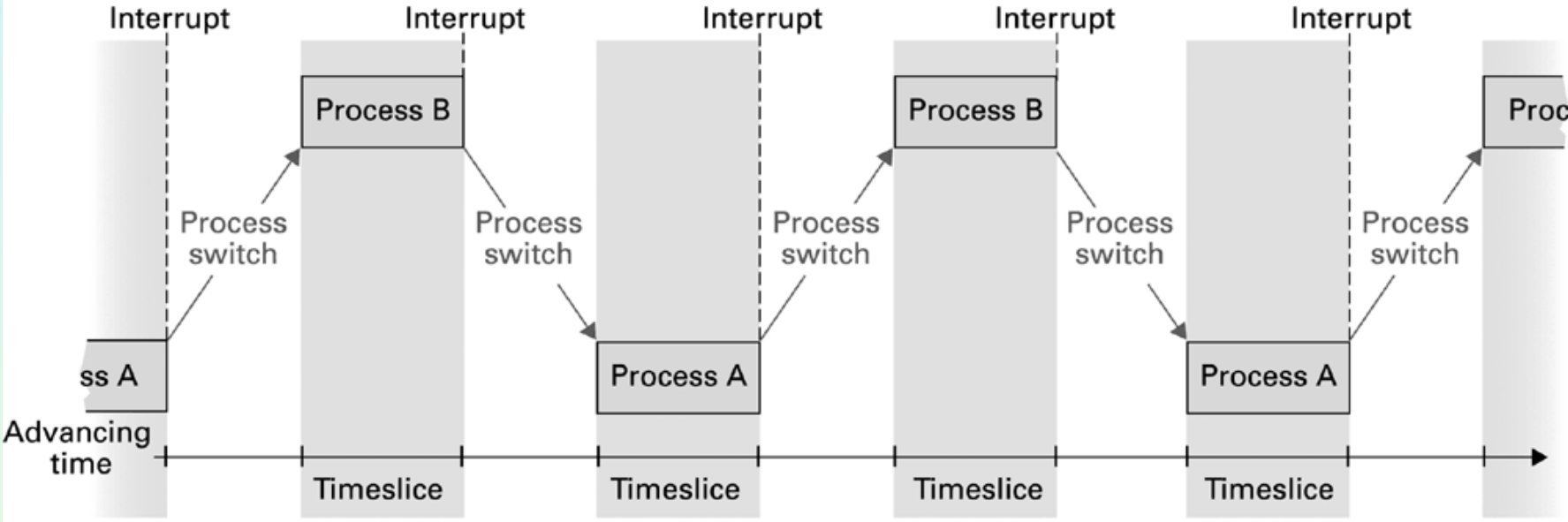
# Non-preemptive scheduling

- Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it **releases the CPU either by terminating or by switching to the waiting state.**
- This scheduling method was used by Microsoft Windows 3.x;
  - Windows 95 introduced preemptive scheduling, and all subsequent versions of Windows operating systems have used preemptive scheduling.
  - The Mac OS X operating system for the Macintosh uses preemptive scheduling; previous versions of the Macintosh operating system relied on cooperative scheduling.

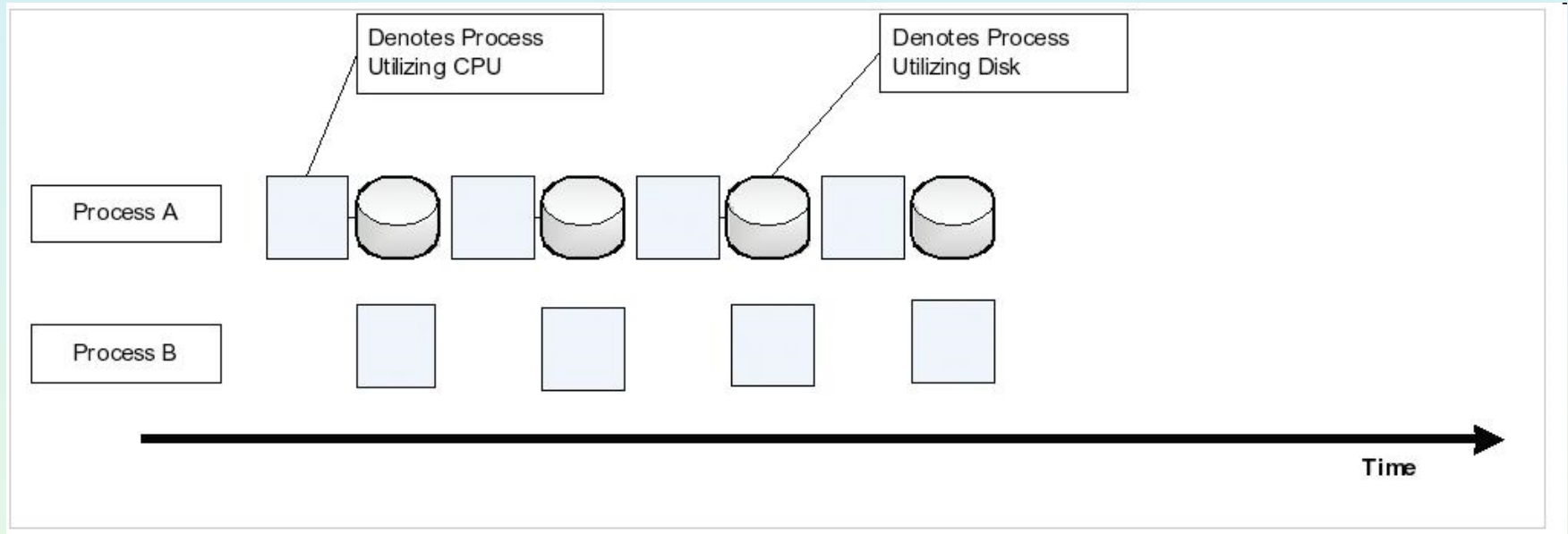
# Dispatcher

- 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

# Dispatcher and scheduler in action



# Dispatcher and scheduler in action: Two processes sharing the CPU



# Scheduling Criteria

- Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.
- In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.
  - **(Analysis for your project)**
- Many criteria have been suggested for comparing CPU scheduling algorithms.
- Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best.

# 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

# Scheduling Algorithm Goals

**CPU scheduling** deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU scheduling algorithms.

## **All systems**

Fairness - giving each process a fair share of the CPU

Policy enforcement - seeing that stated policy is carried out

Balance - keeping all parts of the system busy

## **Batch systems**

Throughput - maximize jobs per hour

Turnaround time - minimize time between submission and termination

CPU utilization - keep the CPU busy all the time

## **Interactive systems**

Response time - respond to requests quickly

Proportionality - meet users' expectations

## **Real-time systems**

Meeting deadlines - avoid losing data

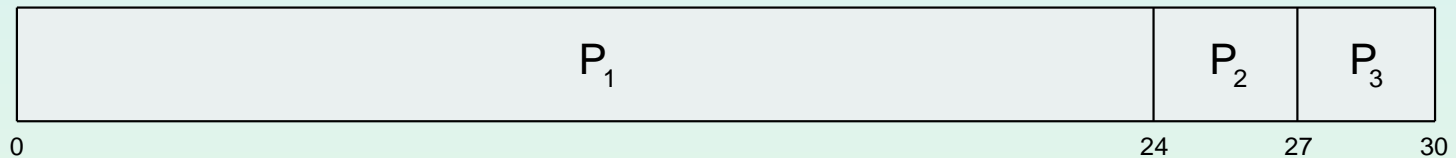
Predictability - avoid quality degradation in multimedia systems



