OPERATING SYSTEM

UNIT-2 - Chapter-5

1. CPU SCHEDULING:

- CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold (in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU.
- The aim of CPU scheduling is to make the system efficient, fast and fair.
- Whenever the CPU becomes idle, the operating system must select one of the processes in the **ready queue** to be executed.
- The selection process is carried out by the short-term scheduler (or CPU scheduler).
- The scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

The idea of multiprogramming is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer, the CPU would then sit 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 given the CPU to another process.

Scheduling is a fundamental operating-system function.

1. 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.
- That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.
- Eventually, the final CPU burst ends with a system request to terminate execution (Figure).
- The durations of CPU bursts have been measured extensively.
- Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in Figure. The curve is generally characterized as exponential or hyper exponential, with a large number of short CPU bursts and a small number of long CPU bursts.

Scanned with CamScanner

Scanned with CamScanner

➢ An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.

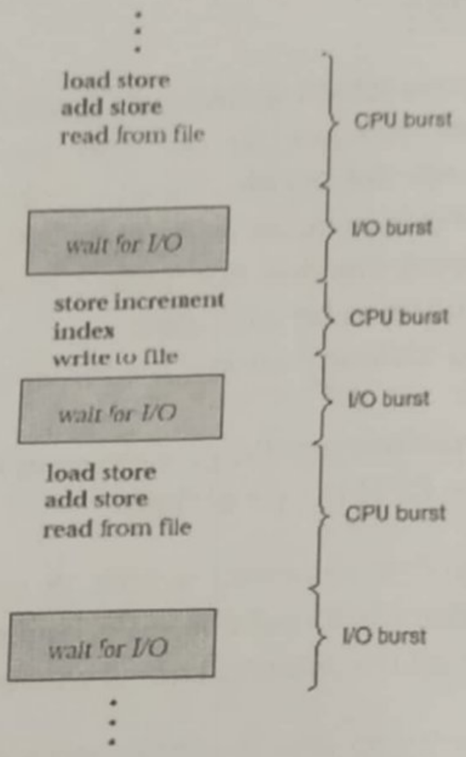


Figure : Alternating sequence of CPU and I/O bursts.

Scanned with CamScanner

Scanned with CamScanner

2. CPU Scheduler:

- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.
- The selection process is carried out by the short-term scheduler (or CPU scheduler).
- 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.
- As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list.
- Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU.
- The records in the queues are generally process control blocks (PCBs) of the processes.

3. Preemptive Scheduling:

- CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the **running state** to the **waiting state** (for example, as the result of an I/O request or an invocation of wait for the termination of one of the child processes)

2. When a process switches from the **running state** to the **ready state** (for example, when an interrupt occurs)

3. When a process switches from the **waiting state** to the **ready state** (for example, at completion of I/O)

4. When a process **terminates**

- In circumstances 1 and 4, there is no choice in terms of scheduling. A new process(if one exists in the ready queue) must be selected for execution. There is a choice, however in circumstances 2 and 3.
- When Scheduling takes place only under circumstances 1 and 4, we say the scheduling scheme is **non-preemptive**; otherwise the scheduling scheme is **preemptive**.

Non-Preemptive Scheduling

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

Scanned with CamScanner

Scanned with CamScanner

- This scheduling method is used by the Microsoft Windows 3.1 and by the Apple Macintosh operating systems.
- It is the only method that can be used on certain hardware platforms, because It does not require the special hardware (for example: a timer) needed for preemptive scheduling.

Preemptive Scheduling

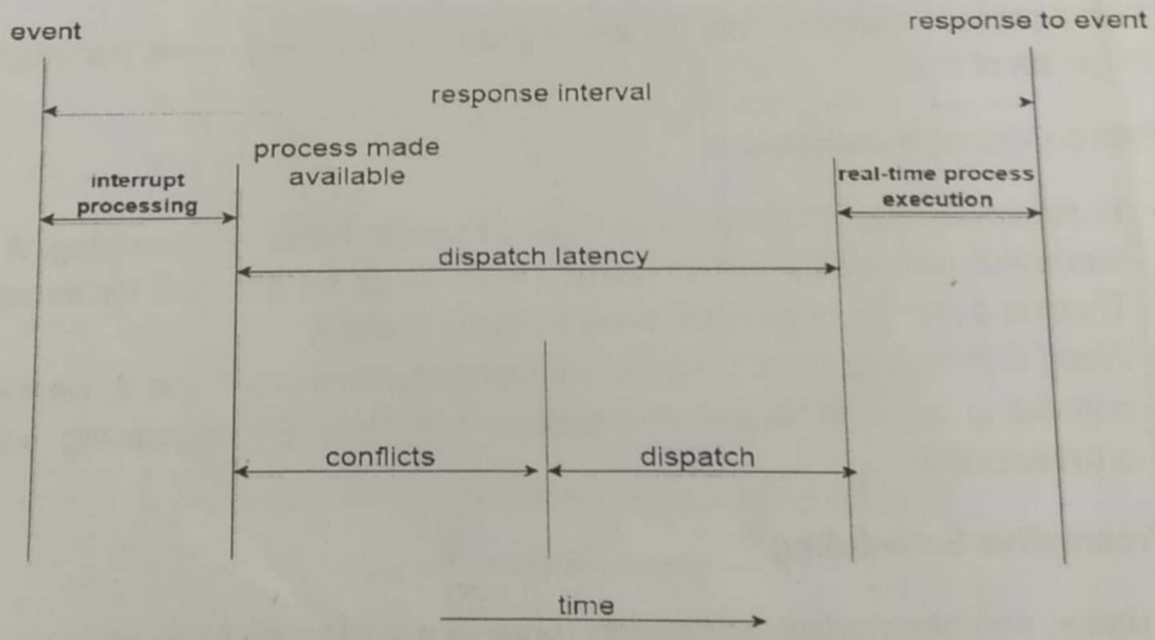
In this type of Scheduling, the tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution.

4. Dispatcher:

- Another component involved in the CPU scheduling function is the **dispatcher.**
- The dispatcher is the module that given control of the CPU to the process selected by the **short-term scheduler**.
    - Switching context
    - Switching to user mode
    - Jumping to the proper location in the user program to restart that program

The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency.**

Dispatch Latency can be explained using the below figure:



Scanned with CamScanner

Scanned with CamScanner

2. Scheduling Criteria:

There are many different criteria to check when considering the **"best"** scheduling algorithm, they are:

1. **CPU Utilization**
2. **Throughput**
3. **Turnaround Time**
4. **Waiting Time**
5. **Response Time**

CPU Utilization: We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).

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 10 processes per second.

Turnaround Time: Turnaround time. From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the ***turnaround time***. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

Waiting time. The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. ***Waiting time*** is the sum of the periods spent waiting in the ready queue.

Response time. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user.

- Thus, another measure is the time from the submission of a request until the first response is produced.
- This measure, called *response time,* is the time it takes to start responding, not the time it takes to output the response.
- The turnaround time is generally limited by the speed of the output device.
- It is desirable to maximize CPU utilization and throughput and to minimize Turn around time, waiting time, and response time.
- For example, to guarantee that all users get good service, we may want to For
- example, to guarantee that all users get good service, we may want to minimize
- the maximum response time.minimize the maximum response time.

Scanned with CamScanner

Scanned with CamScanner

To decide which process to execute first and which process to execute last to achieve maximum CPU utilization, computer scientists have defined some algorithms, and they are:

1. **First Come First Serve(FCFS) Scheduling**
2. **Shortest-Job-First(SJF) Scheduling**
3. **Priority Scheduling**
4. **Round Robin(RR) Scheduling**
5. **Multilevel Queue Scheduling**
6. **Multilevel Feedback Queue Scheduling**

1. First Come First Serve Scheduling:

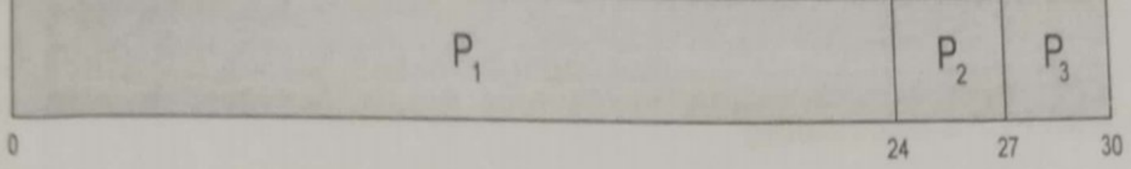
- By far the simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS) scheduling algorithm.**
- With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue.
- When a process enters the 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.
- The code for FCFS scheduling is simple to write and understand.
- The average waiting time under the FCFS policy, however, 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
PI	24
P2	3
P3	3

- If the processes arrive in the order PI, P2, P3, and are served in FCFS order, we get the result shown in the following **Gantt chart:**

Scanned with CamScanner

Scanned with CamScanner



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

➢ The waiting time is 0 milliseconds for process PI, 24 milliseconds for process P2, and 27 milliseconds for process P3.

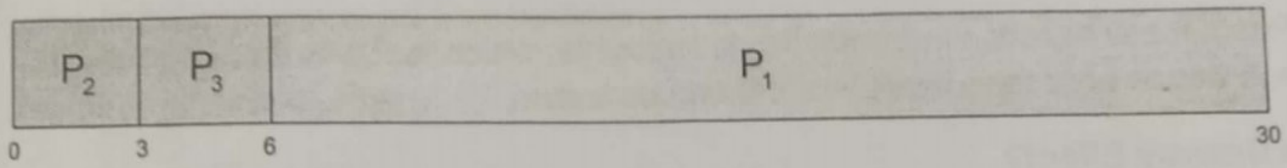
➢ Thus, the average waiting time is (0+ 24 + 27)/3 = 17 milliseconds.

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

➢ Suppose that the processes arrive in the order:

P_2 , P_3 , P_1

➢ The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$

➢ 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 process's CPU burst times vary greatly.

➢ In the "First come first serve" scheduling algorithm, as the name suggests, the process which arrives first, gets executed first, or we can say that the process which requests the CPU first, gets the CPU allocated first.

- First Come First Serve, is just like **FIFO**(First in First out) Queue data structure, where the data element which is added to the queue first, is the one who leaves the queue first.

- This is used in Batch Systems.

- It's **easy to understand and implement** programmatically, using a Queue data structure, where a new process enters through the **tail** of the queue, and the scheduler selects process from the **head** of the queue.

Scanned with CamScanner

Scanned with CamScanner

- A perfect real life example of FCFS scheduling is **buying tickets at ticket counter**.

Problems with FCFS Scheduling

Below we have a few shortcomings or problems with the FCFS scheduling algorithm:

1. It is **Non Pre-emptive** algorithm, which means the **process priority** doesn't matter.

If a process with very least priority is being executed, more like **daily routine backup** process, which takes more time, and all of a sudden some other high priority process arrives, like **interrupt to avoid system crash**, the high priority process will have to wait, and hence in this case, the system will crash, just because of improper process scheduling.

2. Not optimal Average Waiting Time.
3. Resources utilization in parallel is not possible, which leads to **Convoy Effect**, and hence poor resource(CPU, I/O etc) utilization.

What is Convoy Effect?

- Convoy Effect is a situation where many processes, who need to use a resource for short time are blocked by one process holding that resource for a long time.
- This essentially leads to poort utilization of resources and hence poor performance.

- The FCFS scheduling algorithm is **non preemptive**.
- Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting 1/0.
- The FCFS algorithm is thus particularly troublesome for time-sharing systems where it is important that each user get a share of the CPU at regular intervals.
- It would be disastrous to allow one process to keep the CPU for an extended period.

