# Process Management

Mr.S. C. Sagare

# CONTENTS

- Process concept
- Process States
- Process Control Block
- Inter-process communication
- Process scheduling:-
  - Basic concepts
  - Scheduling Criteria
  - Scheduling Algorithms
  - Multiple processor scheduling
  - Real time CPU scheduling

# Introduction

- A process can be thought of as a program in execution.
- A process will need certain resources—such as CPU time, memory, files, and I/O devices —to accomplish its task.
- A process is the unit of work in most systems.
- Systems consist of a collection of processes: operating-system processes execute system Code
- User processes execute user code.
- All these processes may execute concurrently.
- To execute the program, OS needs to take program in the main memory.
- OS has to define a data structure in the main memory for execution that is the process.

- traditionally a process contained only a single thread of control as it ran, most modern operating systems now support processes that have multiple threads.

- The operating system is responsible for several important aspects of process and thread management

- the creation and deletion of both user and system processes; the scheduling of processes; and the provision of mechanisms for synchronization, communication, and deadlock handling for processes.

- Early computers allowed only one program to be executed at a time.This program had complete control of the system and had access to all the system's resources.

# Process Concept

- a process is a program in execution.

- A process is more than the program code, which is sometimes known as the **text section.**

- It also includes the current activity, as represented by the value of the **program counter and the contents of the processor's registers.**

- A process generally also includes the process **stack, which contains temporary data.**

- A process may also include a **heap, which is memory** that is dynamically allocated during process run time

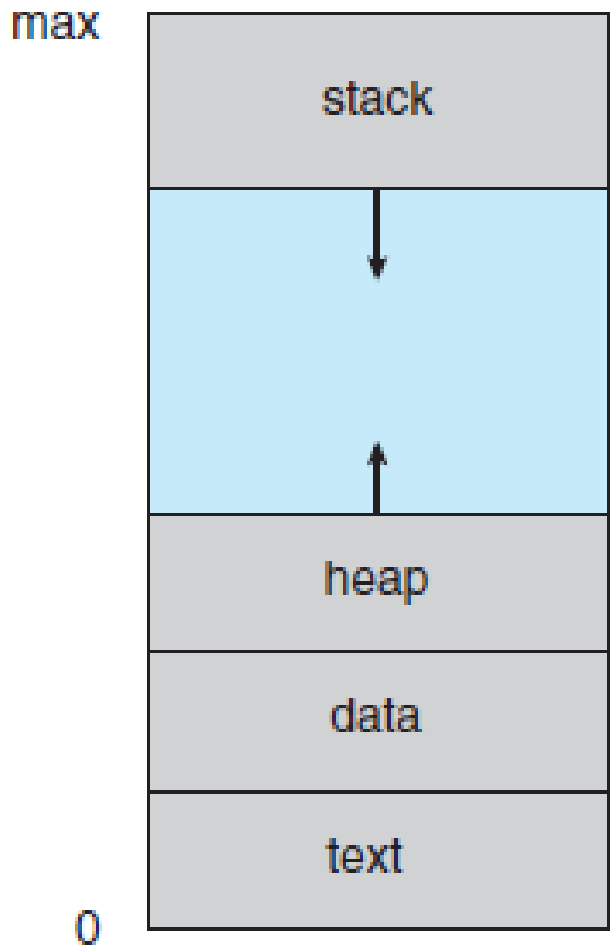



**Figure 3.1** Process in memory.

# Memory : Stack Vs Heap

- **The Stack**
- It's a special region of your computer's memory that stores temporary variables created by each function (including the main()function).
- The stack is a "LIFO" (last in, first out) data structure, that is managed and optimized by the CPU quite closely.
- Every time a function declares a new variable, it is "pushed" onto the stack.Then every time a function exits, **all** of the variables pushed onto the stack by that function, are freed (that is to say, they are deleted).
- Once a stack variable is freed, that region of memory becomes available for other stack variables.
- The advantage of using the stack to store variables, is that memory is managed for you.

- You don't have to allocate memory by hand, or free it once you don't need it any more, the CPU organizes stack memory so efficiently, reading from and writing to stack variables is very fast.

## The Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.

- To allocate memory on the heap, you must use malloc() or calloc(), which are built-in C functions.

- Once you have allocated memory on the heap, you are responsible for using free() to deallocate that memory once you don't need it any more.

- Unlike the stack, the heap does not have size restrictions on variable size.

- Heap memory is slightly slower to be read from and written to, because one has to use **pointers** to access memory on the heap.

- **Stack**
- very fast access
- don't have to explicitly de-allocate variables
- space is managed efficiently by CPU, memory will not become fragmented
- local variables only
- limit on stack size (OS-dependent)
- variables cannot be resized

- **Heap**
- variables can be accessed globally
- no limit on memory size
- (relatively) slower access
- no guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed
- you must manage memory (you're in charge of allocating and freeing variables)
- variables can be resized using realloc()

Example:-short program that creates its variables on the **stack**.

```
#include <stdio.h>

double multiplyByTwo (double input) {
  double twice = input * 2.0;
  return twice;
}

int main (int argc, char *argv[])
{
  int age = 30;
  double salary = 12345.67;
  double myList[3] = {1.2, 2.3, 3.4};

  printf("double your salary is %.3f\n", multiplyByTwo(salary));

  return 0;
}
```

Example:-short program that creates its variables on the **heap**.

```c
#include <stdio.h>
#include <stdlib.h>

double *multiplyByTwo (double *input) {
 double *twice = malloc(sizeof(double));
 *twice = *input * 2.0;
 return twice;
}

int main (int argc, char *argv[])
{
 int *age = malloc(sizeof(int));
 *age = 30;
 double *salary = malloc(sizeof(double));
 *salary = 12345.67;
 double *myList = malloc(3 * sizeof(double));
 myList[0] = 1.2;
 myList[1] = 2.3;
 myList[2] = 3.4;

 double *twiceSalary = multiplyByTwo(salary);

 printf("double your salary is %.3f\n", *twiceSalary);

 free(age);
 free(salary);
 free(myList);
 free(twiceSalary);

 return 0;
}
```

- a program by itself is not a process.
- A program is a ***passive entity, such as a file containing a list of instructions stored on disk*** (often called an **executable file).**
- **In contrast, a process is an *active entity*,** with a program counter specifying the next instruction to execute and a set of associated resources.
- A program becomes a process when an executable file is loaded into memory.
- Two common techniques for loading executable files:-
- double-clicking an icon representing the executable file.
- entering the name of the executable file on the command line

- Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences.
- For instance, several users may be running different copies of the mail program
- the same user may invoke many copies of the web browser program
- It is also common to have a process that handles many processes .
- Note that a process itself can be an execution environment for other code.
- The Java programming environment provides a good example.
- In most circumstances, an executable Java program is executed within the Java virtual machine (JVM).
- The JVM executes as a process that interprets the loaded Java code and takes actions

- For example, to run the compiled Java program Program.class, we would enter **java Program**

- The command java runs the JVM as an ordinary process, which in turns executes the Java program Program in the virtual machine.

# Process State

- As a process executes, it changes state.
- **New.** The process is being created.
- **Running.** Instructions are being executed.
- **Waiting.** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready.** The process is waiting to be assigned to a processor.
- **Terminated.** The process has finished execution

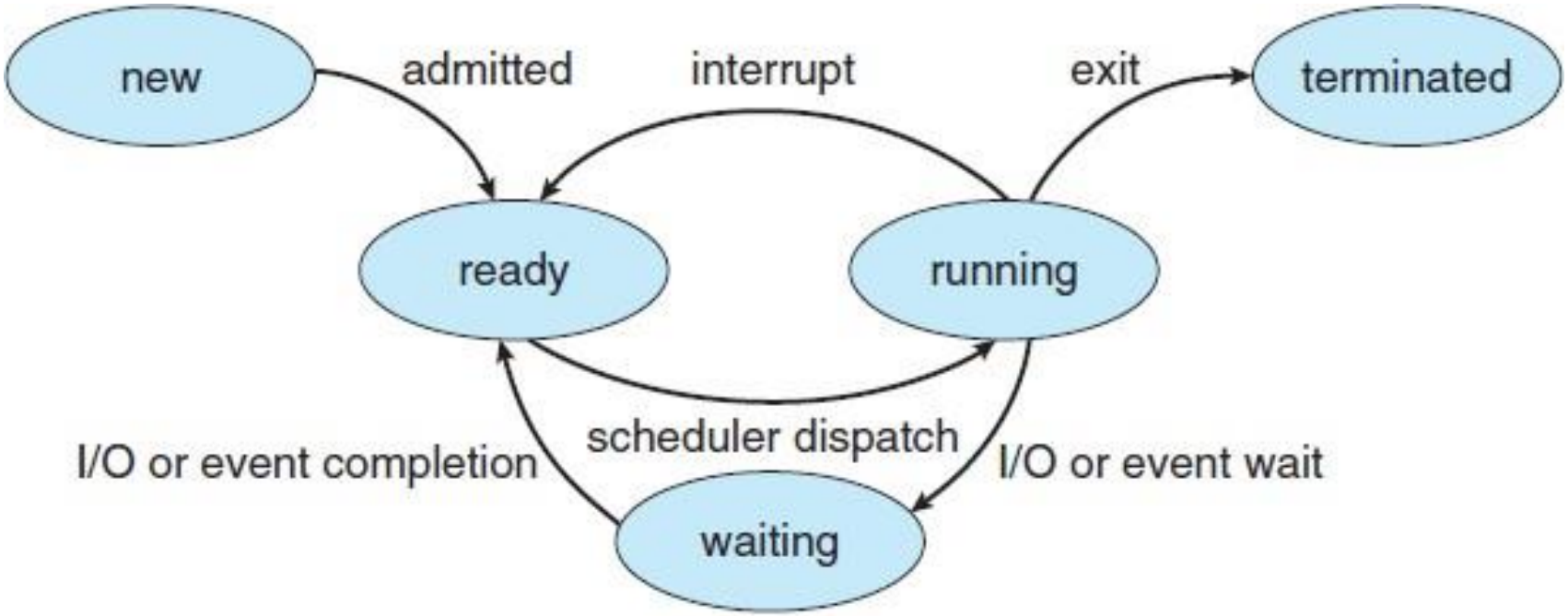



**Figure 3.2** Diagram of process state.

# Process Control Block

- Each process is represented in the operating system by a **process control block (PCB)—also called a task control block.**

- It contains many pieces of information associated with a specific process.

- **Process state.** The state may be new, ready, running, waiting, halted, and so on.

- **Program counter.** The counter indicates the address of the next instruction to be executed for this process

- **CPU registers**.The registers vary in number and type, depending on the computer architecture.They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.

- **CPU-scheduling information.** This information includes a process priority.

- **Memory-management information.** This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.

- **Accounting information. This information includes the amount of CPU** and real time used, time limits, account numbers, job or process numbers, and so on.

- **I/O status information.** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

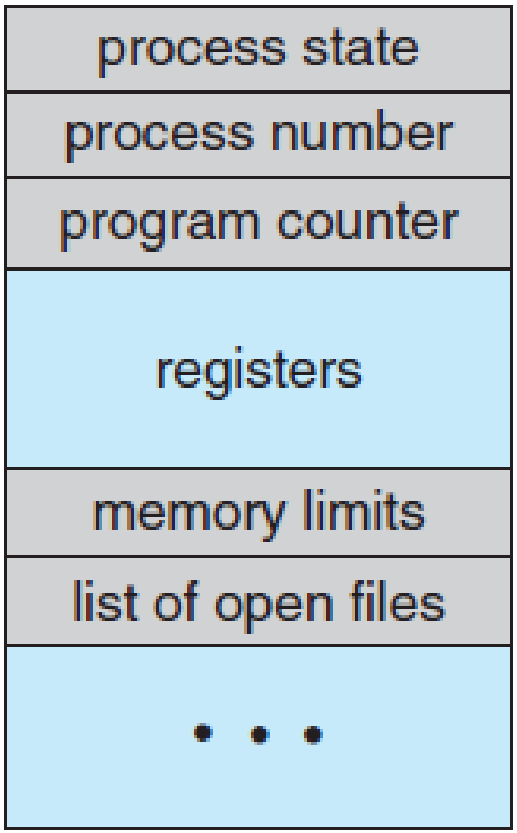



**Figure 3.3** Process control block (PCB).

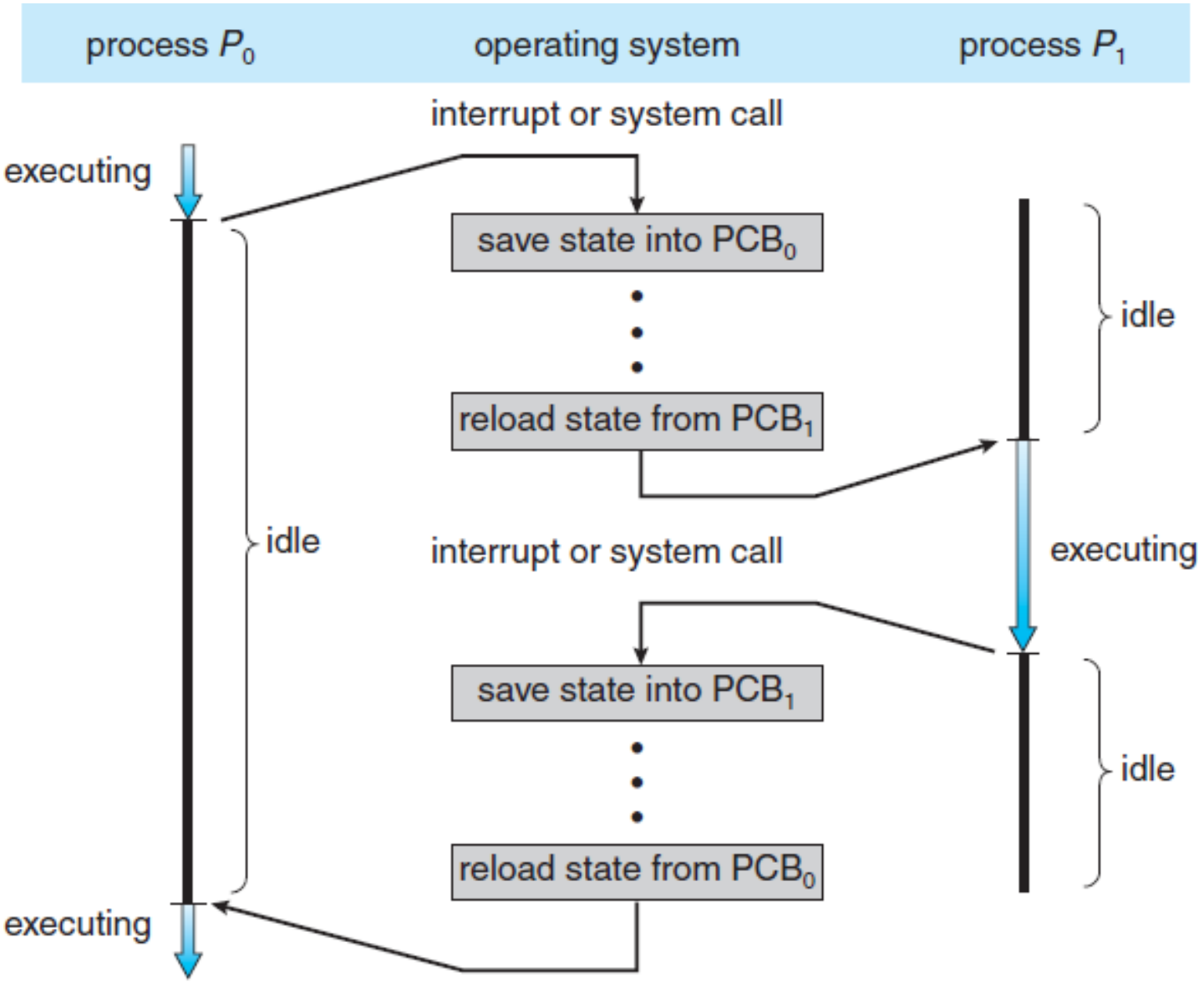



**Figure 3.4** Diagram showing CPU switch from process to process.

# Threads

- a process is a program that performs a single **thread of execution.**
- For example, when a process is running a word-processor program, a single thread of instructions is being executed.
- This single thread of control allows the process to perform only one task at a time.
- Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time

# Process Scheduling

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running..
- To meet these objectives, the **process scheduler selects** an available process (possibly from a set of several available processes) for program execution on the CPU.
- For a single-processor system, there will never be more than one running process.
- If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

# Scheduling Queues

- The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.
- As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue.**
- This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list.
- Each PCB includes a pointer field that points to the next PCB in the ready queue.
- The system also includes other queues. The list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue

- All processes, upon entering into the system, are stored in the **Job Queue**.
- Processes in the Ready state are placed in the **Ready Queue**.
- Processes waiting for a device to become available are placed in **Device Queues**.
- There are unique device queues available for each I/O device.
- A new process is initially put in the **Ready queue**.