# 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:



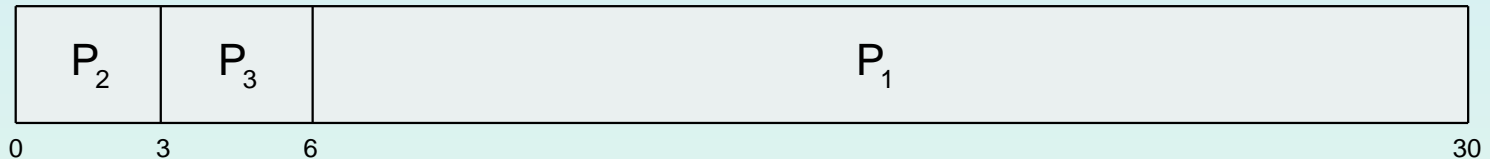
- Waiting time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- Average waiting time:  $(0 + 24 + 27)/3 = 17$

# 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
  - Consider one CPU-bound and many I/O-bound processes

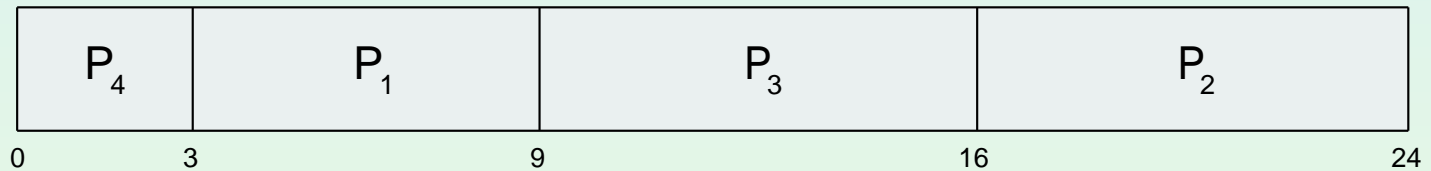
# Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its **next CPU burst**
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives **minimum average waiting time** for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

# Example of SJF

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0.0	6
$P_2$	2.0	8
$P_3$	4.0	7
$P_4$	5.0	3

- SJF scheduling chart

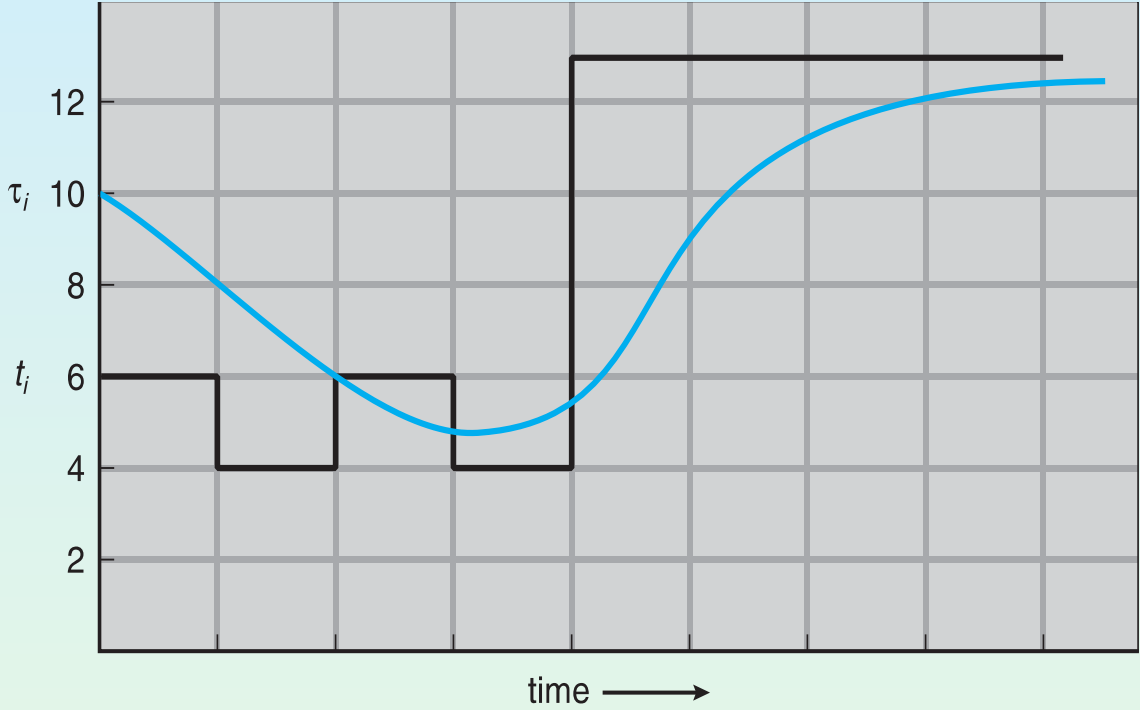


- Average waiting time =  $(3 + 16 + 9 + 0) / 4 = 7$

# Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
  
- Can be done by using the length of previous CPU bursts, using exponential averaging
  1.  $t_n$  = actual length of  $n^{\text{th}}$  CPU burst
  2.  $\tau_{n+1}$  = predicted value for the next CPU burst
  3.  $\alpha, 0 \leq \alpha \leq 1$
  4. Define:  $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$ .
  
- Commonly,  $\alpha$  set to  $\frac{1}{2}$
- Preemptive version called **shortest-remaining-time-first**

# 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:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^{n+1} \tau_0.$$

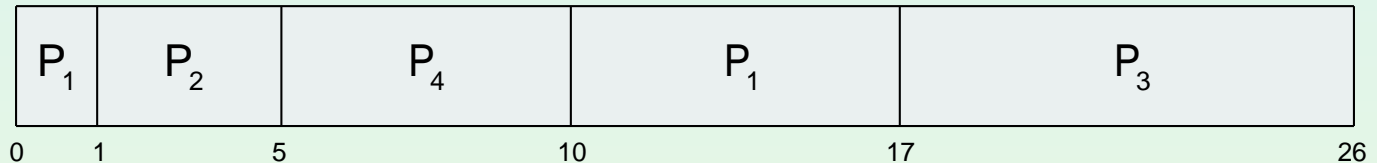
- Since both  $\alpha$  and  $(1 - \alpha)$  are less than or equal to 1, each successive term has less weight than its predecessor

# Example of Shortest-remaining-time-first

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

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0	8
$P_2$	1	4
$P_3$	2	9
$P_4$	3	5

- Preemptive* SJF Gantt Chart



- Average waiting time =  $(P_1 + P_2 + P_3 + P_4) / 4$
- Average waiting time =  $[(10-1)+(1-1)+(17-2)+5-3]/4 = 26/4 = 6.5$  msec



# Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer  $\equiv$  highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem  $\equiv$  **Starvation** – low priority processes may never execute
  - (it is told 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  $\equiv$  **Aging** – as time progresses increase the priority of the process

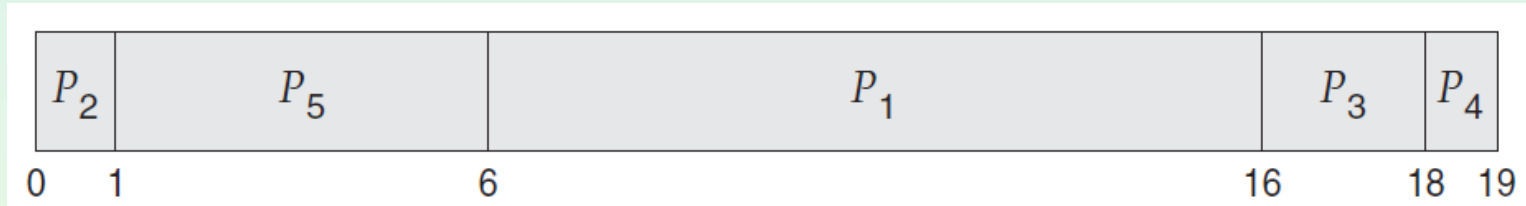
# Aging

- Aging is a technique of **gradually increasing** the priority of processes that wait in the system for a long time.
  - For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.
  - Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed.