<u>Program for FCFS Scheduling:</u>

- Here we have a simple C++ program for processes with **arrival time** as 0.
- In the program, we will be calculating the **Average waiting time** and **Average turn around time** for a given array of **Burst times** for the list of processes.

```
/* Simple C++ program for implementation
of FCFS scheduling */
```

Scanned with CamScanner

Scanned with CamScanner

```
#include<iostream>

using namespace std;

// function to find the waiting time for all processes
void findWaitingTime(int processes[], int n, int bt[], int wt[])
{
   // waiting time for first process will be 0
   wt[0] = 0;

   // calculating waiting time
   for (int i = 1; i < n ; i++)
   {
      wt[i] =  bt[i-1] + wt[i-1];
   }
}

// function to calculate turn around time
void findTurnAroundTime( int processes[], int n, int bt[], int wt[], int tat[])
{
   // calculating turnaround time by adding
   // bt[i] + wt[i]
   for (int i = 0; i < n ; i++)
   {
      tat[i] = bt[i] + wt[i];
   }
}

// function to calculate average time
void findAverageTime( int processes[], int n, int bt[])
{
   int wt[n], tat[n], total_wt = 0, total_tat = 0;

   // function to find waiting time of all processes
   findWaitingTime(processes, n, bt, wt);

   // function to find turn around time for all processes
   findTurnAroundTime(processes, n, bt, wt, tat);

   // display processes along with all details
   cout << "Processes   "<< " Burst time   "<< " Waiting time   " << " Turn around
time\n";

   // calculate total waiting time and total turn around time
   for (int i = 0; i < n; i++)
   {
      total_wt = total_wt + wt[i];
      total_tat = total_tat + tat[i];
      cout << "   " << i+1 << "\t\t" << bt[i] <<"\t    "<< wt[i] <<"\t\t  " << tat[i] <<endl;
   }
```

Scanned with CamScanner

Scanned with CamScanner

```
    cout << "Average waiting time = "<< (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "<< (float)total_tat / (float)n;
}

// main function
int main()
{
    // process ids
    int processes[] = { 1, 2, 3, 4};
    int n = sizeof processes / sizeof processes[0];

    // burst time of all processes
    int  burst_time[] = {21, 3, 6, 2};

    findAverageTime(processes, n,  burst_time);

    return 0;
}
```

```
Processes Burst time Waiting time Turn around time
1 21 0 21
2 3 21 24
3 6 24 30
4 2 30 32


Average waiting time = 18.75
Average turn around time = 26.75
```

Here we have simple formulae for calculating various times for given processes:

Completion Time: Time taken for the execution to complete, starting from arrival time.

Turn Around Time: Time taken to complete after arrival. In simple words, it is the difference between the Completion time and the Arrival time.

Waiting Time: Total time the process has to wait before it's execution begins. It is the difference between the Turn Around time and the Burst time of the process.

For the program above, we have considered the **arrival time** to be 0 for all the processes, try to implement a program with variable arrival times.

Scanned with CamScanner

Scanned with CamScanner

First Come First Serve (FCFS)

Advantages:

- FCFS algorithm doesn't include any complex logic, it just puts the process requests in a queue and executes it one by one.
- Hence, FCFS is pretty simple and easy to implement.
- Eventually, every process will get a chance to run, so starvation doesn't occur.

Disadvantages:

- There is no option for pre-emption of a process. If a process is started, then CPU executes the process until it ends.
- Because there is no pre-emption, if a process executes for a long time, the processes in the back of the queue will have to wait for a long time before they get a chance to be executed.

2. Shortest Job First(SJF) Scheduling:

- A different approach to CPU scheduling is the **shortest-job-first (SJF) scheduling algorithm.**
- This algorithm associates with each process the length of the process's next CPU burst.
- When the CPU is available, it is assigned to the process that has the smallest next CPU burst.
- If two processes have the same length next CPU bursts, FCFS scheduling is used to break the tie.
- Note that a 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, rather than its total length.
- We use the term SJF because most people and textbooks use this term to refer to this type of scheduling.

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

Scanned with CamScanner

Scanned with CamScanner

0 3 9 16 24

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

- Using SJF scheduling, we would schedule these processes according to the following Gantt chart:
- The waiting time is 3 milliseconds for process PI, 16 milliseconds for process *P2,* 9 milliseconds for process *P3,* and 0 milliseconds for process P4 .
- 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.
- The SJF scheduling algorithm is provably *optimal,* in that it gives the minimum average waiting time for a given set of processes.
- Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process.
- Consequently, the *average* waiting time decreases.
- The real difficulty with the SJF algorithm is knowing the length of the next CPU request.
- The SJF algorithm can be either **preemptive or non preemptive.**
- The choice arises when a new process arrives at the ready queue while a previous process is till executing.
- The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.

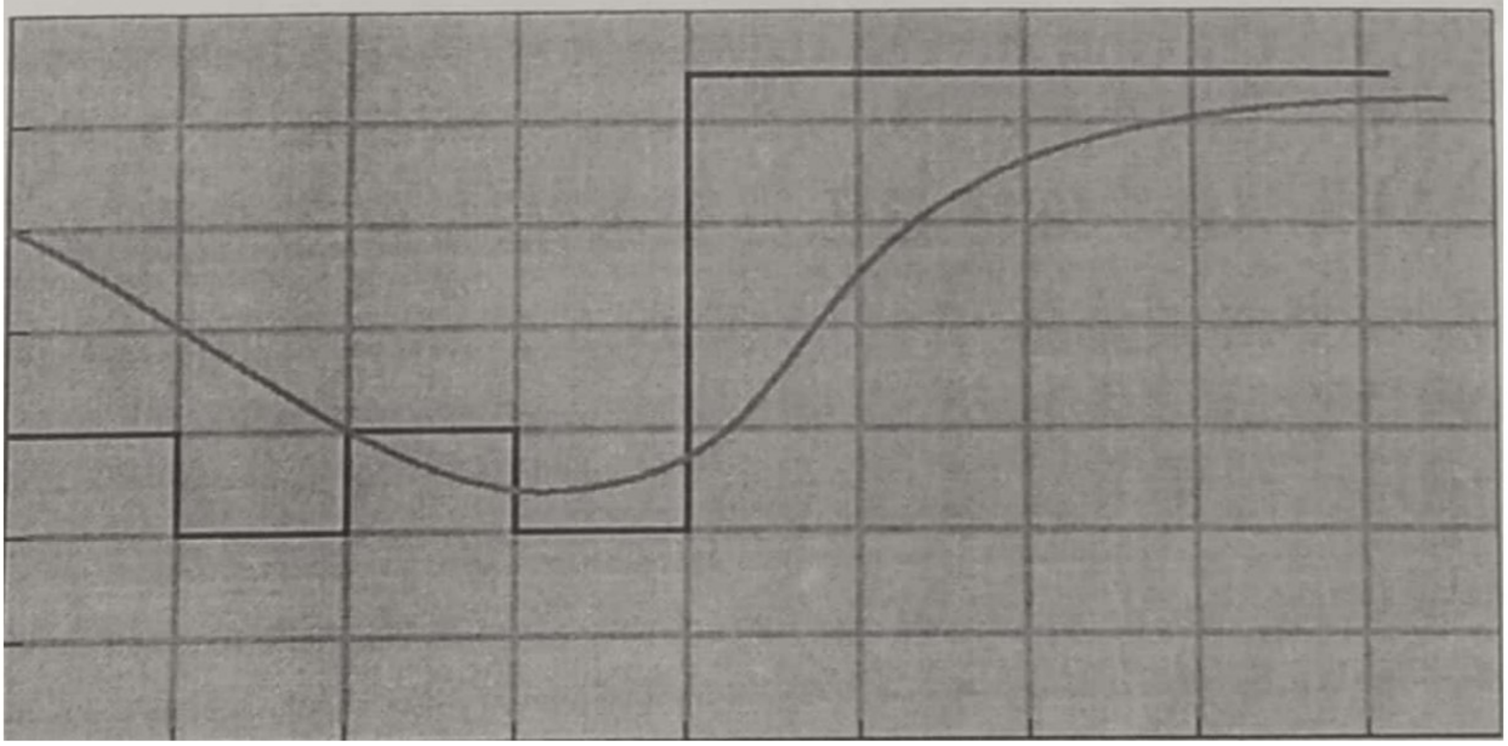
the currently executing process,

Scanned with CamScanner

Scanned with CamScanner

1 5 10 17

rage waiting time = [(10-1)+(1-1)+(17-2)+(5-3))]/4 = 26/4 = 6.5 r
ess P1 is started at time 0, since it is the only process in
ess P2 arrives at time 1.
remaining time for process *P1* (7 milliseconds) is larger tha
ired by process *P2* (4 milliseconds), so process *P1* is pree
ess *P2* is scheduled.
average waiting time for this example is ((10 - 1) + (1 1) +
/4 = 26/4 = 6.5 milliseconds.
preemptive SJF scheduling would result in an average wai
milliseconds.
an only estimate the length
an be done by using the length of previous CPU bursts, using
veraging



Scanned with CamScanner

Scanned with CamScanner

- ➢ $\tau_{n+1} = \tau_n$
- ➢ Recent history does not count.
- ➢ $\alpha = 1$
- ➢ $\tau_{n+1} = \alpha\, t_n$
- ➢ Only the actual last CPU burst counts.
- ➢ If we expand the formula, we get:
- ➢ $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_n - 1 + \ldots + (1 - \alpha)^j \alpha\, t_{n-j} + \ldots + (1 - \alpha)^{n+1} \tau_0$
- ➢ Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor.

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

- This is the best approach to minimize waiting time.
- This is used in Batch Systems.
- It is of two types:
  1. Non Pre-emptive
  2. Pre-emptive
- 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 jobs/processes are available at **Shortest Job First (SJF)**

Starting with the **Advantages:** of Shortest Job First scheduling algorithm.

- According to the definition, short processes are executed first and then followed by longer processes.
- The throughput is increased because more processes can be executed in less amount of time.

And the **Disadvantages:**

- The time taken by a process must be known by the CPU beforehand, which is not possible.
- Longer processes will have more waiting time, eventually they'll suffer starvation.

Scanned with CamScanner

Scanned with CamScanner

3. Priority Scheduling

- Priority is assigned for each process
- Process with highest priority is executed first and so on.
- Processes with same priority are executed in FCFS manner.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the **highest priority** (smallest integer ≡ highest priority)
- **Preemptive**
- **Nonpreemptive**
- SJF is priority scheduling where priority is the next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

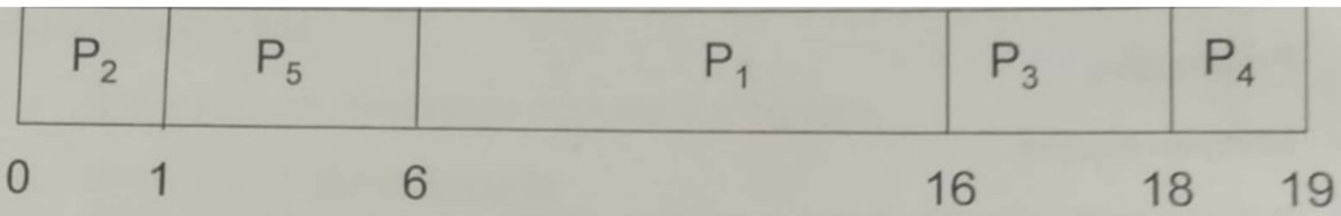
The SJF algorithm is a special case of the general priority scheduling algorithm.