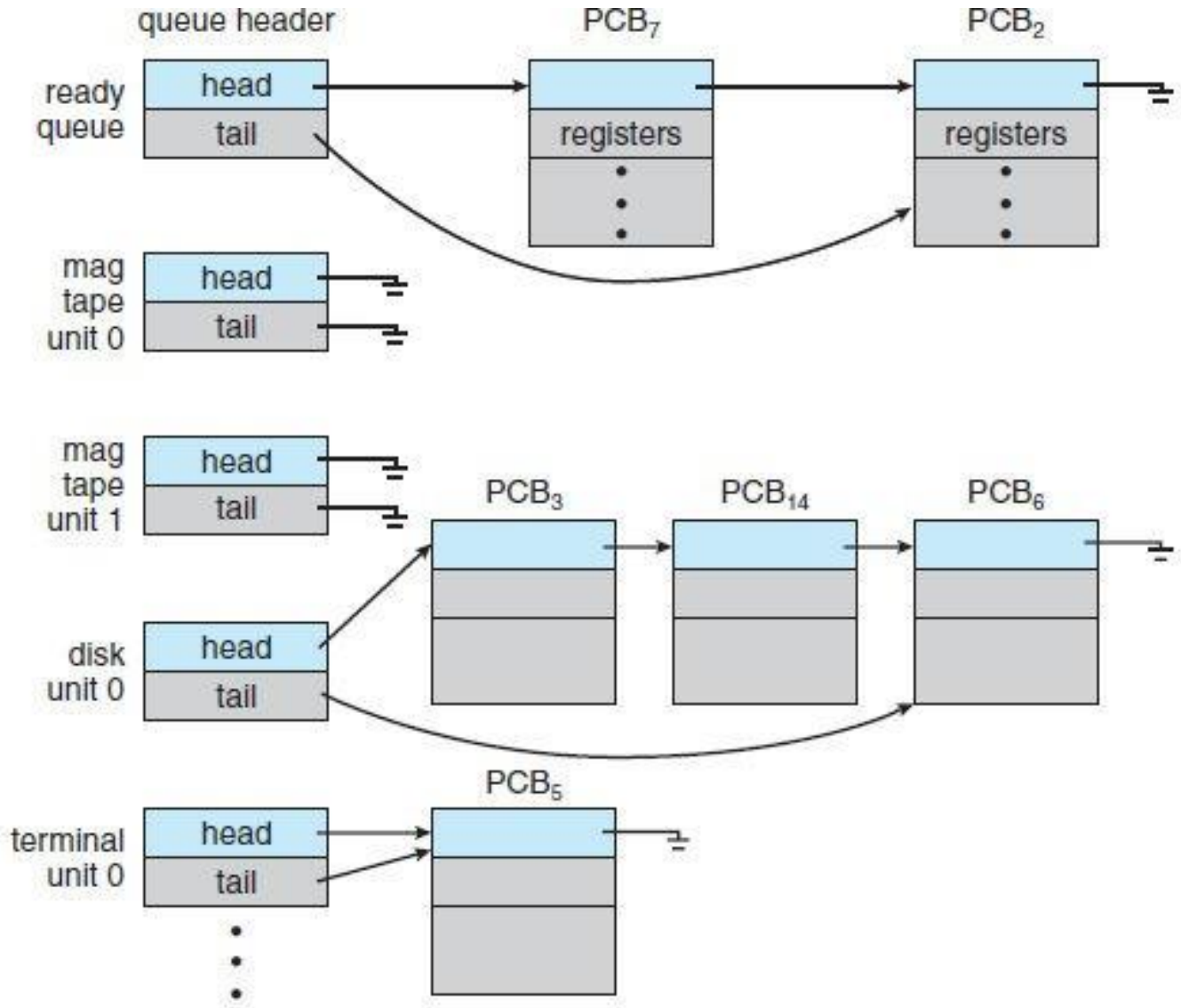



**Figure 3.5** The ready queue and various I/O device queues.

- Once the process is assigned to the CPU and is executing, one of the following several events can occur:
  - The process could issue an I/O request, and then be placed in the **I/O queue**.
  - The process could create a new sub-process and wait for its termination.
  - The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the **ready queue**.

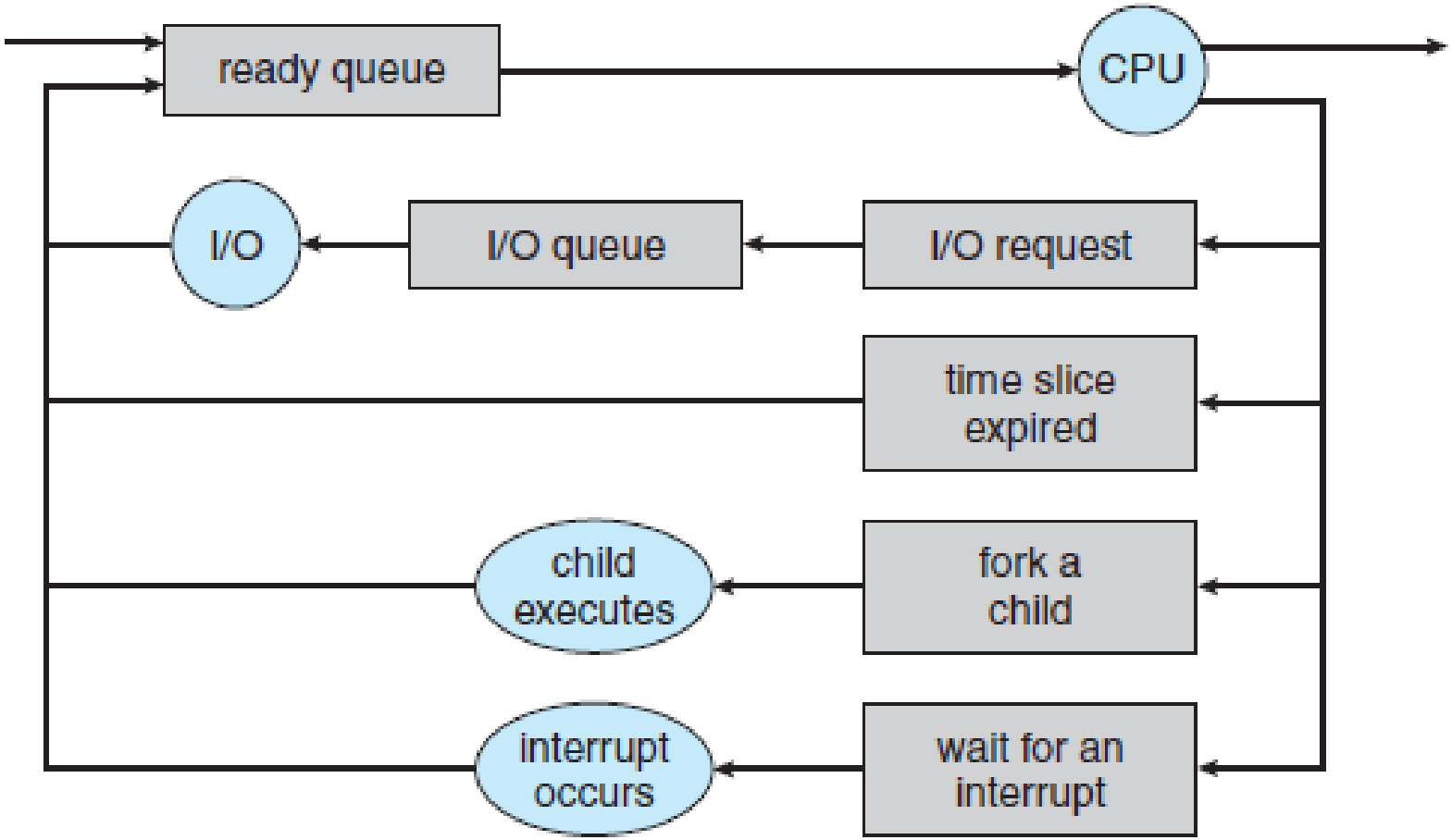



**Figure 3.6** Queueing-diagram representation of process scheduling.

# Schedulers

- A process migrates among the various scheduling queues throughout its lifetime.

- Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run.

- There are three types of schedulers available:
  - Long Term Scheduler
  - Short Term Scheduler
  - Medium Term Scheduler

## Long Term Scheduler

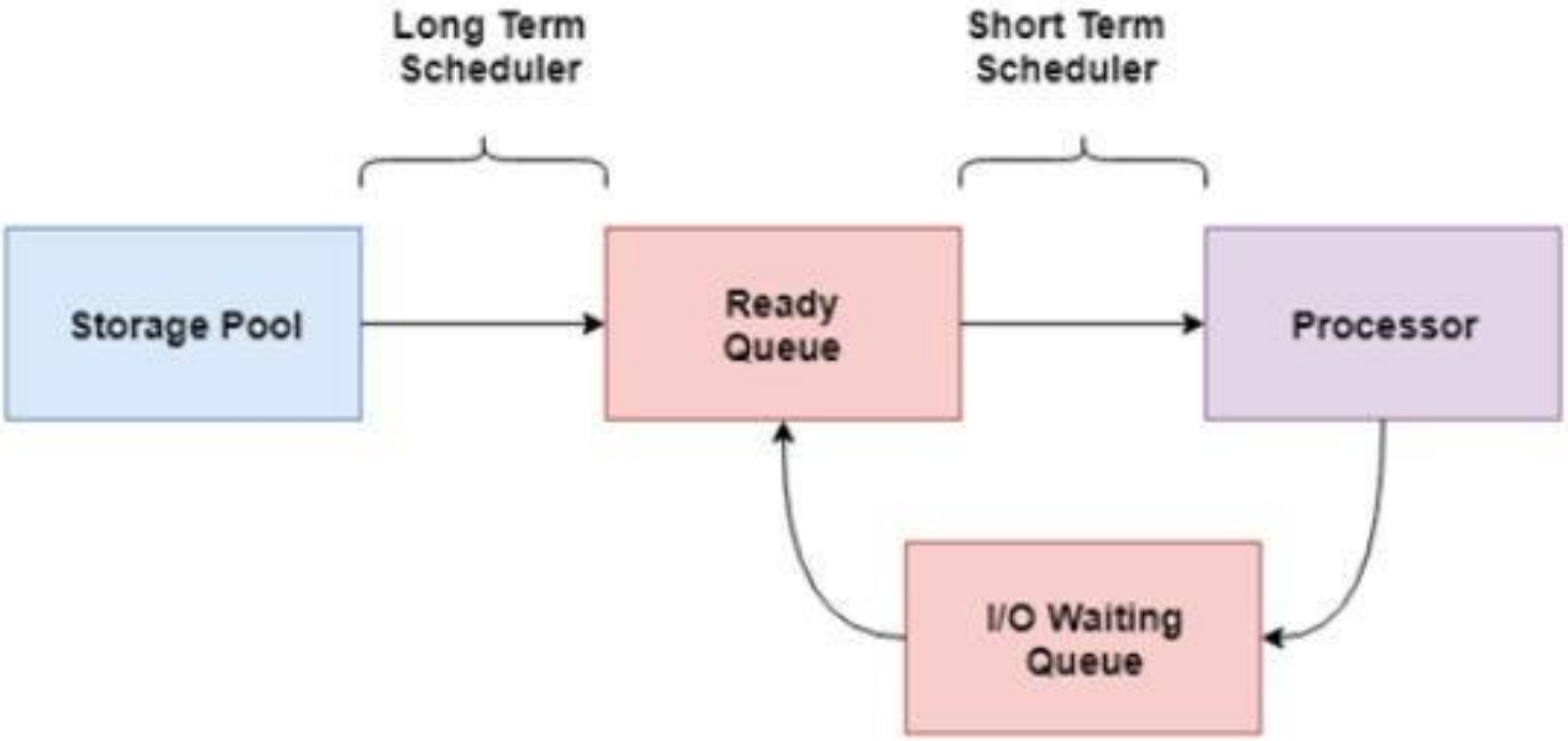
- Long term scheduler runs less frequently.
- It is also called a **job scheduler**. A long-term scheduler determines which programs are admitted to the system for processing
- Long Term Schedulers decide which program must get into the job queue.
- From the job queue, the Job Processor, selects processes and loads them into the memory for execution.
- Primary aim of the Job Scheduler is to maintain a good degree of Multiprogramming.
- An optimal degree of Multiprogramming means the average rate of process creation is equal to the average departure rate of processes from the execution memory

# Short TermScheduler

- Short-term scheduling involves selecting one of the processes from the ready queue and scheduling them for execution.This is done by the short-term scheduler.A scheduling algorithm is used to decide which process will be scheduled for execution next by the short-term scheduler.

- The short-term scheduler executes much more frequently than the long-term scheduler as a process may execute only for a few milliseconds.

- The choices of the short term scheduler are very important. Ifit selects a process with a long burst time, then all the processes after that will have to wait for a long time in the ready queue.This is known as starvation and it may happen if a wrong decision is made by the short-term scheduler.




Representation of Short Term and Long Term Scheduler using a Queuing Diagram

## Medium Term Scheduler

- Medium-term scheduling involves swapping out a process from main memory. The process can be swapped in later from the point it stopped executing. This can also be called as suspending and resuming the process and is done by the medium-term scheduler..

- At some later time, the process can be reintroduced into memory and its execution van be continued where it left off.

- This scheme is called swapping. The process is swapped out, and is later swapped in, by the medium term scheduler.

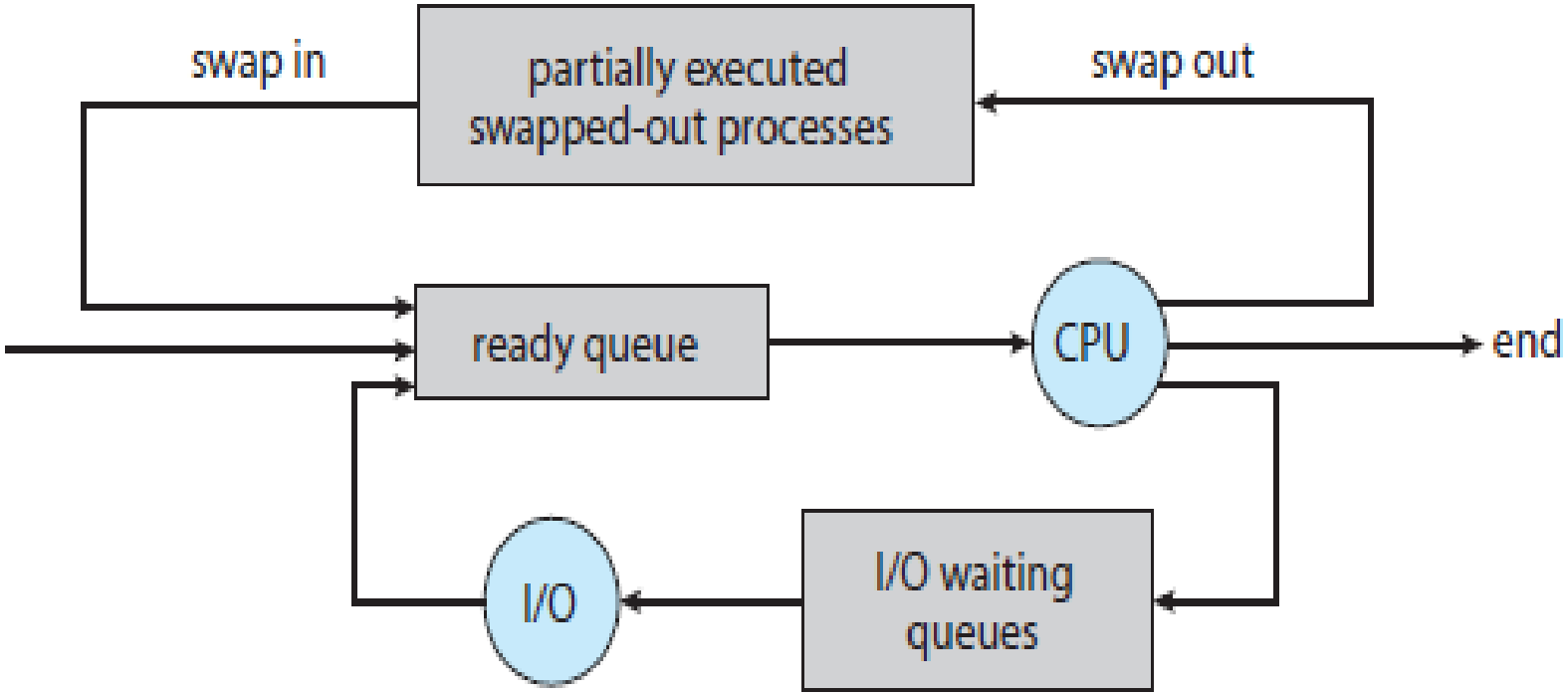



**Figure 3.7** Addition of medium-term scheduling to the queueing diagram.

# What is Context Switch?

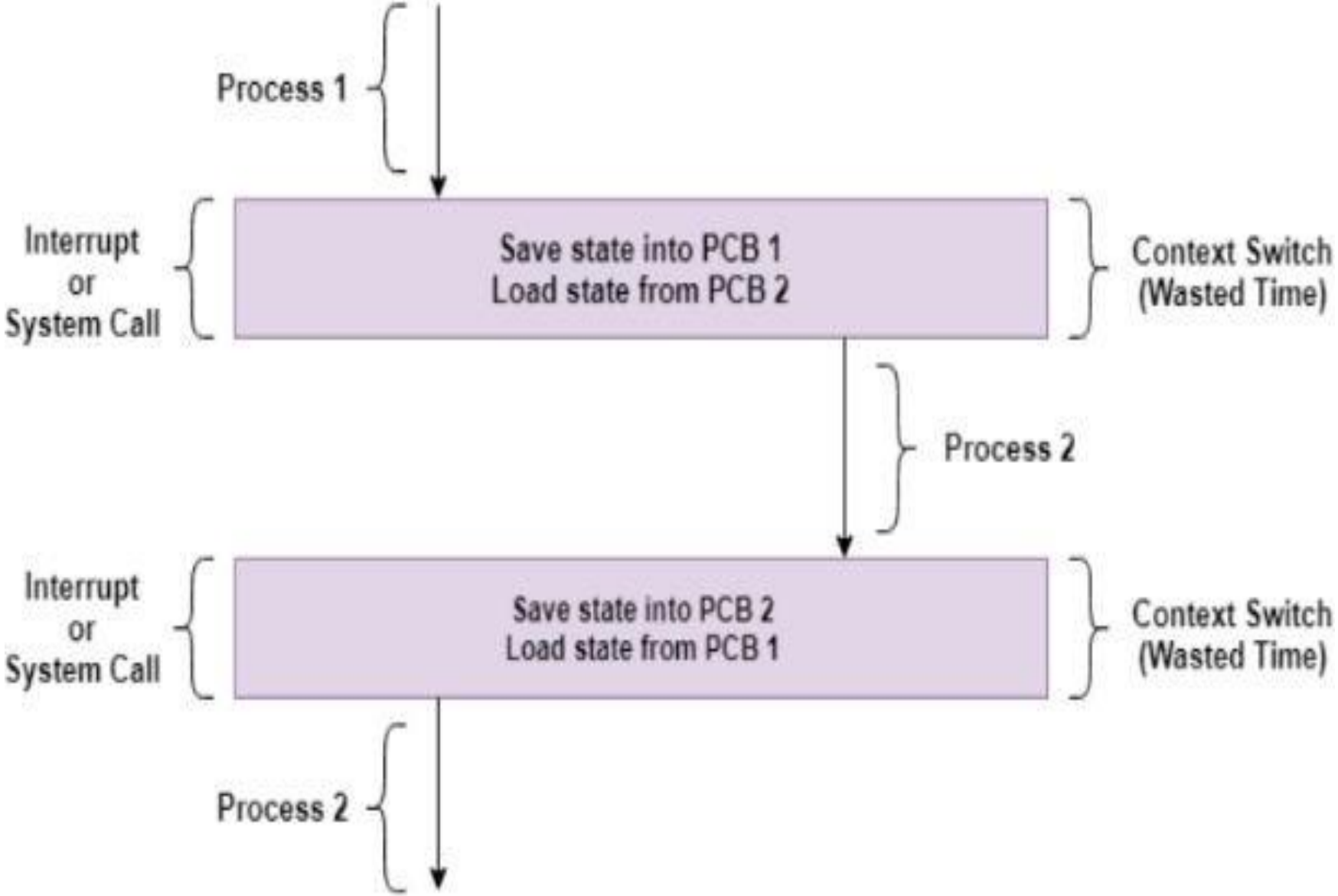
- Switching the CPU to another process requires **saving** the state of the old process and **loading** the saved state for the new process.This task is known as a **Context Switch**.

