while(...) ; means loop will keep looping till this while condition is true

Peterson's solution

- A classic software-based solution to the critical-section problem known as Peterson's solution
- It provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered *Po* and P1. For convenience, when presenting *Pi*, we use *Pj* to denote the other process; that is, j equals 1 - i.

- Peterson's solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

- The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process *Pi* is allowed to execute in its critical section. The flag array is used to indicate if a process *is ready* to enter its critical section.
  For example, if flag [i] is true, this value indicates that *Pi* is ready to enter its critical section.

- To enter the critical section, process *Pi* first sets flag [i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so.

- The structure of process P_i in Peterson's solution

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

if while loop will run till j is inside critical sectin i,e flag[j]=1

- Now the above algorithm will ensures that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

Synchronization hardware

- Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures. Instead, we can generally state that any solution to the critical-section problem requires a simple tool-a **lock.**

- Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.

```
1  while(lock==1);//entry code

2  lock =1;//entry code

   critical section

4  lock=0; //exit code
   remainder section
```

Problem in lock technique : No mutual exclusion
<- If p1 stopped at this point & doesnt go to next line & lock is still 0 then p2 can come & now p1 is already in loop, so it can again do lock=1 & can enter critical . So both are in critical now.

To avoid this problem we can use either TEST AND SET or Swap technique

- The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified. In this manner, we could be sure 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.

- Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

- Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words atomically—that is, as one uninterruptible unit.

- We can use these special instructions to solve the critical-section problem in a relatively simple manner.

- The TestAndSet () instruction can be defined as shown.

We combinaed 1st & 2nd line of lock technique so that no process can be between that 2 lines

```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

- The important characteristic of this instruction is that it is executed atomically. Thus, if two TestAndSet () instructions are executed simultaneously (each on a different CPU) they will be executed sequentially in some arbitrary order.

- If the machine supports the TestAndSet () instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of process *P;* is shown here.

```
do {
   while (TestAndSet(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
} while (TRUE);
```

- The Swap() instruction, in contrast to the TestAndSet () instruction, operates on the contents of two words; it is defined as shown

```
void Swap(boolean *a, boolean *b) {
   boolean temp = *a;
   *a = *b;
   *b = temp;
}
```

- Like the TestAndSet () instruction, it is executed atomically. If the machine supports the Swap() instruction, then mutual exclusion can be provided as follows.

```
do {
   key = TRUE;
   while (key == TRUE)
      Swap(&lock, &key);

      // critical section

   lock = FALSE;

      // remainder section
} while (TRUE);
```

Combinaed 1st & 2nd line of lock technique so that no process can be between that 2 lines using swap function

Semaphore

- The hardware-based solutions to the critical-section problem are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a semaphore.
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: **wait () and signal ().**
- The definition of wait () is as follows:

```
wait(S) {
    while S <= 0
      ; // no-op
    S--;
}
```

- The definition of signal() is as follows:

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

- All modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

- In addition, in the case of wait (S), the testing of the integer value of S (S :S 0), as well as its possible modification (S--), must be executed without interruption.

- 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.

- Binary semaphores are also known as mutex locks, as they are locks that provide mutual exclusion. We can use binary semaphores to deal with the critical-section problem or multiple processes. Then processes share a semaphore, mutex, initialized to 1. Each process *Pi* is organized as shown

```
do {
   wait(mutex);        or wait (S)

      // critical section

   signal(mutex);

      // remainder section
} while (TRUE);
```

- Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count).

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.

- To overcome the need for busy waiting, we can modify the definition of the wait() and signal() semaphore operations. When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait.
- The process can *block* itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.
- Then control is transferred to the CPU scheduler, which selects another process to execute.
- A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation.
- The process is restarted by a wakeup () operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

To implement semaphores under this definition, we define a semaphore as a "C' struct:

```
typedef struct {
int value;
struct process *list;
} semaphore
```

The wait() semaphore operation can now be defined as

```
wait(semaphore *S) {
S->value--;
if (S->value < 0) {
add this process to S->list;
block();
}
}
```

The signal () semaphore operation can now be defined as:

```
signal(semaphore *S) {
S->value++;
if (S->value <= 0) {
remove a process P from S->list;
wakeup(P);
}
}
```

The block () operation suspends the process that invokes it.
The wakeup (P) operation resumes the execution of a blocked process P.
These two operations are provided by the operating system as basic system calls.

The Bounded-Buffer Problem / Producer consumer Problem

- The *bounded-buffer problem* is commonly used to illustrate the power of synchronization primitives.
- We assume that the pool consists of *n* buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.
- The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The code for the producer process is shown below

```
do {
    . . .
   // produce an item
    . . .
   wait(empty);
   wait(mutex);
    . . .
    add an item to buffer
    . . .
   signal(mutex);
   signal(full);
} while (TRUE);
```

wait(empty) means wait if the empty unit is 0 & If wait overs then decrement empty--.

wait(mutex) is used to take control of critical section (acquire lock

release lock

increment full

The code for the consumer process is shown below.

```
do {
   wait(full);
   wait(mutex);
    . . .
   // remove an item from buffer
    . . .
   signal(mutex);
   signal(empty);
    . . .
   // consume the item
    . . .
} while (TRUE);
```

Increment empty

- Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

The Readers-Writers Problem

- Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update the database.
- We distinguish between these two types of processes by referring to the former as **readers** and to the latter as **writers**.
- If two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

- To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the ***readers-writers problem.***

- In the solution to the readers-writers problem, the processes share the following data structures:

```
semaphore mutex, wrt;
int readcount;
```

  - ❖ The semaphores mutex and wrt are initialized to 1; read count is initialized to 0.
  - ❖ The semaphore wrt is common to both reader and writer processes.
  - ❖ The mutex semaphore is used to ensure mutual exclusion when the variable read count is updated.
  - ❖ The read count variable keeps track of how many processes are currently reading the object.
  - ❖ The semaphore wrt functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

The code for a writer process is shown in Figure

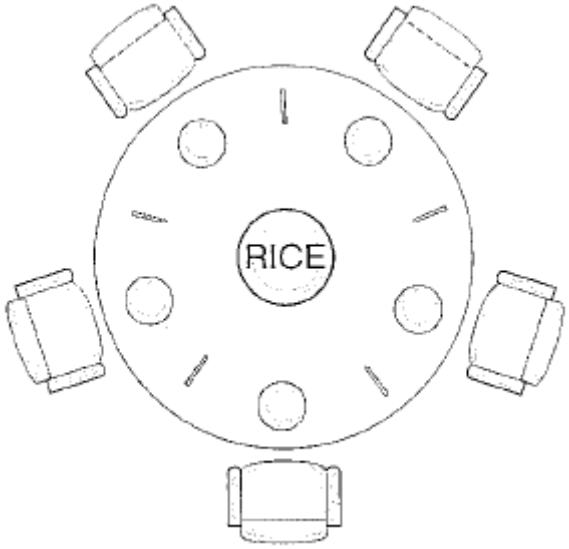
```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
} while (TRUE);
```

The code for a reader process is shown below

```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
      wait(wrt);   //Checking if there are no writers
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
      signal(wrt);  If no readers then only increment writers
   signal(mutex);
} while (TRUE);
```

The Dining-Philosophers Problem

- Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.



- When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).
- *A* philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

Solution

One simple solution is to represent each chopstick with a semaphore. *A* philosopher tries to grab a chopstick by executing a wait () operation on that semaphore; she releases her chopsticks by executing the signal () operation on the appropriate semaphores. Thus, the shared data are

semaphore chopstick[5];

where all the elements of chopstick are initialized to 1.

The structure of philosopher *i* is shown in Figure

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   // eat
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   // think
      . . .
} while (TRUE);
```

MONITORS

- The monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control.
- A monitor is a software module consisting of one or more procedures, an initialization sequence, and local data.
- The monitor construct has been implemented in a number of programming languages, including Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, and Java. It has also been implemented as a program library. This allows programmers to put a monitor lock on any object.
- The chief characteristics of a monitor are the following:
  1. The local data variables are accessible only by the monitor's procedures and not by any external procedure.
  2. A process enters the monitor by invoking one of its procedures.
  3. Only one process may be executing in the monitor at a time; any other processes that have invoked the monitor are blocked, waiting for the monitor to become available.
- By enforcing the discipline of one process at a time, the monitor is able to provide a mutual exclusion facility. The data variables in the monitor can be accessed by only one process at a time. Thus, a shared data structure can be protected by placing it in a monitor.