- 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.
- Note that we discuss 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. 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

Scanned with CamScanner

Scanned with CamScanner



Average waiting time = 8.2 msec

Priority based Scheduling

Advantages of Priority Scheduling:

- The priority of a process

Scanned with CamScanner

Scanned with CamScanner

4. Round Robin Scheduling:

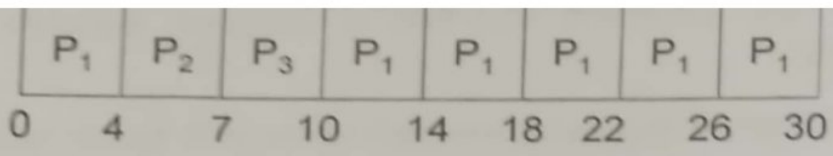
- The round-robin (RR) scheduling algorithm is designed especially for timesharing systems.
- It is similar to FCFS scheduling, but preemption is added to switch between processes.
- A small unit of time, called a time quantum or time slice, is defined.
- A time quantum is generally from 10 to 100 milliseconds.
- 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 1 time quantum.
- A fixed time is allotted to each process, called **quantum**, for execution.
- Once a process is executed for given time period that process is preempted and other process executes for given time period.
- Context switching is used to save states of preempted processes.
- 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**
P1	24
P2	3
P3	3

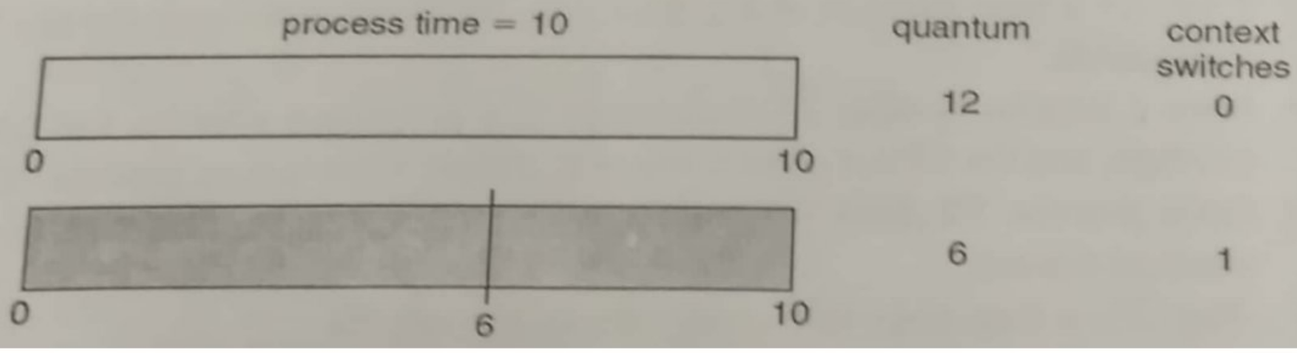
- ➢ If we use a time quantum of 4 milliseconds, then process PI gets the first 4 milliseconds.
- ➢ Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2.
- ➢ Since process P2 does not need 4 milliseconds, it quits before its time quantum expires.
- ➢ The CPU is then given to the next process, process *P3*.
- ➢ Once each process has received 1 time quantum, the CPU is returned to process PI for an additional time quantum. The resulting RR schedule is

Scanned with CamScanner

Scanned with CamScanner

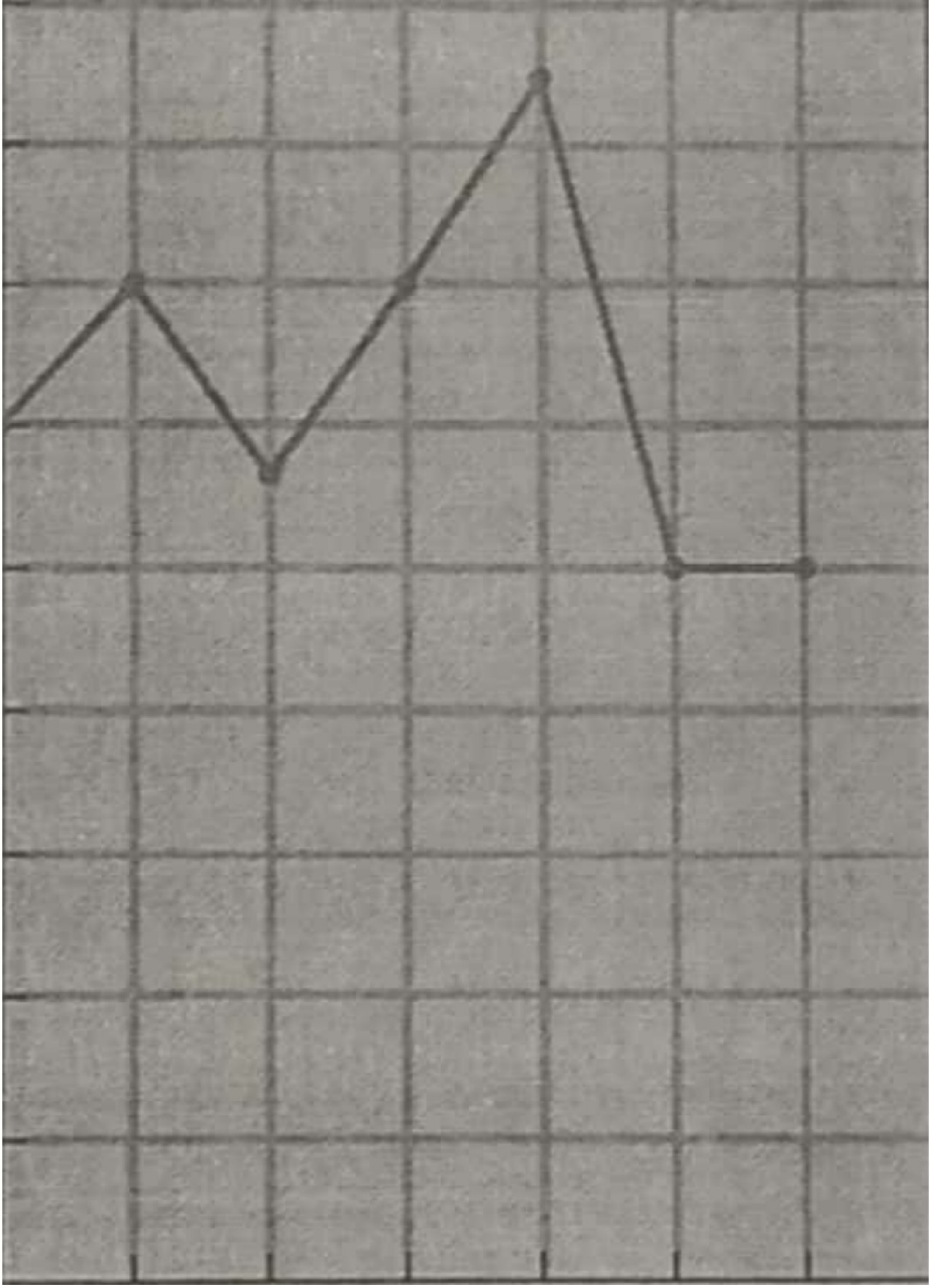


- The **average waiting time is 17/3 = 5.66 milliseconds.**
- In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row.
- If a 5.3 Scheduling Algorithms 165 process's CPU burst exceeds 1 time quantum, that process is **preempted** and is put back in the ready queue.
- The RR scheduling algorithm is thus preemptive.
- If there are 11 processes in the ready queue and the time quantum is *q*, then each process gets **1/n** of the CPU time in chunks of at most *q* time units.
- Each process must wait no longer than **(n - 1) x q** time units until its next time quantum.
- **For example**, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.



Scanned with CamScanner

Scanned with CamScanner



Scanned with CamScanner

Scanned with CamScanner

Round Robin (RR)

Advantages: of using the Round Robin Scheduling:

- Each process is served by the CPU for a fixed time quantum, so all processes are given the same priority.
- Starvation doesn't occur because for each round robin cycle, every process is given a fixed time to execute. No process is left behind.

Disadvantages:

- The throughput in RR largely depends on the choice of the length of the time quantum. If time quantum is longer than needed, it tends to exhibit the same behavior as FCFS.
- If time quantum is shorter than needed, the number of times that CPU switches from one process to another process, increases. This leads to decrease in CPU efficiency.

4. 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 (or interactive)** processes and **background (or batch)** processes.
- ➢ These two types of processes have different response-time requirements, and so might have different scheduling needs.
- ➢ In addition, foreground processes may have priority over background processes.
- ➢ **A multi-level 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 Round Robin algorithm, while the background queue is scheduled by an **FCFS algorithm**.
- ➢ In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.
- ➢ **For example:** The foreground queue may have absolute priority over the background queue.

Scanned with CamScanner

Let us consider an example of a multilevel queue-scheduling algorithm with five queues:

1. System Processes
2. Interactive Processes
3. Interactive Editing Processes
4. Batch Processes
5. Student Processes

- Each queue has absolute priority over lower-priority queues.
- No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty.
- If an interactive editing process entered the ready queue while a batch process was running, the batch process will be preempted.

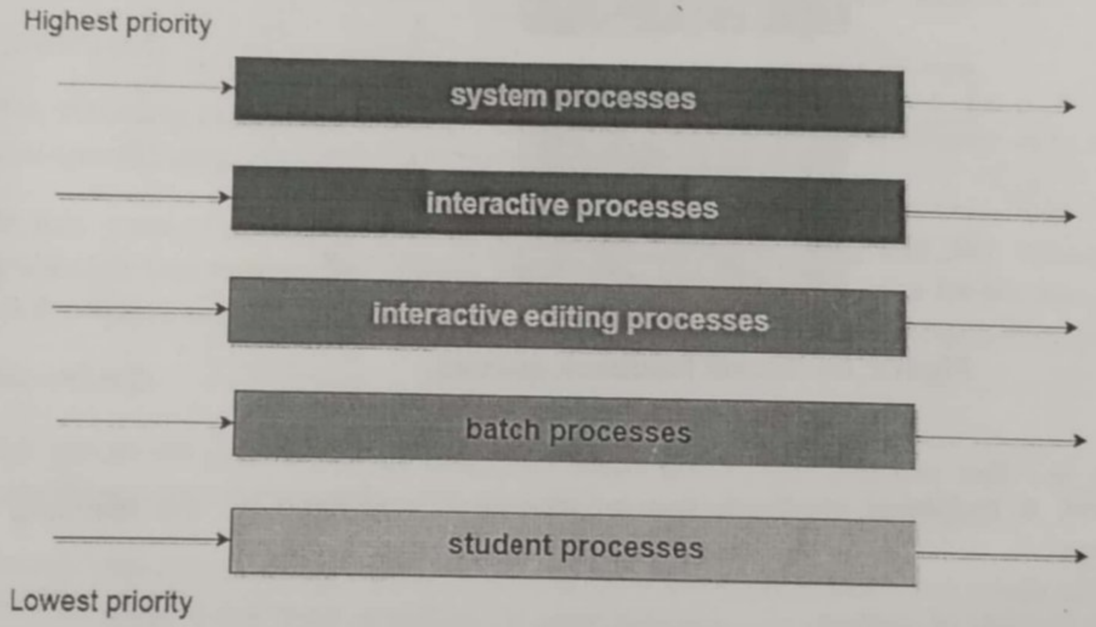


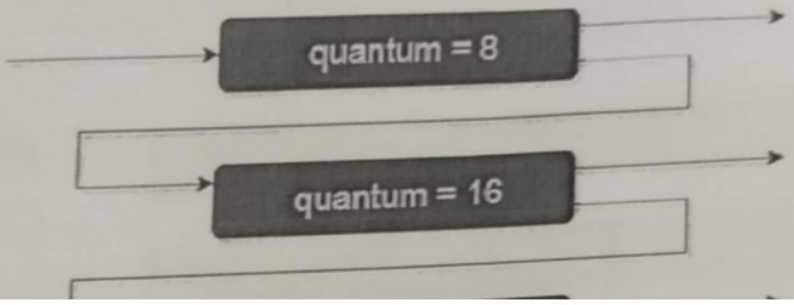
Figure: Multilevel queue scheduling.

- Another possibility is to time-slice among the queues.
- Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes.

Scanned with CamScanner

Scanned with CamScanner

- In a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system.
- Processes do not move between queues.
- This setup has the advantage of low scheduling overhead, but the disadvantage of being inflexible.
- Multilevel feedback queue scheduling, however, allows a process to move between queues.
- The idea is to separate processes with different CPU-burst characteristics.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- Similarly, 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.