# Example of Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	10	3
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

## ■ Priority scheduling Gantt Chart



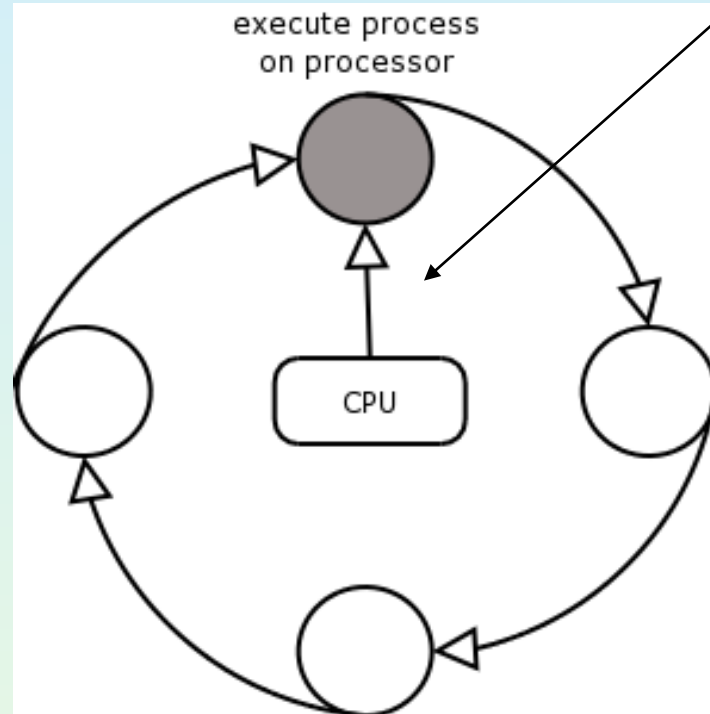
## ■ Average waiting time = 8.2 msec

# Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum**  $q$ ), usually 10-100 milliseconds.
- After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are  $n$  processes in the ready queue and the time quantum is  $q$ , then each process gets  $1/n$  of the CPU time in chunks of at most  $q$  time units at once. No process waits more than  $(n-1)q$  time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - $q$  large  $\Rightarrow$  FIFO
  - $q$  small  $\Rightarrow q$  must be large with respect to context switch, otherwise overhead is too high

# Round-Robin

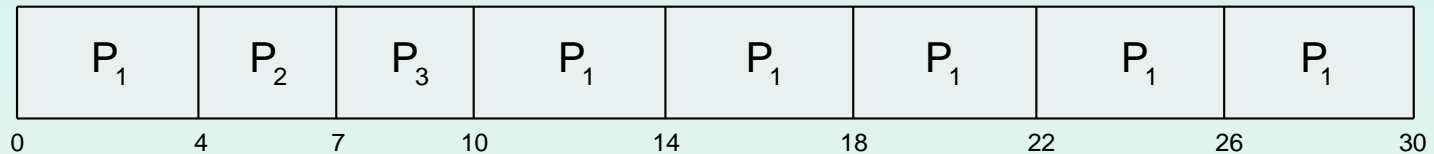
A process keeps the CPU for a quantum of time  $q$



# Example of RR with Time Quantum = 4

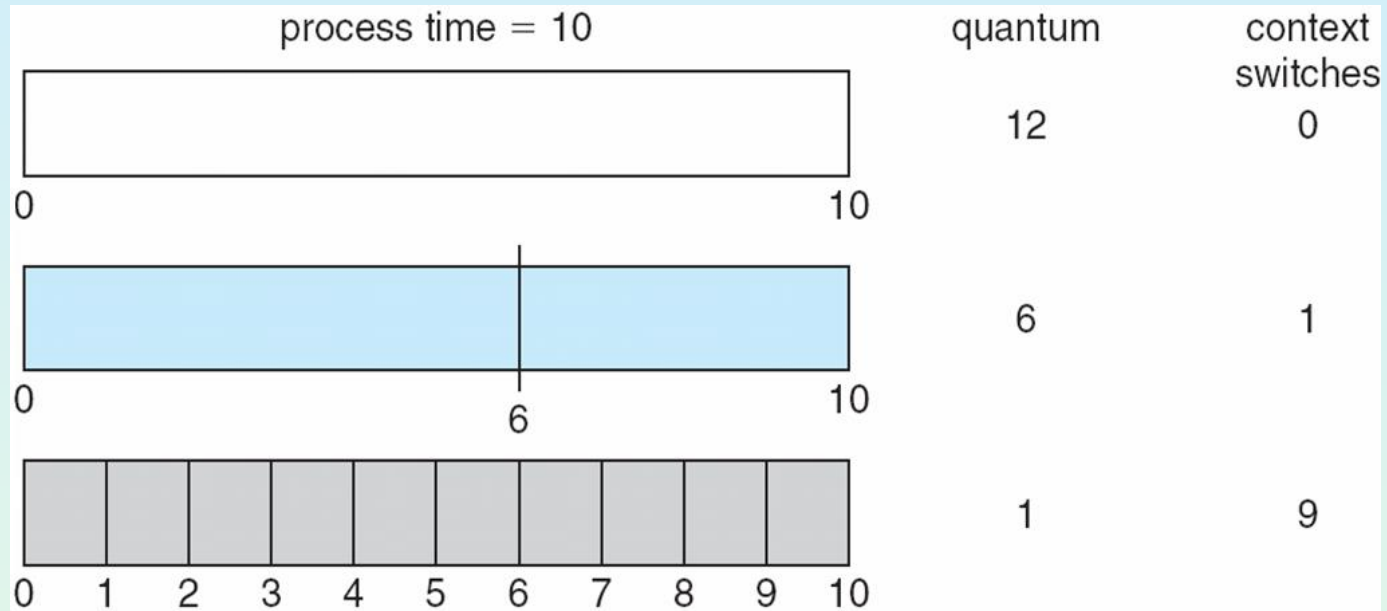
<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

- The Gantt chart is:

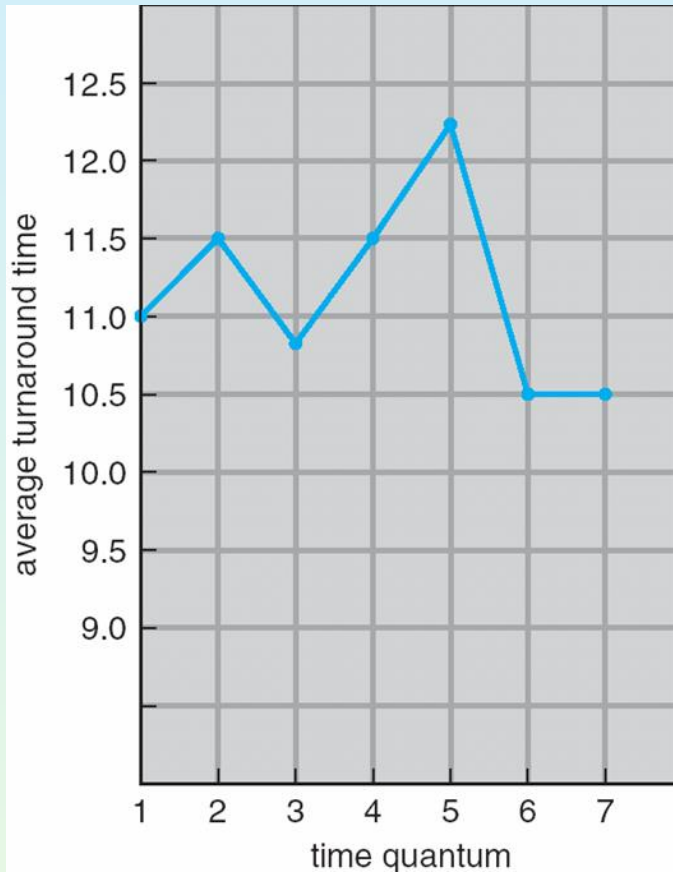


- Typically, higher average turnaround than SJF, but better **response**
- $q$  should be large compared to context switch time
- $q$  usually 10ms to 100ms, context switch < 10 usec

# 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

80% of CPU bursts  
should be shorter than  $q$

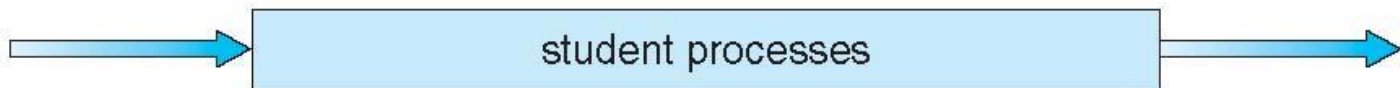
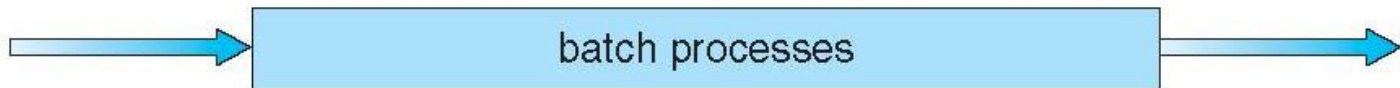
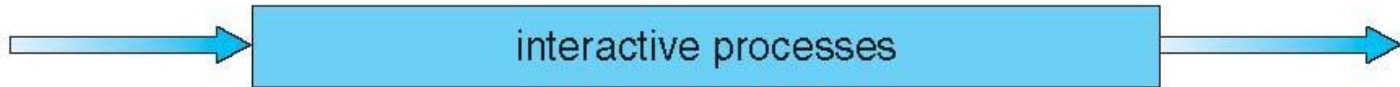
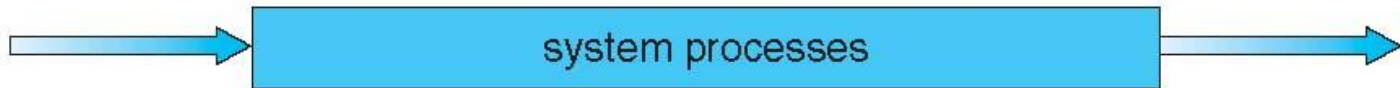


# Multilevel Queue

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

# Multilevel Queue Scheduling

highest priority



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

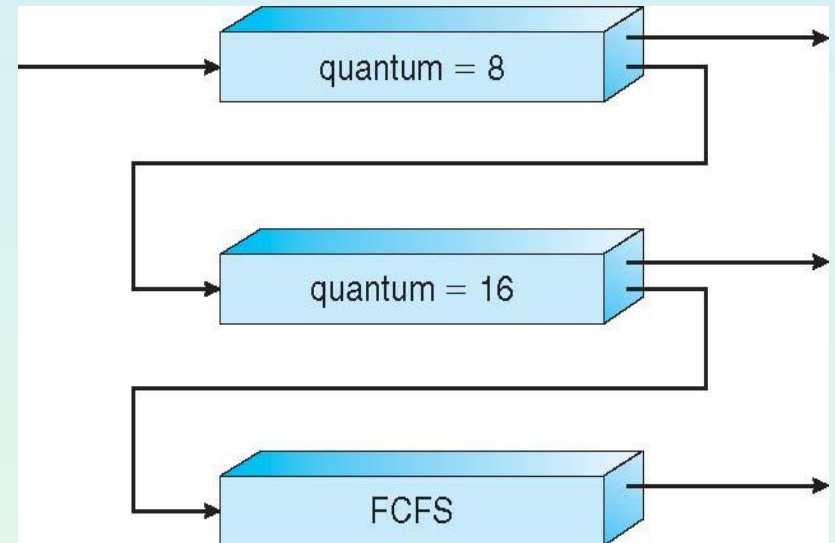
# Example of Multilevel Feedback Queue

## ■ 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  $Q_0$ 
  - ▶ 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  $Q_2$



# Thread Scheduling

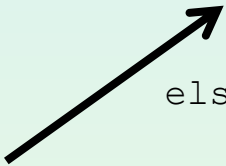
- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope (PCS)** since scheduling competition is within 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

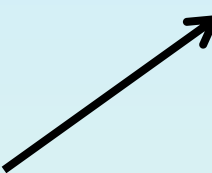
# Pthread Scheduling API

```
#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);
}
```





# All schedulers comparison

Scheduling algorithm	CPU Utilization	Throughput	Turnaround time	Response time	Deadline handling	Starvation free
First In First Out	Low	Low	High	High	No	Yes
Shortest remaining time	Medium	High	Medium	Medium	No	No
Fixed priority pre-emptive scheduling	Medium	Low	High	High	Yes	No
Round-robin scheduling	High	Medium	Medium	Low	No	Yes

There is no universal “best” scheduling algorithm, and many operating systems use extended or combinations of the scheduling algorithms above.