- A monitor supports synchronization by the use of **condition variables** that are contained within the monitor and accessible only within the monitor. Condition variables are a special data type in monitors, which are operated on by two functions:
    - cwait(c) : Suspend execution of the calling process on condition *c* . The monitor is now available for use by another process.
    - csignal(c) : Resume execution of some process blocked after a cwait on the same condition. If there are several such processes, choose one of them; if there is no such process, do nothing.

CPU-scheduling algorithms

- Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms

    - **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 ten processes per second.

    - **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/0.

    - **Waiting time**. The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/0; it affects only the an1.ount 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 since it takes to start responding, not the time it takes to output the response.

First-Come, First-Served Scheduling/ (FCFS) scheduling algorithm

- 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, it 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.
- On the negative side, the average waiting time under the FCFS policy is often quite long.

Example

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

Process	Burst Time
P_1	24
P_2	3
P_3	3

The processes arri*ve* in the order *P1, P2, P3,* and are served in FCFS order,

Gantt Chart

- The waiting time is 0 milliseconds for process P1, 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:

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

Q1. Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	3	4
P2	5	3
P3	0	2
P4	5	1
P5	4	3

If the CPU scheduling policy is FCFS, calculate the average waiting time and average turn around time.

Gantt Chart-

Gantt Chart

- **Turn Around time = Exit time – Arrival time**
- **Waiting time = Turn Around time – Burst time**

Process Id	Exit time	Turn Around time	Waiting time
P1	7	7 – 3 = 4	4 – 4 = 0
P2	13	13 – 5 = 8	8 – 3 = 5
P3	2	2 – 0 = 2	2 – 2 = 0
P4	14	14 – 5 = 9	9 – 1 = 8
P5	10	10 – 4 = 6	6 – 3 = 3

Now,

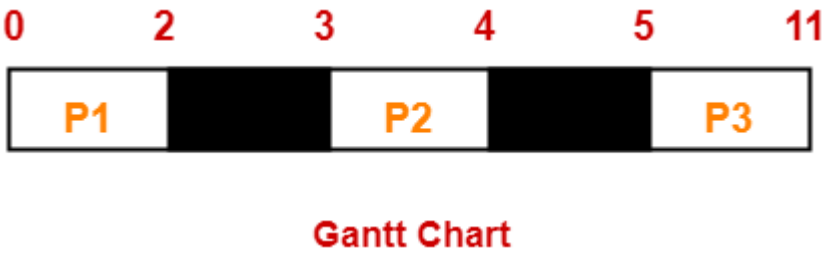
- Average Turn Around time = (4 + 8 + 2 + 9 + 6) / 5 = 29 / 5 = 5.8 unit
- Average waiting time = (0 + 5 + 0 + 8 + 3) / 5 = 16 / 5 = 3.2 unit

Q2. Consider the set of 3 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	0	2
P2	3	1
P3	5	6

If the CPU scheduling policy is FCFS, calculate the average waiting time and average turn around time.

Gantt Chart-



Gantt Chart

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Process Id	Exit time	Turn Around time	Waiting time
P1	2	2 – 0 = 2	2 – 2 = 0
P2	4	4 – 3 = 1	1 – 1 = 0
P3	11	11- 5 = 6	6 – 6 = 0

Now,

- Average Turn Around time = (2 + 1 + 6) / 3 = 9 / 3 = 3 unit
- Average waiting time = (0 + 0 + 0) / 3 = 0 / 3 = 0 unit

Shortest-Job-First Scheduling

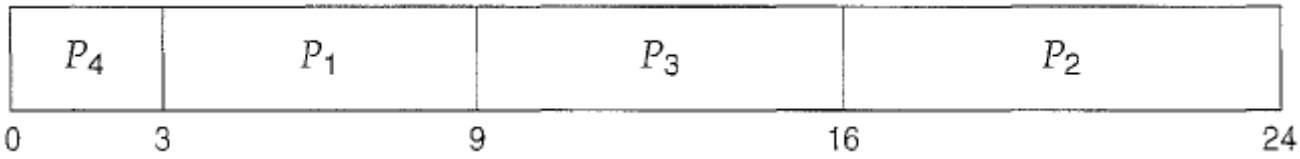
- This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst.
- If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