Scanned with CamScanner

Scanned with CamScanner

- The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm.
- It can be configured to match a specific system under design.
- Unfortunately, it also requires some means of selecting values for all the parameters to define the best scheduler.
- Although a multilevel feedback queue is the **most general scheme**, it is also the **most complex.**

Comparison of Scheduling Algorithms

By now, you must have understood how CPU can apply different scheduling algorithms to schedule processes. Now, let us examine the advantages and disadvantages of each scheduling algorithms that we have studied so far.

Usage of Scheduling Algorithms in Different Situations

Every scheduling algorithm has a type of a situation where it is the best choice. Let's look at different such situations:

Situation 1:

The incoming processes are short and there is no need for the processes to execute in a specific order.

In this case, FCFS works best when compared to SJF and RR because the processes are short which means that no process will wait for a longer time. When each process is executed one by one, every process will be executed eventually.

Situation 2:

The processes are a mix of long and short processes and the task will only be completed if all the processes are executed successfully in a given time.

Round Robin scheduling works efficiently here because it does not cause starvation and also gives equal time quantum for each process.

Situation 3:

The processes are a mix of user based and kernel based processes.

Priority based scheduling works efficiently in this case because generally kernel based processes have higher priority when compared to user based processes.

Scanned with CamScanner

Scanned with CamScanner

- If multiple CPUs are available, load sharing among them becomes possible.
- The scheduling problem becomes more complex.
- We concentrate in this discussion on systems in which the processors are identical (homogeneous) in terms of their functionality.
- We can use any available processor to run any process in the queue.

1.Approaches to 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.

- CPU scheduling more complex when multiple CPUs are available
- Homogeneous processors within a multiprocessor
- Two approaches: **Asymmetric** processing and **symmetric** processing.

- **Asymmetric multiprocessing**
- **Symmetric multiprocessing (SMP)** Processor affinity – process has affinity for processor on which it is currently running
- **soft affinity**
- **hard affinity**

Asymmetric multiprocessing (ASMP)

- One processor handles all scheduling decisions, I/O processing, and other system activities
- The other processors execute only user code
- Because only one processor accesses the system data structures, the need for data sharing is reduced

Symmetric multiprocessing (SMP)

- Each processor schedules itself
- All processes may be in a common ready queue or each processor may have its own ready queue
- Either way, each processor examines the ready queue and selects a process to execute
- Efficient use of the CPUs requires load balancing to keep the workload evenly distributed
- In a **Push** migration approach, a specific task regularly checks the processor loads and redistributes the waiting processes as needed
- In a **Pull** migration approach, an idle processor pulls a waiting job from the queue of a busy processor

Scanned with CamScanner

Scanned with CamScanner

- Virtually all modern operating systems support SMP, including Windows XP, Solaris, Linux, and Mac OS X.

2. Processor Affinity:

- Processor affinity – process has affinity for processor on which it is currently running
- **soft affinity**
- **hard affinity**

- The high cost of invalidating and re-populating 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, meaning that a process has an affinity for the processor on which it is currently running.

- Processor affinity takes several forms.

- 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, it is possible for a process to migrate between processors. Some systems -such as Linux-also provide system calls that support **hard affinity**, thereby allowing a process to specify that it is not to migrate to other processors.

5. Real-Time CPU Scheduling:

Real-time system:

- "A real-time system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computation, but also on the physical instant at which these results are produced".
- "A real-time system is a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment".
- Real-time computing is divided into two types.
- **Hard real time**
- **Soft real time**

Scanned with CamScanner

Scanned with CamScanner

- **Hard real time:** Hard real time systems are required to complete a critical task within a guaranteed amount of time. Generally, a process is submitted along with a statement of the amount of time in which it needs to complete or perform I/O.

- The scheduler then either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible. This is known as **resource reservation.**

- **Soft real-time:** Soft real-time computing is less restrictive. It requires that critical processes receive priority over less fortunate ones. Although adding soft real-time functionality to a time-sharing system may cause an unfair allocation of resoures and unfor

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**. This function involves:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program from where it left last time.

The dispatcher should be as fast as possible, given that it is invoked during every process switch. The time taken by the dispatcher to stop one process and start another process is known as the **Dispatch Latency**. Dispatch Latency can be explained using the below figure:

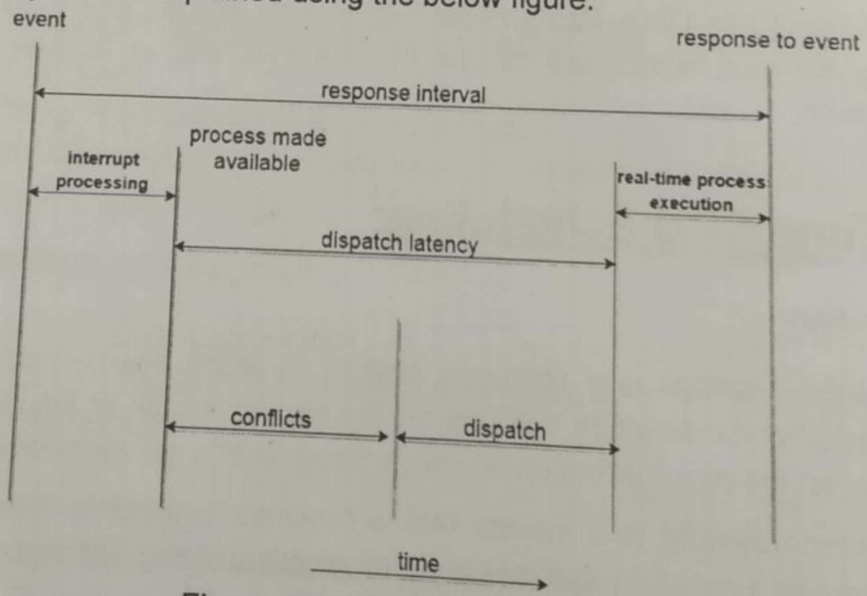


Figure: Dispatch Latency

Scanned with CamScanner

CHAPTER – 6

Process Synchronization

TOPICS:

- **The critical – section problem**
- **Synchronization Hardware**
- **Semaphores**
- **Classic problems of synchronization**
- **Critical Regions**
- **Monitors**

Introduction to process synchronization:

Process Synchronization was introduced to handle problems that arose while multiple process executions.

Process is categorized into two types on the basis of synchronization and these are given below:

- Independent Process
- Cooperative Process

Independent Processes

Two processes are said to be independent if the execution of one process does not affect the execution of another process.

Cooperative Processes

Two processes are said to be cooperative if the execution of one process affects the execution of another process. These processes need to be synchronized so that the order of execution can be guaranteed.

It is the task phenomenon of coordinating the execution of processes in such a way that no two processes can have access to the same shared data and resources.

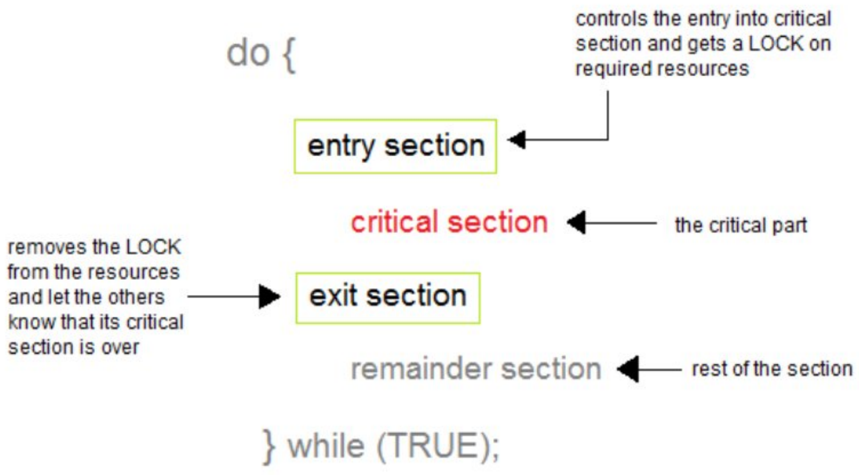
- It is a procedure that is involved in order to preserve the appropriate order of execution of cooperative processes.
- In order to synchronize the processes, there are various synchronization mechanisms.
- Process Synchronization is mainly needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or any data at the same time.

Scanned with CamScanner

- Process synchronization is the technique to overcome the problem of concurrent access to shared data which can result in data inconsistency. A cooperating process is the one which can affect or be affected by other process which will lead to inconsistency in processes data therefore Process synchronization is required for consistency of data.

❖ The Critical-Section Problem:

- Every process has a reserved segment of code which is known as **Critical Section**. In this section, process can change common variables, update tables, write files, etc.
- The key point to note about critical section is that when one process is executing in its critical section, no other process can execute in its critical section.
- Each process must request for permission before entering into its critical section and the section of a code implementing this request is the **Entry Section**, the end of the code is the **Exit Section** and the remaining code is the **remainder section.**
- A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section.
- If any other process also wants to execute its critical section, it must wait until the first one finishes.
- The entry to the critical section is mainly handled by wait() function while the exit from the critical section is controlled by the signal() function.


do {
controls the entry into critical section and gets a LOCK on required resources
entry section
critical section
the critical part
removes the LOCK from the resources and let the others know that its critical section is over
exit section
remainder section
rest of the section
} while (TRUE);

Scanned with CamScanner

Entry Section

- In this section mainly the process requests for its entry in the critical section.

Exit Section

- This section is followed by the critical section.

The solution to the Critical Section Problem

A solution to the critical section problem must satisfy the following **three conditions**:

1. Mutual Exclusion

Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. Progress

If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

3. Bounded Waiting

After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, the system must grant the process permission to get into its critical section.

Synchronization Hardware:

It is implemented using two types of instructions −

- Test and Set()
- swap()

Test and Set () is a hardware solution to solve the problem of synchronization. In this, there is a shared variable which is shared by multiple processes known as Lock which can have one value from 0 and 1 where 1 represents Lock gained and 0 represents Lock released.

Whenever the process is trying to enter their critical sections they need to enquire about the value of lock. If the value of lock is 1 then they have to wait until the value of lock won't get changed to 0.

Scanned with CamScanner

Given below is the mutual-exclusion implementation with TestAndSet()

```
do
{
        while(TestAndSetLock(& lock))
        ; //do nothing
                Critical section

        Lock = FALSE;
                Remainder section

}while(TRUE);
```

- In Synchronization hardware, we explore several more solutions to the critical-section problem using techniques ranging from hardware to software based APIs available to application programmers.
- These solutions are based on the premise of locking; however, the design of such locks can be quite sophisticated.
- These Hardware features can make any programming task easier and improve system efficiency. Here, we present some simple hardware instructions that are available on many systems and show how they can be used effectively in solving the critical-section problem.
- If we could prevent interrupts from occurring while a shared variable was being modified. The critical-section problem could be solved simply in a uniprocessor environment.
- In this manner, we would be assuring that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.
- This is the approach taken by non-preemptive kernels. But unfortunately, this solution is not as feasible in a multiprocessor environment. Since the message is passed to all the processors, disabling interrupts on a multiprocessor can be time consuming.
- We may use these special instructions to solve the critical-section problem in a relatively simple manner. Now we abstract the main concepts behind these types of instructions. The TestAndSet() instruction can be defined as shown in below code.

```
boolean test and set(boolean *target){

   boolean rv = *target;
```

Scanned with CamScanner

```
    *target = true;

    return rv;

}
```

Definition of the test and set() instruction.

The essential characteristic is that this instruction is executed atomically. So, if two TestAndSet C) instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order. we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false, if the machine supports the TestAndSet () instruction.

Semaphores:

Semaphores

Semaphore is a synchronization tool that is used to overcome the problems generated by TestAndSet() and Swap() instructions. A semaphore S is an integer variable that can be accessed through two standard atomic operations that are wait() and signal()

Function for wait():

```
wait(S) {
   While S <= 0
   ; // no operation
   S--;
}
```

Function for Signal():

```
signal(S) {
   S++;
}
```

When one process is modifying the value of semaphore then no other process can simultaneously manipulate the same semaphore value.

