Operating systems

4th class

م. منار عبدالكريم زيدان العباجي   (نظري + عملي)

م. كنار محمد سامي (عملي)

What is an Operating System?

An operating system is software that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware.

Operating system goals:

- Execute user programs and make solving user problems easier.
- Make the computer system (S.W) convenient to use.
- Use the computer hardware in an efficient manner.

A fundamental responsibility of an operating system is to allocate the resources such as CPU, memory, and I/O devices, as well as storage to programs.

Computer System Components

A computer system can be divided roughly into four components: the hardware, the operating system, the application programs, and the user (Figure 1). The operating system controls the hardware and coordinates its use among the various application programs for the various users.

1. Hardware – provides basic computing resources (CPU, memory, I/O devices).
2. Operating system – controls and coordinates the use of the hardware among the various application programs for the various users.
3. Applications programs – define the ways in which the system resources are used to solve the computing problems of the users (compilers, database systems, video games, business programs).
4. Users (people, machines, other computers).

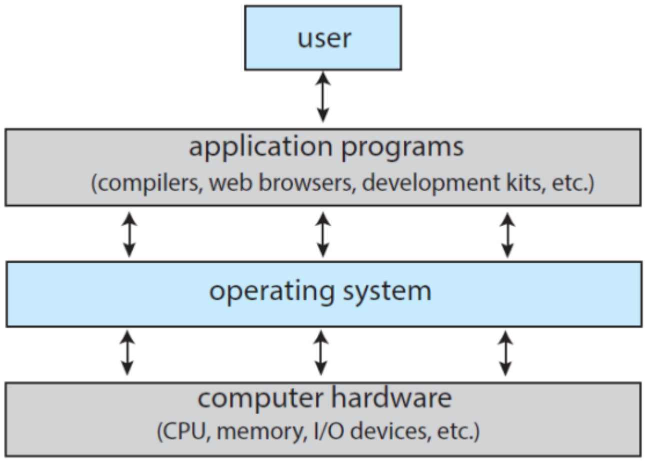

user
application programs
(compilers, web browsers, development kits, etc.)
operating system
computer hardware
(CPU, memory, I/O devices, etc.)

Figure 1: Abstract view of the components of a computer system

Operating System Definitions

1. **O.S is a Resource allocator – manages and allocates resources.**

an operating system can be viewed as a resource allocator. A computer system has many resources that may be required to solve a problem: CPU time, memory space, storage space, I/O devices, and so on. The operating system acts as the manager of these resources by allocating them to specific programs and users so that it can operate the computer system efficiently and fairly.

2. **O.S is a Control program – controls the execution of user programs and operations of I/O devices .**

an operating system can be viewed as control program that manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

3. **Kernel – the one program running at all times (all else being application programs).**

For a computer to start running it needs to have an initial or bootstrap program to run. A bootstrap program, tends to be simple. Typically, it is stored within the computer hardware in firmware. It initializes all aspects of the system, from CPU to registers to device controllers to memory contents. The bootstrap program must know where to locate the operating-system kernel and how to load it into memory.

Once the kernel is loaded and executing, it can start providing services to the system and its users. Once this phase is complete, the system is fully booted, and the system waits for some event to occur.

Functions of Operating System

The functions can be summarized as follows (will be explained later in more details):

1- Management of computer resources (processors, memory, disks, I/O devices, programs, etc.).
2- Scheduling resources among users (time-sharing).
3- Protection of programs being executed in memory from one another.
4- Providing a proper user interface e.g Graphics User Interface (GUI).
5- File management.
6- Network communication.
7- Many others.

Types of Operating Systems

Simple Batch Systems
Multiprogramming Batch Systems
Time-Sharing Systems
Parallel Systems
Distributed Systems
Real-Time Systems

Simple Batch Systems

similar types of jobs were batched together and executed in time one by one sequentially. People were used to having a single computer, which was called a mainframe. each user wrote their work on <u>punch cards or tapes</u>. A <u>special program monitor</u> manages the execution of each programming batch. Batch: set of jobs/ processes with similar needs and requirements.

Disadvantage

- Priority can not be set for the jobs (All the jobs of a batch are executed sequentially)
- There is <u>no direct interaction</u> between users and the computer.
- CPU is often idle, because the speeds of the mechanical I/O devices are slower than CPU.

- It is difficult to provide the desired priority.

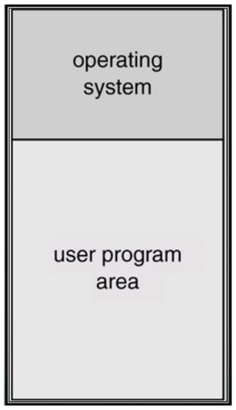


Figure 2:Memory Layout for a Simple Batch System

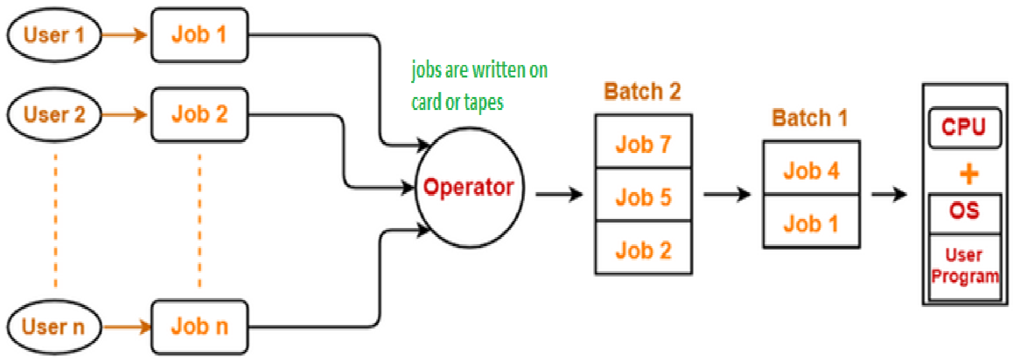


Figure 3: Simple Batch System

Multiprogramming Batch Systems

An operating system can execute multiple programs on a single processor machine. In a Multiprogramming system, a program in execution is termed a job (process). The jobs are executed one by one (Sequential). When a process does I/O, the CPU can start executing another process. Multiprogramming is the process of switching the processor between several programs; then, the CPU will never be idle (increases CPU utilization).

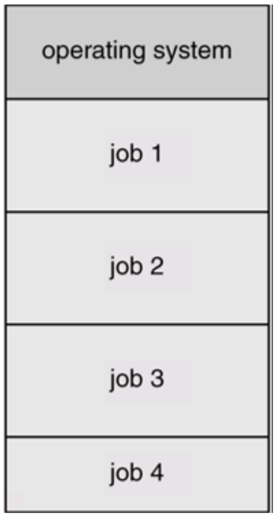


Figure 4: Multiprogramming system with three jobs in memory

Time Sharing System

This type of OS provides online communication between the user and the system. The user gives instructions to the OS directly (usually from a terminal) and receives an immediate response. Therefore, it is sometimes called an <u>interactive system</u> or a <u>multitasking system.</u>

The CPU executes multiple jobs by switching between them, but the switches occur frequently. Each job is given a time quantum to execute, and the CPU switches to the following job depending on the time interval. If n users are present, each user can get a time quantum. When the user submits the command, the response time is usually a few seconds. The operating system uses CPU scheduling and multiprogramming to give each user a small amount of time.

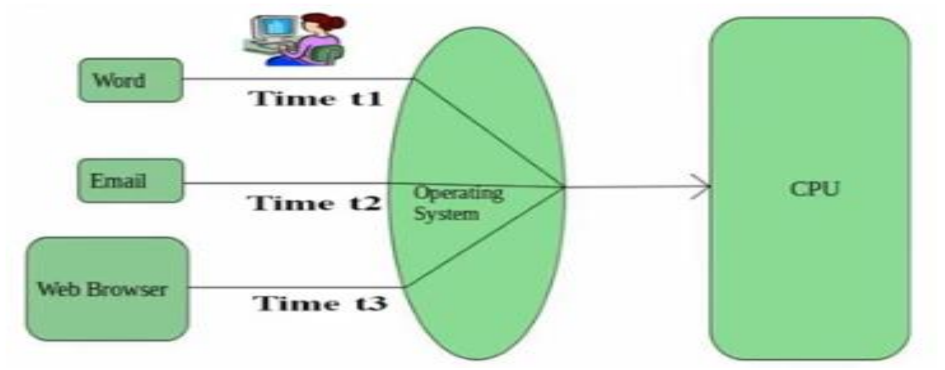


Figure 5: Time-Sharing System

Advantage:
- reduce the CPU idle time.
- Provide a quick response.

Disadvantage:
- more complex.

Parallel Systems

a. Most systems previously were simple processor systems that have one main CPU.

b. There is a trend to have multiprocessor systems, where such systems have more than one processor in close communication, sharing the computer Bus, the clock, and some memory and peripheral devices. This system called: *Tightly coupled system*.

c. Parallel operating systems break large tasks into smaller pieces that are done simultaneously in different places and by different mechanisms.

The advantages of this type of system are:
- Increase throughput (speed up).
- save money.
- Increase reliability.

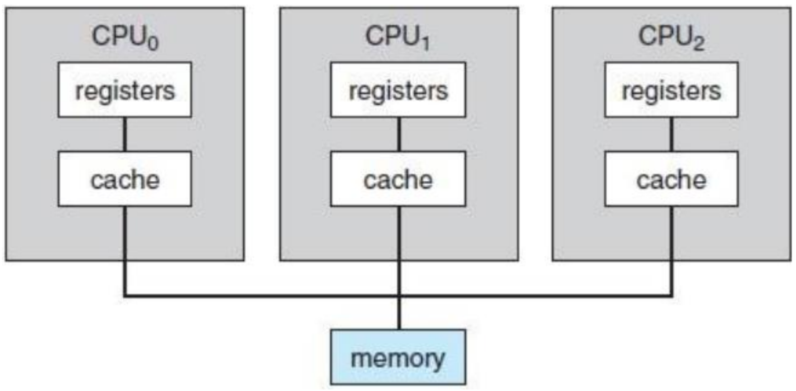


Figure 6: Parallel System Layout

There are two types of processors in multiprocessors systems:

1. Asymmetric multiprocessor:

    - Each processor is assigned a specific task; master processor schedules and allocated work to slave processors.

2. Symmetric multiprocessor:

    - Each processor runs an identical copy of the operating system.
    - Many processes can run at once without performance degradation.

Distributed Systems

A distributed operating system is an operating system that runs on several machines whose purpose is to provide a useful set of services. It distributes the computation among them.

It's called a “loosely coupled system” because each processor has its own local memory. Processors communicate with one another through various communications lines, such as high-speed buses or telephone lines.

Advantages of distributed systems.

- ➢ Resources Sharing
- ➢ Computation speed up
- ➢ Reliability
- ➢ Communications

distributed system requirements:

- Requires networking infrastructure.
- Local area networks (LAN) or Wide area networks (WAN)
- May be either client-server or peer-to-peer systems.