- The **context** of a process is represented in the **Process Control Block(PCB)** of a process; it includes the value of the CPU registers, the process state and memory-management information.

- When a context switch occurs, the Kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

- Context Switching involves storing the context or state of a process so that it can be reloaded when required and execution can be resumed from the same point as earlier. This is a feature of a multitasking operating system and allows a single CPU to be shared by multiple processes.

- A diagram that demonstrates context switching is asfollows


Process 1
Interrupt or System Call
Save state into PCB 1
Load state from PCB 2
Context Switch (Wasted Time)
Process 2
Interrupt or System Call
Save state into PCB 2
Load state from PCB 1
Context Switch (Wasted Time)
Process 2

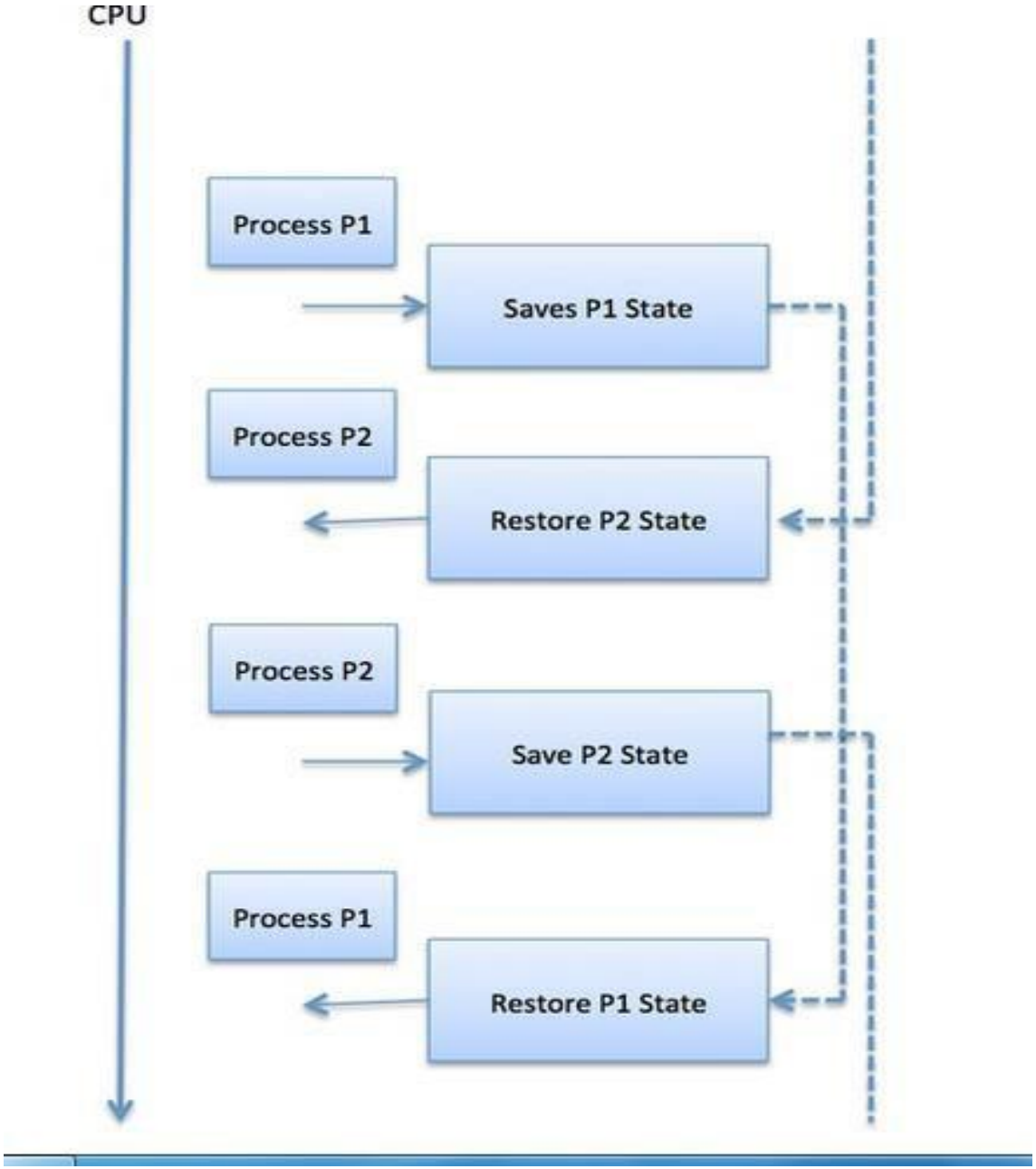
## Context Switching Triggers

There are three major triggers for context switching. These are given as follows:

- **Multitasking:** In a multitasking environment, a process is switched out of the CPU so another process can be run. The state of the old process is saved and the state of the new process is loaded.
- **Interrupt Handling:** The hardware switches a part of the context when an interrupt occurs. This happens automatically. Only some of the context is changed to minimize the time required to handle the interrupt.
- **User and Kernel Mode Switching:** A context switch may take place when a transition between the user mode and kernel mode is required in the operating system.

## Context Switching Steps:-

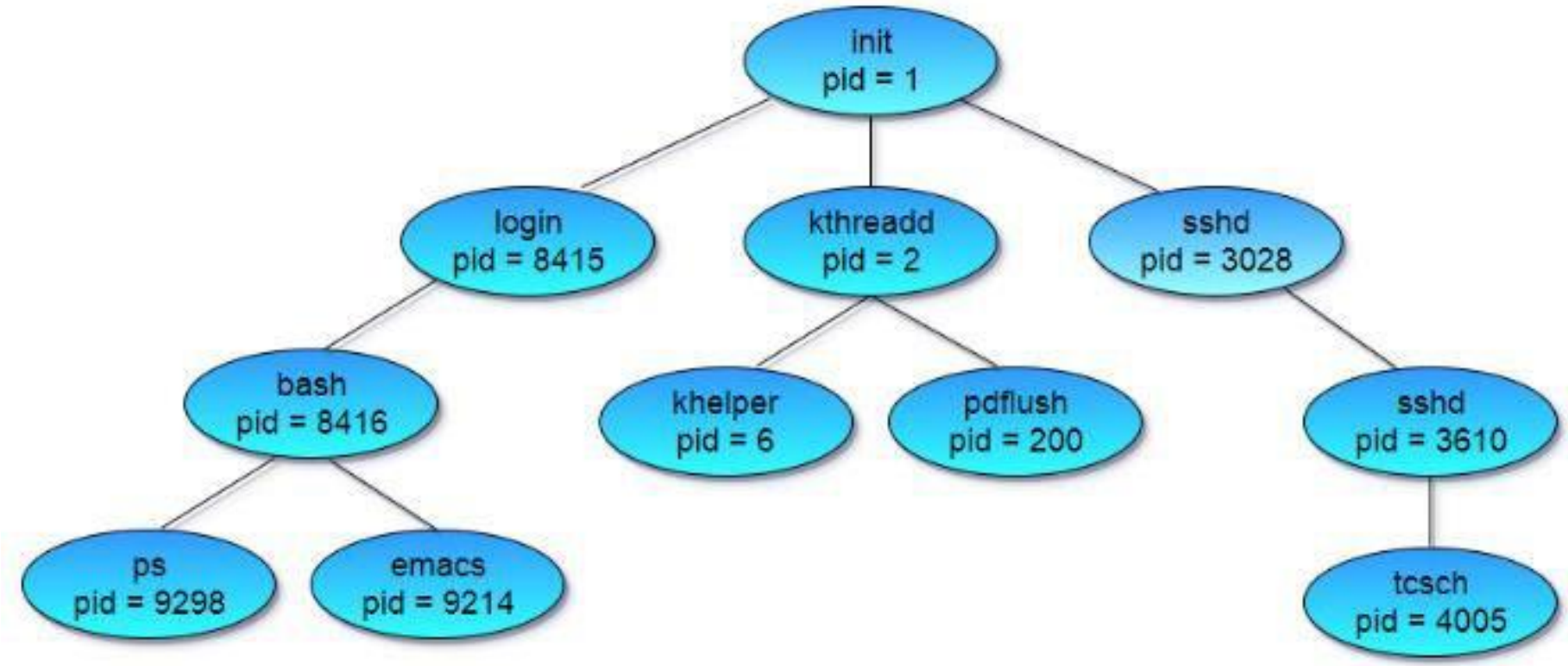
- The steps involved in context switching are as follows:
- Save the context of the process that is currently running on the CPU. Update the process control block and other important fields.
- Move the process control block of the above process into the relevant queue such as the ready queue, I/O queue etc.
- Select a new process for execution.
- Update the process control block of the selected process.This includes updating the process state to running.
- Update the memory management data structures as required.
- Restore the context of the process that was previously running when it is loaded again on the processor.This is done by loading the previous values of the process control block and registers.


CPU
Process P1
Saves P1 State
Process P2
Restore P2 State
Process P2
Save P2 State
Process P1
Restore P1 State

# Operations on Processes

- **Process Creation:-**
  - Through appropriate system calls, such as fork or spawn, processes may create other processes.
  - The process which creates other process, is termed the **parent** of the other process, while the created sub-process is termed its **child**.
  - Each process is given an integer identifier, termed as process identifier, or PID.The parent PID (PPID) is also stored for each process.
  - On a typical UNIX systems the process scheduler is termed as sched, and is given PID 0.
  - The first thing done by it at system start-up time is to launch init, which gives that process PID 1. Further Init launches all the system daemons and user logins, and becomes the ultimate parent of all other processes.

In Unix-based computer operating systems, **init** is the first **process** started during booting of the computer system. **Init** is a daemon **process** that continues running until the system is shut down.



- There are two options for the parent process after creating the child :
  - The parent continues to execute concurrently with its children. Parent process makes a wait() system call, for either a specific child process or for any particular child process
  - The parent waits until some or all of its children have terminated.

- There are also two possibilities in terms of the address space of the new process:
  - The child process is a duplicate of the parent process (it has the same program and data as the parent).
  - The child process has a new program loaded into it.

- In UNIX, each process is identified by its **process identifier**, which is a unique integer.
- A new process is created by the **fork** system call.
- The new process consists of a copy of the address space of the original process.
- This mechanism allows the parent process to communicate easily with its child process.
- Both processes continue execution at the instruction after the fork system call, with one difference: **The return code for the fork system call is zero for the new(child) process, whereas the(non zero) process identifier of the child is returned to the parent.**

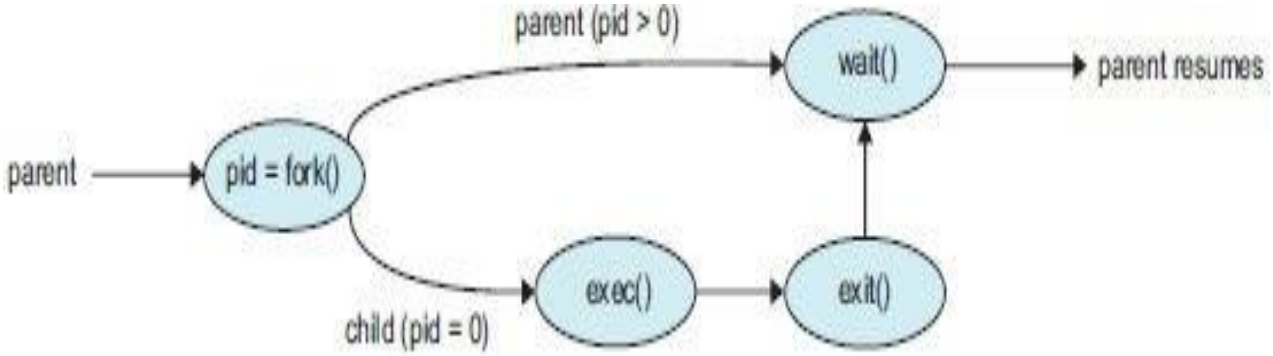



**Figure 3.10** Process creation using the `fork()` system call.

# Process Termination

- By making the exit(system call), typically returning an int, processes may request their own termination.
- This int is passed along to the parent if it is doing a wait(), and is typically zero on successful completion and some non-zero code in the event of any problem.

- Processes may also be terminated by the system for a variety of reasons, including :
  - The inability of the system to deliver the necessary system resources.
  - In response to a KILL command or other unhandled process interrupts.
  - A parent may kill its children if the task assigned to them is no longer needed i.e. if the need of having a child terminates.
  - If the parent exits, the system may or may not allow the child to continue without a parent

# Process Management

Mr.S. C. Sagare

# CONTENTS

- Process concept
- Process States
- Process Control Block
- Inter-process communication
- Process scheduling:-
  - Basic concepts
  - Scheduling Criteria
  - Scheduling Algorithms
  - Multiple processor scheduling
  - Real time CPU scheduling

# Inter-process Communication

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- A process is ***independent*** if it cannot affect or be affected by the other processes executing in the system.
- Any process that does not share data with any other process is independent.
- A process is ***cooperating*** *if it can affect or be affected by the other processes* executing in the system.
- Clearly, any process that shares data with other processes is a cooperating process.

- **There are several reasons for providing an environment that allows process cooperation:**

- **Information sharing:** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

- **Computation speedup:** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.

- **Modularity:** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads

- **Convenience:** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

- Cooperating processes require an **inter-process communication (IPC) mechanism** that will allow them to exchange data and information.

- There are two fundamental models of inter process communication: **shared memory and message passing.**

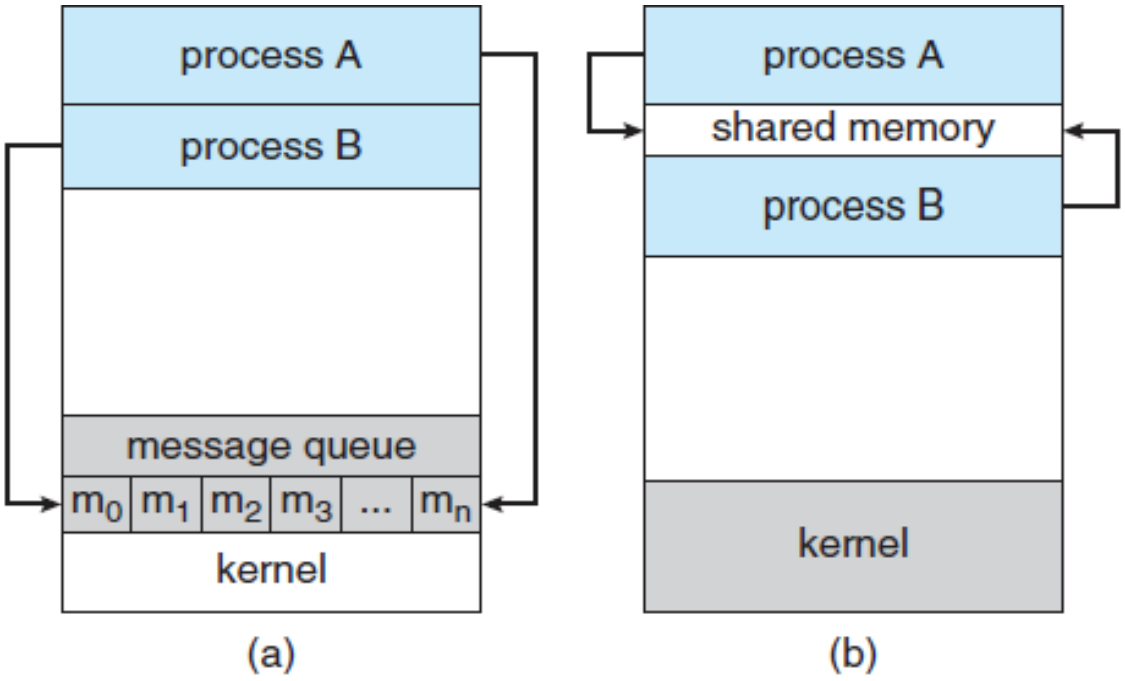



**Figure 3.12** Communications models. (a) Message passing. (b) Shared memory.

- Shared-Memory Systems:-
- Inter-process communication using shared memory requires communicating processes to establish a region of shared memory.
- Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space

## Message-Passing Systems

- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.

- It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network

- an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages

# *MULTIPROCESS ARCHITECTURE—CHROME BROWSER*

Many websites contain active content such as JavaScript, Flash, and HTML5 to provide a rich and dynamic web-browsing experience. Unfortunately, these web applications may also contain software bugs, which can result in sluggish response times and can even cause the web browser to crash. This isn't a big problem in a web browser that displays content from only one website. But most contemporary web browsers provide tabbed browsing, which allows a single instance of a web browser application to open several websites at the same time, with each site in a separate tab. To switch between the different sites , a user need only click on the appropriate tab. This arrangement is illustrated below:


A problem with this approach is that if a web application in any tab crashes, the entire process—including all other tabs displaying additional websites—crashes as well.

Google's Chrome web browser was designed to address this issue by using a multiprocess architecture. Chrome identifies three different types of processes: browser, renderers, and plug-ins.

- The browser process is responsible for managing the user interface as well as disk and network I/O. A new browser process is created when Chrome is started. Only one browser process is created.

- Renderer processes contain logic for rendering web pages. Thus, they contain the logic for handling HTML, Javascript, images, and so forth. As a general rule, a new renderer process is created for each website opened in a new tab, and so several renderer processes may be active at the same time.
- A plug-in process is created for each type of plug-in (such as Flash or QuickTime) in use. Plug-in processes contain the code for the plug-in as well as additional code that enables the plug-in to communicate with associated renderer processes and the browser process.

The advantage of the multiprocess approach is that websites run in isolation from one another. If one website crashes, only its renderer process is affected; all other processes remain unharmed. Furthermore, renderer

- **Naming:-**
- Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

- Under **direct communication,each process that wants to communicate** must explicitly name the recipient or sender of the communication.

- send(P, message)—Send a message to process P.
- receive(Q, message)—Receive a message from process Q.

- With **indirect communication**, the messages are sent to and received from mailboxes, or ports.

- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.

- send(A, message)—Send a message to mailbox A.
- receive(A, message)—Receive a message from mailbox A.

# Process Management

Mr.S. C. Sagare

# CONTENTS

- Process concept
- Process States
- Process Control Block
- Inter-process communication
- Process scheduling:-
  - Basic concepts
  - Scheduling Criteria
  - Scheduling Algorithms
  - Multiple processor scheduling
  - Real time CPU scheduling

# Communication in Client-Server Systems

- **1. Sockets:-**
  - A socket is defined as an endpoint for communication.
  - A pair of processes communicating over a network employs a pair of sockets—one for each process.
  - socket is identified by an IP address concatenated with a port number.
  - sockets use a client-server architecture.
  - The server waits for incoming client requests by listening to a specified port.
  - Once a request is received, the server accepts a connection from the client socket to complete the connection.

- All ports below 1024 are considered well known; we can use them to implement standard services.
- Servers implementing specific services (such as telnet, FTP, and HTTP) listen to well-known ports.
- A telnet server listens to port 23; an FTP server listens to port 21; and a web, or HTTP, server listens to port 80

- When a client process initiates a request for a connection, an arbitrary number is assigned a port by its host computer greater than 1024

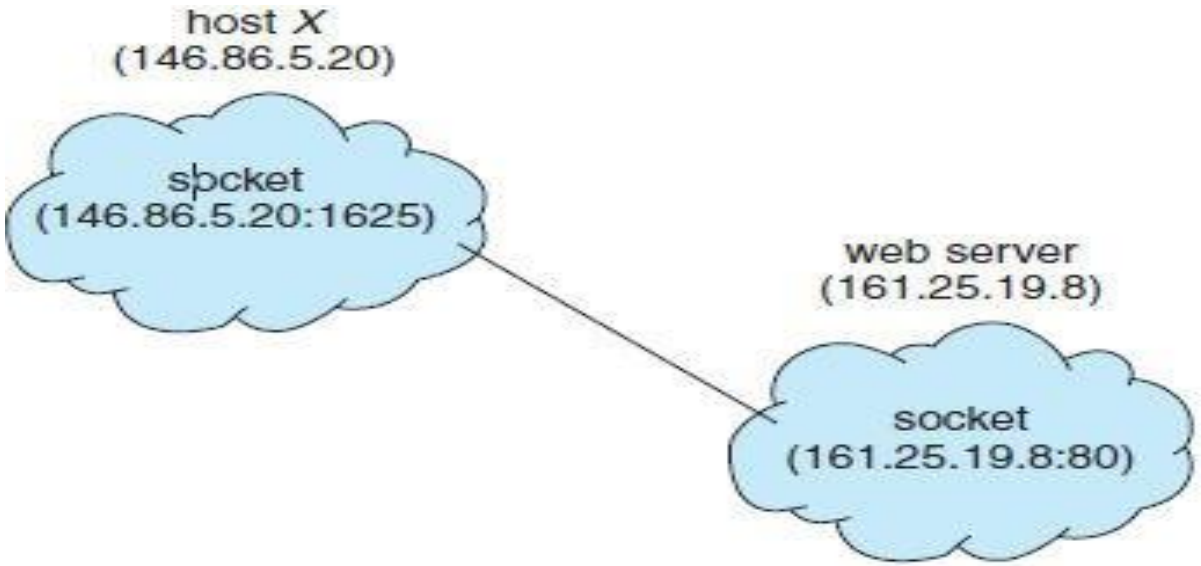



**Figure 3.20** Communication using sockets.

# Communication in Client-Server Systems

- **2. Remote Procedure Calls:-**
  - The RPC was designed as a way to abstract the procedure-call mechanism for use between systems with network connections.
  - It is similar in many respects to the IPC mechanism however, because we are dealing with an environment in which the processes are executing on separate systems, we must use a message-based communication scheme to provide remote service.
  - In contrast to IPC messages, the messages exchanged in RPC communication are well structured and are thus no longer just packets of data.
  - Each message is addressed to an RPC daemon listening to a port on the remote system, and each contains an identifier specifying the function to execute and the parameters to pass to that function.
  - The function is then executed as requested, and any output is sent back to the requester in a separate message.

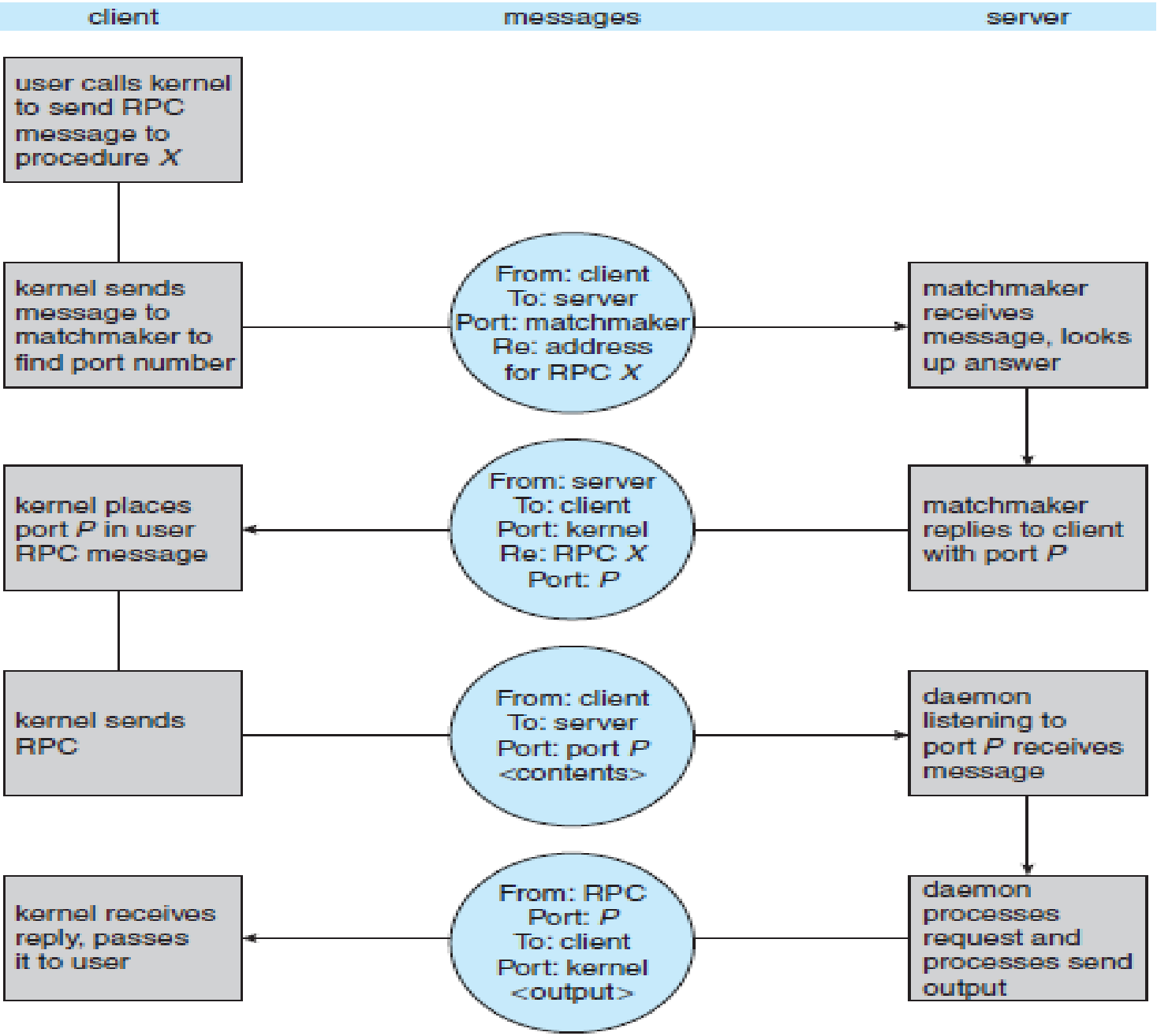



**Figure 3.23** Execution of a remote procedure call (RPC).

# Communication in Client-Server Systems

- **3. Pipes:-**
- A **pipe** acts as a conduit allowing two processes to communicate.
- Pipes were one of the first IPC mechanisms in early UNIX systems.
- They typically provide one of the simpler ways for processes to communicate with one another, although they also have some limitations.
- In implementing a pipe, four issues must be considered:
  - **1.** Does the pipe allow bidirectional communication, or is communication unidirectional?
  - **2.** If two-way communication is allowed, is it half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions at the same time)?
  - **3.** Must a relationship (such as ***parent-child***) exist between the communicating processes?
  - **4.** Can the pipes communicate over a network, or must the communicating processes reside on the same machine?

- **Ordinary Pipes**
- Ordinary pipes allow two processes to communicate in standard producer-consumer fashion: the producer writes to one end of the pipe (the **write-end**) and the consumer reads from the other end (the **read-end**).
- As a result, ordinary pipes are unidirectional, allowing only one-way communication.
- If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction.
- On UNIX systems, ordinary pipes are constructed using the function: **pipe(int fd[])**
- This function creates a pipe that is accessed through the file descriptors: **int fd[]**
- File descriptors:
  - **fd[0]** is the read-end of the pipe, and
  - **fd[1]** is the write-end

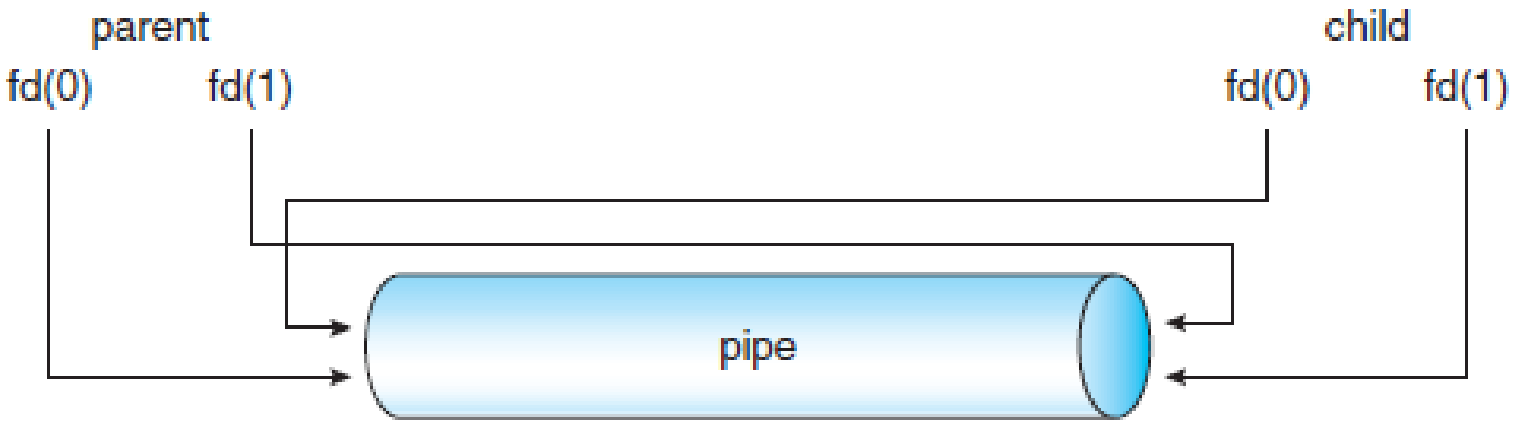



**Figure 3.24** File descriptors for an ordinary pipe.

- **Named Pipes**
- Ordinary pipes provide a simple mechanism for allowing a pair of processes to communicate. However, ordinary pipes exist only while the processes are communicating with one another.
- On both UNIX and Windows systems, once the processes have finished communicating and have terminated, the ordinary pipe ceases to exist.
- Named pipes provide a much more powerful communication tool. Communication can be **bidirectional**, and **no parent-child relationship** is required.
- Once a named pipe is established, **several processes** can use it for communication.
- In fact, in a typical scenario, a named pipe has **several writers**.
- Additionally, named pipes **continue to exist** after communicating processes have finished.
- Both UNIX and Windows systems support named pipes, although the details of implementation differ greatly,
- It will continue to exist until it is explicitly deleted from the file system.

# Pipes in UNIX Vs Pipes in Windows

- **UNIX:**
- Named pipes are referred to as FIFOs in UNIX systems. Although FIFOs allow bidirectional communication, only half-duplex transmission is permitted.
- If data must travel in both directions, two FIFOs are typically used. Additionally, the communicating processes must reside on the same machine. If inter-machine communication is required, sockets must be used.
- **Windows:**
- Named pipes on Windows systems provide a richer communication mechanism than their UNIX counterparts.
- Full-duplex communication is allowed, and the communicating processes may reside on either the same or different machines.
- Additionally, only byte-oriented data may be transmitted across a UNIX FIFO, whereas Windows systems allow either byte- or message-oriented data.

# Process scheduling

- CPU scheduling is the basis of multi-programmed operating systems.
- By switching the CPU among processes, the operating system can make the computer more productive.
- In a single-processor system, only one process can run at a time. Others must wait until the CPU is free and can be rescheduled.
- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- A process is executed until it must wait, typically for the completion of some I/O request.
- In a simple computer system, the CPU then just sits idle. All this waiting time is wasted;

- With multiprogramming, we try to use this time productively.
- Several processes are kept in memory at one time.
- When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process.This pattern continues.

## CPU-I/O Burst Cycle:-

- The success of CPU scheduling depends on an observed property of processes
- process execution consists of a **cycle of CPU execution and I/O wait.**
- **Processes** alternate between these two states.
- Process execution begins with a **CPU burst.**
- **CPU burst** is when the process is being executed in the **CPU.**
- **I/O burst** is when the CPU is waiting for **I/O** for further execution.
- That is followed by an **I/O burst, which is followed by another CPU burst, then** another I/O burst, and so on.

# CPU Scheduler:-

- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.
- In an operating system (OS), a process scheduler performs the important activity of scheduling a process between the ready queue and waiting queue and allocating them to the CPU
- The OS assigns priority to each process and maintains these queues.
- **The scheduler selects a process** from the processes in memory that are ready to execute and allocates the CPU to that process.
- Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue.
- Depends on scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list.

- Preemptive Scheduling:-
- CPU-scheduling decisions may take place under the following four circumstances:
  - When a process switches from the running state to the waiting state.
  - When a process switches from the running state to the ready state. (for example, when an interrupt occurs).
  - When a process switches from the waiting state to the ready state.
  - When a process terminates.

# Preemptive Scheduling:-

- The scheduling in which a running process can be interrupted if a high priority process enters the queue and is allocated to the CPU is called **preemptive scheduling**.

- In this case, the current process switches from the running queue to ready queue, and the high priority process utilizes the CPU cycle.

- A process P2 arrives at time 0 and allocated the CPU. Process P3 arrives at time 1, before P2 finishes execution. The time remaining for P2 is 5 milliseconds, which is larger than a time required for P3 (4 milliseconds). So CPU is allocated to process P3. Process P1 arrives at time 2. P3 continues to execute because the remaining time for P3 (3 milliseconds) is less than the time required by processes P1 (4 milliseconds) and P2 (5 milliseconds).

- Process P0 arrives at time 3. Now P3 continues to run because the remaining time for P3 (2 milliseconds) is equal to the time required by P0 (2 milliseconds). After P3 finishes, the CPU is allocated to P0 as it has smaller burst time than other processes. Later, the CPU is allocated to P1 and then to P2.

## Non-Preemptive Scheduling:-

- The scheduling in which a running process cannot be interrupted by any other process is called **non-preemptive scheduling**.

- Any other process which enters the queue has to wait until the current process finishes its CPU cycle.

- Process P2 arrives at time 0 and is allocated the CPU until it finishes execution. While P2 is executing, processes P0, P1, P3 arrive into the ready queue. But all other processes have to wait until process P2 finishes its execution. After P2 finishes its CPU cycle, based on the arrival time, process P3 is allocated the CPU. After P3 finishes, P1 executes and then P0.

## Dispatcher:-

- Another component involved in the CPU-scheduling function is the **dispatcher.**

- The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.

# Scheduling Criteria

- Different CPU-scheduling algorithms have different properties.
- In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.
- Many criteria have been suggested for comparing CPU-scheduling algorithms.
- The criteria include… …
- **CPU utilization**.We want to keep the CPU as busy as possible.
- **Throughput**. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

- **Turnaround time.** The interval from the time of submission of a process to the time of completion is the turnaround time.

- Turnaround time = periods spent waiting to get into memory + waiting in the ready queue + executing on the CPU + and doing I/O.

- **Waiting time:-** Waiting time is the sum of the periods spent waiting in the ready queue.

- The CPU-scheduling algorithm affects only the amount of time that a process spends waiting in the ready queue

- **Response time.:-** the time from the submission of a request until the first response is produced

# Scheduling Algorithms

- CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.
- First-Come, First-Served Scheduling:-
- The simplest CPU-scheduling algorithm is the **first-come,first-served (FCFS) scheduling algorithm.**
- **With this scheme,the process that requests the** CPU first is allocated the CPU first.
- The implementation of the FCFS policy is easily managed with a FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.When the CPU is free, it is allocated to the process at the head of the queue.
- The running process is then removed from the queue.

- Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |


The waiting time is 0 milliseconds for process $P_1$, 24 milliseconds for process $P_2$, and 27 milliseconds for process $P_3$. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order $P_2$, $P_3$, $P_1$, however, the results will be as shown in the following Gantt chart:


The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.

# Shortest-Job-First Scheduling

- Shortest Job First scheduling works on the process with the shortest **burst time** or **duration** first.

- When the CPU is available, it is assigned to the process that has the smallest next CPU burst.

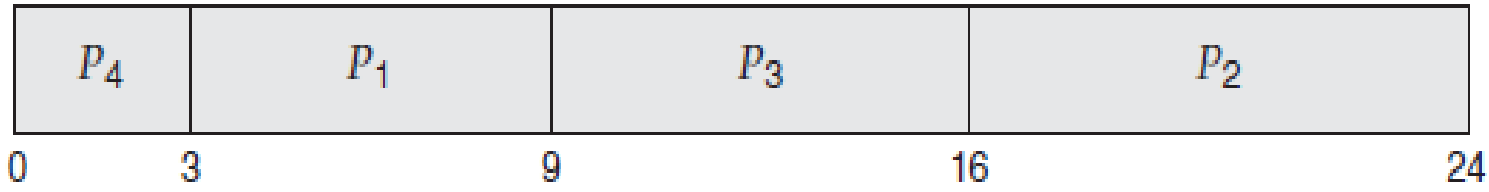
- If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

- more appropriate term for this scheduling method would be the ***shortest-next- CPU-burst algorithm, because scheduling depends on the length of the next*** CPU burst of a process.