Usage

Operating systems often distinguish between counting and binary semaphores. The value of a **counting semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range only between 0 and 1. On some systems, binary semaphores are known as **mutex locks**, as they are locks that provide mutual exclusion. We can use binary semaphores to deal with the critical-section problem for multiple processes

Scanned with CamScanner

```
do
{
     waiting(mutex);
          Critical section
     signal(mutex);
          Remainder section
}while(TRUE);
```

Implementation:

The main disadvantage of the semaphore definition given here is that it requires busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.

This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a spinlock because the process "spins" while waiting for the lock.

Properties of Semaphores

1. It's simple and always have a non-negative integer value.
2. Works with many processes.
3. Can have many different critical sections with different semaphores.
4. Each critical section has unique access semaphores.
5. Can permit multiple processes into the critical section at once, if desirable.

Types of Semaphores

Semaphores are mainly of two types in Operating system:

1. **Binary Semaphore:**

   It is a special form of semaphore used for implementing mutual exclusion, hence it is often called a **Mutex**. A binary semaphore is initialized to 1 and only takes the values 0 and 1 during the execution of a program. In Binary Semaphore, the wait operation works only if the value of semaphore = 1, and the signal operation

Scanned with CamScanner

succeeds when the semaphore= 0. Binary Semaphores are easier to implement than counting semaphores.

2. **Counting Semaphores:**

   These are used to implement **bounded concurrency**. The Counting semaphores can range over an **unrestricted domain**. These can be used to control access to a given resource that consists of a finite number of Instances. Here the semaphore count is used to indicate the number of available resources. If the resources are added then the semaphore count automatically gets incremented and if the resources are removed, the count is decremented. Counting Semaphore has no mutual exclusion.

Advantages of Semaphores

- With the help of semaphores, there is a flexible management of resources.
- Semaphores are machine-independent and they should be run in the machine-independent code of the microkernel.
- Semaphores do not allow multiple processes to enter in the critical section.
- They allow more than one thread to access the critical section.
- As semaphores follow the mutual exclusion principle strictly and these are much more efficient than some other methods of synchronization.

Disadvantages of Semaphores

- One of the biggest limitations is that semaphores may lead to priority inversion; where low priority processes may access the critical section first and high priority processes may access the critical section later.
- To avoid deadlocks in the semaphore, the Wait and Signal operations are required to be executed in the correct order.
- Using semaphores at a large scale is impractical; as their use leads to loss of modularity and this happens because the wait() and signal() operations prevent the creation of the structured layout for the system.
- Their use is not enforced but is by convention only.
- With improper use, a process may block indefinitely. Such a situation is called **Deadlock**.

Classical Problems of Synchronization:

Semaphore can be used in other synchronization problems besides Mutual Exclusion.

Below are some of the classical problems depicting flaws of process synchronization in systems where cooperating processes are present.

We will discuss the following three problems:

Scanned with CamScanner

1. Bounded Buffer (Producer-Consumer) Problem
2. Dining Philosophers Problem
3. The Readers Writers Problem

Bounded Buffer Problem:

Because the buffer pool has a maximum size, this problem is often called the **Bounded buffer problem**.

- This problem is generalized in terms of the **Producer-Consumer problem**, where a **finite** buffer pool is used to exchange messages between producer and consumer processes.
- The solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.
- This Producers mainly produce a product and consumers consume the product, but both can use one of the containers each time.
- The main complexity of this problem is that we must have to maintain the count for both empty and full containers that are available.

```
do {

II produce an item in nextp

wait (empty) ;

wait (mutex) i

II add nextp to buffer

signal (mutex) ;

signal (full) ;

 }while (TRUE) i
```

Figure :The structure of the producer process.

```
 do {

wait(full) ;

 wait (mutex) ;

II remove an item from buffer to nextc
```

Scanned with CamScanner

```
signal (mutex) ;

 signal (empty) ;

II consume the item in nextc

}while (TRUE);
```

Figure : The structure of the consumer process.

Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization. Let's start by understanding the problem here, before moving on to the solution and program code.

There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.

Dining philosopher's problem:

- The dining philosopher's problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.
- There are five philosophers sitting around a table, in which there are five chopsticks/forks kept beside them and a bowl of rice in the center, When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

The dining philosophers problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.

```
do {

wait (chopstick[i] );

wait (chopstick[(i+l) % 5]);

II eat

signal (chopstick[i] );
```

Scanned with CamScanner

```
signal (chopstick[(i+l) % 5]);

II think

}while (TRUE);
```

Figure :The structure of philosopher i.

At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever

Figure : The situation of the dining philosophers

Scanned with CamScanner

The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

The Readers Writers Problem

- In this problem, there are some processes (called **readers**) that only read the shared data, and never change it, and there are other processes (called **writers**) that may change the data in addition to reading, or instead of reading it.
- There is various type of readers-writers problems, most centered on relative priorities of readers and writers.
- The main complexity of this problem occurs from allowing more than one reader to access the data at the same time.Readers writer problem is another example of a classic synchronization problem.
- There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**.
- Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource.
- When a **writer** is writing data to the resource, no other process can access the resource.
- A **writer** cannot write to the resource if there are non zero number of readers accessing the resource at that time.

The semaphore wrt functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last

```
do {

 wait (wrt) ;

II writing is performed

signal (wrt) ;

}while (TRUE);
```

Figure :The structure of a writer process.

Scanned with CamScanner

```
do {
wait (mutex) ;
readcount++;
 if (readcount ==1)
 wait (wrt);
 signal (mutex) ;
II reading is performed
wait (mutex) i
 readcount - -;
 if (readcount = =0)
 signal (wrt) ;
 signal (mutex) ;
}while (TRUE) ;
```

Figure : The structure of a reader process.

reader that enters or exits the critical section. It is not used by readers.

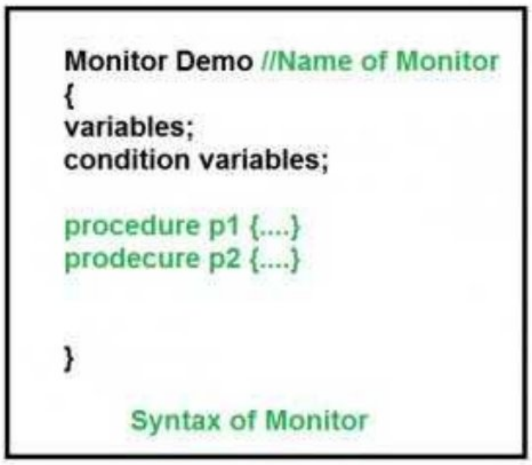
Monitors:

- Another high-level synchronization construct is the monitor type.
- The monitor is one of the ways to achieve Process synchronization.
- The monitor is supported by programming languages to achieve mutual exclusion between processes.
- **For example** Java Synchronized methods. Java provides wait() and notify() constructs.

1. It is the collection of condition variables and procedures combined together in a special kind of module or a package.
2. The processes running outside the monitor can't access the internal variable of the monitor but can call procedures of the monitor.
3. Only one process at a time can execute code inside monitors.

Scanned with CamScanner

Syntax:



```
Monitor Demo //Name of Monitor
{
variables;
condition variables;

procedure p1 {....}
prodecure p2 {....}


}
```

Syntax of Monitor

Condition Variables:
Two different operations are performed on the condition variables of the monitor.
Wait.

signal.

condition x, y;

Wait operation
x.wait() : Process performing wait operation on any condition variable are suspended. The suspended processes are placed in block queue of that condition variable.

Note: Each condition variable has its unique block queue.

Signal operation
x.signal(): When a process performs signal operation on condition variable, one of the blocked processes is given chance.

If (x block queue empty)

Scanned with CamScanner

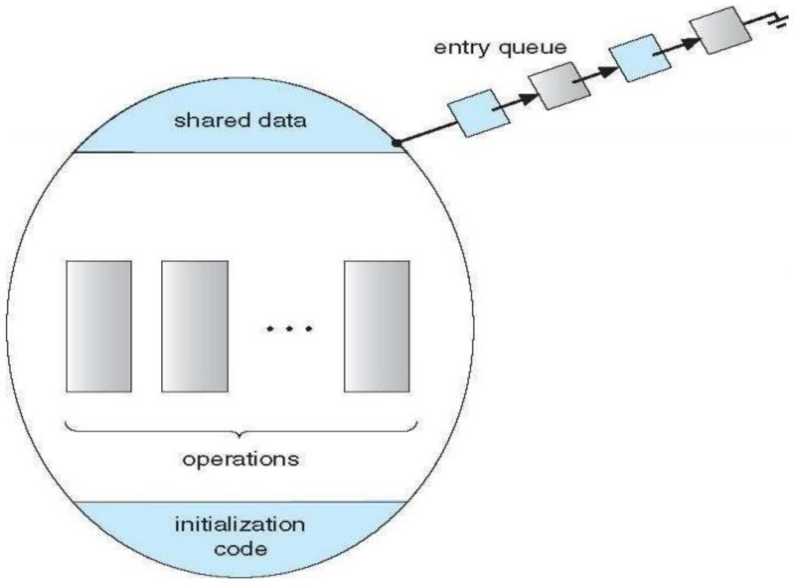


Figure: Schematic view of a monitor

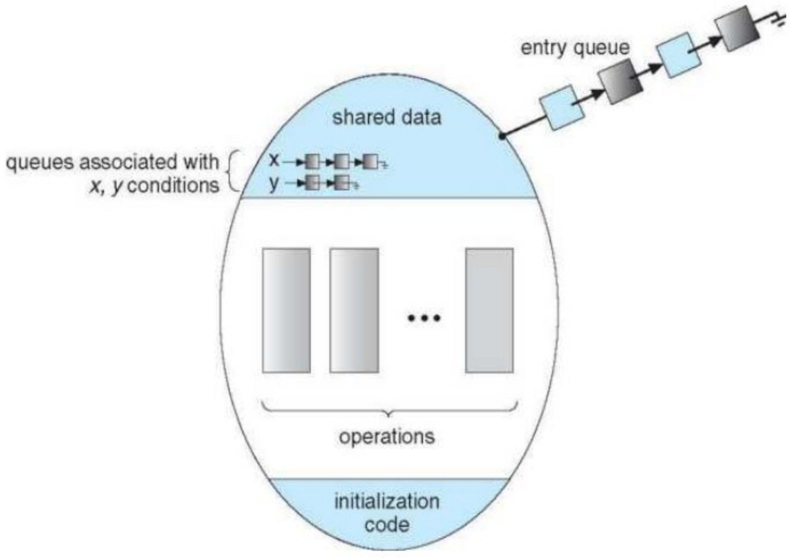


Figure: Monitor with condition variable

Characteristics of Monitors.

- Inside the monitors, we can only execute one process at a time.
- Monitors are the group of procedures, and condition variables that are merged together in a special type of module.

Scanned with CamScanner

- If the process is running outside the monitor, then it cannot access the monitor's internal variable. But a process can call the procedures of the monitor.
- Monitors offer high-level of synchronization
- Monitors were derived to simplify the complexity of synchronization problems.
- There is only one process that can be active at a time inside the monitor.

Components of Monitor

There are four main components of the monitor:

- Initialization
- Private data
- Monitor procedure
- Monitor entry queue

Initialization: – Initialization comprises the code, and when the monitors are created, we use this code exactly once.

Private Data: – Private data is another component of the monitor. It comprises all the private data, and the private data contains private procedures that can only be used within the monitor. So, outside the monitor, private data is not visible.

Monitor Procedure: – Monitors Procedures are those procedures that can be called from outside the monitor.

Monitor Entry Queue: – Monitor entry queue is another essential component of the monitor that includes all the threads, which are called procedures.

Advantages of Monitor:

Monitors have the advantage of making parallel programming easier and less error prone than using techniques such as semaphore.

Disadvantages of Monitor:

- Monitors have to be implemented as part of the programming language.