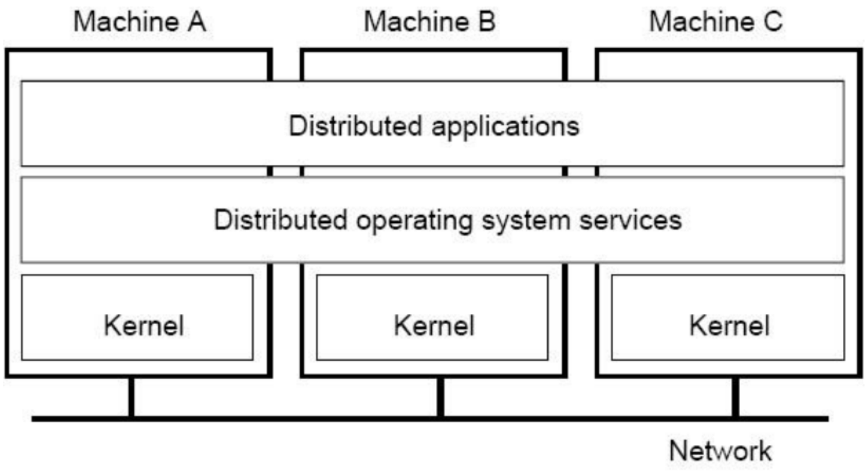


Figure 7: Distributed operating system

Real-Time System

Real-time systems are used when there are rigid time requirements on a processor's operation or data flow. A real-time operating system must have well-defined, fixed time constraints; otherwise, it will fail. Examples include medical imaging systems, industrial control systems, weapon systems, robots, and air traffic control systems.

There are two types of real-time operating systems:

a. Hard real-time systems:
   guarantee that critical tasks are completed on time. In hard real-time systems, secondary storage is limited or missing. and data stored in short-term memory, or read-only memory (ROM).

b. Soft real-time systems:
   A critical real-time task gets priority over other tasks and retains the priority until it is completed. Limited utility in industrial control of robotics. Useful in applications (multimedia).

Operating System Components

We can create a system as large and complex as an operating system only by partitioning it into smaller pieces. Each piece should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions. Following are some of the important functions of an operating system:-

- Process Management
- Main Memory Management
- File Management
- I/O System Management
- Secondary Storage Management (Disk Management)
- Networking (Distributed Systems)
- Protection System
- Command-Interpreter System

Process Management

A *process* is a program in execution. A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task.

The operating system is responsible for the following activities concerning process management:

1. Process creation and deletion.
2. process suspension and resumption.
3. Provision of mechanisms for process synchronization
4. Provision of mechanisms for process communication
5. Provision of mechanisms for deadlock handling.

Main Memory Management

Memory is a large array of words or bytes, each with its own address. It is a repository of quickly accessible data shared by the CPU and I/O devices. Main memory is a volatile storage device. It loses its contents in the case of system failure.

The operating system is responsible for the following activities in connections with memory management:

1. Keep track of which parts of memory are currently being used and by whom.
2. Decide which processes to load when memory space becomes available.
3. Allocate and deallocate memory space as needed.

File Management

Files represent programs and data. The operating system is responsible for the following activities in connections with file management:

1. File creation and deletion.
2. Directory creation and deletion.
3. Support of primitives for manipulating files and directories.
4. Mapping files onto secondary storage.
5. File backup on stable (nonvolatile) storage media.

I/O System Management

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. The I/O subsystem consists of:

1. A memory-management component that includes buffering, caching, and spooling
2. A general device-driver interface
3. Drivers for specific hardware devices

Secondary Storage Management (disk management)

Since main memory (*primary storage*) is volatile and too small to accommodate all data and programs permanently, the computer system must provide *secondary storage* to back up main memory.

Most modern computer systems use disks as the principle on-line storage medium, for both programs and data. The operating system is responsible for the following activities in connection with disk management:

1. Free space management
2. Storage allocation
3. Disk scheduling

Networking (Distributed Systems)

A *distributed* system is a collection processors that do not share memory or a clock. Each processor has its own local memory. The processors in the system are connected through a communication network. Communication takes place using a *protocol.*
A distributed system provides user access to various system resources, Access to a shared resource allows:

1. Computation speed-up
2. Increased data availability
3. Enhanced reliability

Protection System

If a computer system has multiple users and allows the concurrent execution of multiple processes, then the various processes must be protected from one another's activities.

Protection refers to a mechanism for controlling access by programs, processes, or users to both system and user resources. The protection mechanism must:

1. distinguish between authorized and unauthorized usage.
2. specify the controls to be imposed.
3. provide a means of enforcement.

Command-Interpreter System

One of the most important system programs is the command interpreter, which is the interface between the user and the operating system. Some operating systems include the command interpreter in the kernel. Other operating systems, such as MS-DOS and UNIX, treat the command interpreter as a special program that runs when a job is initiated or when a user first logs on (on time-sharing systems).

Many commands are given to the operating system by control statements which deal with:

1. process creation and management
2. I/O handling

3. secondary-storage management
4. main-memory management
5. file-system access
6. protection
7. networking

Operating System Services

An operating system provides an environment for program execution and makes certain services available to programs and their users.

One set of operating system services provides functions that are helpful to the user:

1- User interface

Almost all operating systems have a user interface (UI). This interface can take several forms. Most commonly, a graphical user interface (GUI) is used. Here, the interface is a window system with a mouse that serves as a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Mobile systems such as phones and tablets provide a touch-screen interface, enabling users to slide their fingers across the screen or press buttons on the screen to select choices. Another option is a command-line interface (CLI), which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options). Some systems provide two or all three of these variations.

2- Program execution

The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).

3- I/O operations

A running program may require I/O, which may involve a file or an I/O device. For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.

4- File-system manipulation

The file system is of particular interest. Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information

5- Communications

exchange of information between processes executing either on the same computer or on different systems tied together by a network. Communications may be implemented via *shared memory*, in which two or more processes read and write to a shared section of memory, or *message passing*, in which packets of information in predefined formats are moved between processes by the operating system.

6- Error detection

The operating system needs to constantly detect and correct errors. Errors may occur in the CPU and memory hardware, in I/O devices, and in the user program. For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

Another set of operating-system functions exists not to help the user but rather to ensure the efficient operation of the system itself:

7- Resource allocation

When multiple processes are running at the same time, resources must be allocated to each of them. The operating system manages many different types of resources. Some (such as CPU cycles, main memory, and file storage)

8- Accounting

keep track of and record which users use how much and what kinds of computer resources for account billing or for accumulating usage statistics.

9- Protection and security

The owners of information stored in a multiuser or networked computer system may want to control use of that information. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself.

Protection involves ensuring that all access to system resources is controlled.

Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate himself or herself to the system, usually by means of a password, to gain access to system resources.

Processes

A process is a program in execution. A process is more than the program code, It also includes the current activity, as represented by:

- The value of the program counter,
- The contents of the processor's registers. In addition,
- A process generally includes the process stack, which contains temporary data (such as method parameters, return addresses, and local variables), and a data section, which contains global variables.

a program is a passive entity, whereas a process is an active entity,

Process State

As a process executes, it changes state. The state of a process is defined in part by its current activity. A process may be in one of the following states:

- **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.

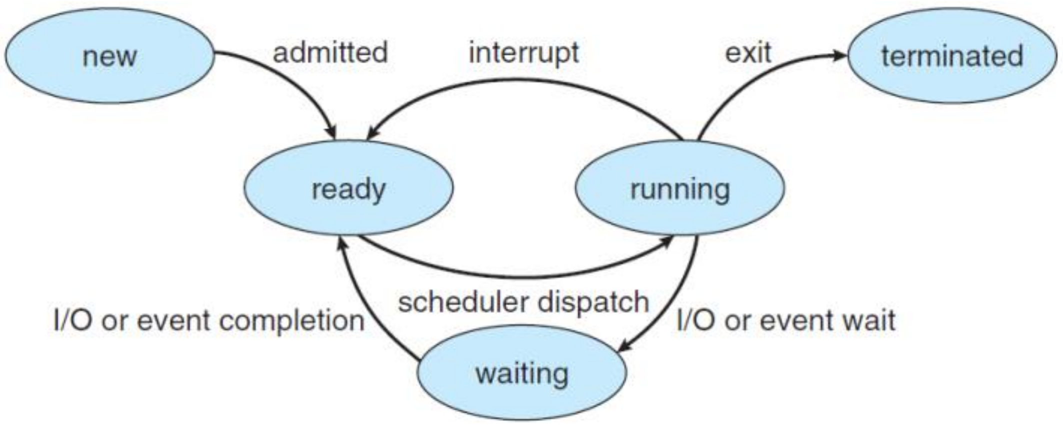


Figure: Diagram of process state

Process Control Block (PCB)

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, including these:

- **Process state:** The state may be new, ready, running, waiting, etc.
- **Process number (ID):** each process has a unique number.
- **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. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.
- **CPU scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **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.

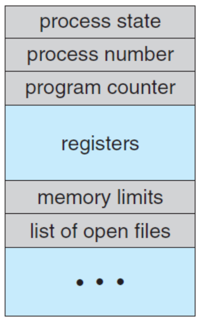


Figure: Process Control Block(PCB)

Process Scheduling

To meet the objectives of multiprogramming and multitasking, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on a core. Each CPU core can run one process at a time. If there are more processes than cores, excess processes will have to wait until a core is free and can be rescheduled. The number of processes currently in memory is known as the degree of multiprogramming.

In general, most processes can be described as either I/O bound or CPU bound. An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations. A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations.

Scheduling Queues

As processes enter the system, they are put into a ready queue, where they are ready and waiting to execute on a CPU's core. Processes that are waiting for a certain event to occur — such as completion of I/O — are placed in a wait queue.

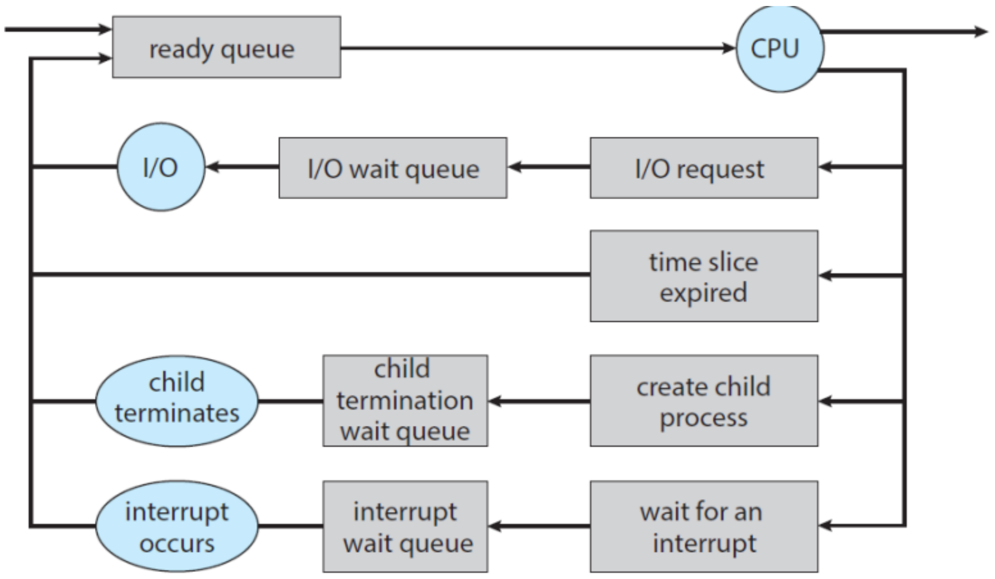


Figure: Queuing-diagram representation of process scheduling

Schedulers

A process migrates between the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The appropriate scheduler carries out the selection process.