# 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
  - Variations including **processor sets**

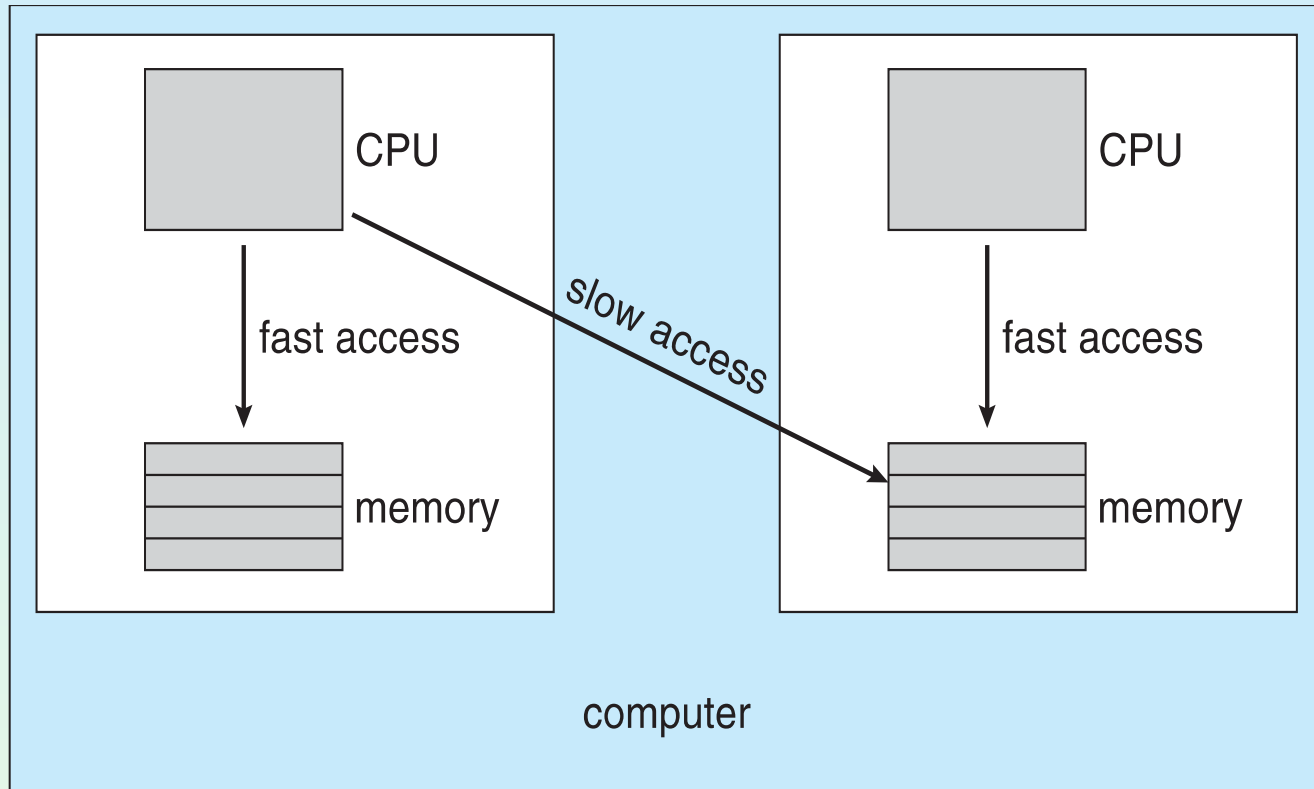
# Processor affinity

- **Processor affinity** – process has affinity for processor on which it is currently running
- **Soft affinity** - The operating system has a policy of attempting to keep a process running on the same processor — but not guaranteeing that it will do so
- **Hard affinity** – Some systems provide system calls that allow a process to specify a subset of processors on which it may run.
- Many systems provide both soft and hard affinity.
- For example, Linux implements soft affinity, but it also provides the **`sched_setaffinity()` system call**, which supports hard affinity.

# NUMA

- The **main-memory architecture** of a system can affect processor affinity.
- Non-uniform memory access (NUMA), in which a CPU has faster access to some parts of main memory than to other parts.
- Typically, this occurs in systems containing **combined CPU and memory boards**.
- CPUs on a board can access the memory on that board **faster** than they can access memory on other boards in the system.
- If the operating system's CPU scheduler and memory-placement algorithms **work together**, then a process that is assigned affinity to a particular CPU can be allocated memory on the board where that CPU resides.

# NUMA and CPU Scheduling



Note that memory-placement algorithms can also consider affinity

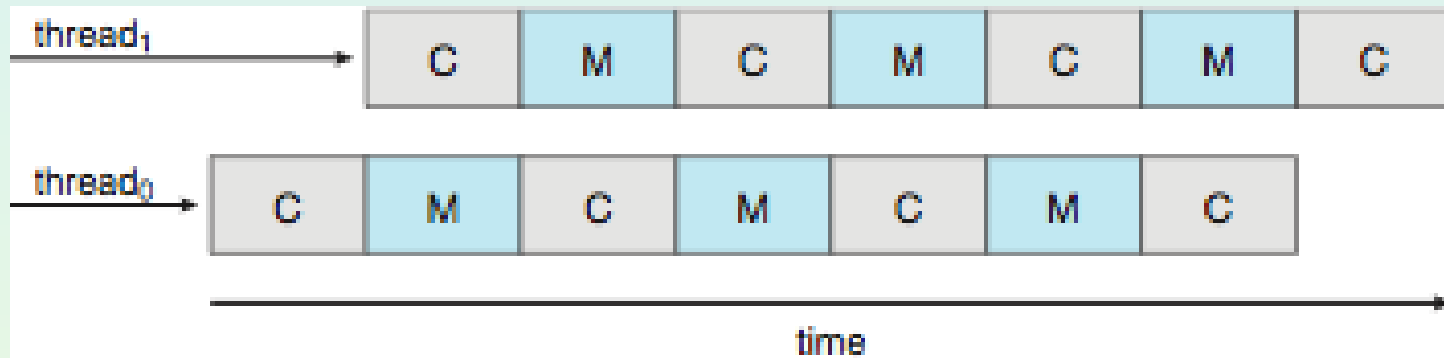
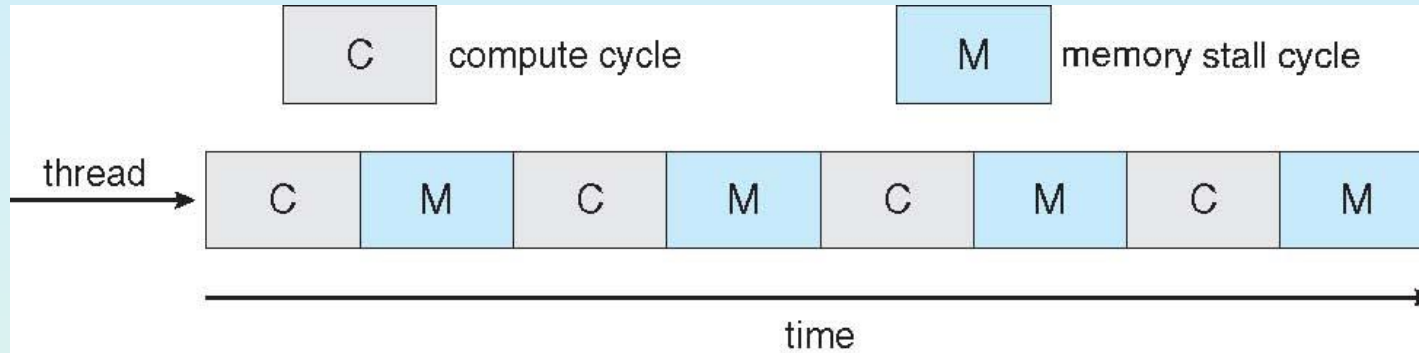
# Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor

# Multicore Processors

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

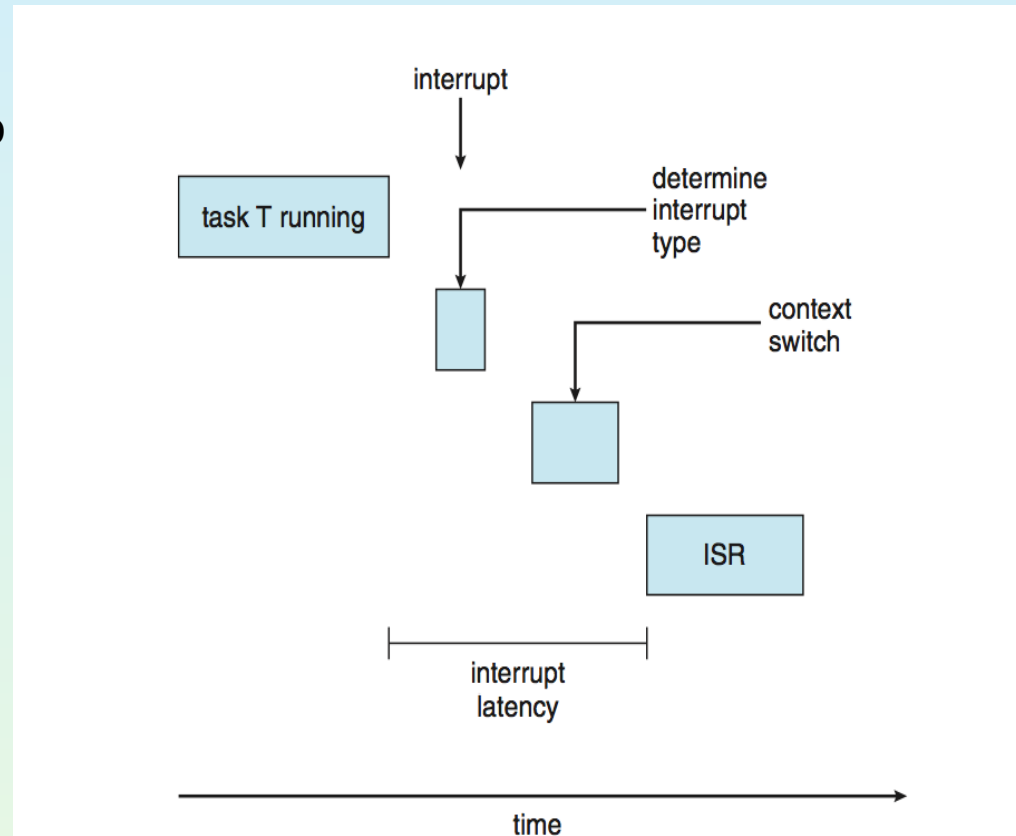
# Multithreaded Multicore System





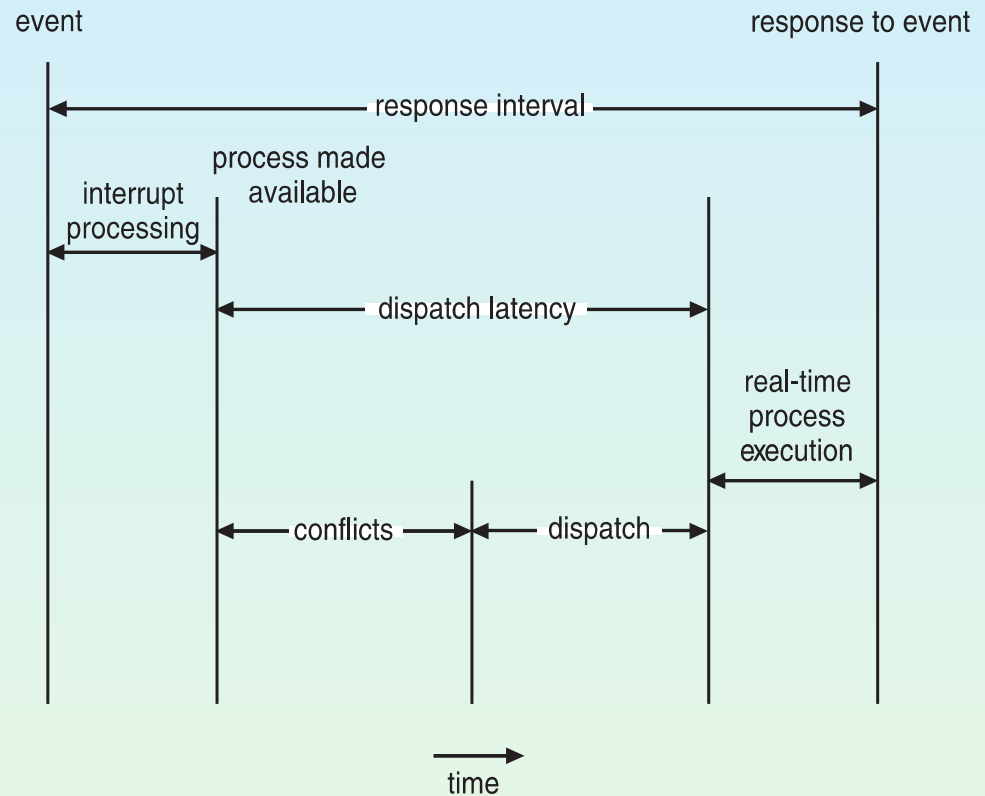
# Real-Time CPU Scheduling

- 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



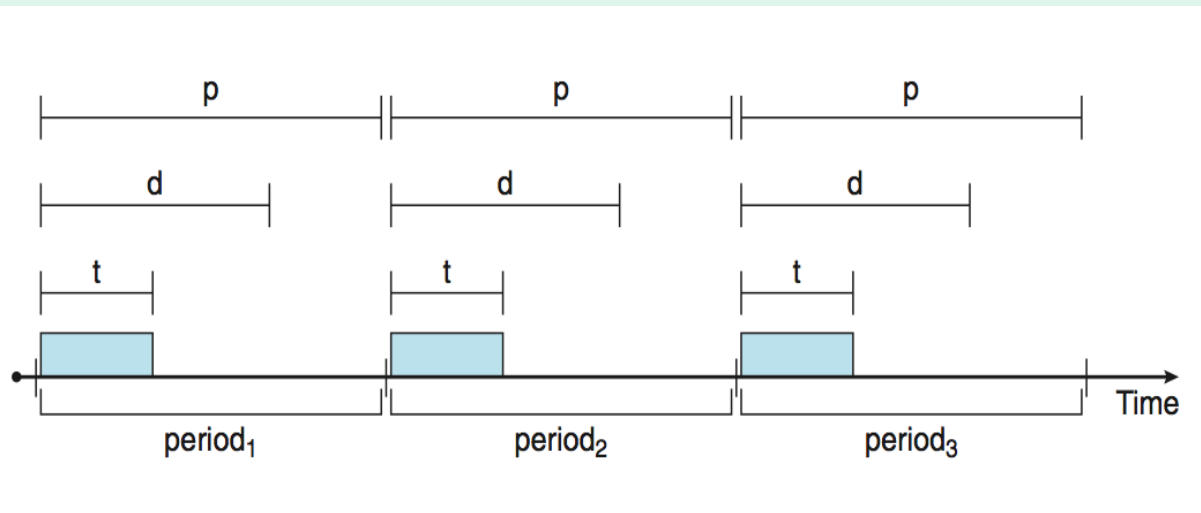
# Real-Time CPU Scheduling (Cont.)

- 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 or **periods**
  - Has processing time  $t$ , deadline  $d$ , period  $p$
  - $0 \leq t \leq d \leq p$
  - **Rate** of periodic task is  $1/p$

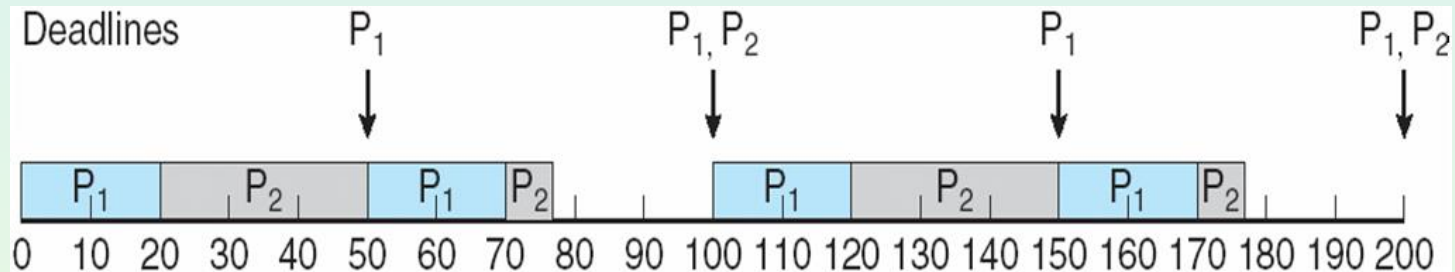


# Admission-control algorithm

- What is unusual about this form of scheduling is that a process may have to **announce its deadline requirements** to the scheduler.
- Then, using a technique known as an **admission-control** algorithm, the scheduler does one of two things.
- It either **admits** the process, guaranteeing that the process will complete on time, or **rejects** the request as impossible if it cannot guarantee that the task will be serviced by its deadline.