The compiler must generate code for them. This gives the compiler the additional burden of having to know what operating system facilities are available to control access to critical sections in concurrent processes. Some languages that do support monitors are Java, C#, Visual Basic, Ada .

Scanned with CamScanner

CHAPTER -7

DEADLOCKS

TOPICS:

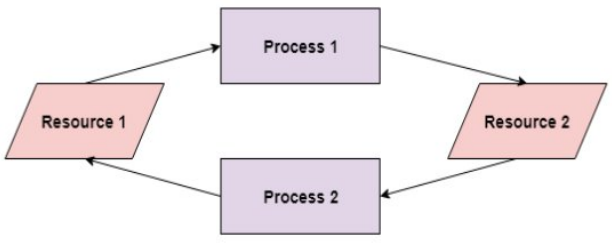
- **Deadlock Characterization**
- **Deadlock Handling**
- **Deadlock Prevention**
- **Deadlock Avoidance**
- **Deadlock Detection**
- **Deadlock Recovery**

Deadlock definition:

In a multiprogramming environment, several processes may complete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting pro -cesses. This situation is called a **deadlock.**

A deadlock is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function.

A deadlock happens in operating system when two or more processes need some resource to complete their execution that is held by the other process.



Deadlock in Operating System

- In the above diagram, the process 1 has resource 1 and needs to acquire resource 2. Similarly process 2 has resource 2 and needs to acquire resource 1.

Scanned with CamScanner

- Process 1 and process 2 are in deadlock as each of them needs the other's resource to complete their execution but neither of them is willing to relinquish their resources.

Deadlock Characterization:

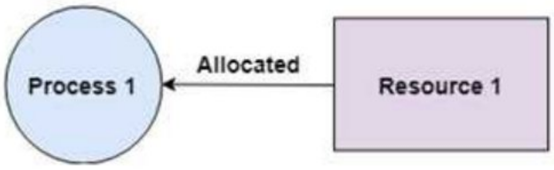
Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in the system.

The four conditions are given as follows −

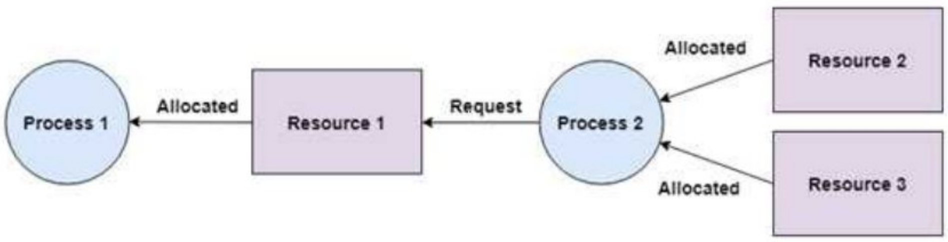
- **Mutual Exclusion**

  There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.



- **Hold and Wait**

  A process can hold multiple resources and still request more resources from other processes which are holding them. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.

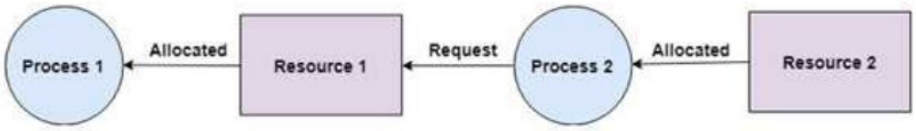


- **No Preemption**

  A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt

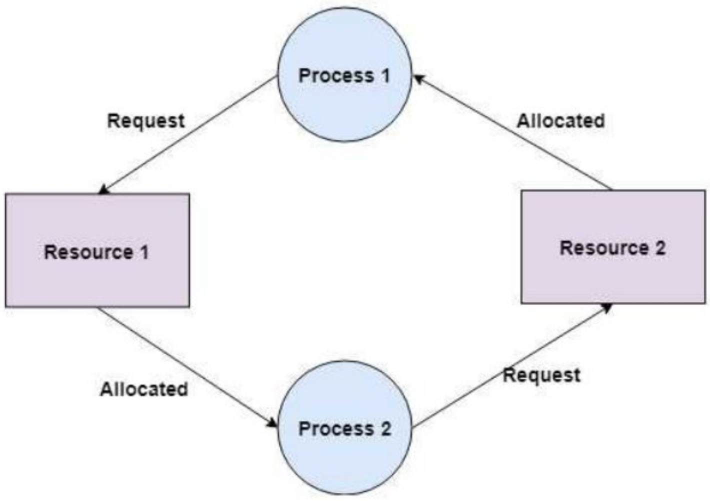
Scanned with CamScanner

Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.



- **Circular Wait**

  A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain. For example: Process 1 is allocated Resource2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.



Resource – Allocation Graph :

Deadlocks can be described more precisely in terms of a directed graphs called a **system Resource Allocation Graph.**

- This Graph acts as the pictorial representation of the state of the system.
- The Resource Allocation graph mainly consists of a **set of vertices V** and **a set of Edges E.**

Scanned with CamScanner

- This graph mainly contains all the information related to the processes that are holding some resources and also contains the information of the processes that are waiting for some more resources in the system.
- Also, this graph contains all the information that is related to all the instances of the resources which means the information about available resources and the resources which are being used by the process
- In this graph, the **circle is used to represent the process**, and the **rectangle is used to represent the resource.**

Components of Resource Allocation Graph

Given below are the components of RAG:

1. Vertices
2. Edges

1.Vertices

There is two kinds of vertices used in the resource allocation graph and these are:

- Process Vertices
- Resource Vertices

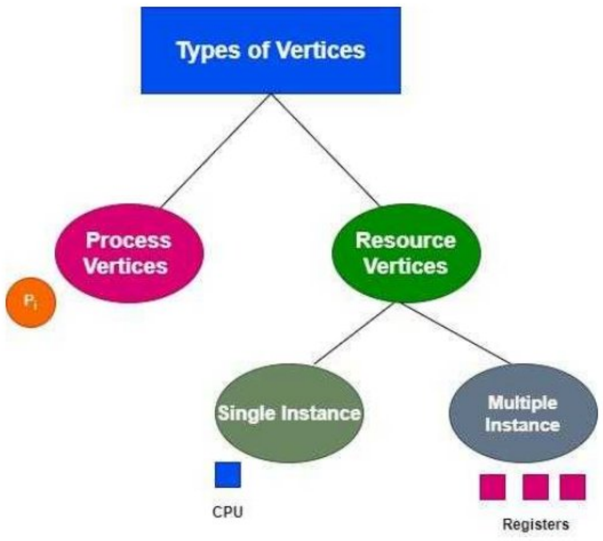
Process Vertices

These vertices are used in order to represent process vertices. The circle is used in order to draw the process vertices and the name of the process is mentioned inside the circle.

Resource Vertices

These vertices are used in order to represent resource vertices. The rectangle is used in order to draw the resource vertices and we use dots inside the circle to mention the number of instances of that resource.

Scanned with CamScanner



In the system, there may exist a number of instances and according to them, there are two types of resource vertices and these are single instances and multiple instances.

Single Instance

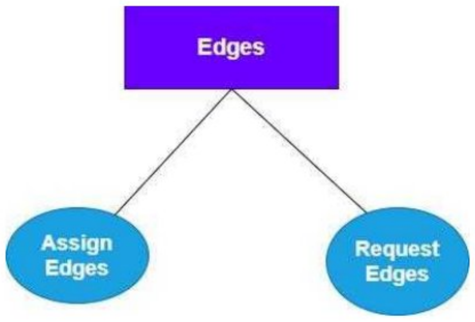
In a single instance resource type, there is a single dot inside the box. The single dot mainly indicates that there is one instance of the resource.

Multiple Instance

In multiple instance resource types, there are multiple dots inside the box, and these Multiple dots indicate that there are multiple instances of the resources.

2. Edges

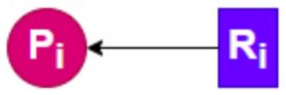
In the Resource Allocation Graph, Edges are further categorized into two:



1. Assign Edges

Scanned with CamScanner

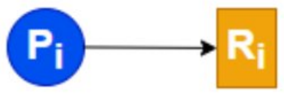
Assign Edges are mainly used to represent the allocation of resources to the process. We can draw assign edges with the help of an arrow in which mainly the arrowhead points to the process, and the process mainly tail points to the instance of the resource.



In the above Figure, the resource is assigned to the process

2. Request Edges

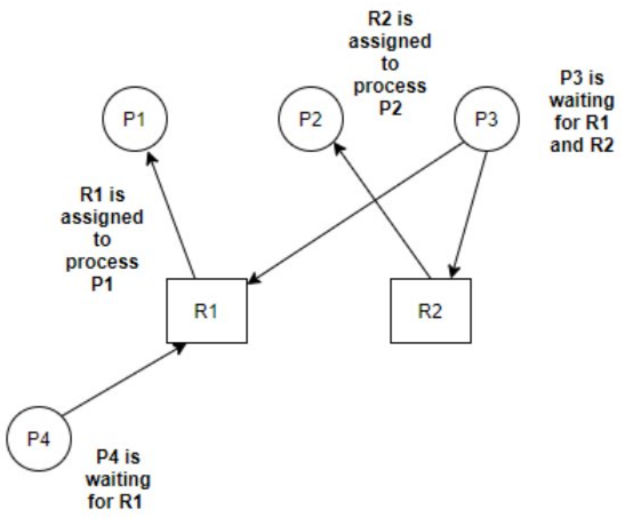
Request Edge is mainly used to signify the waiting state of the process. Likewise in assigned edge, an arrow is used to draw an arrow edge. But Here the arrowhead points to the instance of a resource, and the tail of the process points to the process.



In the above figure, the process is requesting a resource

Single Instance RAG Example

Suppose there are Four Processes P1, P2, P3, P4, and two resources R1 and R2, where P1 is holding R1 and P2 is holding R2, P3 is waiting for R1 and R2 while P4 is waiting for resource R1.


R2 is assigned to process P2
P1
P2
P3
P3 is waiting for R1 and R2
R1 is assigned to process P1
R1
R2
P4
P4 is waiting for R1

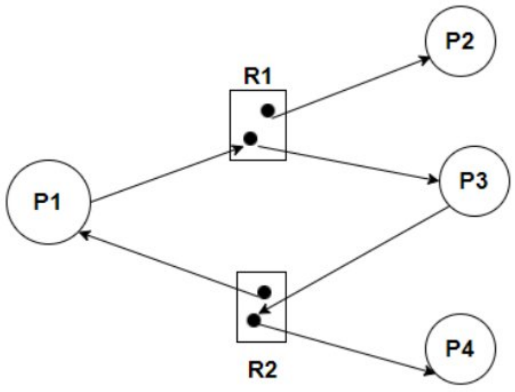
Scanned with CamScanner

In the above example, there is no circular dependency so there are no chances for the occurrence of deadlock.

Thus having cycled in single-instance resource type must be the sufficient condition for deadlock.

Multiple Instance RAG Example

Suppose there are four processes P1, P2, P3, P4 and there are two instances of resource R1 and two instances of resource R2:



Multiple Instance Resource Allocation graph with a cycle but no deadlock

One instance of R2 is assigned to process P1 and another instance of R2 is assigned to process P4, Process P1 is waiting for resource R1.

One instance of R1 is assigned to Process P2 while another instance of R2 is assigned to process P3, Process P3 is waiting for resource R2.

- **For example:**

Resource -allocation graph shown in below figure the following situation**.**

❖ The set P,R and E:
  - P = {P1,P2,P3}
  - R = {R1,R2,R3,R4}
  - E = {P1→R1, P2→R3, R1 →P2, R2 →P2,R2→P1, R3→P3}

❖ Resource instances:
  - One instance of resource type R1
  - Two instance of resource type R2
  - One instance of resource type R3
  - Three instance of resource type R4

Scanned with CamScanner

❖ Process state:
  - Process P1 is holding an instance of resource type R2, and is waiting for an instance of resource type R1.
  - Process P2 is holding an instance of R1 and R2, and is waiting for an instance of resource type R3.
  - Process P3 is holding an instance of R3.