Types of schedulers:

1. Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue.
2. Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU.
3. Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This medium-term scheduler,

removes processes from memory (and from active contention for the CPU), and thus reduces the degree of multiprogramming. At some later time, the process can be reintroduced into memory and its execution can 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. Swapping may be necessary to improve the process mix for system balancing, or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast).
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow).
- The long-term scheduler controls the *degree of multiprogramming.*
- Processes can be described as either:
    - I/O-*bound process* – spends more time doing I/O than computations, many short CPU bursts.
    - *CPU-bound process* – spends more time doing computations; few very long CPU bursts.

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 PCB of a process.

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-switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (Context-switch times are highly dependent on hardware support).

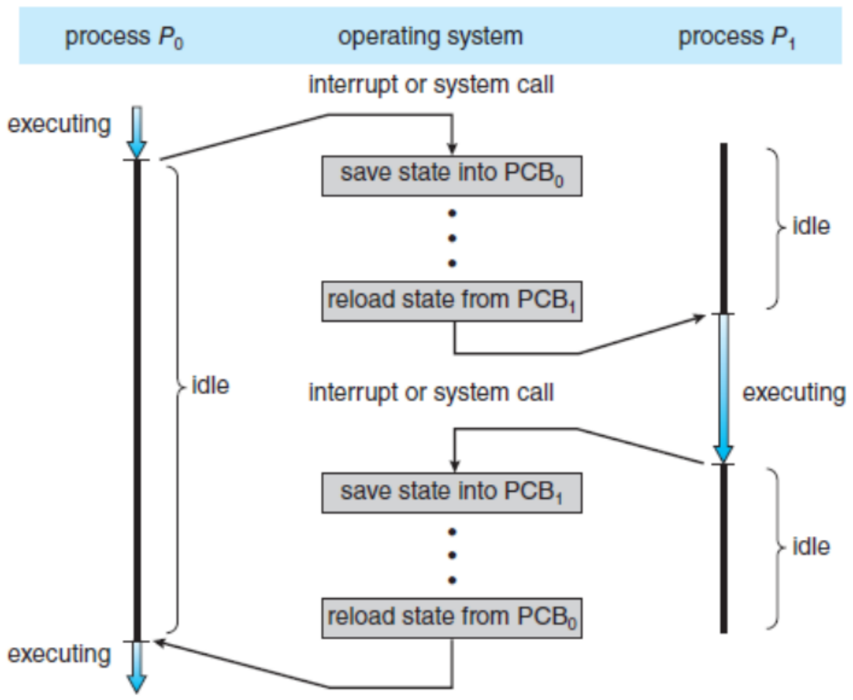


Figure: Diagram showing context switch from process to process

CPU Scheduling

CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive.

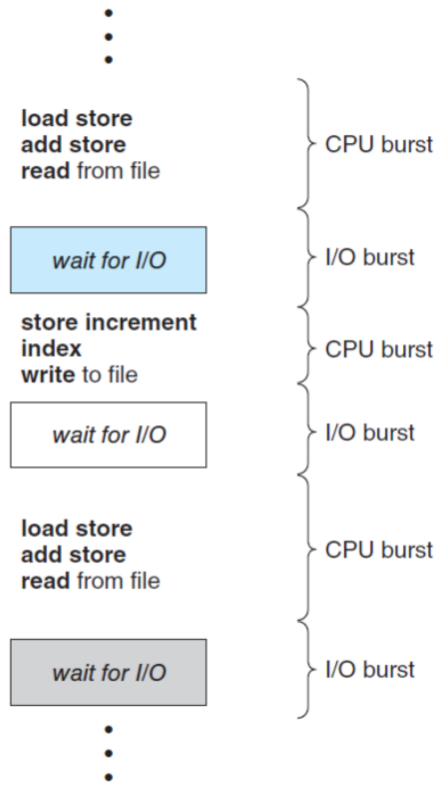


Figure: Alternating sequence of CPU and I/O bursts

When the CPU becomes idle, the CPU scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state.
2. Switches from running to ready state.
3. Switches from waiting to ready.
4. Terminates.

- Scheduling under 1 and 4 is *nonpreemptive*.
- All other scheduling is *preemptive.*

Preemptive VS. Nonpreemptive Scheduling

Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.

Under preemptive scheduling, the CPU is taken from the process abandomly and given to other process.

Dispatcher

Dispatcher module gives control of the CPU to the process selected by the short-term scheduler (CPU scheduler) ; this involves:

- switching context
- switching to user mode
- jumping to the proper location in the user program to restart that program

- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running.

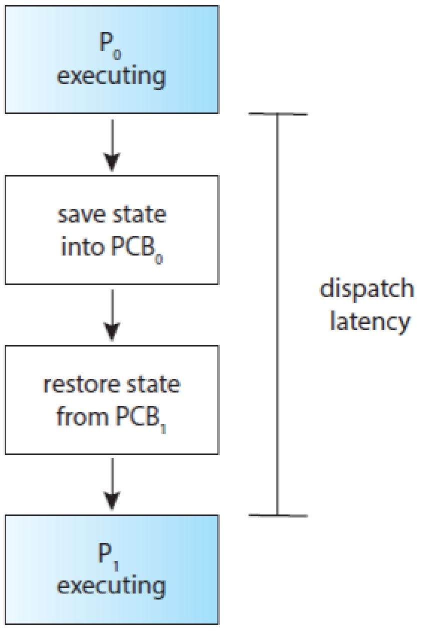


Figure: The role of the dispatcher

Scheduling Criteria

Many criteria have been suggested for comparing CPU scheduling algorithms. The criteria include the following:

- CPU utilization – keep the CPU as busy as possible
- Throughput – no. of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process (Turnaround time is the sum of the periods spent waiting in the ready queue, executing on the CPU, and doing I/O).
- Waiting time – amount of time a process has been waiting in the ready queue (Waiting time is the sum of the periods spent waiting in the ready queue).
- Response time – amount of time it takes from when a request was submitted until the first response is produced, not the time it takes to output the response.

Optimization Criteria

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.

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

Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU's core. There are many different CPU scheduling algorithms. we describe these scheduling algorithms in the context of only one processing core available.

1. First-Come, First-Served Scheduling

• 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.

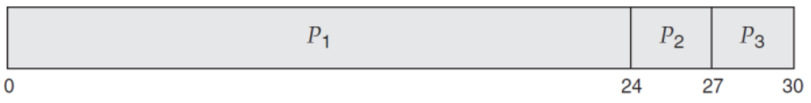
• The code for FCFS scheduling is simple to write and understand.

• the average waiting time under the FCFS policy is often quite 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 the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following Gantt chart

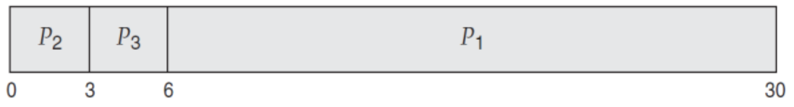


- Waiting time for P1(WT_{P1}) is start time(p1) – arrival time(p1):

WT_{P1}=0 - 0= 0
WT_{P2}= 24 – 0 =24
WT_{P3} = 27 – 0 =27

- Average waiting time(AWT) is (0+ 24 + 27)/3 = 17 millisecondes.
- Turnaround time (TAT_P) for each process is calculated as end time(p) – Arrival time(p)
- TAT_{P1}= 24 – 0 = 24
- TAT_{P2}= 27 – 0 = 27
- TAT_{P3}= 30 – 0 = 30
- Average turnaround time(ATAT) is (24 + 27 +30) / 3 = 27 milliseconds

If the processes arrive in the order P2, P3, P1, however, the results will be as shown in the following Gantt chart



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds.
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.

- The FCFS scheduling algorithm is nonpreemptive.
- The FCFS algorithm is thus particularly troublesome for time-sharing systems.

Example: Consider the following set of processes

Process	Burst time	Arrival time
P1	10	0
P2	6	6
P3	3	7
P4	7	22

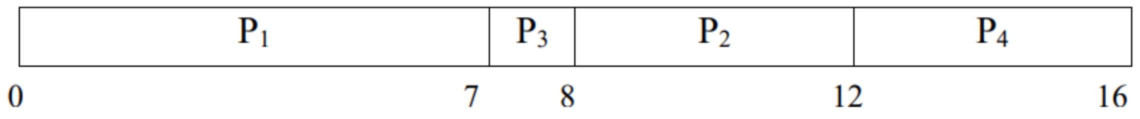
1. Draw the Gantt chart for these processes applying FCFS algorithm.
2. What is the waiting time for each process?
3. What is the Average waiting time?
4. What is the Turnaround time for each process?
5. What is the Average turnaround time?

2. Shortest Job First (SJF)

It is a non-preemptive scheduling discipline in which the waiting job/process with the smallest estimated run time to completion is run next. If two jobs have the same run time, FCFS is used. SJF reduces average waiting time over FCFS.

Example:

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



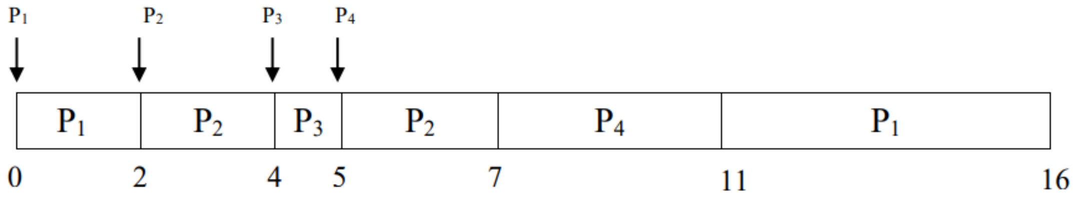
Average waiting time = (0 + 6 + 3 + 7)/4 = 4
Turnaround time for P_1= **7**
Turnaround time for P_2= (8-2)+4=10
Turnaround time for P_3= (7-4)+1=4
Turnaround time for P_4=(12-5)+ 4=11
The average =(7+10+4+11)/4=8

3. Shortest Remaining Time First(SRTF)

Preemptive SJF scheduling is sometimes called shortest-remaining-time-first (SRTF) scheduling. In SRTF, the algorithm will preempt the currently executing process when the newly arrived process is shorter than what is left of the currently executing process.