Example

Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

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

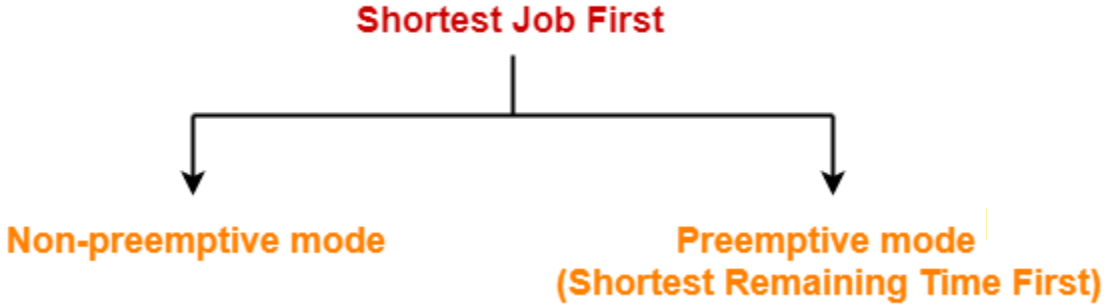


- The waiting time is 3 milliseconds for process P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process P4 .
- Thus, the average waiting time is (3 + 16 + 9 + 0) *I* 4 = 7 milliseconds.
- By comparison, if we were using the FCFS scheduling scheme, the average 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.

- SJF Scheduling can be used in both preemptive and non-preemptive mode.

- Preemptive mode of Shortest Job First is called as **Shortest Remaining Time First (SRTF)**.



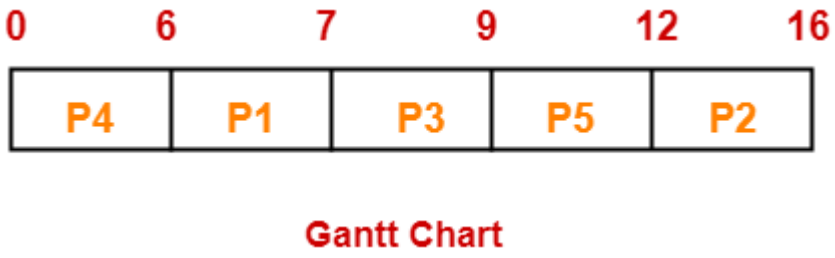
Problem-01: **Consider the set of 5 processes whose arrival time and burst time are given below-**

Process Id	Arrival time	Burst time
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3

If the CPU scheduling policy is SJF non-preemptive, calculate the average waiting time and average turn around time.

Solution-

Gantt Chart-



Gantt Chart

Now, we know-

- **Turn Around time = Exit time – Arrival time**
- **Waiting time = Turn Around time – Burst time**

Process Id	Exit time	Turn Around time	Waiting time
P1	7	7 – 3 = 4	4 – 1 = 3
P2	16	16 – 1 = 15	15 – 4 = 11
P3	9	9 – 4 = 5	5 – 2 = 3
P4	6	6 – 0 = 6	6 – 6 = 0
P5	12	12 – 2 = 10	10 – 3 = 7

Now,

- Average Turn Around time = (4 + 15 + 5 + 6 + 10) / 5 = 40 / 5 = 8 unit
- Average waiting time = (3 + 11 + 3 + 0 + 7) / 5 = 24 / 5 = 4.8 unit

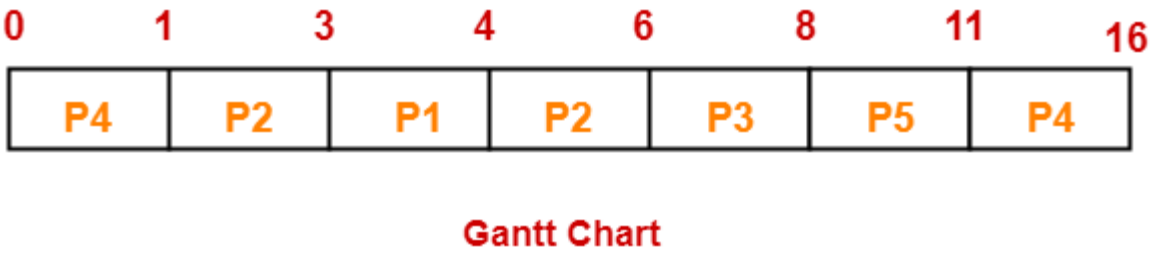
Problem-02: Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3

If the CPU scheduling policy is SJF preemptive, calculate the average waiting time and average turn around time.