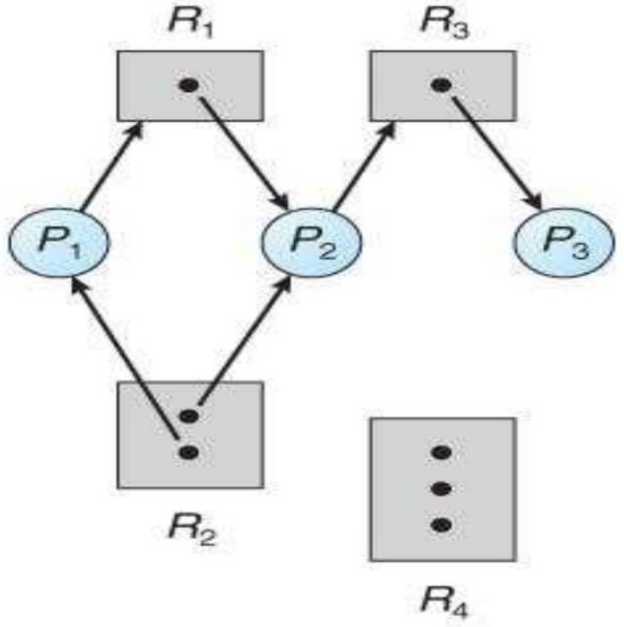


Figure - Resource allocation graph

- If a resource-allocation graph contains no cycles, then the system is not deadlocked. (When looking for cycles, remember that these are ***directed*** graphs.) See the example in Figure above.
- If a resource-allocation graph does contain cycles **AND** each resource category contains only a single instance, then a deadlock exists.
- If a resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the *possibility* of a deadlock, but does not guarantee one.
- Consider, for example, Figures below:
- The two minimal cycle exist in the system

P1→ R1→P2→ R3→P3→R2→P1

P2→R3 → P3 →R2→ P2

❖ Processes P1,P2,and P3 are deadlocked.
❖ Process P2 is waiting for the resource R3,which is held by process P3.

Scanned with CamScanner

- ❖ Process P3, on the other hand, is waiting for either process P1 or Process P2 to release resource R2.
- ❖ Process P1 is waiting for process P2 to release resource R1.
- ❖ The resource –allocation graph in figure also have a cycle.
- ❖ P1→R1→P3→R2→P1

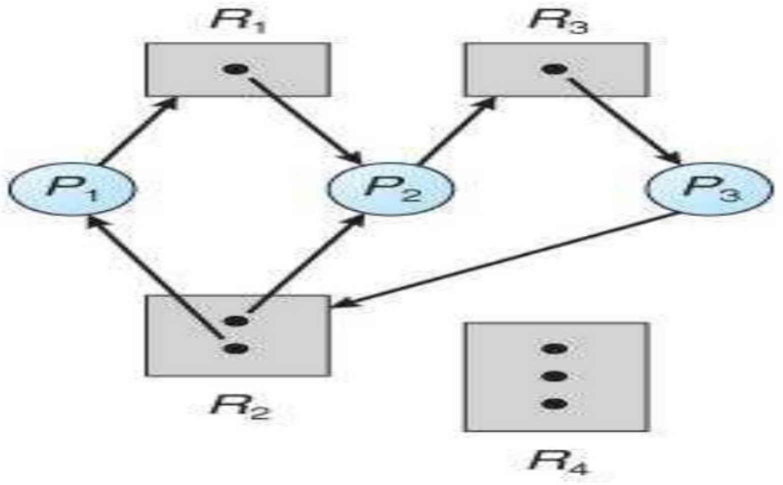


Figure - Resource allocation graph with a deadlock.

Methods for Handling Deadlocks:

- Generally speaking there are three ways of handling deadlocks:
- Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.

- Deadlock detection and recovery - Abort a process or preempt some resources when deadlocks are detected.
- Ignore the problem all together - If deadlocks only occur once a year or so, it may be better to simply let them happen and reboot as necessary than to the constant overhead and system performance associated with deadlock prevention or detection. This is the approach that both Windows and UNIX take.

Scanned with CamScanner

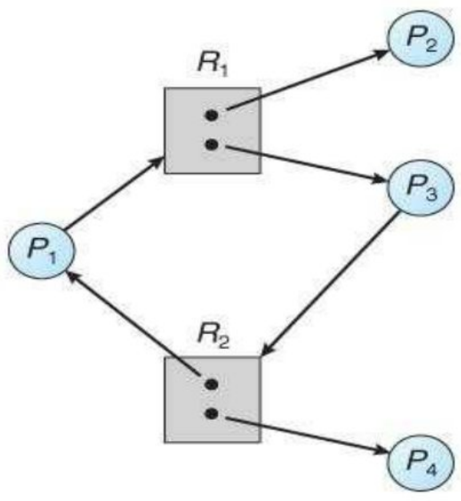


Figure - Resource allocation graph with a cycle but no deadlock

- In order to avoid deadlocks, the system must have additional information about all processes. In particular, the system must know what resources a process will or may request in the future.
- Deadlock detection is fairly straight forward, but deadlock recovery requires either aborting processes.
- If deadlocks are neither prevented nor detected, then when a deadlock occurs the system will gradually slow down, as more and more processes become stuck waiting for resources currently held by the deadlock and by other waiting processes.

Deadlock Prevention

- Deadlocks can be prevented by preventing at least one of the four required conditions:

Mutual Exclusion

- Shared resources such as read-only files do not lead to deadlocks.
- Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

Hold and Wait

- To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others.

Scanned with CamScanner

- **There are several possibilities for this:**
- Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.
- Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.
- Either of the methods described above can lead to starvation if a process requires one or more popular resources.

No Preemption

- Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.
- One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion.
- Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources *and* are them selves blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.
- Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.

Circular Wait

- One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order.
- In other words, in order to request resource Rj, a process must first release all Ri such that i >= j.
- One big challenge in this scheme is determining the relative ordering of the different resources

Deadlock Avoidance:

- The general idea behind deadlock avoidance is to prevent deadlocks from ever happening, by preventing at least one of the aforementioned conditions.
- This requires more information about each process, AND tends to lead to low device utilization.

Scanned with CamScanner

- In some algorithms the scheduler only needs to know the *maximum* number of each resource that a process might potentially use. In more complex algorithms the scheduler can also take advantage of the *schedule* of exactly what resources may be needed in what order.
- When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted.
- A resource allocation ***state*** is defined by the number of available and allocated resources, and the maximum requirements of all processes in the system.

Safe State

- A state is ***safe*** if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.
- More formally, a state is safe if there exists a ***safe sequence*** of processes { P0, P1, P2, ..., PN } such that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j < i. (I.e. if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they have freed up.)
- If a safe sequence does not exist, then the system is in an unsafe state, which ***MAY*** lead to deadlock. (All safe states are deadlock free, but not all unsafe states lead to deadlocks.)

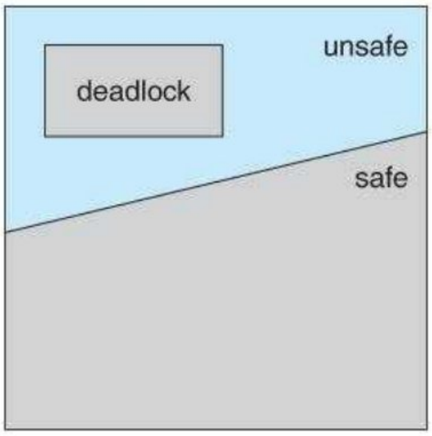


Figure: Safe, unsafe, and deadlocked state spaces.

- For example, consider a system with 12 tape drives, allocated as follows. Is this a safe state? What is the safe sequence?

	Maximum Needs	**Current Allocation**
P0	10	5

Scanned with CamScanner

P1	4	2
P2	9	2

- What happens to the above table if process P2 requests and is granted one more tape drive?
- Key to the safe state approach is that when a request is made for resources, the request is granted only if the resulting allocation state is a safe one.

Resource-Allocation Graph Algorithm

- If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs.
- In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with ***claim edges***, noted by dashed lines, which point from a process to a resource that it may request in the future.
- In order for this technique to work, all claim edges must be added to the graph for any particular process before that process is allowed to request any resources. (Alternatively, processes may only make requests for resources for which they have already established claim edges, and claim edges cannot be added to any process that is currently holding resources.)
- When a process makes a request, the claim edge Pi->Rj is converted to a request edge. Similarly when a resource is released, the assignment reverts back to a claim edge.
- This approach works by denying requests that would produce cycles in the resource-allocation graph, taking claim edges into effect.
- Consider for example what happens when process P2 requests resource R2:

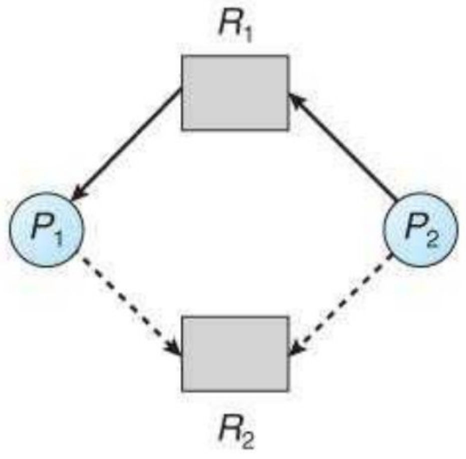


Figure : Resource allocation graph for deadlock avoidance

Scanned with CamScanner

- The resulting resource-allocation graph would have a cycle in it, and so the request cannot be granted.

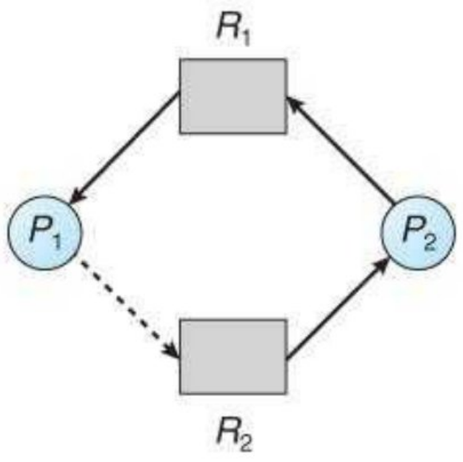


Figure - An unsafe state in a resource allocation graph

Banker's Algorithm

❖ The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The deadlock avoidance algorithm that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the **banker's algorithm**.

- **The banker's algorithm relies on several key data structures: (where 'n' is the number of processes and 'm' is the number of resource categories.)**
  - **Available**: [m] indicates how many resources are currently available of each type.
  - **Max:** [n][m] indicates the maximum demand of each process of each resource.
  - **Allocation:** [n][m] indicates the number of each resource category allocated to each process.
  - **Need:** [n][m] indicates the remaining resources needed of each type for each process. (Note that Need[i][j] = Max[i][j] - Allocation[i][j] for all i, j.)
- For simplification of discussions, we make the following notations / observations:
  - One row of the Need vector, Need[i], can be treated as a vector corresponding to the needs of process i, and similarly for Allocation and Max.
  - A vector X is considered to be <= a vector Y if X[i] <= Y[i] for all i.

Scanned with CamScanner

Safety Algorithm

- In order to apply the Banker's algorithm, we first need an algorithm for determining whether or not a particular state is safe.
- This algorithm determines if the current state of a system is safe, according to the following steps:

We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

1. Let Work and Finish be vectors of length m and n respectively. Initialize Work := Available and Finish[i] =false for i = 1, 2, …,n.

2. Find an i such that both

a. Finish[i] == false

b. Need i < or = Work

If no such i exists, go to step 4.

3.Work :=Work + Allocation;
Finish[i] = true
Go to step 2.
4.If Finish[i] == true for all i, then the system is in a safe state.

This algorithm may require an order of m x n 2 operations to decide whether a state is safe.

7.5.3.2 Resource-Request Algorithm

We nm-v describe the algorithm which determines if requests can be safely granted.