Example 1:

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



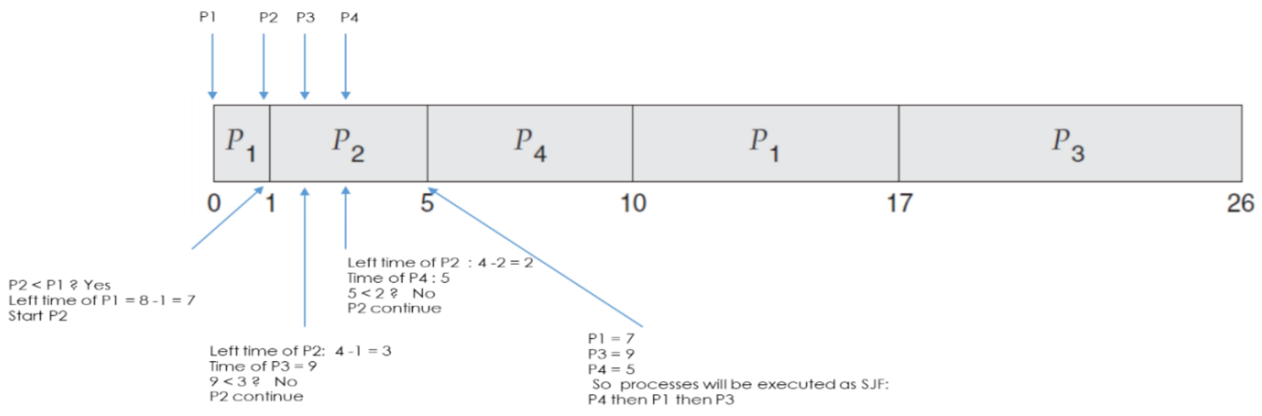
Process1 starts at time 0, since it is the only job in the queue. Process2 arrives at time 2. The remaining time for process1 (5 time units) is larger than the time required by process2 (4 time units), so process1 is preempted, and process2 is scheduled. The average turnaround time for this example is:

((16-0) +(7-2)+(5-4)+(11-5))/4 =7 time units

Example 2:

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

the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



Waiting time for process P is the summation of waiting times for it in the ready queue

$WT_{P1} = (0 - 0) + (10 - 1) = 9$

$WT_{P2} = 1 - 1 = 0$

$WT_{P3} = 17 - 2 = 15$

$WT_{P4} = 5 - 3 = 2$

Average waiting time(AWT) is (9+ 0 + 15+2) / 4 = 6.5 millisecondes.

Turnaround time (TAT_P) for each process is calculated as end time(p) – Arrival time(p) OR summation of waiting times(P) + Burst time(P)

$TAT_{P1} = 9 + 8 = 17$

$TAT_{P2} = 0 + 4 = 4$

$TAT_{P3} = 15 + 9 = 24$

$TAT_{P4} = 2 + 5 = 7$

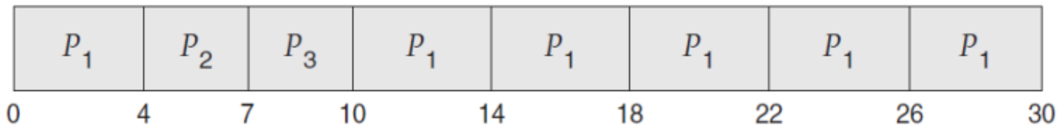
Average turnaround time(ATAT) is (17 + 4 +24+ 7) / 4 = 13 milliseconds

4. Round-Robin Scheduling

The round-robin (RR) scheduling algorithm 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 from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to one-time quantum. 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



- Waiting time for P1 = (0-0) +(10-4) = 6 milliseconds
  P2 =4 – 0 =4 milliseconds
  P3 = 7 – 0 =7 milliseconds
- The average waiting time = (6 + 4 + 7) / 3 =5.66 milliseconds

The performance of the RR algorithm depends heavily on the size of the time quantum. To improve performance, the time quantum must be large with respect to the context switch time.

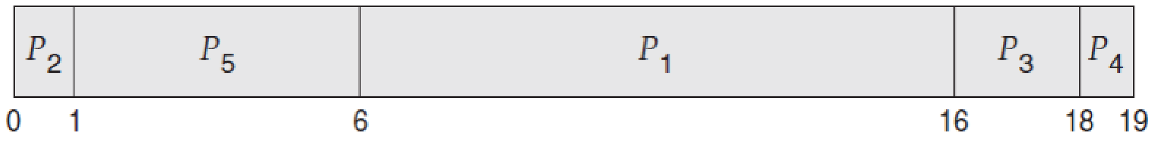
5. Priority Scheduling

Each process has a priority, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7. We will assume that a low number represents high priority. Priority scheduling can be either preemptive or non-preemptive.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, · · ·, P5, 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 (non-preemptive), we would schedule these processes according to the following Gantt chart:



The average waiting time is 8.2 milliseconds.

A major problem with priority scheduling algorithms is indefinite blocking or starvation. A priority scheduling algorithm can leave some low priority processes waiting indefinitely.

A solution to the problem of indefinite blockage of low-priority processes:

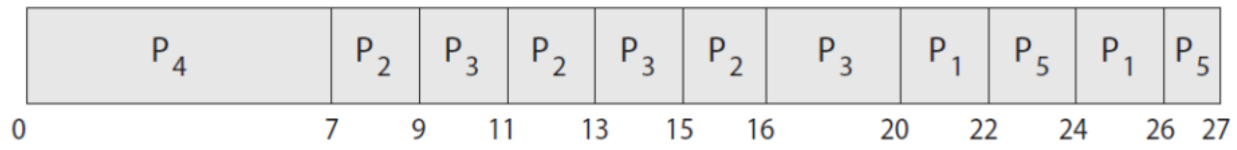
- **Aging:** Aging involves gradually increasing the priority of processes that have been waiting in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could periodically (say, every second) increase the priority of the awaiting process by 1.

- **combine round-robin and priority scheduling** in such a way that the system executes the highest-priority process and runs processes with the same priority using round-robin scheduling.

Let's illustrate with an example using the following set of processes, with the burst time in milliseconds:

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

Using priority scheduling with round-robin for processes with equal priority, we would schedule these processes according to the following Gantt chart using a time quantum of 2 milliseconds:



In this example, process P4 has the highest priority, so it will run to completion. Processes P2 and P3 have the next-highest priority, and they will execute in a round-robin fashion. Notice that when process P2 finishes at time 16, process P3 is the highest-priority process, so it will run until it completes execution.

Now, only processes P1 and P5 remain, and as they have equal priority, they will execute in round-robin order until they complete.

6. Multilevel Queue Scheduling

A multilevel queue scheduling algorithm can also be used to partition processes into several separate queues based on the process type. For example, a division between foreground (interactive) processes and background (batch) processes. Each type may have different scheduling needs.

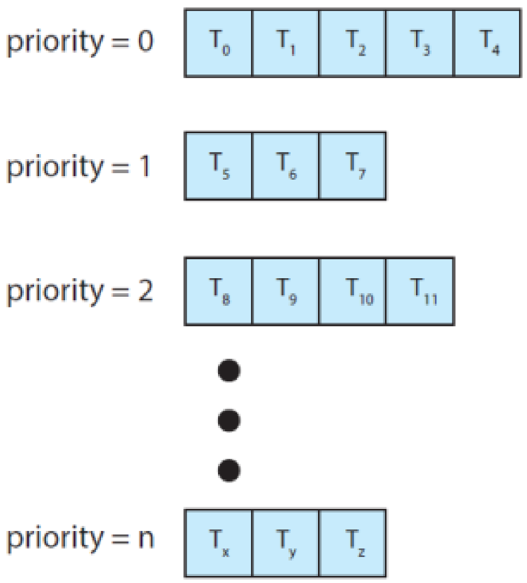


Figure: Separate queues for each priority.

Separate queues might be used for foreground and background processes, and each queue might have its own scheduling algorithm. In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.

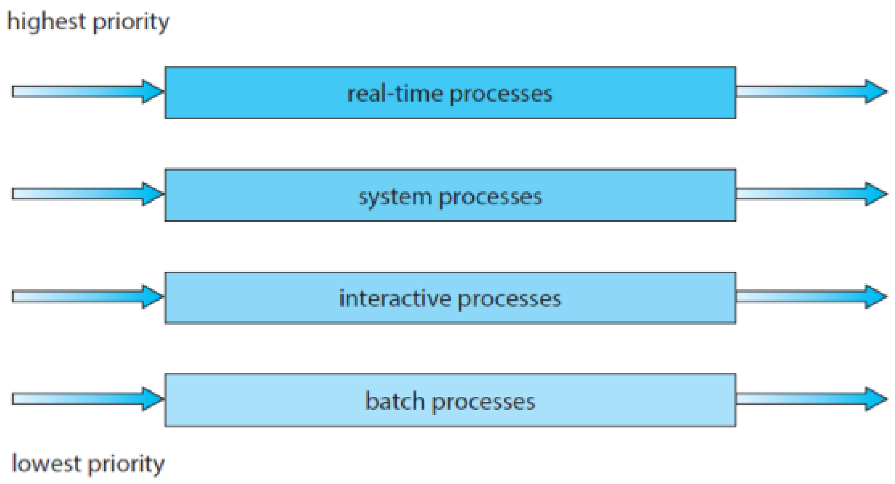


Figure: Multilevel queue scheduling.

Let’s look at an example of a multilevel queue scheduling algorithm with four queues, listed below in order of priority:

1. Real-time processes

2. System processes

3. Interactive processes

4. Batch processes

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for real-time processes, system processes, and interactive processes were all empty. If an interactive process entered the ready queue while a batch process was running, the batch 36 process would be preempted. Another possibility in this algorithm is to time-slice among the queues.

7. 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. This setup has the advantage of low scheduling overhead, but it is inflexible. The multilevel feedback queue scheduling algorithm, in contrast, allows a process to move between queues.

If a process uses too much CPU time, it will be moved to a lower-priority queue. 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.

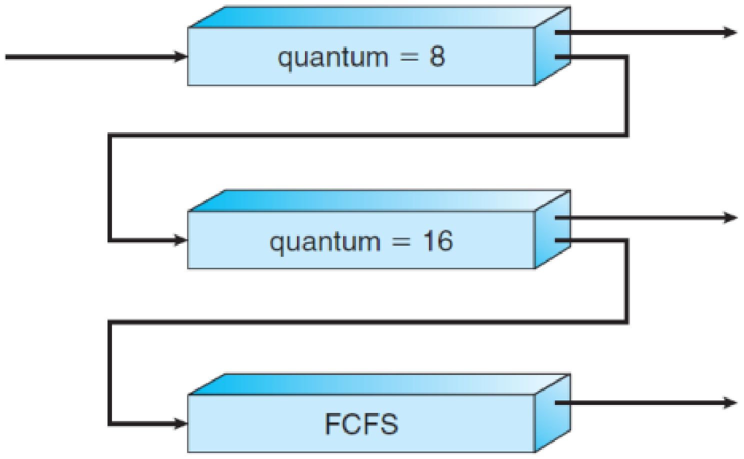


Figure: Multilevel feedback queues.