As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

| Process | Burst Time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 8 |
| $P_3$ | 7 |
| $P_4$ | 3 |

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:




The waiting time is 3 milliseconds for process $P_1$, 16 milliseconds for process $P_2$, 9 milliseconds for process $P_3$, and 0 milliseconds for process $P_4$. Thus, the average waiting time is $(3 + 16 + 9 + 0)/4 = 7$ milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

- To successfully implement it, the burst time/duration time of the processes should be known to the processor in advance, which is practically not feasible all the time.

- This scheduling algorithm is optimal if all the processes are available at the same time. (either Arrival time is 0 for all, or Arrival time is same for all).

- **It is of two types**
- **Non Pre-emptive**
- **Pre-emptive**
- The choice arises when a new process arrives at the ready queue while a previous process is still executing.

## Non Pre-emptive:-

- A non-preemptive SJFalgorithm will allow the currently running process to finish its CPU burst.

- This leads to the problem of **Starvation**, where a shorter process has to wait for a long time until the current longer process gets executed.This happens if shorter jobs keep coming, but this can be solved using the concept of aging.

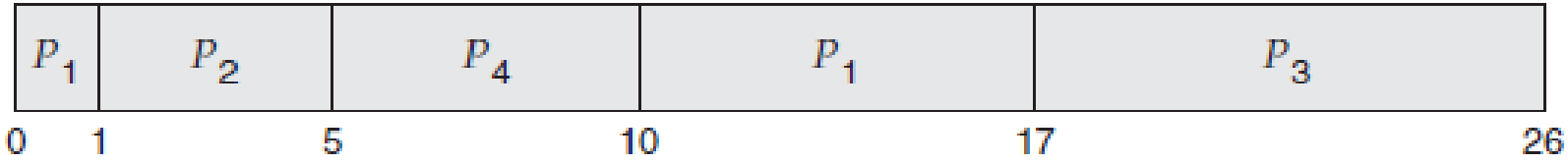
## Pre-emptive:-

- Preemptive SJFscheduling is sometimes called **shortest-remaining-time-first** scheduling.

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0 | 8 |
| $P_2$ | 1 | 4 |
| $P_3$ | 2 | 9 |
| $P_4$ | 3 | 5 |

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:




Process $P_1$ is started at time 0, since it is the only process in the queue. Process $P_2$ arrives at time 1. The remaining time for process $P_1$ (7 milliseconds) is larger than the time required by process $P_2$ (4 milliseconds), so process $P_1$ is preempted, and process $P_2$ is scheduled. The average waiting time for this

**The average waiting time for this example is [(10 − 1) + (1 − 1) + (17 − 2) + (5 − 3)]/4 = 26/4 = 6.5**

# Priority Scheduling:-

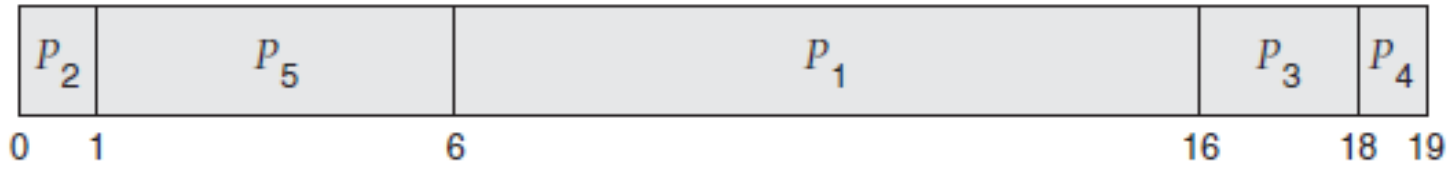
- A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order.
- An SJF algorithm is simply a priority algorithm where the priority (*p) is the* inverse of the (predicted) next CPU burst.
- The larger the CPU burst, the lower the priority, and vice versa.
- scheduling in terms of ***high priority and low priority.***
- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095.
- However, there is no general agreement on whether 0 is the highest or lowest priority

- Some systems use low numbers to represent low priority; others use low numbers for high priority.
- This difference can lead to confusion.
- In this text, we assume that low numbers represent high priority.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order $P_1, P_2, \cdots, P_5$, with the length of the CPU burst given in milliseconds:

| Process | Burst Time | Priority |
|---|---|---|
| $P_1$ | 10 | 3 |
| $P_2$ | 1 | 1 |
| $P_3$ | 2 | 4 |
| $P_4$ | 1 | 5 |
| $P_5$ | 5 | 2 |

Using priority scheduling, we would schedule these processes according to the following Gantt chart:




The average waiting time is 8.2 milliseconds

- The average waiting time is 8.2 milliseconds.
- Priorities can be defined either internally or externally.
- Internally defined priorities use some measurable quantity or quantities to compute the priority of a process.
- For example, time limits, memory requirements, the number of open files.
- External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work and other, often political, factors.
- Priority scheduling can be either preemptive or non-preemptive.

- A major problem with priority scheduling algorithms is **indefinite blocking,** or **starvation.**
- **A process that is ready to run but waiting for the CPU can** be considered blocked.
- A priority scheduling algorithm can leave some low priority processes waiting indefinitely.
- A solution to the problem of indefinite blockage of low-priority processes is **aging.**
- **Aging involves gradually increasing the priority of processes that wait** in the system for a long time.

## Round-Robin Scheduling:-

- The **round-robin (RR) scheduling algorithm is designed especially for timesharing** systems.

- It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.

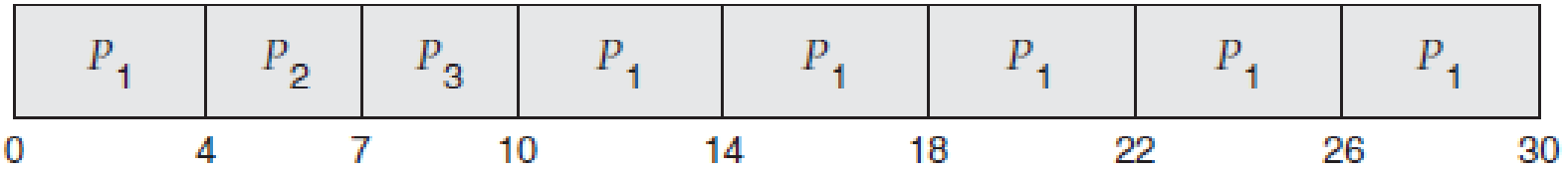
- A small unit of time, called a **time quantum or time slice, is defined.**

- **A time quantum is generally from10** to 100 milliseconds in length.The ready queue is treated as a circular queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

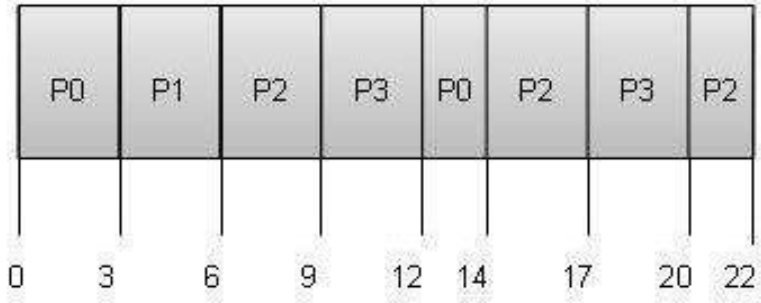
| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

If we use a time quantum of 4 milliseconds, then process $P_1$ gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process $P_2$. Process $P_2$ does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process $P_3$. Once each process has received 1 time quantum, the CPU is returned to process $P_1$ for an additional time quantum. The resulting RR schedule is as follows:




Let's calculate the average waiting time for this schedule. $P_1$ waits for 6 milliseconds (10 - 4), $P_2$ waits for 4 milliseconds, and $P_3$ waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds.

- For RR
- Waiting time = Last start time - arrival time - (preemption * quantum)




**Wait time** of each process is as follows −

| Process | Wait Time : Service Time - Arrival Time |
|---|---|
| P0 | (0 - 0) + (12 - 3) = 9 |
| P1 | (3 - 1) = 2 |
| P2 | (6 - 2) + (14 - 9) + (20 - 17) = 12 |
| P3 | (9 - 3) + (17 - 12) = 11 |

Average Wait Time: (9+2+12+11) / 4 = 8.5

| Process | Arrival Time | Execute Time | Priority | Service Time |
|---|---|---|---|---|
| P0 | 0 | 5 | 1 | 9 |
| P1 | 1 | 3 | 2 | 6 |
| P2 | 2 | 8 | 1 | 14 |
| P3 | 3 | 6 | 3 | 0 |

# Multilevel Queue Scheduling:-

- Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups.
- For example, a common division is made between **foreground (interactive) processes and background (batch) processes.**
- **These two types of processes have different** response-time requirements and so may have different scheduling needs.
- In addition, foreground processes may have priority (externally defined) over background processes.
- A **multilevel queue scheduling algorithm partitions the ready queue into** several separate queues.
- The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.

- Each queue has its own scheduling algorithm.
- For example, separate queues might be used for foreground and background processes.
- The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.
- Let's look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority.

1. **System processes**
2. **Interactive processes**
3. **Interactive editing processes**
4. **Batch processes**
5. **Student processes**

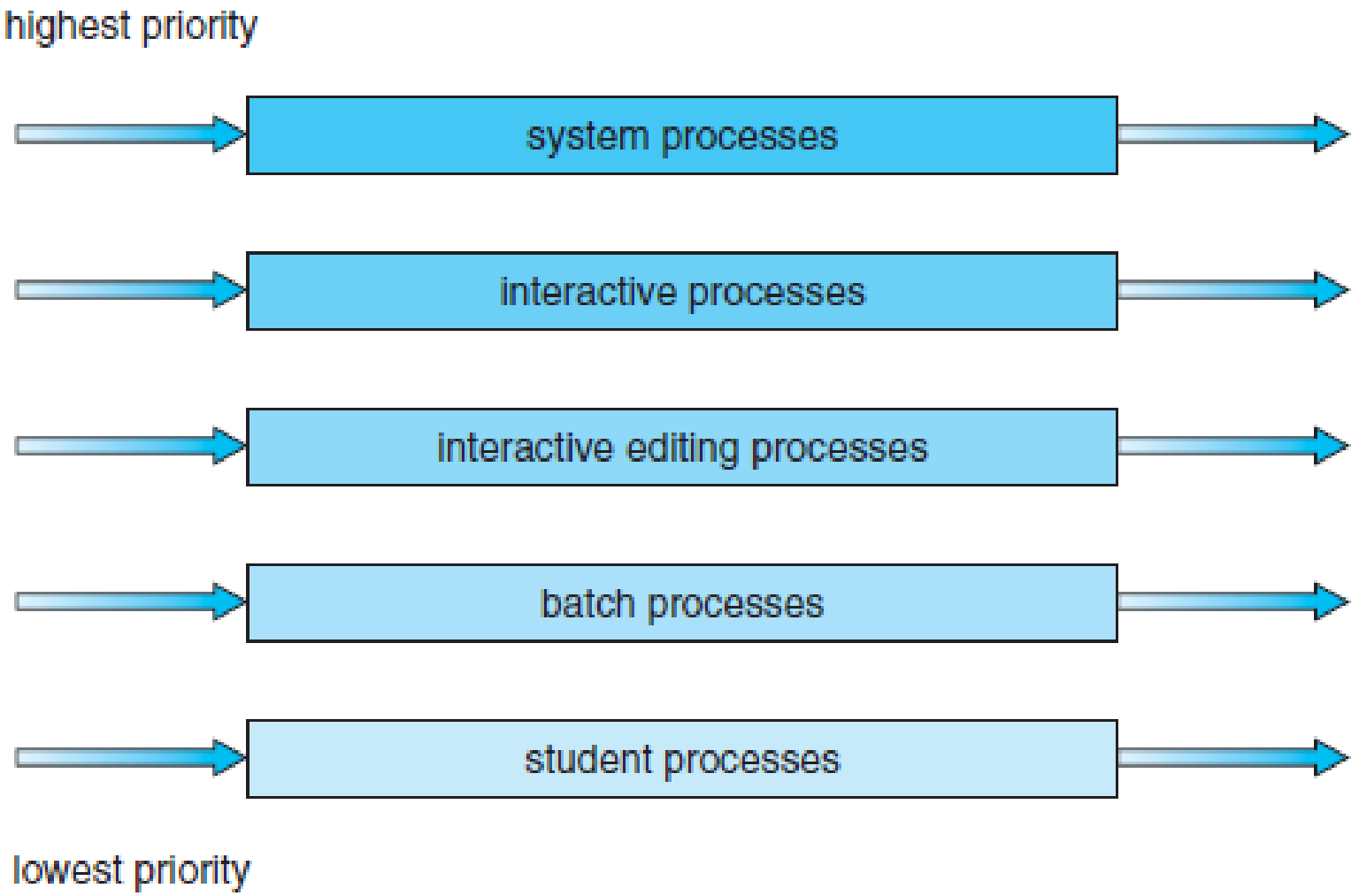



**Figure 6.6** Multilevel queue scheduling.

# Multilevel Feedback Queue Scheduling:-

- Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system.
- The **multilevel feedback queue scheduling algorithm,in contrast, allows** a process to move between queues.
- The idea is to separate processes according to the characteristics of their CPU bursts.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
- In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue

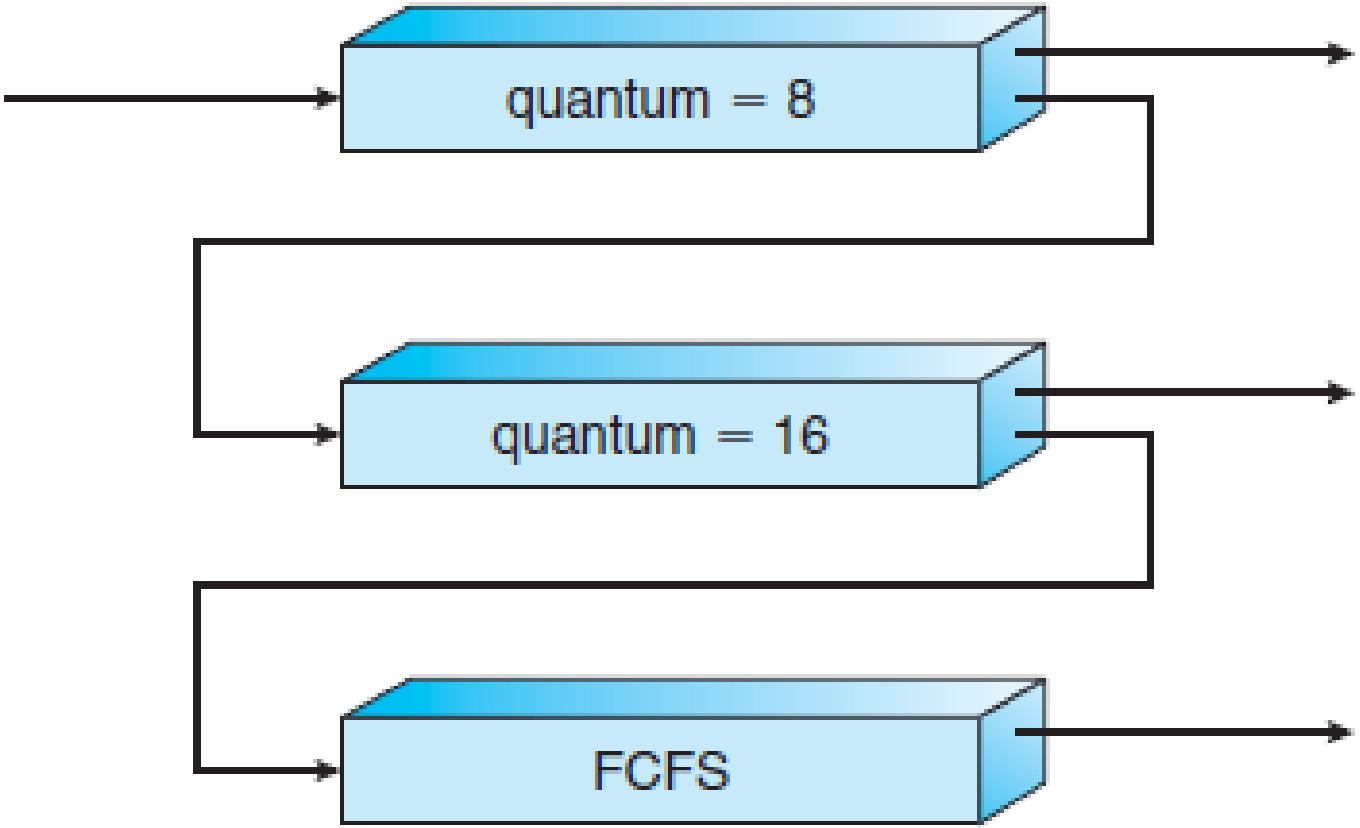



**Figure 6.7** Multilevel feedback queues.

- For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2.
- The scheduler first executes all processes in queue 0.
- Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes In queue 2 will be executed only if queues 0 and 1 are empty.
- A process entering the ready queue is put in queue 0.
- A process in queue 0 is given a time quantum of 8 milliseconds.
- If it does not finish within this time, it is moved to the tail of queue 1.
- If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2.
- Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

# Multiple-Processor Scheduling

- One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor—the master server.
- The other processors execute only user code.
- This **asymmetric multiprocessing is simple because only one processor** accesses the system data structures, reducing the need for data sharing.
- A second approach uses **symmetric multiprocessing (SMP), where each** processor is self-scheduling.
- All processes may be in a common ready queue, or each processor may have its own private queue of ready processes. Regardless,

- scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute.

- If we have multiple processors trying to access and update a common data structure, the scheduler must be programmed carefully.

- We must ensure that two separate processors do not choose to schedule the same process and that processes are not lost from the queue.

- processor affinity
- **—that is, a process** has an affinity for the processor on which it is currently running.

## Load Balancing:-

- On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor.
- Otherwise, one or more processors may sit idle while other processors have
- high workloads, along with lists of processes awaiting the CPU.
- Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system.
- It is important to note that load balancing is typically necessary only on systems where each processor has its own private queue of eligible processes to execute.
- On systems with a common run queue, load balancing is often unnecessary, because once a processor becomes idle, it immediately extracts a runnable process from the common run queue.

- There are two general approaches to load balancing: push migration and pull migration.