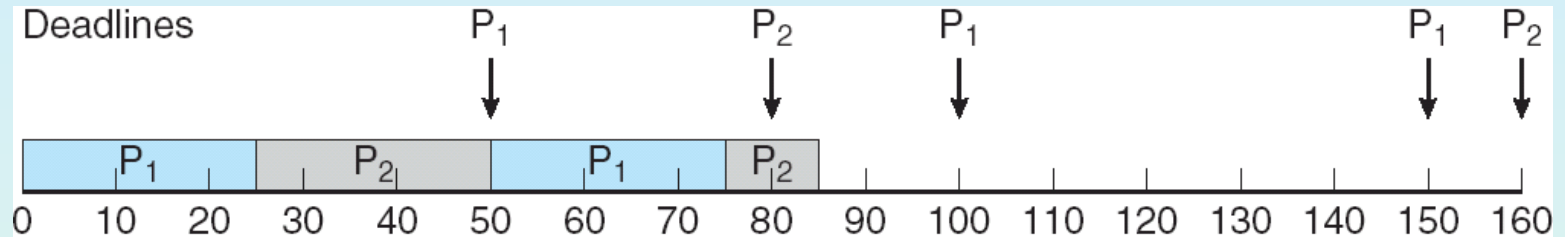
# Rate Monotonic Scheduling

- 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  $P_2$ :
  - The periods for  $P_1$  and  $P_2$  are 50 and 100



The deadline for each process requires that it complete its CPU burst by the start of its next period ( $t_1 = 20$  for  $P_1$  and  $t_2 = 35$  for  $P_2$ )

# Missed Deadlines with Rate Monotonic Scheduling



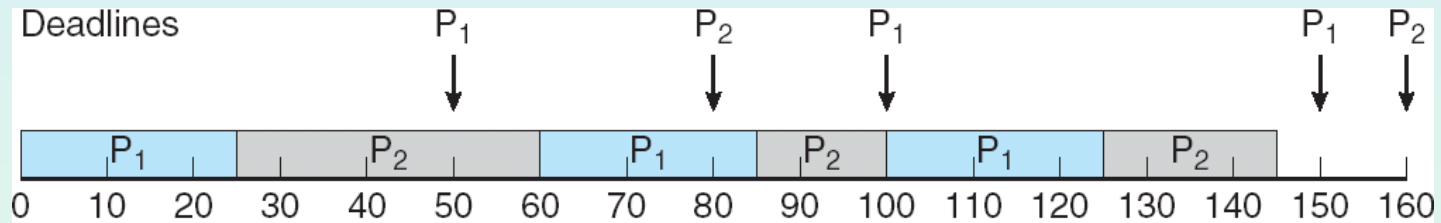
$P_1$  has values of  $p_1 = 50$  and  $t_1 = 25$  and that  $P_2$  has values of  $p_2 = 80$  and  $t_2 = 35$ .

# 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
- Assume that a total of  $T = 100$  shares is to be divided among three processes,  $A$ ,  $B$ , and  $C$ .
- $A$  is assigned 50 shares,  $B$  is assigned 15 shares, and  $C$  is assigned 20 shares.
- This scheme ensures that  $A$  will have **50 percent** of total processor time,  $B$  will have 15 percent, and  $C$  will have 20 percent.



# POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads:
  1. SCHED\_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
  2. SCHED\_RR - similar to SCHED\_FIFO except time-slicing occurs for threads of equal priority
- Defines two functions for getting and setting scheduling policy:
  1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
  2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