In general, a multilevel feedback queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher priority queue
- The method used to determine when to demote a process to a lower priority queue
- The method used to determine which queue a process will enter when that process needs service

The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm. Unfortunately, it is also the most complex algorithm, since defining the best scheduler requires some means by which to select values for all the parameters.

Deadlocks

We say that a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each of which consists of some number of identical instances. Memory space, CPU cycles, files, and I/O devices (such as printers and tape drives) are examples of resource types. If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances.

A process must request a resource before using it, and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. Request: If the request cannot be granted immediately (for example, the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

2. Use: The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

3. Release: The process releases the resource.

Necessary Conditions

Deadlock can arise if four conditions hold simultaneously.

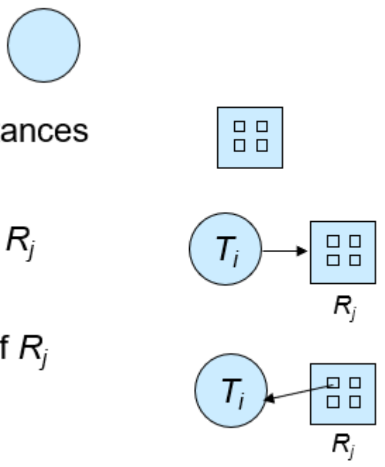
- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{T_0, T_1, \ldots, T_n\}$ of waiting processes such that T_0 is waiting for a resource that is held by T_1, T_1 is waiting for a resource that is held by T_2, …, T_{n-1} is waiting for a resource that is held by T_n, and T_n is waiting for a resource that is held by T_0.

Resource-Allocation Graph

A set of vertices V and a set of edges E.

- V is partitioned into two types:
    - $T = \{T_1, T_2, \ldots, T_n\}$, the set consisting of all the processes in the system
    - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $T_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow T_i$

- Process
- Resource Type with 4 instances
- T_i requests an instance of R_j
- T_i is holding an instance of R_j



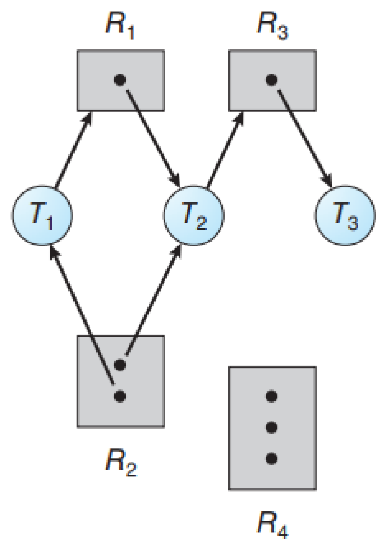


Figure: Resource-allocation graph

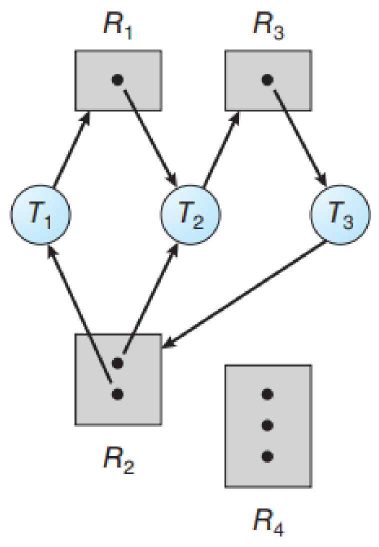


Figure 8.5 Resource-allocation graph with a deadlock.

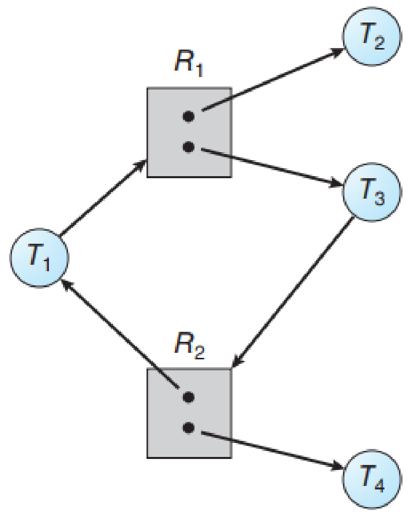


Figure 8.6 Resource-allocation graph with a cycle but no deadlock.

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
    - if only one instance per resource type, then deadlock
    - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
    - Deadlock prevention
    - Deadlock avoidance
- Allow the system to enter a deadlock state, detect it, and recover.
- Ignore the problem and pretend that deadlocks never occur in the system

Deadlock prevention provides a set of methods to ensure that at least one of the necessary conditions cannot hold.

Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime.

………………

Deadlock Prevention

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock. We elaborate on this approach by examining each of the four necessary conditions separately.

- **Mutual Exclusion**

not required for sharable resources (e.g., read-only files); must hold for non-sharable resources

- **Hold and Wait**

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, *whenever a process requests a resource, it does not hold any other resources.*

- One protocol that we can use requires each process to request and be allocated all its resources before it begins execution.
- An alternative protocol allows a process to request resources only when it has none.

Both these protocols have two main disadvantages:

- First, resource utilization may be low, since resources may be allocated but unused for a long period.
- Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

- **No Preemption**

We can use the following protocol to ensure this condition does not hold. If a process holds some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process currently holds are preempted.
The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources and the new ones it is requesting, which may cause starvation.

- **Circular Wait**

impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

Possible side effects of preventing deadlocks by this method, however, are:

1. low device utilization.
2. reduced system throughput.

An alternative method for avoiding deadlocks requires that the system has some additional ***a priori*** information available

- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition

- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A sequence of processes <T1,T2,…,Tn> is a safe sequence for the current allocation state if, for each Ti, the resource requests that Ti can still make can be satisfied by the currently available resources plus the resources held by all Tj, with j < i.

<u>A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks.</u>

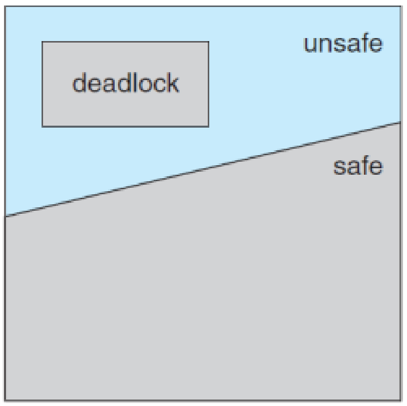


Figure: Safe, unsafe, and deadlocked state spaces.

Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

Resource-Allocation-Graph Algorithm

In resource-allocation graph, we introduce a new type of edge, called a claim edge. A claim edge Ti→Rj indicates that process Ti may request resource Rj at some time in the future. When process Ti requests resource Rj, the claim edge Ti → Rj is converted to a request edge. Similarly, when a resource Rj is released by Ti, the assignment edge Rj →Ti is reconverted to a claim edge Ti →Rj. The request can be granted only if converting the request edge Ti → Rj to an assignment edge Rj → Ti does not result in the formation of a cycle in the resource-allocation graph

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process Ti will have to wait for its requests to be satisfied.

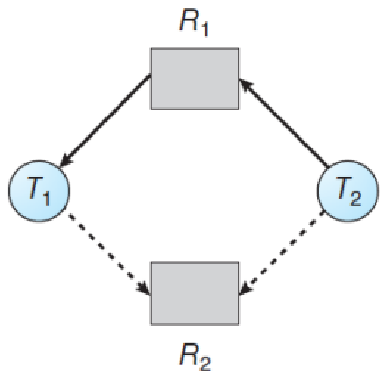


Figure: Resource-allocation graph for deadlock avoidance

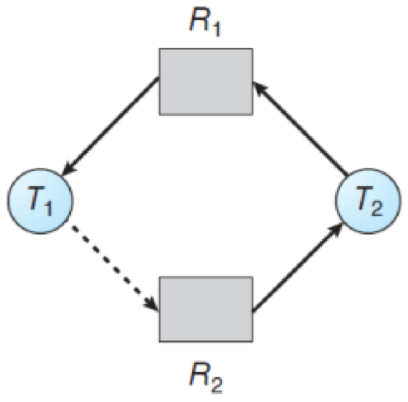


Figure: An unsafe state in a resource-allocation graph

Banker's Algorithm

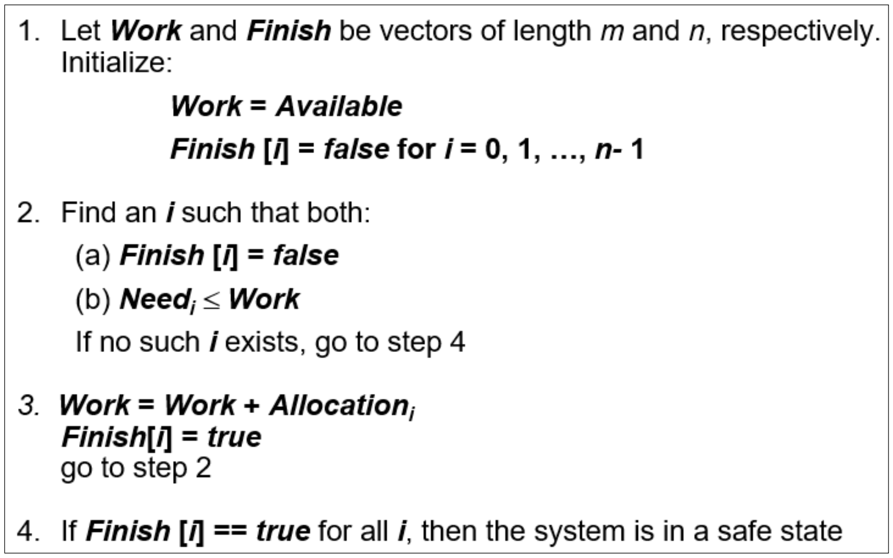
The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. Several data structures must be maintained to implement the banker's algorithm.