- With push migration, **a specific task periodically checks the** load on each processor and—if it finds an imbalance—evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors.

- Pull migration is where a scheduler finds that there are no more processes in the run queue for the processor. In this case, it raids another processor's run queue and transfers a process onto its own queue so it will have something to run

# Process Management

Mr.S. C. Sagare

# CONTENTS

- Process concept
- Process States
- Process Control Block
- Inter-process communication
- Process scheduling:-
  - Basic concepts
  - Scheduling Criteria
  - Scheduling Algorithms
  - Multiple processor scheduling
  - Real time CPU scheduling

# Process scheduling

- CPU scheduling is the basis of multi-programmed operating systems.
- By switching the CPU among processes, the operating system can make the computer more productive.
- In a single-processor system, only one process can run at a time. Others must wait until the CPU is free and can be rescheduled.
- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- A process is executed until it must wait, typically for the completion of some I/O request.
- In a simple computer system, the CPU then just sits idle. All this waiting time is wasted;

# Process scheduling

- With multiprogramming, we try to use this time productively.
- Several processes are kept in memory at one time.
- When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues.

- CPU-I/O Burst Cycle:-
  - The success of CPU scheduling depends on an observed property of processes
  - process execution consists of a **cycle of CPU execution and I/O wait.**
  - **Processes** alternate between these two states.
  - Process execution begins with a **CPU burst.**
    - **CPU burst** is when the process is being executed in the **CPU.**
  - **CPU burst** is followed by an **I/O burst, which is followed by another CPU burst, then** another I/O burst, and so on.
    - **I/O burst** is when the CPU is waiting for **I/O** for further execution.
  - Eventually, the final **CPU burst** ends with a system request to **terminate** execution (Figure 6.1).

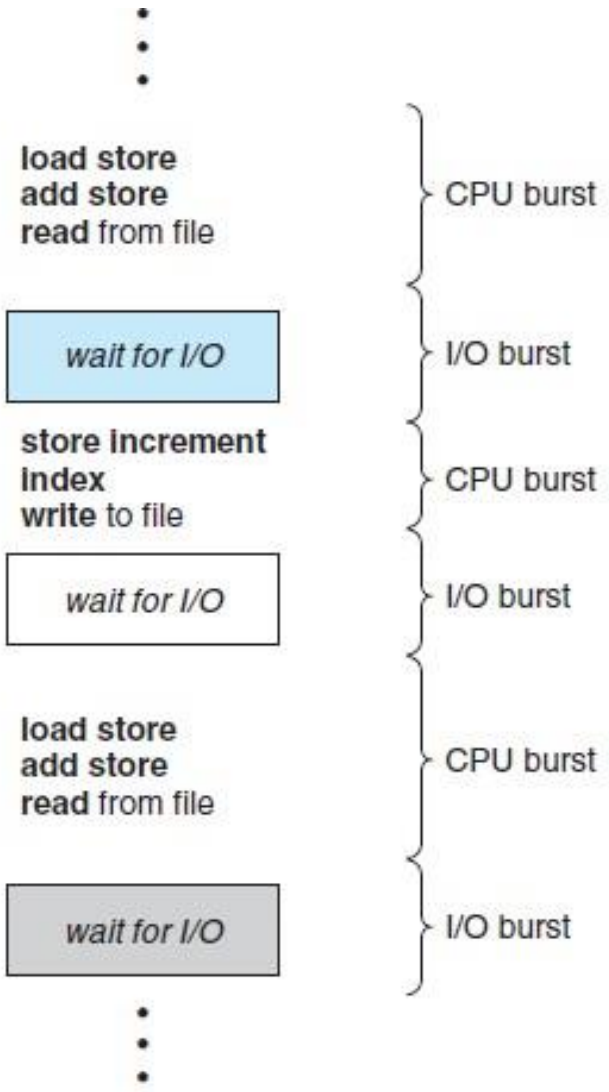



**Figure 6.1** Alternating sequence of CPU and I/O bursts.

- CPU Scheduler:-
  - Whenever the CPU becomes idle, the operating system must select one of the processes in the **ready queue** to be executed.
  - In an operating system (OS), a process scheduler performs the important activity of scheduling a process between the **ready queue** and **waiting queue** and allocating them to the CPU
  - The OS assigns priority to each process and maintains these queues.
  - **The scheduler selects a process** from the processes in memory that are ready to execute and allocates the CPU to that process.
  - Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue.
  - Depends on scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list.

- Preemptive Scheduling:-
- CPU-scheduling decisions may take place under the following four circumstances:
  - When a process switches from the running state to the waiting state.
  - When a process switches from the running state to the ready state. (for example, when an interrupt occurs).
  - When a process switches from the waiting state to the ready state.
  - When a process terminates.

- Preemptive Scheduling:-
  - The scheduling in which a running process can be interrupted if a high priority process enters the queue and is allocated to the CPU is called **preemptive scheduling**.
  - In this case, the current process switches from the running queue to ready queue, and the high priority process utilizes the CPU cycle.
  - A process P2 arrives at time 0 and allocated the CPU. Process P3 arrives at time 1, before P2 finishes execution. The time remaining for P2 is 5 milliseconds, which is larger than a time required for P3 (4 milliseconds). So CPU is allocated to process P3. Process P1 arrives at time 2. P3 continues to execute because the remaining time for P3 (3 milliseconds) is less than the time required by processes P1 (4 milliseconds) and P2 (5 milliseconds).
  - Process P0 arrives at time 3. Now P3 continues to run because the remaining time for P3 (2 milliseconds) is equal to the time required by P0 (2 milliseconds). After P3 finishes, the CPU is allocated to P0 as it has smaller burst time than other processes. Later, the CPU is allocated to P1 and then to P2.

- Non-Preemptive Scheduling:-
  - The scheduling in which a running process cannot be interrupted by any other process is called **non-preemptive scheduling**.
  - Any other process which enters the queue has to wait until the current process finishes its CPU cycle.
  - Process P2 arrives at time 0 and is allocated the CPU until it finishes execution. While P2 is executing, processes P0, P1, P3 arrive into the ready queue. But all other processes have to wait until process P2 finishes its execution. After P2 finishes its CPU cycle, based on the arrival time, process P3 is allocated the CPU. After P3 finishes, P1 executes and then P0.

## Dispatcher:-

- Another component involved in the CPU-scheduling function is the **dispatcher.**

- The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.

# Process Management

Mr. S. C. Sagare

# CONTENTS

- Process concept
- Process States
- Process Control Block
- Inter-process communication
- Process scheduling:-
  - Basic concepts
  - Scheduling Criteria
  - Scheduling Algorithms
  - Multiple processor scheduling
  - Real time CPU scheduling

# Scheduling Criteria

- Different CPU-scheduling algorithms have different properties.
- In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.
- Many criteria have been suggested for comparing CPU-scheduling algorithms.
- The criteria include… …
- **CPU utilization**. We want to keep the CPU as busy as possible.
- **Throughput**. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

- **Turnaround time.**The interval from the time of submission of a process to the time of completion is the turnaround time.

- Turnaround time = periods spent waiting to get into memory + waiting in the ready queue + executing on the CPU + and doing I/O.

- **Waiting time:-** Waiting time is the sum of the periods spent waiting in the ready queue.

- The CPU-scheduling algorithm affects only the amount of time that a process spends waiting in the ready queue

- **Response time.:-** the time from the submission of a request until the first response is produced

# Scheduling Algorithms

- CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.

## First-Come, First-Served Scheduling:

- The simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS) scheduling algorithm.**

- **With this scheme, the process that requests the CPU first is allocated the CPU first.**

- The implementation of the **FCFS** policy is easily managed with a **FIFO queue**.

- When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue.
- The running process is then removed from the queue.

- Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |


The waiting time is 0 milliseconds for process $P_1$, 24 milliseconds for process $P_2$, and 27 milliseconds for process $P_3$. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order $P_2$, $P_3$, $P_1$, however, the results will be as shown in the following Gantt chart:


The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.

# Shortest-Job-First Scheduling:

- Shortest Job First scheduling works on the process with the shortest **burst time** or **duration** first.

- When the CPU is available, it is assigned to the process that has the smallest next CPU burst.

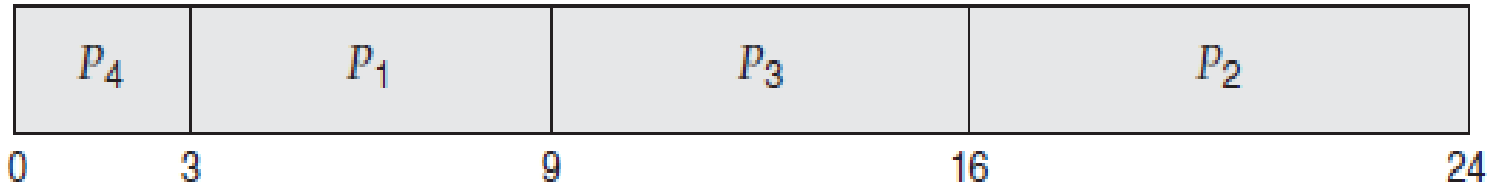
- If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

- more appropriate term for this scheduling method would be the ***shortest-next- CPU-burst algorithm, because scheduling depends on the length of the next*** CPU burst of a process.

As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

| Process | Burst Time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 8 |
| $P_3$ | 7 |
| $P_4$ | 3 |

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:




The waiting time is 3 milliseconds for process $P_1$, 16 milliseconds for process $P_2$, 9 milliseconds for process $P_3$, and 0 milliseconds for process $P_4$. Thus, the average waiting time is $(3 + 16 + 9 + 0)/4 = 7$ milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

- To successfully implement it, the burst time/duration time of the processes should be known to the processor in advance, which is practically not feasible all the time.

- This scheduling algorithm is optimal if all the processes are available at the same time. (either Arrival time is 0 for all, or Arrival time is same for all).

- **It is of two types**
  - Non Pre-emptive
  - Pre-emptive
- The choice arises when a new process arrives at the ready queue while a previous process is still executing.

## Non Pre-emptive:-

- A non-preemptive SJFalgorithm will allow the currently running process to finish its CPU burst.

- This leads to the problem of **Starvation**, where a shorter process has to wait for a long time until the current longer process gets executed. This happens if shorter jobs keep coming, but this can be solved using the concept of **aging**.

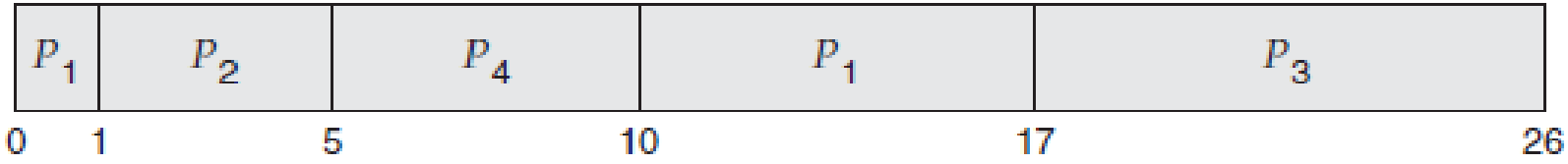
## Pre-emptive:-

- Preemptive SJFscheduling is sometimes called **shortest-remaining-time-first** scheduling.

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0 | 8 |
| $P_2$ | 1 | 4 |
| $P_3$ | 2 | 9 |
| $P_4$ | 3 | 5 |

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:




Process $P_1$ is started at time 0, since it is the only process in the queue. Process $P_2$ arrives at time 1. The remaining time for process $P_1$ (7 milliseconds) is larger than the time required by process $P_2$ (4 milliseconds), so process $P_1$ is preempted, and process $P_2$ is scheduled. The average waiting time for this

**The average waiting time for this example is [(10 − 1) + (1 − 1) + (17 − 2) + (5 − 3)]/4 = 26/4 = 6.5**

# Process Management

Mr. S. C. Sagare

# CONTENTS

- Process concept
- Process States
- Process Control Block
- Inter-process communication
- Process scheduling:-
  - Basic concepts
  - Scheduling Criteria
  - Scheduling Algorithms
  - Multiple processor scheduling
  - Real time CPU scheduling

# Priority Scheduling:

- A **priority** is associated with each process, and the CPU is allocated to the process with the highest priority.
- **Equal-priority** processes are scheduled in **FCFS** order.
- An SJF algorithm is simply a priority algorithm where the priority (*p) is the* inverse of the (predicted) next CPU burst.
- The larger the CPU burst, the lower the priority, and vice versa.
- Scheduling in terms of ***high priority and low priority.***
- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095.
- However, there is no general agreement on whether 0 is the highest or lowest priority

- Some systems use low numbers to represent low priority; others use low numbers for high priority.
- This difference can lead to confusion.
- Here, we assume that low numbers represent high priority.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order $P_1, P_2, \cdots, P_5$, with the length of the CPU burst given in milliseconds:

| Process | Burst Time | Priority |
|---|---|---|
| $P_1$ | 10 | 3 |
| $P_2$ | 1 | 1 |
| $P_3$ | 2 | 4 |
| $P_4$ | 1 | 5 |
| $P_5$ | 5 | 2 |

Using priority scheduling, we would schedule these processes according to the following Gantt chart:


- The average waiting time is 8.2 milliseconds

- Priorities can be defined either **internally or externally**.
- Internally defined priorities use some measurable quantity or quantities to compute the priority of a process.
- For example, following have been used in computing priorities.
  - **Time limits**
  - **Memory requirements**
  - **Number of open files**
  - **Ratio of average I/O burst to average CPU burst**

- External priorities are set by criteria outside the operating system, such as
  - **The importance of the process**
  - **The type and amount of funds being paid for computer use**
  - **The department sponsoring the work and**
  - **Other, often political, factors.**

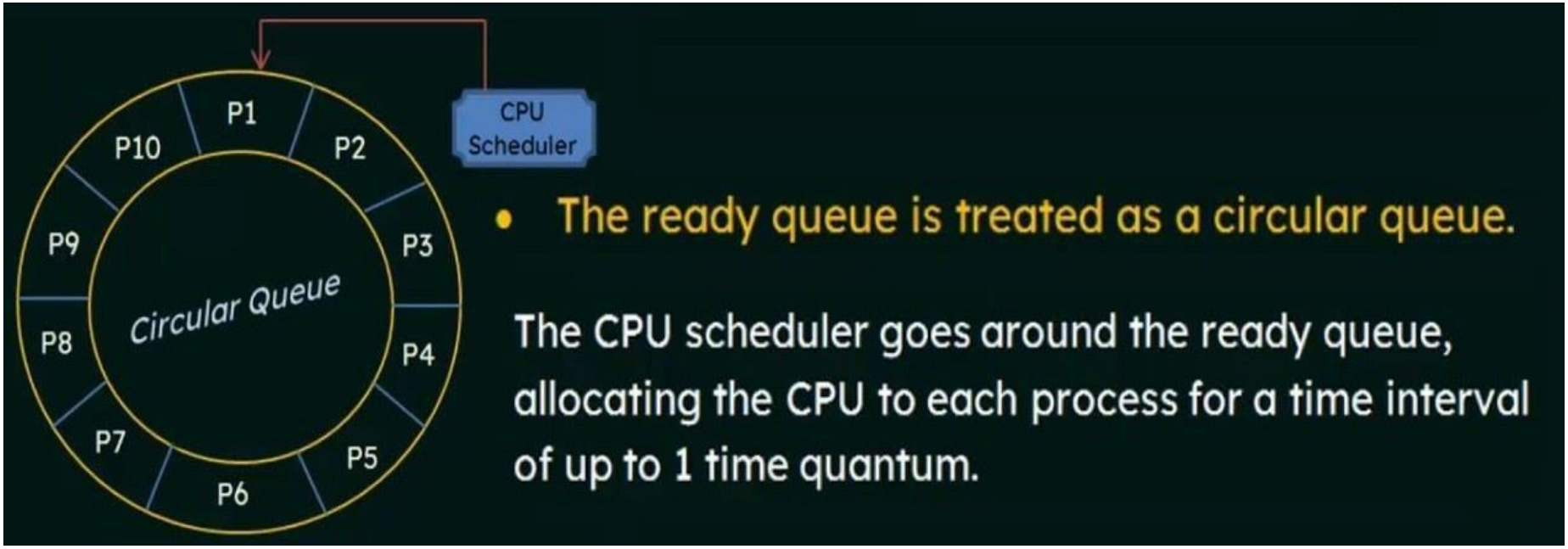
- Priority scheduling can be either preemptive or non-preemptive.
  - A **preemptive priority scheduling** algorithm will **preempt** the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
  - A **non-preemptive priority scheduling** algorithm will simply put the new process at the head of the ready queue.

- A major problem with priority scheduling algorithms is **indefinite blocking,** or **starvation.**
- **A process that is ready to run but waiting for the CPU can be considered as blocked.**
- In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.
- Generally, one of two things will happen.
  - Either the process will eventually be run when system will be lightly-loaded at some point of time,
  - Or the computer system will eventually crash and lose all unfinished low-priority processes

- A solution to the problem of indefinite blockage of low-priority processes is **aging.**
- **Aging involves 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.
  - In fact, it would take no more than 32 hours for a priority-127 process to age to a priority-0 process.

# Round-Robin Scheduling:

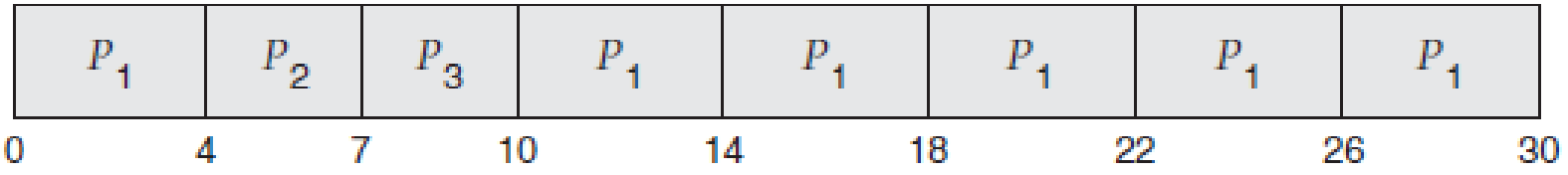
- The **round-robin (RR) scheduling algorithm is designed especially for timesharing** systems.
- It is similar to FCFS scheduling, but **preemption** is added to enable the system to switch between processes.
- A small unit of time, called a **time quantum** or **time slice,** is defined.
- A time slice is generally from **10 to 100 milliseconds** in length.