# POSIX Real-Time Scheduling API

```
#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");
    }
}
```

# POSIX Real-Time Scheduling API (Cont.)

```
/* 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

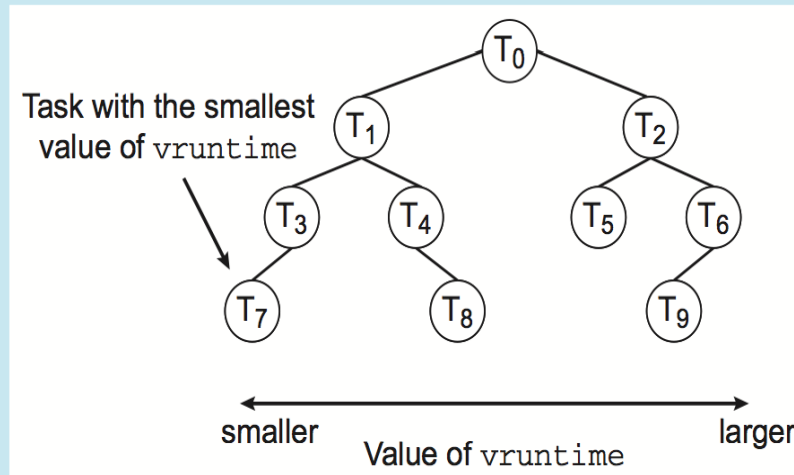
- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order  $O(1)$  scheduling time
  - **Preemptive, priority based**
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99 and **nice** value from 100 to 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: balanced tree

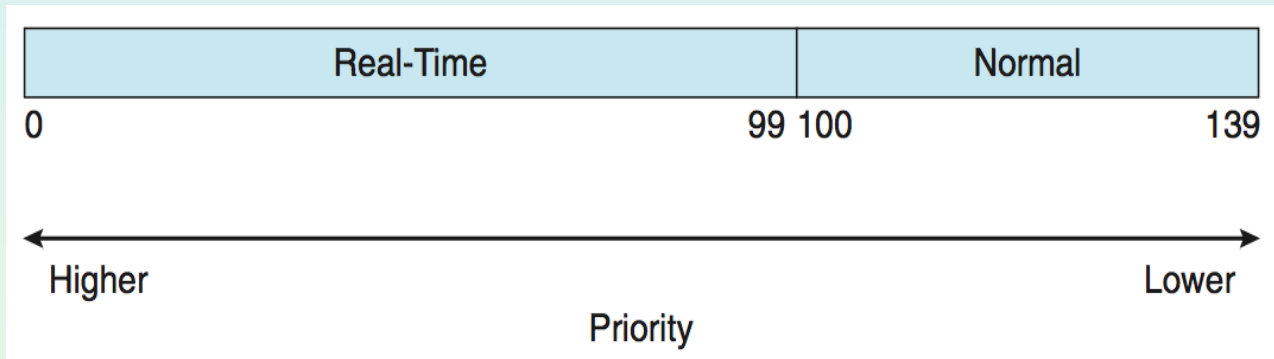
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



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





# 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

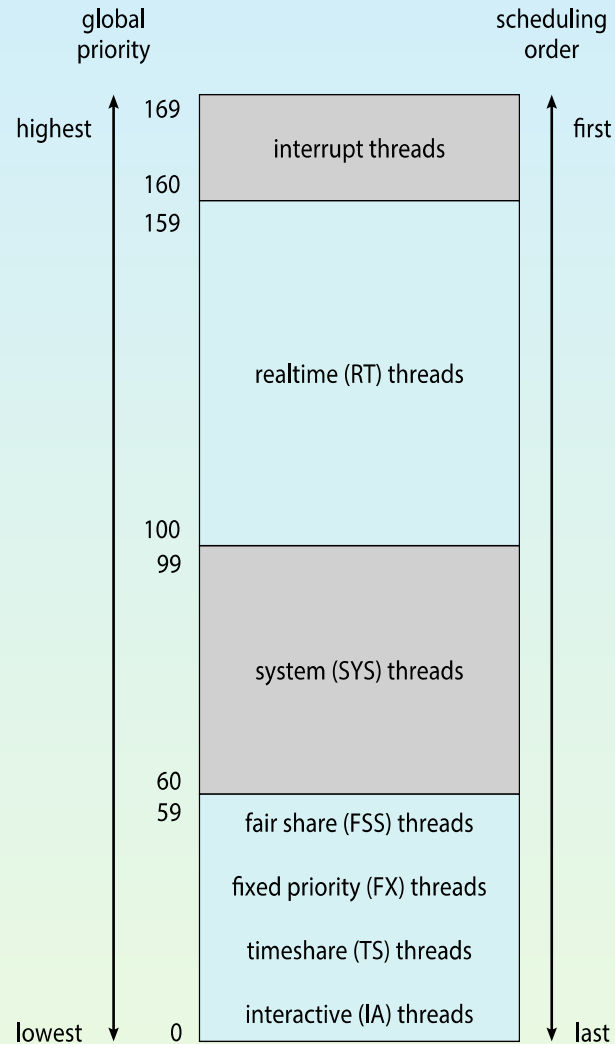
# Solaris

- 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



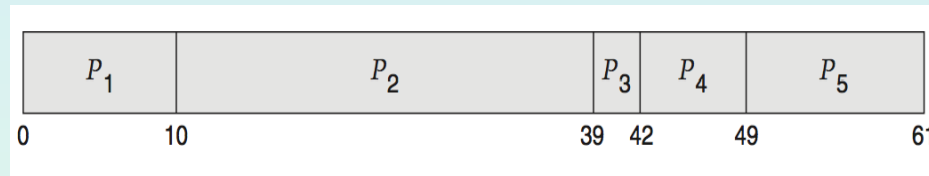
# Algorithm Evaluation

- 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:

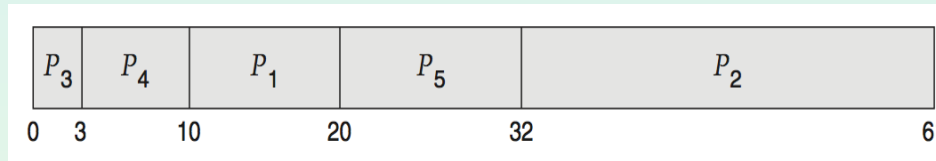
<u>Process</u>	<u>Burst Time</u>
$P_1$	10
$P_2$	29
$P_3$	3
$P_4$	7
$P_5$	12

# Deterministic Evaluation

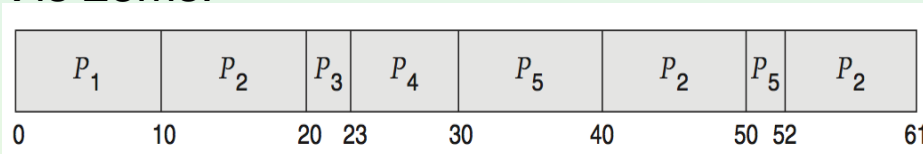
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:



- Non-preemptive SFJ is 13ms:



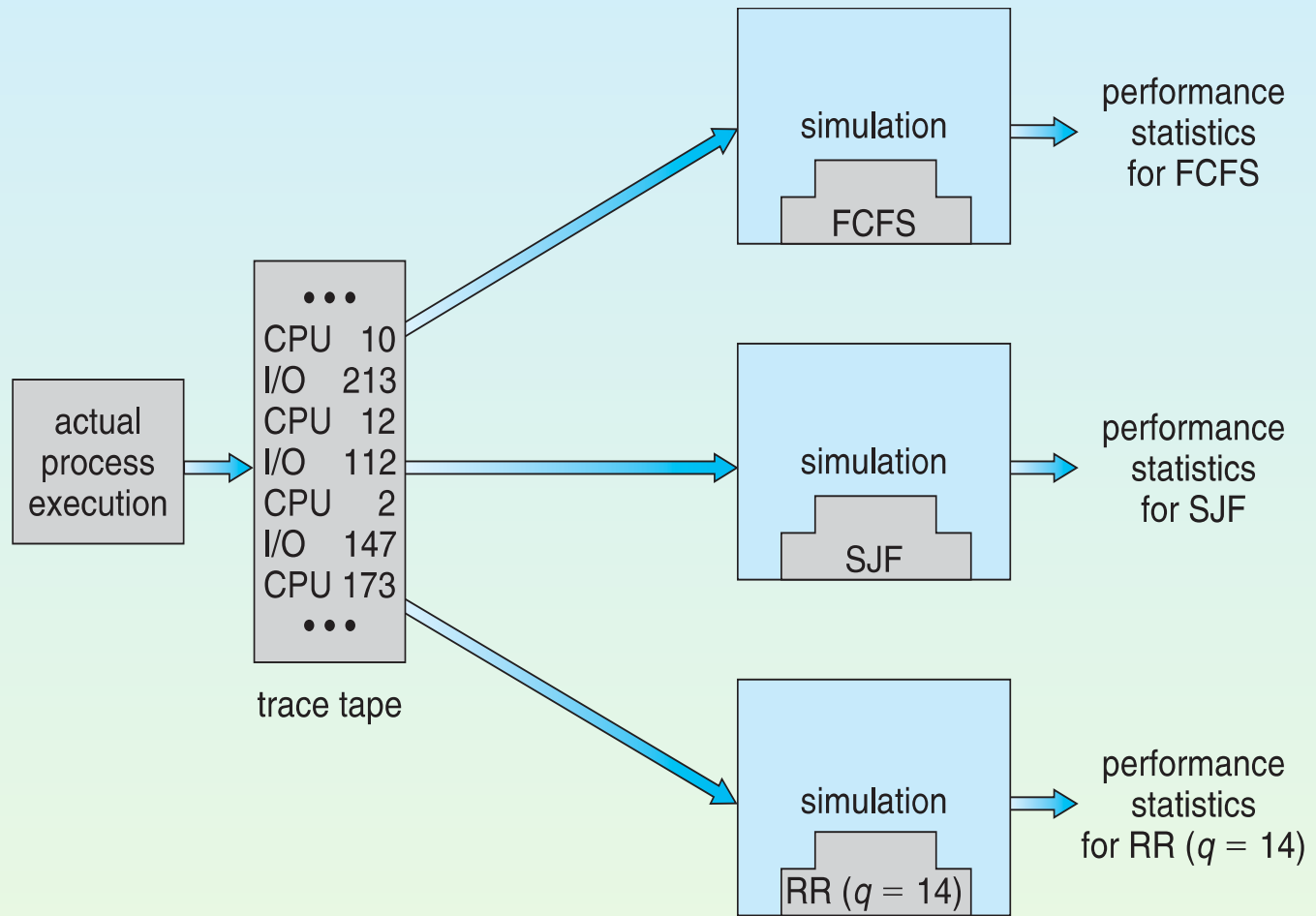
- RR is 23ms:



# Simulations

- **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 Lesson 5**  
**Chapter 6**