We need the following data structures, where n is the number of processes in the system and m is the number of resource types:

- **Available**: Vector of length *m*. If available [*j*] = *k*, there are *k* instances of resource type R_j available
- **Max**: *n x m* matrix. If *Max* [*i,j*] = *k*, then process P_i may request at most *k* instances of resource type R_j
- **Allocation**: *n* x *m* matrix. If Allocation[*i,j*] = *k* then P_i is currently allocated *k* instances of R_j
- **Need**: *n* x *m* matrix. If *Need*[*i,j*] = *k*, then P_i may need *k* more instances of R_j to complete its task

Need [*i,j*] = *Max*[*i,j*] – *Allocation* [*i,j*]

a- Safety Algorithm



1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:

   Work = Available

   ***Finish [i] = false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

   (a) ***Finish [i] = false***

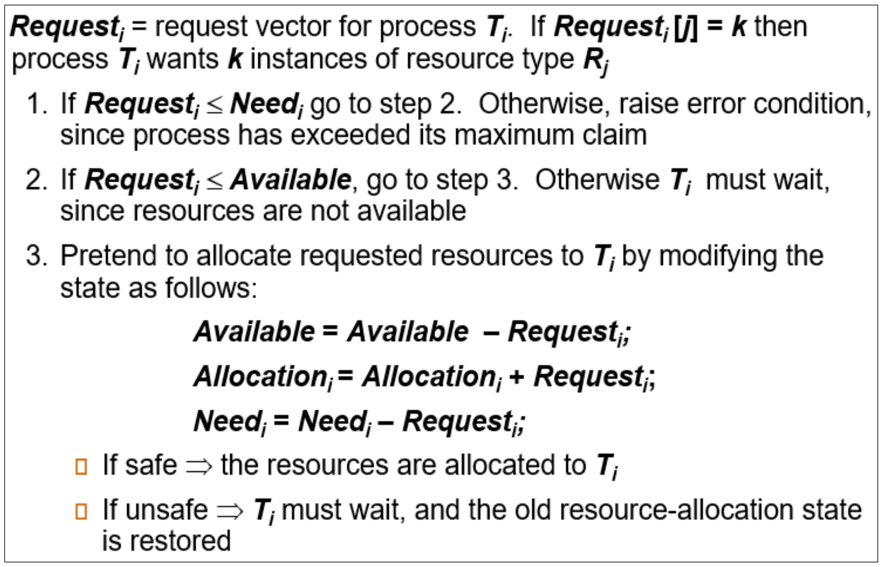
   (b) ***Need$_i$*** $\leq$ ***Work***

   If no such ***i*** exists, go to step 4

3. ***Work = Work + Allocation$_i$***
   Finish[i] = true
   go to step 2

4. If ***Finish [i] == true*** for all ***i***, then the system is in a safe state

b- Resource-Request Algorithm



Request$_i$ = request vector for process ***T$_i$***. If ***Request$_i$ [j] = k*** then process ***T$_i$*** wants ***k*** instances of resource type ***R$_j$***

1. If ***Request$_i$*** $\leq$ ***Need$_i$*** go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If ***Request$_i$*** $\leq$ ***Available***, go to step 3. Otherwise ***T$_i$*** must wait, since resources are not available
3. Pretend to allocate requested resources to ***T$_i$*** by modifying the state as follows:

   Available = Available – Request$_i$;

   Allocation$_i$ = Allocation$_i$ + Request$_i$;

   Need$_i$ = Need$_i$ – Request$_i$;

   - If safe $\Rightarrow$ the resources are allocated to ***T$_i$***
   - If unsafe $\Rightarrow$ ***T$_i$*** must wait, and the old resource-allocation state is restored

Example

consider a system with five processes T0 through T4 and three resource types A, B, and C. Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances. Suppose that the following snapshot represents the current state of the system:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	7 5 3	3 3 2
T_1	2 0 0	3 2 2	
T_2	3 0 2	9 0 2	
T_3	2 1 1	2 2 2	
T_4	0 0 2	4 3 3	

المصفوفة Need تمثل بالشكل التالي:

	Need
	A B C
T_0	7 4 3
T_1	1 2 2
T_2	6 0 0
T_3	0 1 1
T_4	4 3 1

We must determine whether this system state is safe. To do so, we execute our safety algorithm. Executing safety algorithm shows that sequence < T_1, T_3, T_4, T_0, T_2> satisfies safety requirement.

Suppose now that T1 request is Request1 = (1,0,2). To decide whether this request can be immediately granted, we first check that Request1 ≤ Available—that is, that (1,0,2) ≤ (3,3,2), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	7 4 3	2 3 0
T_1	3 0 2	0 2 0	
T_2	3 0 2	6 0 0	
T_3	2 1 1	0 1 1	
T_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we execute our safety algorithm to decide if we can immediately grant the request of process T1 or to postponed granting it. Executing safety algorithm shows that sequence < T_1, T_3, T_0, T_2 ,T_4 > satisfies safety requirement.

H.W

- Can request for (3,3,0) by T_4 be granted?
- Can request for (0,2,0) by T_0 be granted?

Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

• An algorithm that examines the state of the system to determine whether a deadlock has occurred

• An algorithm to recover from the deadlock.

Next, we discuss algorithms pertain to systems with only a single instance of each resource type, as well as to systems with several instances of each resource type.

1- Single Instance of Each Resource Type

- Maintain **wait-for** graph
  - Nodes are processes
  - $\boldsymbol{T_i \rightarrow T_j}$ if $\boldsymbol{T_i}$ is waiting for $\boldsymbol{T_j}$
- To detect deadlocks, the system needs to maintain the wait for graph and Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

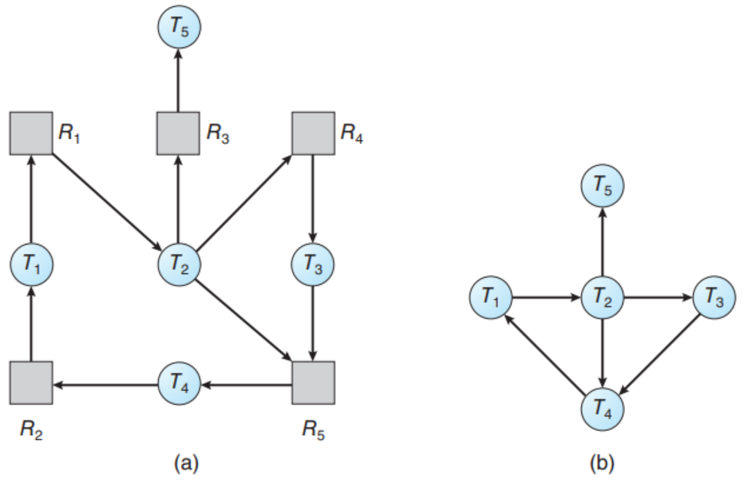
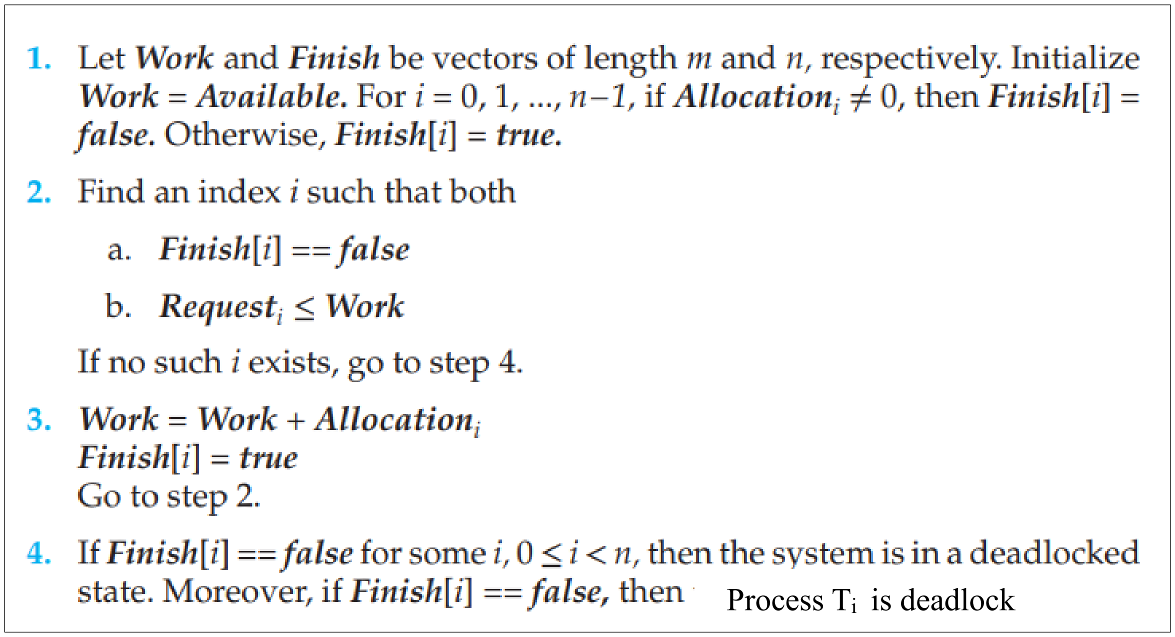


Figure: (a) Resource-allocation graph. (b) Corresponding wait-for graph.

2- Several Instances of a Resource Type

- **Available**: A vector of length ***m*** indicates the number of available resources of each type
- **Allocation**: An ***n* x *m*** matrix defines the number of resources of each type currently allocated to each process
- **Request**: An ***n* x *m*** matrix indicates the current request of each process. If ***Request* [*i*][*j*] = *k***, then process T_i is requesting ***k*** more instances of resource type R_j.

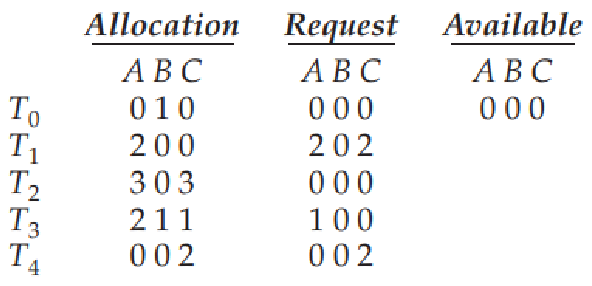


1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize *Work* = *Available*. For $i = 0, 1, ..., n-1$, if $Allocation_i \neq 0$, then *Finish*[i] = *false*. Otherwise, *Finish*[i] = *true*.
2. Find an index i such that both
   a. *Finish*[i] == *false*
   b. $Request_i \leq Work$

   If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
   Finish[i] = *true*
   Go to step 2.
4. If *Finish*[i] == *false* for some i, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if *Finish*[i] == *false*, then Process T_i is deadlock

Example of Detection Algorithm

Five processes T_0 through T_4; three resource types: A (7 instances), *B* (2 instances), and *C* (6 instances). The following snapshot represents the current state of the system:



	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	0 0 0	0 0 0
T_1	2 0 0	2 0 2	
T_2	3 0 3	0 0 0	
T_3	2 1 1	1 0 0	
T_4	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence <T0,T2,T3,T1,T4> results in ***Finish[i]* = *true*** for all ***i***

Suppose now that process T2 makes one additional request for an instance of type C. The Request matrix is modified as follows:

	Request
	A B C
T_0	0 0 0
T_1	2 0 2
T_2	0 0 1
T_3	1 0 0
T_4	0 0 2

We claim that the system is now deadlocked. Although we can reclaim the resources held by process T0, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes T1, T2, T3, and T4.

Recovery from Deadlock

1-Process Termination
2-Resource Preemption

1-Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
    - Priority of the process
    - How long process has computed, and how much longer to completion
    - Resources the process has used
    - Resources process needs to complete
    - How many processes will need to be terminated
    - Is process interactive or batch?