The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

If we use a time quantum of 4 milliseconds, then process $P_1$ gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process $P_2$. Process $P_2$ does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process $P_3$. Once each process has received 1 time quantum, the CPU is returned to process $P_1$ for an additional time quantum. The resulting RR schedule is as follows:




Let's calculate the average waiting time for this schedule. $P_1$ waits for 6 milliseconds (10 - 4), $P_2$ waits for 4 milliseconds, and $P_3$ waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds.

- If there are **n** processes in the ready queue and the time slice is **q**, then each process gets **1/n** of the CPU time in parts of at most **q** time units.
- Each process must wait no longer than **(n − 1) × q** time units until its next time quantum.
  - For example, with five processes and a time slice of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds
- **The performance of the RR algorithm depends heavily on the size of the time slice (time quantum)**
  - If the time slice 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.

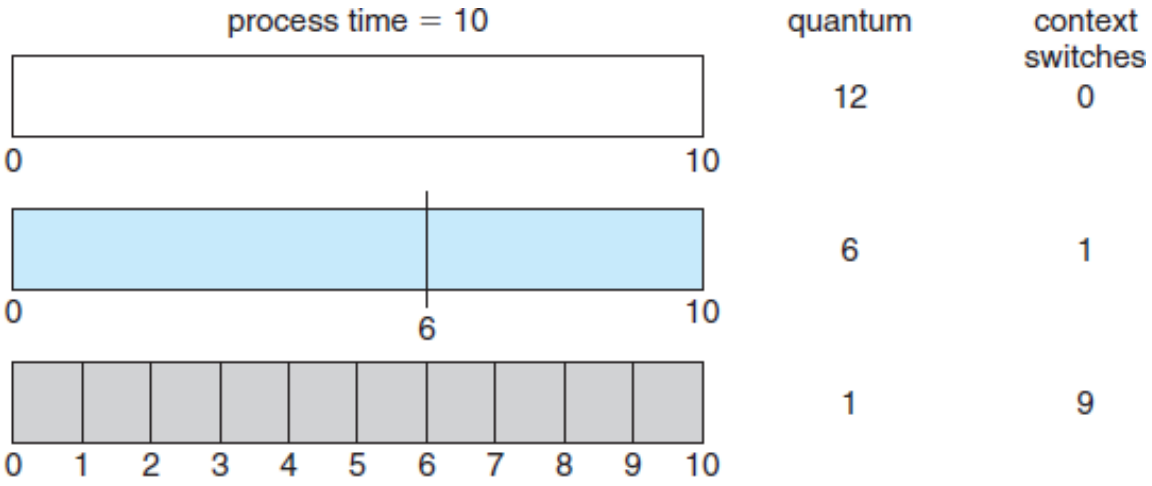
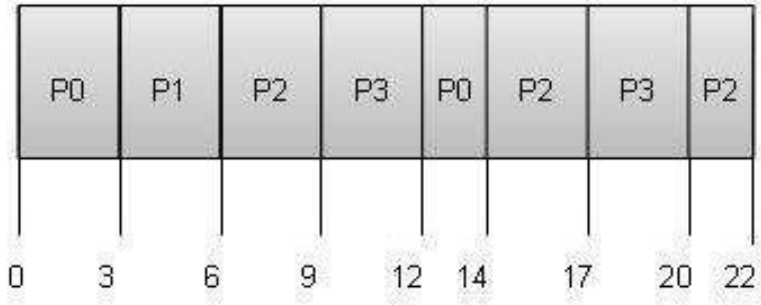



**Figure 6.4** How a smaller time quantum increases context switches.




**Wait time** of each process is as follows –

| Process | Wait Time : Service Time - Arrival Time |
|---|---|
| P0 | (0 - 0) + (12 - 3) = 9 |
| P1 | (3 - 1) = 2 |
| P2 | (6 - 2) + (14 - 9) + (20 - 17) = 12 |
| P3 | (9 - 3) + (17 - 12) = 11 |

Average Wait Time: (9+2+12+11) / 4 = 8.5

| Process | Arrival Time | Execute Time | Priority | Service Time |
|---|---|---|---|---|
| P0 | 0 | 5 | 1 | 9 |
| P1 | 1 | 3 | 2 | 6 |
| P2 | 2 | 8 | 1 | 14 |
| P3 | 3 | 6 | 3 | 0 |

# Multilevel Queue Scheduling:

- Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups.
- For example, a common division is made between **foreground (interactive) processes and background (batch) processes.**
- **These two types of processes have different** response-time requirements and so may have different scheduling needs.
- In addition, foreground processes may have priority (externally defined) over background processes.
- A **multilevel queue scheduling algorithm partitions the ready queue into** several separate queues.
- The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.

- Each queue has its own scheduling algorithm.
- For example, separate queues might be used for foreground and background processes.
- The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.
- Let's look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority.

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

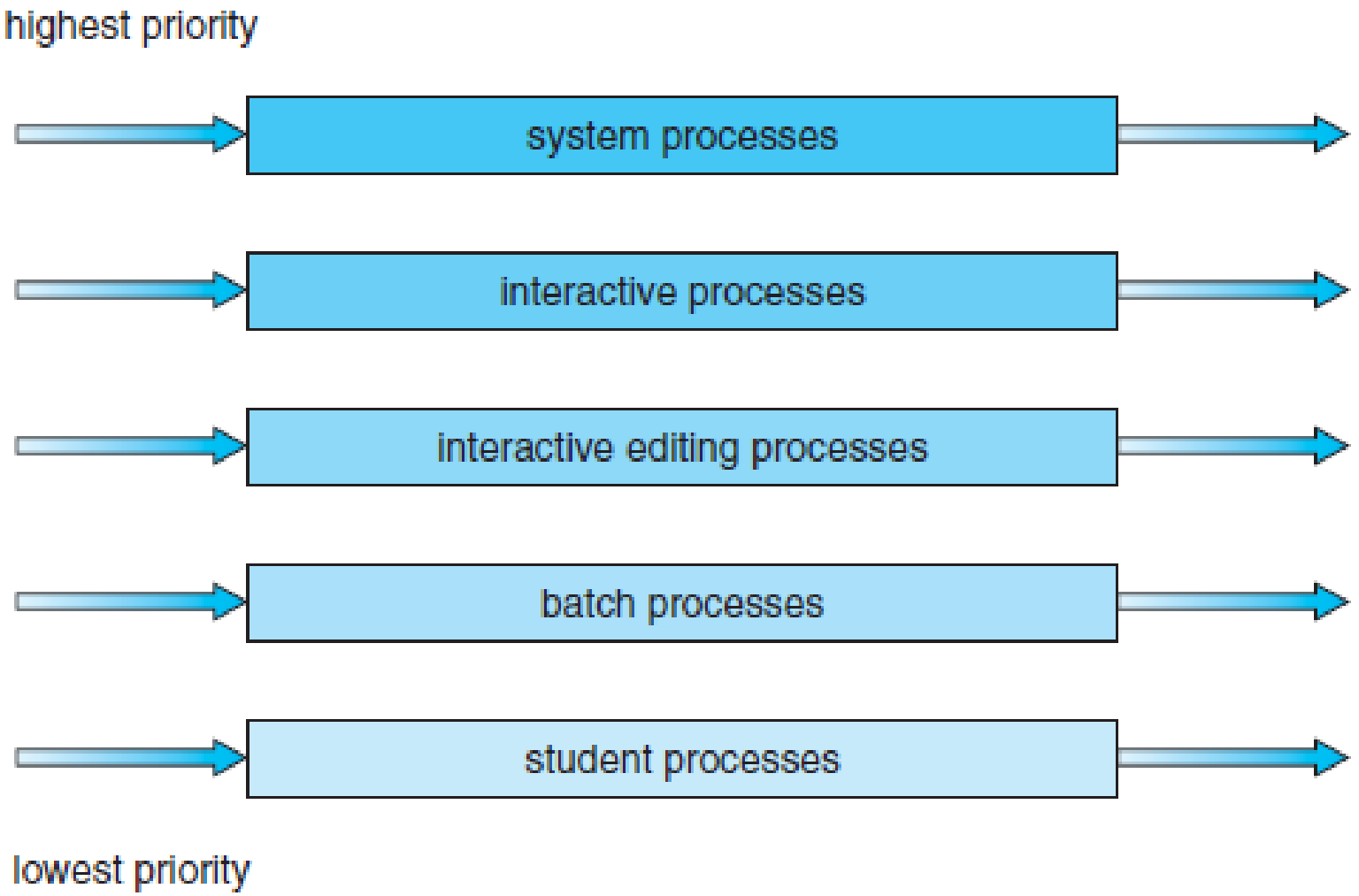



**Figure 6.6** Multilevel queue scheduling.

# Multilevel Feedback Queue Scheduling:

- Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system.
- The **multilevel feedback queue scheduling algorithm, in contrast, allows** a process to move between queues.
- The idea is to separate processes according to the characteristics of their CPU bursts.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
- In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

- For example, consider a multilevel feedback queue scheduler with **three queues**, numbered from 0 to 2.
- The scheduler first executes all processes in queue 0.
- Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes In queue 2 will be executed only if queues 0 and 1 are empty.
- A process entering the ready queue is put in queue 0.
- A process in queue 0 is given a time quantum of **8 milliseconds**.
- If it does not finish within this time, it is moved to the tail of queue 1.
- If queue 0 is empty, the process at the head of queue 1 is given a quantum of **16 milliseconds.** If it does not complete, it is preempted and is put into queue 2.
- Processes in queue 2 are run on an **FCFS basis** but are run only when queues 0 and 1 are empty.

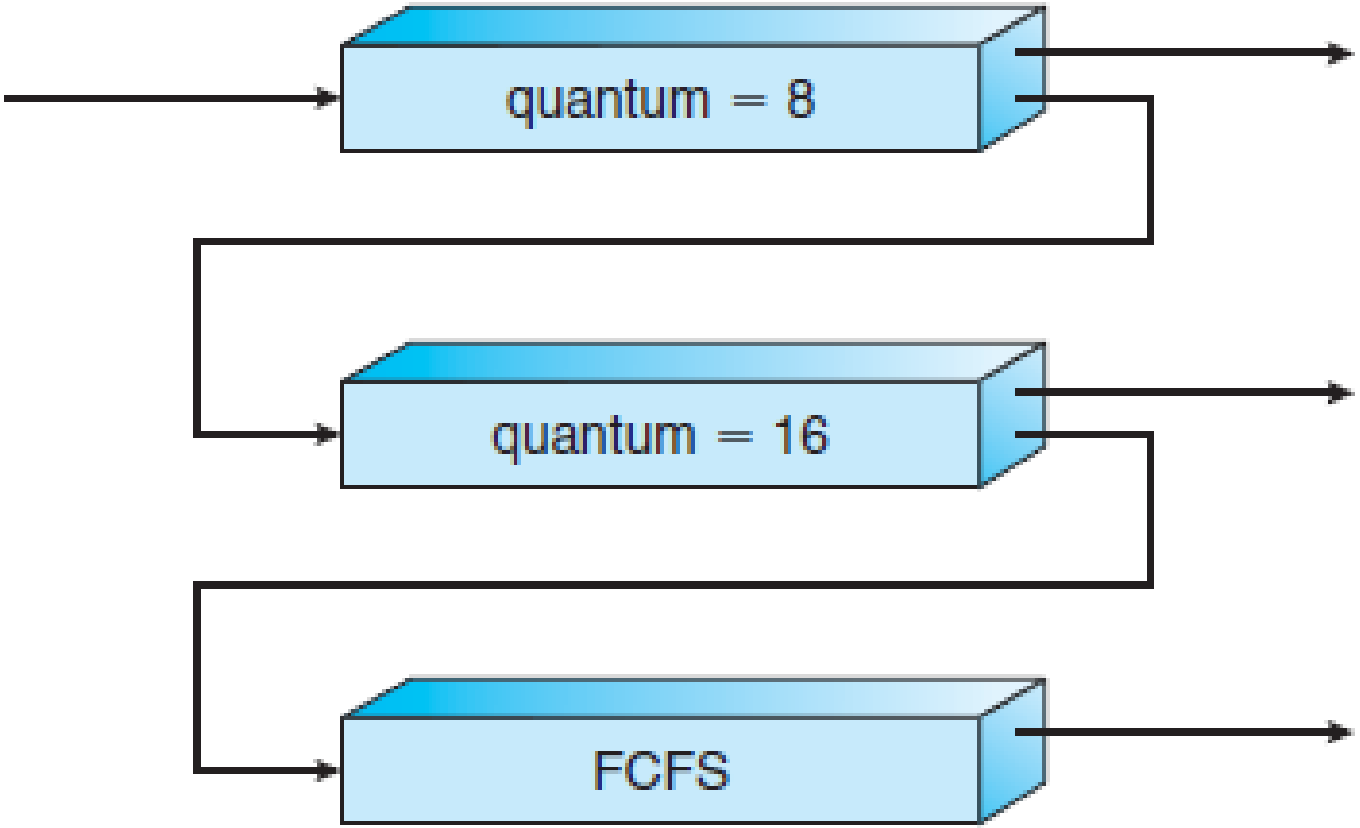



**Figure 6.7** Multilevel feedback queues.

# Multiple-Processor Scheduling

- **Asymmetric multiprocessing:**
- One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor—the **master server**.
- The other processors execute only user code.
- This **Asymmetric multiprocessing is simple because only one processor** accesses the system data structures, reducing the need for data sharing.

- **Symmetric multiprocessing:**
- A second approach uses **symmetric multiprocessing (SMP), where each** processor is self-scheduling.
- All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.
- Scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute.

- If we have multiple processors trying to access and update a common data structure, the scheduler must be programmed carefully.

- We must ensure that two separate processors do not choose to schedule the same process and that processes are not lost from the queue.

- **Processor Affinity** – Successive memory accesses by the process are often satisfied in **cache memory**.
- Now consider what happens if the process migrates to another processor.
  - The contents of cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated.
  - Because of the high cost of invalidating and repopulating caches, most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor.
  - This is known as **processor affinity**

- **Soft affinity:** When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing that it will do so—we have a situation known as Soft affinity.
  - Here, the operating system will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors.
- **Hard affinity:** Some systems provide system calls that support hard affinity, thereby allowing a process to specify a **subset of processors** on which it mayrun.

- 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**.

# Load Balancing:-

- On Symmetric Multiprocessing (SMP) systems, it is important to keep the **workload balanced** among all processors to fully utilize the benefits of having more than one processor.
- Otherwise, one or more processors may sit idle while other processors have high workloads, along with lists of processes awaiting the CPU.
- **Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system.**
- It is important to note that load balancing is typically necessary only on systems where **each processor has its own private queue of eligible processes to execute**.
- **On systems with a common run queue, load balancing is often unnecessary**, because once a processor becomes idle, it immediately extracts a runnable process from the common run queue.

- There are two general approaches to load balancing:
  - **Push migration**
  - **Pull migration.**

- **Push migration**: In this, **a specific task periodically checks the** load on each processor and—if it finds an imbalance—evenly distributes the load by **moving (or pushing)** processes from overloaded to idle or less-busy processors.

- **Pull migration:**
- Pull migration occurs when an idle processor pulls a waiting task from a busy processor.
- Here, If a scheduler finds that there are no more processes in the run queue for the processor. In this case, it **raids** another processor's run queue and transfers a process onto its own queue so it will have something to run

- **Load balancing often counteracts the benefits of processor affinity.**
  - That is, the benefit of keeping a process running on the same processor is that the process can take advantage of its data being in that processor's cache memory.
  - Either pulling or pushing a process from one processor to another removes this benefit

# Real-Time CPU Scheduling

- CPU scheduling for real-time operating systems involves special issues.
- We can distinguish between soft real-time systems and hard real-time systems.
- **Soft real-time systems** provide no guarantee as to when a critical real-time process will be scheduled.
- They guarantee only that the process will be given preference over noncritical processes.
- **Hard real-time systems** have stricter requirements. A task must be serviced by its deadline; service after the deadline has expired is the same as no service at all

# Real-Time CPU Scheduling

- **1. Minimizing Latency:**
- The system is typically waiting for an event in real time to occur.
- Events may arise either in software—as when a timer expires—or in hardware—as when a remote-controlled vehicle detects that it is approaching an obstruction
- When an event occurs, the system must respond to and service it as quickly as possible.
- **Event latency:** The amount of time that elapses from when an event occurs to when it is serviced.