Let Request) be the request vector for process P, If Request i [j] == k,then process Pi wants k instances of resource type Rj. When a request for resources is made by process Pi , the following actions are taken:

1. If Request i< or = Need i, go to step 2. Otherwise/ raise an error condition, since the process has exceeded its maximum claim.
2. If Request i < or = Available, go to step 3. Otherwise, Pi must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

Available := Available – Request i;

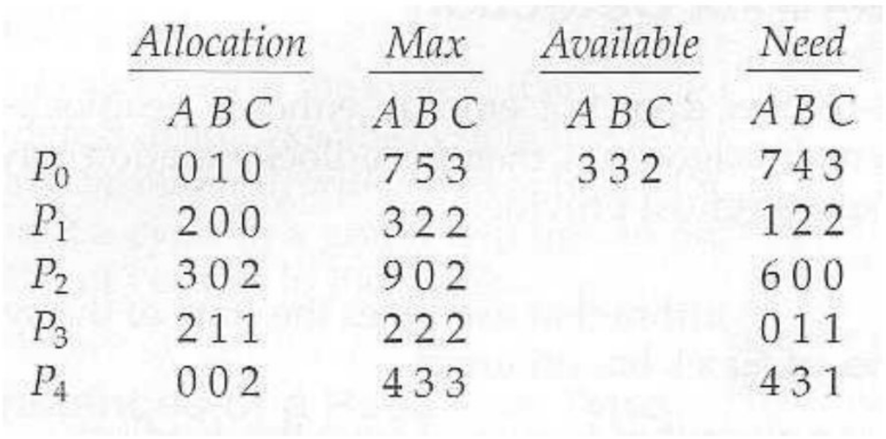
Scanned with CamScanner

Allocation i = Allocation i + Request i;
Need i =Need i – Request i;

4. If the resulting resource-allocation state is safe, the transaction is completed, and process Pi is allocated its resources. However, if the new state is unsafe, then Pi must wait for Request i, and the old resource-allocation state is restored.

An Illustrative Example:

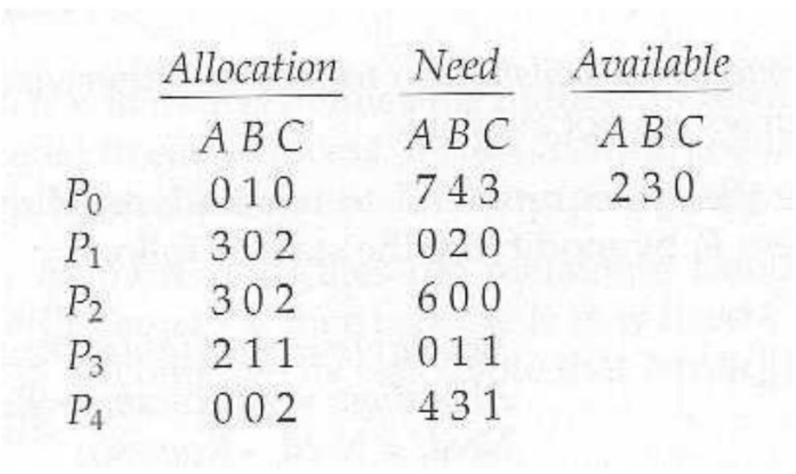
- Consider a system with five processes P0 through P4 and three resource types A,B,C. Resource type A has 10 instance, resource type B has 5 instances, and resource type C has 7 instances. Suppose that, at a time T0, the following snapshot of the system has been taken:



	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

- The consider of the matrix Need is defined to be Max-Allocation and is follow:
- We claim that the system is currently in a safe state. Indeed, the sequence <P1, P3, P4, P2, P0> satisfies the safety criteria.
- Suppose now that process P1 requests one additional instance of resource type A and two instances of resource type C, so Request1= (1,0,2). To decide whether this request can be immediately granted, we first check that Request1 < or = Available (that is, that (1,0,2) < or = (3,3,2)), which is true. We then pretend. that this request has been fulfilled, and we arrive at the following new state:

Scanned with CamScanner



	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_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 and find that the sequence <P1, P3, P4, P0, P2> satisfies our safety requirements.
- Hence, we can immediately grant the request of process P1.

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 must provide:
- **An algorithm that examines the state of the system to determine whether a deadlock has occurred**
- **An algorithm to recover from the deadlock**
- In the following discussion, we elaborate on these two requirements as they pertain to systems with only a **single instance of each resource type**, as well as to systems with **several instances of each resource type.**

- **Single Instance of Each Resource Type:**

- If each resource category has a single instance, then we can define a deadlock detection algorithm that uses a variation of the resource-allocation graph known as a ***wait-for graph***.
- A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges, as shown in the figure below.
- An edge from Pi to Pi in a wait-for graph implies that process Pi is waiting for process Pi to release a resource that Pi needs.
- An edge Pi → Pj exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pi→ Rq and Rq→ Pj for some resource Rq.

Scanned with CamScanner

- For example, in Figure we present a resource-allocation graph and the corresponding wait-for graph.
- As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle.
- To detect deadlocks, the system needs to maintain the wait-for graph and periodically to invoke an algorithm that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n2 operations, where n is the number of vertices in the graph.

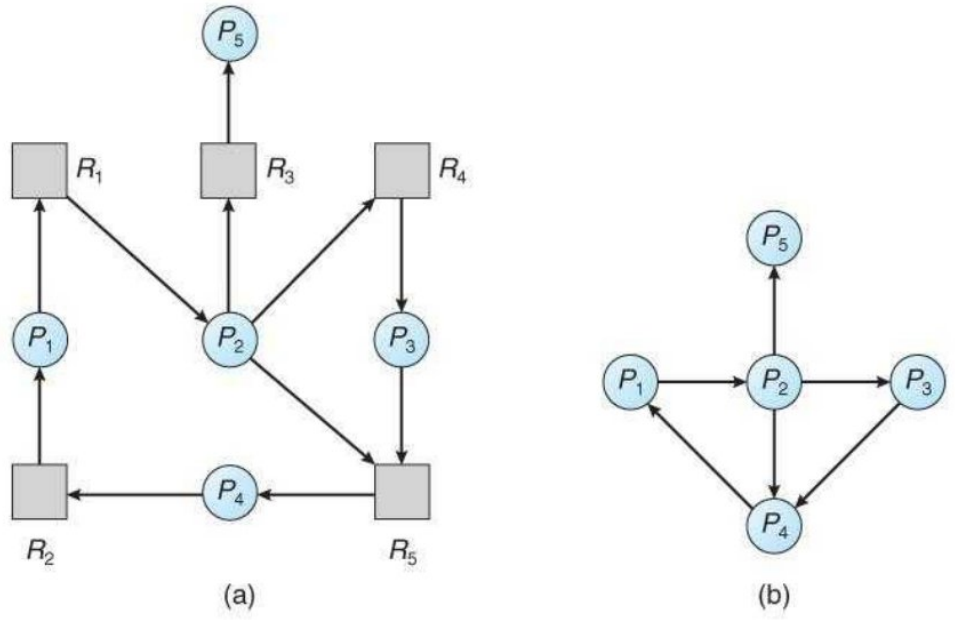


Figure - (a) Resource allocation graph. (b) Corresponding wait-for graph

Several Instances of a Resource Type:

- ❖ The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. We turn now to a deadlock detection algorithm that is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm.
- ❖ **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 Pi is requesting k more instances of resource type Rj .

Scanned with CamScanner

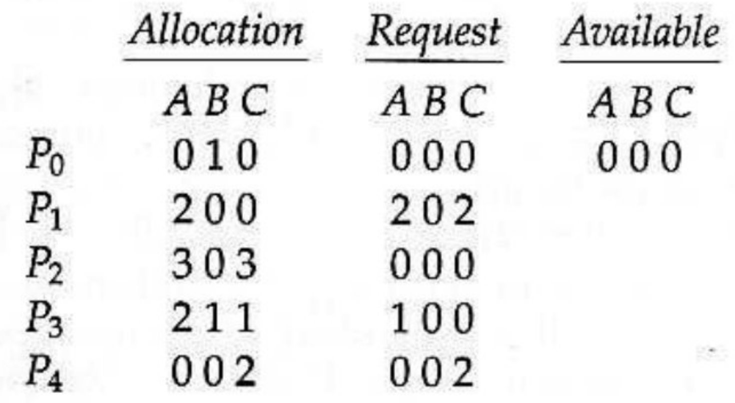
1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work:=Available. For i=1,2,…,n if Allocation i =/ 0, then Finish[i] := false otherwise, Finish[i] :=true.

2. Find an index i such that both
a. Finish[i] = false.
b. Request i < or = Work.
If no such i exists, go to step 4.

3. Work := Work + Allocation;
Finish[i] := true
Go to step 2.

4. If Finish[i] = false, for some i, 1< or = i < or =n, then the system is in a deadlocked state. Moreover, if Finish[i] = false, then process Pi is deadlocked.

❖ This algorithm requires an order of m x n2 operations to detect whether the system is in a deadlocked state.
❖ To illustrate this algorithm, we consider a system with **five processes Po** through **P4** and **three resource types A, B, and C.**
❖ Resource type A has **seven** instances, resource type B has **two** instances, and resource type C has **six** instances.
❖ Suppose that, at time To, we have the following resource-allocation state:

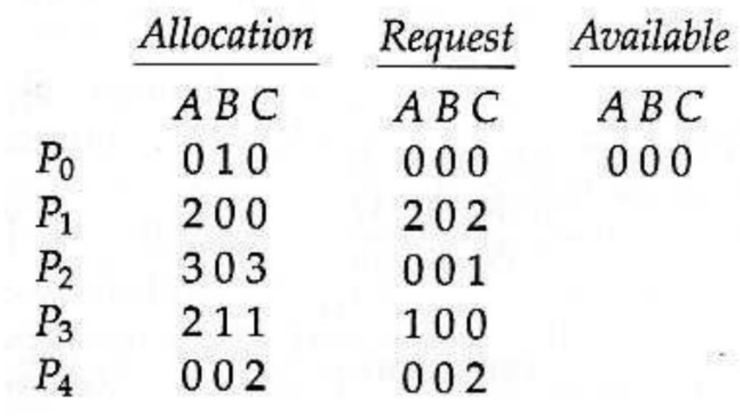


	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence < P0, P2, P3, P1, P4> results in Finish[i] = true for all i.

Suppose now that process P2 makes one additional request for an instance of type C. The Request matrix is modified as follows:

Scanned with CamScanner



	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

We claim that the system is now deadlocked. Although we can reclaim the resources held by process P0, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P1, P2, P3, and P4.

Detection-Algorithm Usage

When should we invoke the detection algorithm? The answer depends on two factors:

1. How often is a deadlock likely to occur?

2. How many processes will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow.

Recovery From Deadlock

- There are three basic approaches to recovery from deadlock:
  1. Inform the system operator, and allow him/her to take manual intervention.
  2. Terminate one or more processes involved in the deadlock
  3. Preempt resources.

Process Termination

- Two basic approaches, both of which recover resources allocated to terminated processes:
  - Terminate all processes involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.

Scanned with CamScanner

  - Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.

- **Abort all deadlocked processes**. This method clearly will break the deadlock cycle, but at great expense; the deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- **Abort one process at a time until the deadlock cycle is eliminated**. This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine wl1f'ther any processes are still deadlocked
- In the latter case there are many factors that can go into deciding which processes to terminate next:
  1. Process priorities.
  2. How long the process has been running, and how close it is to finishing.
  3. How many and what type of resources is the process has used (for example, whether the resources are simple to preempt)
  4. How many more resources the process needs in order to complete.
  5. How many processes will need to be terminated
  6. Whether the process is interactive or batch.

Resource Preemption

- When preempting resources to relieve deadlock, there are three important issues to be addressed:
- **Selecting a victim** - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.

- **Rollback** - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (i.e. abort the process and make it start over.)

- **Starvation** - How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get preempted any more.

Scanned with CamScanner