2-Resource Preemption

- **Selecting a victim:** The order of preemption must be determined to minimize cost. Cost factors may include parameters such as the number of resources a deadlocked process is holding and the amount of time the process has consumed thus far.
- **Rollback:** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state.
- **Starvation:** In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation any practical system must address. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times.

The most common solution is to include the number of rollbacks in the cost factor.

Operating systems

(عملـي)

م. منار عبدالكريم زيدان العباجي

م. كنار محمد سامي

طرق تقسيم الهارد ديسك

١. تقسيم أثناء الفرمتة.
٢. تقسيم بدون فرمتة.

قبل البدء يجدر الذكر أن شكل القوائم والخيارات قد يختلف بين حاسبٍ وآخر لذلك عليك فهم الخطوات الأساسية.

تقسيم الهارد أثناء الفرمتة

أثناء تنصيب نظام ويندوز يتطلب الأمر اختيار الجزء من الهارد الذي تود أن تحفظ ملفات النظام فيه، حيث تكون الخطوات كالتالي:

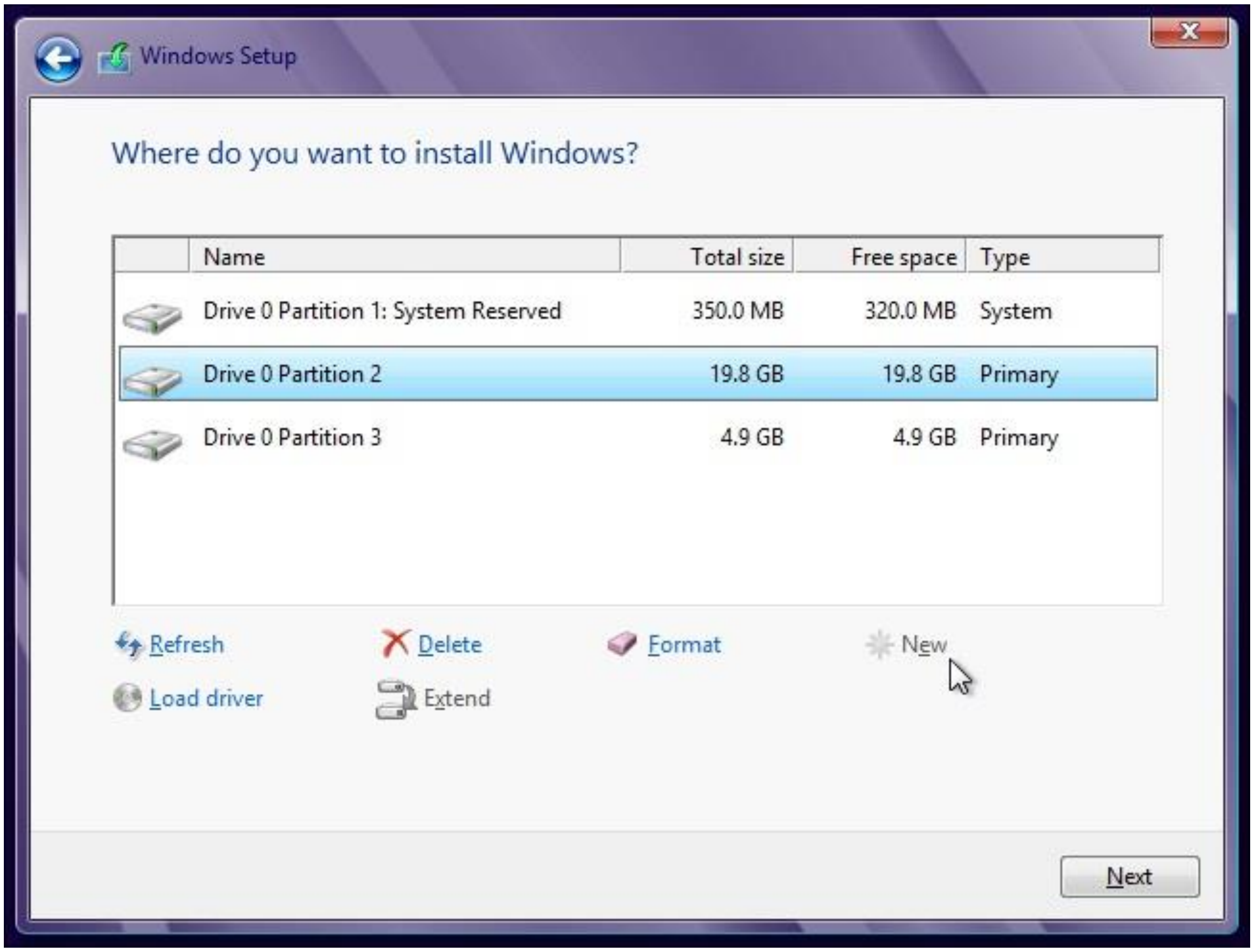
١. تبدأ العملية بإدراج DVD أو فلاشة إقلاعية تحوي النظام قبل تشغيل الحاسب ثم الاقلاع منها، وتتسلسل الواجهات بعدها حسب نسخة النظام الذي تقوم بتشغيله لتصل إلى واجهة اختيار القرص.
٢. تظهر الأقراص الموجودة على الحاسب لديك بأسماءٍ تدل على محتواها غالبًا، وليس **الأسماء** المعتادة C,D... وهنا لديك حرية الاختيار ولكن إليك بعض التوضيح:

- حذف قرص عن طريق Delete والذي يجعل منه مساحةً غير مخصصةٍ، وهذه الخطوة هامةٌ حيث يجب الحذر من أن تحذف الملفات الخاصة بك فهي في أحد الأقراص أمامك.
- إضافة قرصٍ جديدٍ وتحديد المساحة المخصصة له بشكلٍ دقيقٍ، أو ترك الرقم كما هو في حال لم تحذف سوى قرصٍ واحدٍ ولا تريد تقسيمها.
- اختيار القرص الجديد والضغط على زر التالي أو تنصيب حسب الواجهة الظاهرة أي البدء بعملية الفرمتة.

الحفاظ على ملفاتك هنا هو بألا تحذف الأقراص الموجودة فيها. وفي حال أردت إعادة التقسيم بشكلٍ كاملٍ قم بأخذ نسخةٍ احتياطيةٍ من الملفات الهامة لأن عملية التقسيم أثناء الفرمتة تؤدي إلى حذف كافة ملفات الحاسب بشكلٍ كاملٍ غير قابلٍ للاستعادة.[1]

تذكير :قم بالتأكد من تحديد القرص الصحيح قبل أن تضغط على زر تنصيب النظام الجديد.

ملاحظة :سجل على ورقةٍ خارجيةٍ حجم كل قرصٍ والمساحة المستخدمة منه قبل البدء بالفرمتة، ذلك سيساعدك في اختيار القرص الصحيح لتقوم بحذفه والمحافظة على بقية الأقراص من الخطأ.



Where do you want to install Windows?

	Name	Total size	Free space	Type
	Drive 0 Partition 1: System Reserved	350.0 MB	320.0 MB	System
	Drive 0 Partition 2	19.8 GB	19.8 GB	Primary
	Drive 0 Partition 3	4.9 GB	4.9 GB	Primary

تقسيم الهارد ديسك دون فرمتة

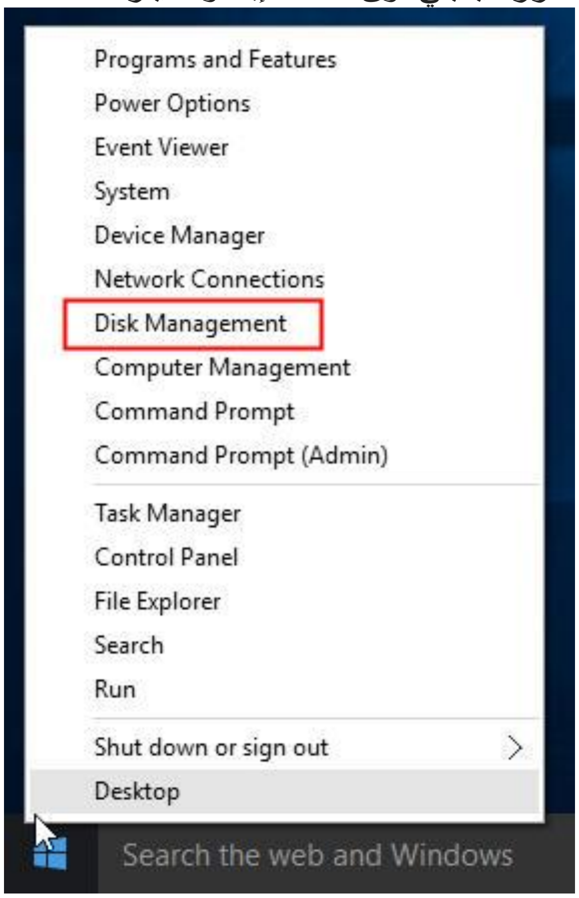
يمكنك استخدام هذه الطريقة في حال رغبتك في تقسيم الهارد الخاص بك دون تنصيب نظام التشغيل من جديد والمحافظة أيضًا على ملفاتك، ويتم ذلك عن طريق أداةٍ برمجيةٍ متوفرةٍ ضمن النظام تدعى Disk Management وإليك الخطوات:

الوقت اللازم: 3 دقائق.