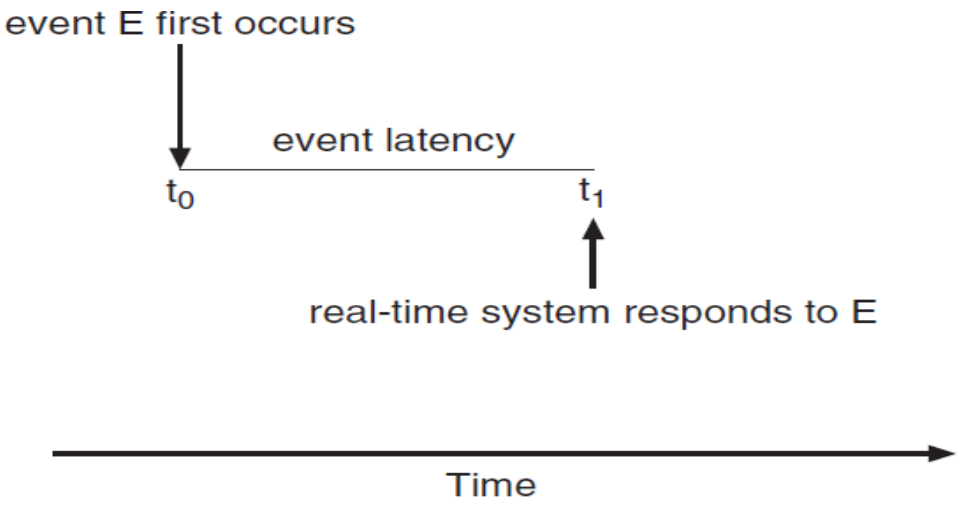



**Figure 6.12** Event latency.

# Real-Time CPU Scheduling

- Two types of latencies affect the performance of real-time systems: **1. Interrupt Latency 2. Dispatch latency**
- **1. Interrupt latency** refers to the period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt

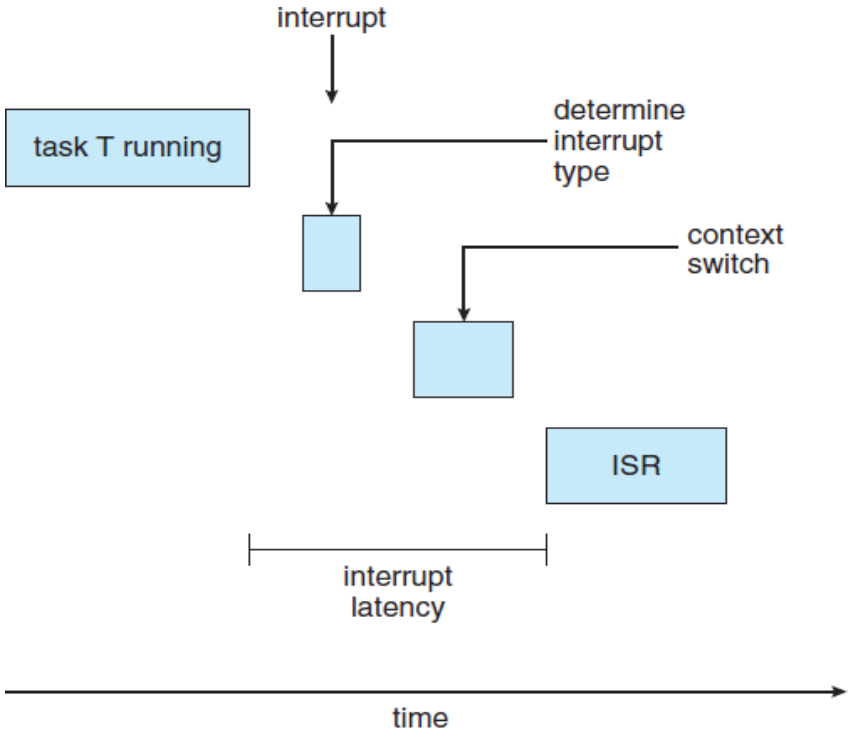



**Figure 6.13** Interrupt latency.

# Real-Time CPU Scheduling

- **2. Dispatch latency**
- The amount of time required for the scheduling dispatcher to stop one process and start another is known as **dispatch latency**.
- The most effective technique for keeping dispatch latency low is to provide preemptive kernels.
- The **conflict phase** of dispatch latency has two components:
  - **1.** Preemption of any process running in the kernel
  - **2.** Release by low-priority processes of resources needed by a high-priority process

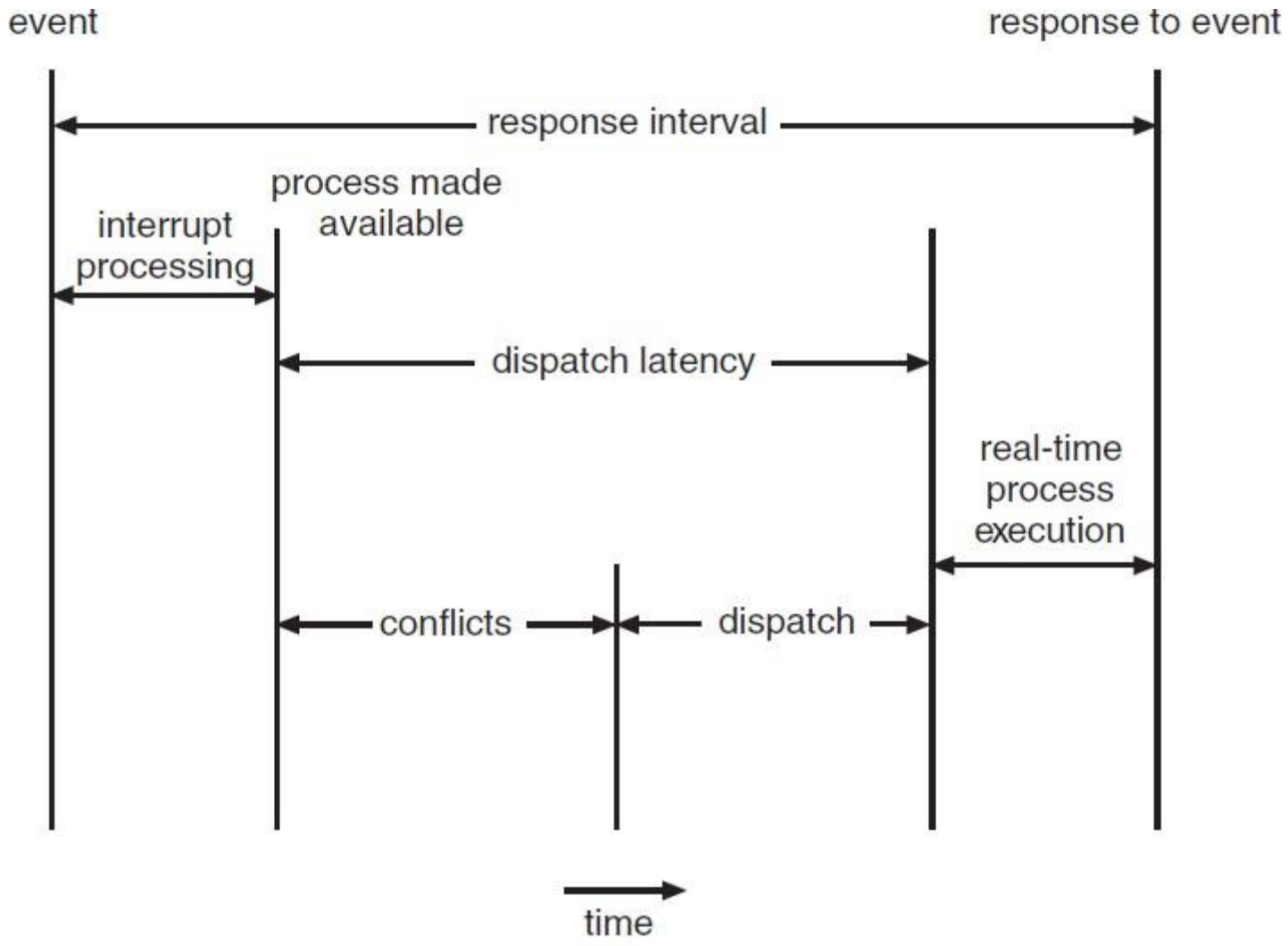



**Figure 6.14** Dispatch latency.

# Real-Time CPU Scheduling

- **Priority-Based Scheduling for Real-Time Systems:**
- Preemptive, priority-based scheduling algorithms presents examples of the soft real-time scheduling features of the Linux,Windows, and Solaris operating systems.
- Each of these systems assigns real-time processes the highest scheduling priority.
- But providing a preemptive, priority-based scheduler only guarantees **soft real-time** functionality
- **Hard real-time** systems must further guarantee that real-time tasks will be serviced in accord with their **deadline requirements**, and making such guarantees requires additional scheduling features.

# Real-Time CPU Scheduling

- First, the processes are considered **periodic**. That is, they require the CPU at constant intervals (periods).
- Once a periodic process has acquired the CPU, it has a fixed processing time *t*, a deadline *d* by which it must be serviced by the CPU, and a period *p*.
- The relationship of the processing time, the deadline, and the period can be expressed as $0 \leq t \leq d \leq p$.
- The **rate** of a periodic task is $1/p$.
- Schedulers can take advantage of these characteristics and assign priorities according to a process's deadline or rate requirements.
- **Admission-control** algorithm 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.

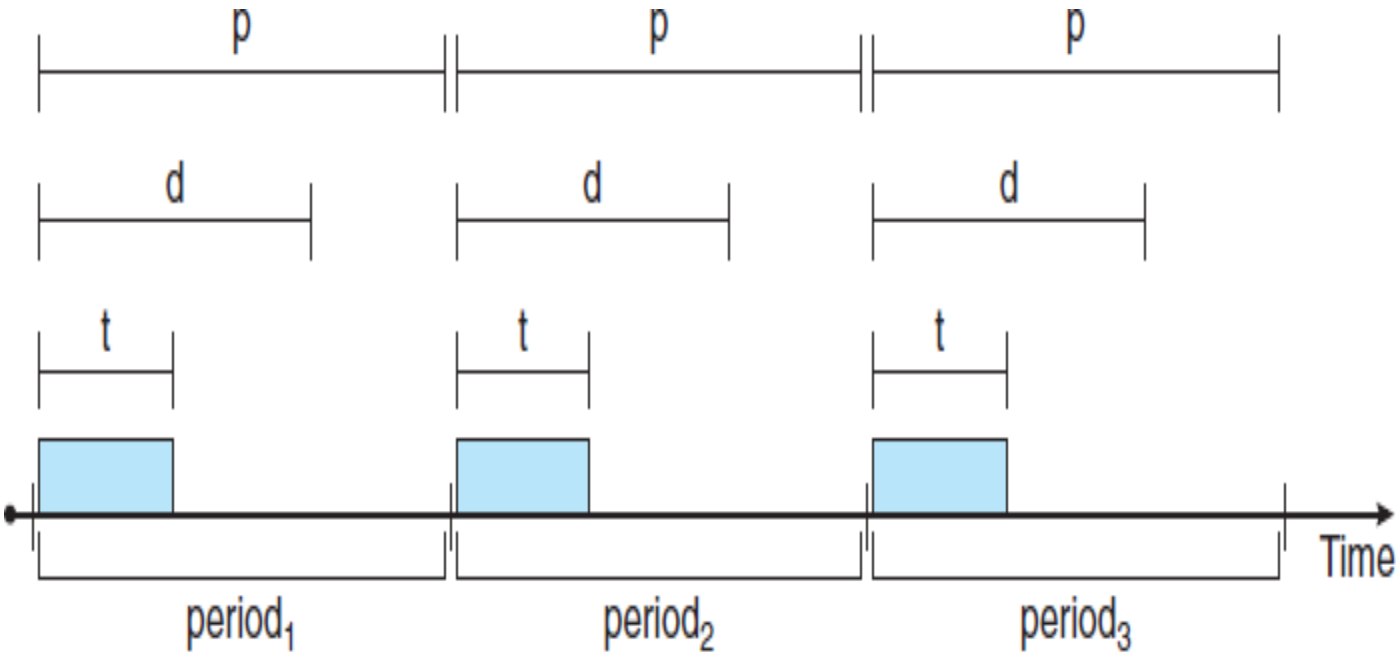



**Figure 6.15** Periodic task.

# Real-Time CPU Scheduling

- **Rate-Monotonic Scheduling:**
- The **rate-monotonic** scheduling algorithm schedules periodic tasks using a static priority policy with preemption.
- If a lower-priority process is running and a higher-priority process becomes available to run, it will **preempt** the lower-priority process.
- Upon entering the system, each periodic task is assigned a priority inversely based on its period.
  - **The shorter the period, the higher the priority; the longer the period, the lower the priority.**
  - The rationale behind this policy is to assign a higher priority to tasks that require the CPU more often.
- Furthermore, rate-monotonic scheduling assumes that the processing time of a periodic process is the **same for each CPU burst**.
  - That is, every time a process acquires the CPU, the duration of its CPU burst is the same.

- Example: We have two processes, $P1$ and $P2$. The periods for $P1$ and $P2$ are 50 and 100, respectively—that is, $p1 = 50$ and $p2 = 100$. The processing times are $t1 = 20$ for $P1$ and $t2 = 35$ for $P2$.
- The deadline for each process requires that it complete its CPU burst by the start of its next period.
- Suppose we assign $P2$ a higher priority than $P1$. The execution of $P1$ and $P2$ in this situation is shown in Figure

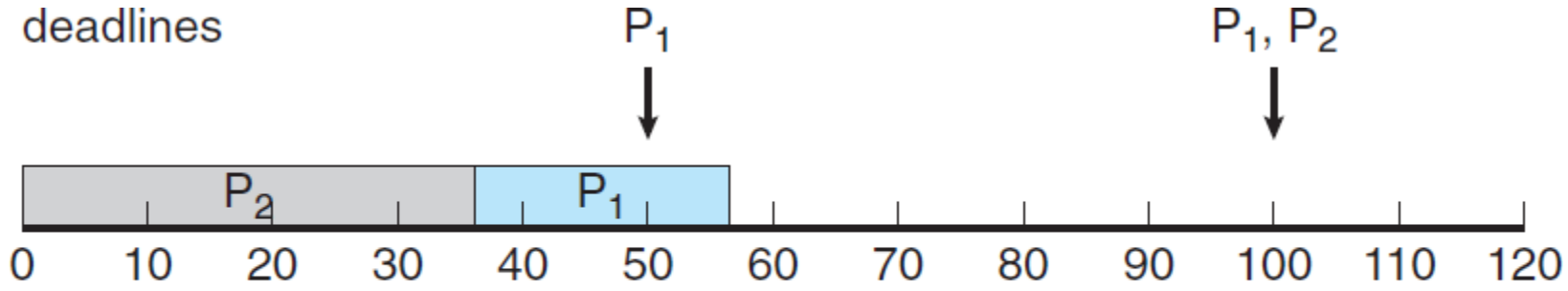



**Figure 6.16** Scheduling of tasks when $P_2$ has a higher priority than $P_1$.

- As we can see, $P2$ starts execution first and completes at time 35. At this point, $P1$ starts; it completes its CPU burst at time 55.
- However, the first deadline for $P1$ was at time 50, so the scheduler has caused $P1$ to miss its deadline.

- If we use Rate-monotonic scheduling, in which we assign P1 a higher priority than P2 because the period of P1 is shorter than that of P2. The execution of these processes in this situation is shown in Figure

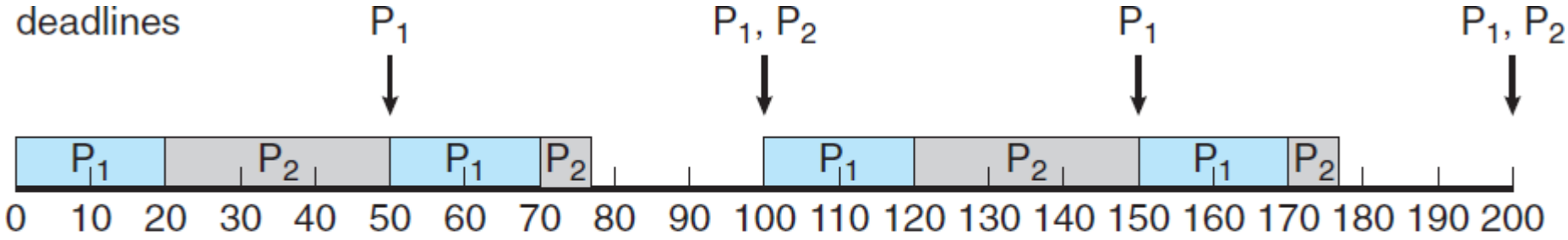



**Figure 6.17** Rate-monotonic scheduling.

- P1 starts first and completes its CPU burst at time 20, thereby meeting its first deadline.
- P2 starts running at this point and runs until time 50. At this time, it is preempted by P1,
- Although it still has 5 milliseconds remaining in its CPU burst. P1 completes its CPU burst at time 70, at which point the scheduler resumes *P2*.
- *P2* completes its CPU burst at time 75, also meeting its first deadline.
- The system is idle until time 100, when *P1* is scheduled again.
- Rate-monotonic scheduling is considered **optimal** in that if a set of processes cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that assigns static priorities.

# Real-Time CPU Scheduling

- **Earliest-Deadline-First Scheduling (EDF):**
- **EDF** scheduling dynamically assigns priorities according to deadline.
- **The earlier the deadline, the higher the priority; the later the deadline, the lower the priority.**
- Under the EDF policy, when a process becomes runnable, it must announce its deadline requirements to the system.
- Priorities may have to be adjusted to reflect the deadline of the newly runnable process.
- Note how this differs from rate-monotonic scheduling, where priorities are fixed.
- **Example:**
  - Recall that $P1$ has values of $p1 = 50$ and $t1 = 25$ and that $P2$ has values of $p2 = 80$ and $t2 = 35$.
  - The EDF scheduling of these processes is shown in Figure

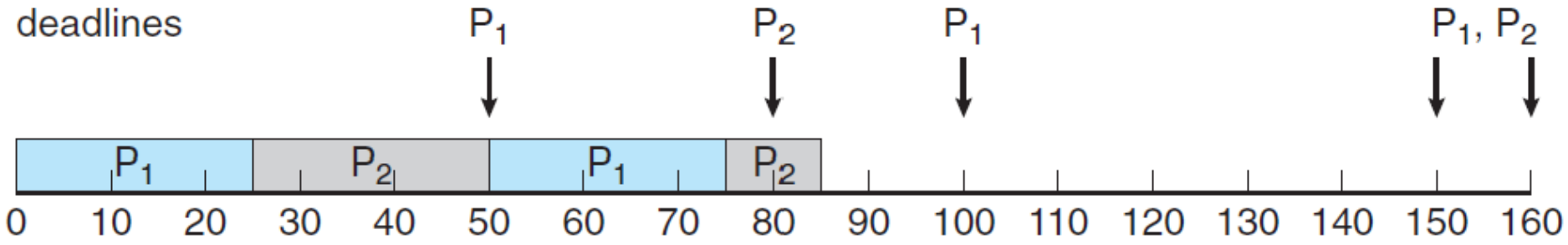



**Figure 6.18** Missing deadlines with rate-monotonic scheduling.

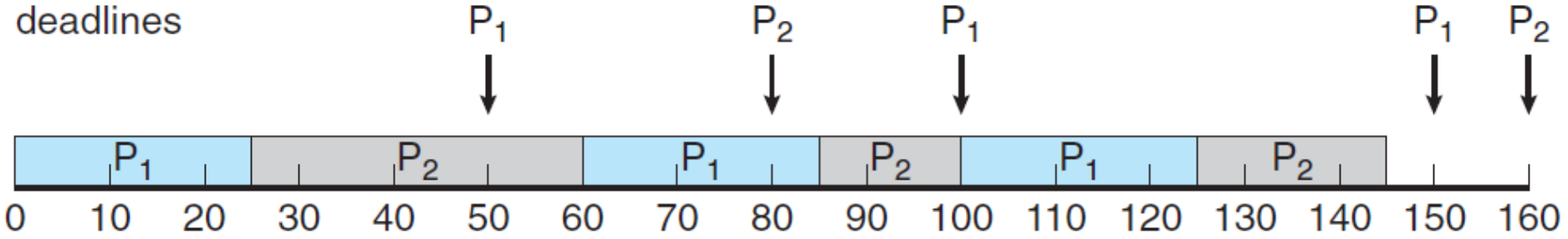



**Figure 6.19** Earliest-deadline-first scheduling.

- Unlike the rate-monotonic algorithm, EDF scheduling does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst.
- The only requirement is that a process announce its deadline to the scheduler when it becomes runnable.