تقسيم الهارد ديسك دون فرمتة

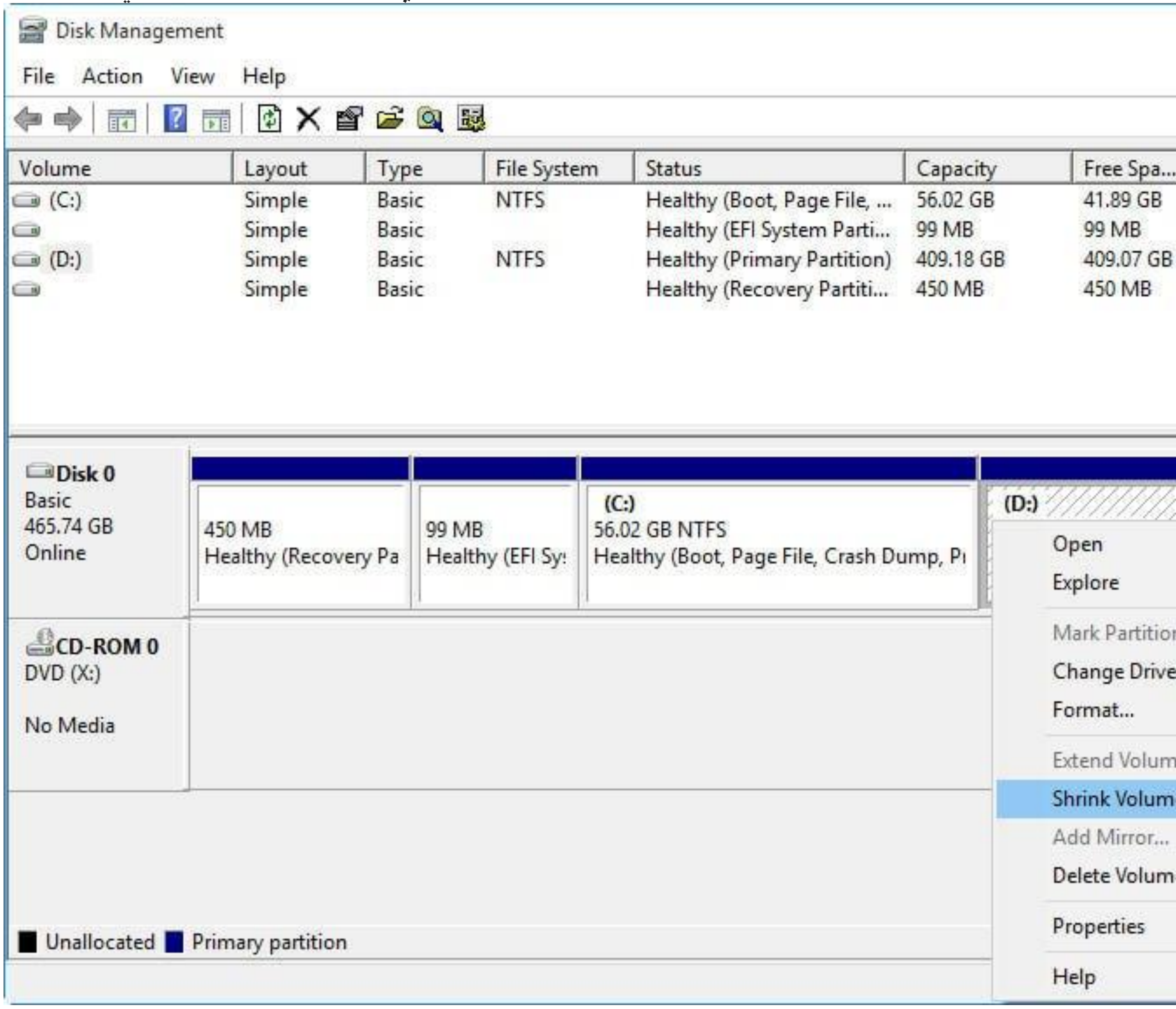
1. **فتح أداة إدارة الأقراص**

أو بالضغط على **Disk Management إدارة الأقراص** ابحث ضمن القائمة إبدأ على أداة الزر اليميني فوق القائمة إبدأ واختيارها.



2. تقليص القرص

اختر القرص الذي تود تقسيمه وبالزر اليميني اختر **تقليص القرص Shrink Volume.**


Disk Management

File Action View Help

Volume	Layout	Type	File System	Status	Capacity	Free Spa...
(C:)	Simple	Basic	NTFS	Healthy (Boot, Page File, ...	56.02 GB	41.89 GB
	Simple	Basic		Healthy (EFI System Parti...	99 MB	99 MB
(D:)	Simple	Basic	NTFS	Healthy (Primary Partition)	409.18 GB	409.07 GB
	Simple	Basic		Healthy (Recovery Partiti...	450 MB	450 MB

Disk 0
Basic
465.74 GB
Online

450 MB
Healthy (Recovery Pa

99 MB
Healthy (EFI Sy:

(C:)
56.02 GB NTFS
Healthy (Boot, Page File, Crash Dump, Pı

(D:)

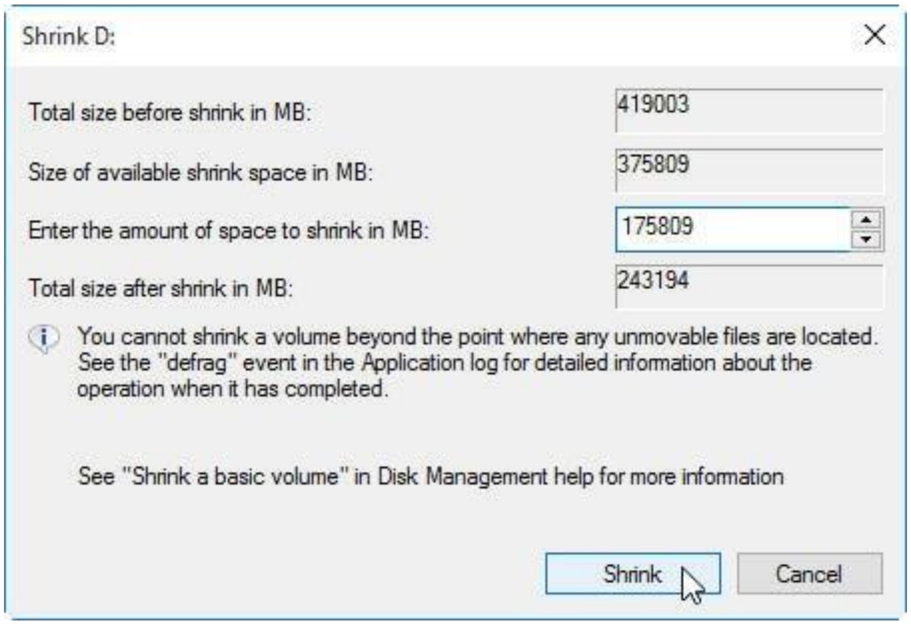
CD-ROM 0
DVD (X:)

No Media

■ Unallocated ■ Primary partition

3. البدء بتقسيم القرص

بعد انتهاء الحاسب من حساب المساحة المتاحة للتقسيم على القرص اختر المساحة التي تريد تخصيصها واضغط **تقليص Shrink.**


Shrink D:

Total size before shrink in MB: 419003

Size of available shrink space in MB: 375809

Enter the amount of space to shrink in MB: 175809

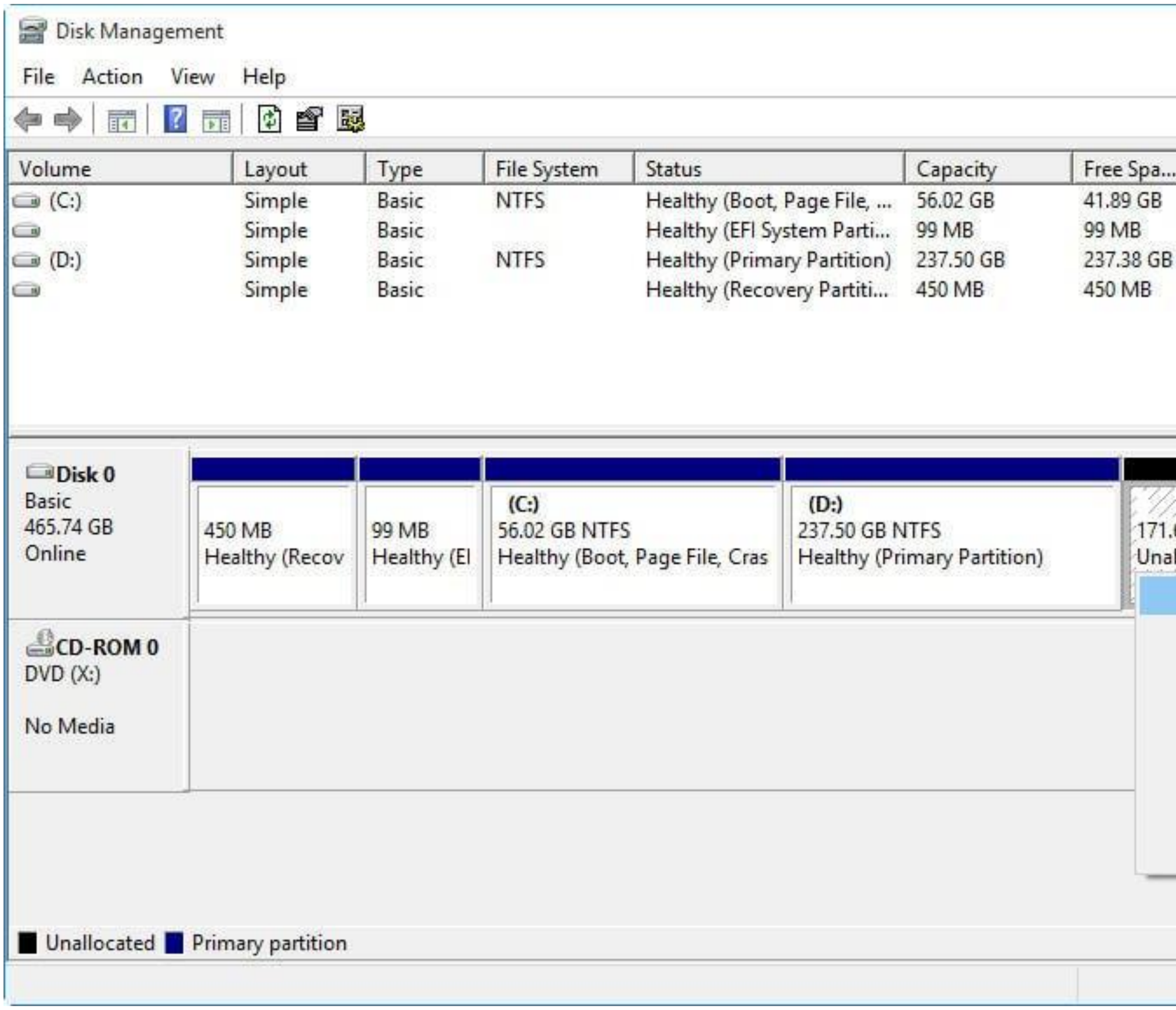
Total size after shrink in MB: 243194

You cannot shrink a volume beyond the point where any unmovable files are located. See the "defrag" event in the Application log for detailed information about the operation when it has completed.

See "Shrink a basic volume" in Disk Management help for more information

4. **إنشاء قرص بسيط**

بعد انتهاء العملية ستجد مساحة غير مخصصةٍ قم بالضغط عليها بالزر اليميني واختر **New Simple Volume**، واتبع العملية لإنشاء القرص الجديد.


Disk Management

File Action View Help

Volume	Layout	Type	File System	Status	Capacity	Free Spa...
(C:)	Simple	Basic	NTFS	Healthy (Boot, Page File, ...	56.02 GB	41.89 GB
	Simple	Basic		Healthy (EFI System Parti...	99 MB	99 MB
(D:)	Simple	Basic	NTFS	Healthy (Primary Partition)	237.50 GB	237.38 GB
	Simple	Basic		Healthy (Recovery Partiti...	450 MB	450 MB

Disk 0
Basic
465.74 GB
Online

450 MB
Healthy (Recov

99 MB
Healthy (EI

(C:)
56.02 GB NTFS
Healthy (Boot, Page File, Cras

(D:)
237.50 GB NTFS
Healthy (Primary Partition)

171.
Unal

CD-ROM 0
DVD (X:)

No Media

يتطلب إتمام العملية الضغط على زر التالي عدة مراتٍ وتحصل في النهاية على قرصٍ جديدٍ