Solution-

Gantt Chart-



Gantt Chart

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Process Id	Exit time	Turn Around time	Waiting time
P1	4	4 – 3 = 1	1 – 1 = 0
P2	6	6 – 1 = 5	5 – 4 = 1
P3	8	8 – 4 = 4	4 – 2 = 2
P4	16	16 – 0 = 16	16 – 6 = 10
P5	11	11 – 2 = 9	9 – 3 = 6

Now,

- Average Turn Around time = (1 + 5 + 4 + 16 + 9) / 5 = 35 / 5 = 7 unit
- Average waiting time = (0 + 1 + 2 + 10 + 6) / 5 = 19 / 5 = 3.8 unit

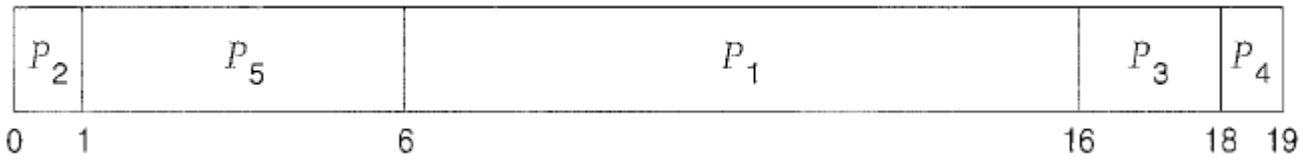
Priority Scheduling

- A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order.
- An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst.

Example
consider the following set of processes, assumed to have arrived at time 0 in the order P1, *P2*, · · ·, *Ps*, with the length of the CPU burst given in milliseconds:

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

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



The average waiting time is 8.2 milliseconds.

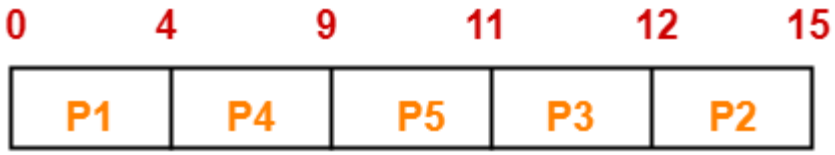
Problem-01 Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time	Priority
P1	0	4	2
P2	1	3	3
P3	2	1	4
P4	3	5	5
P5	4	2	5

If the CPU scheduling policy is priority non-preemptive, calculate the average waiting time and average turn around time. (Higher number represents higher priority)

Solution-

Gantt Chart-



Gantt Chart

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Process Id	Exit time	Turn Around time	Waiting time
P1	4	4 – 0 = 4	4 – 4 = 0
P2	15	15 – 1 = 14	14 – 3 = 11
P3	12	12 – 2 = 10	10 – 1 = 9
P4	9	9 – 3 = 6	6 – 5 = 1
P5	11	11 – 4 = 7	7 – 2 = 5

Now,

- Average Turn Around time = (4 + 14 + 10 + 6 + 7) / 5 = 41 / 5 = 8.2 unit
- Average waiting time = (0 + 11 + 9 + 1 + 5) / 5 = 26 / 5 = 5.2 unit

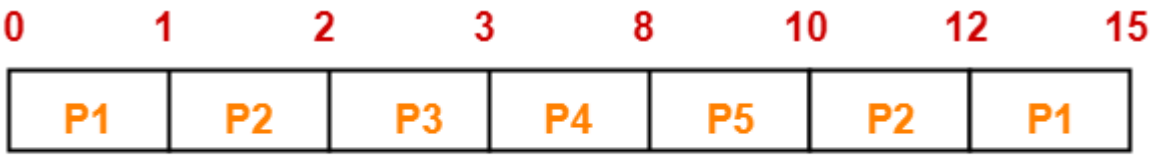
Problem-02: Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time	Priority
P1	0	4	2
P2	1	3	3
P3	2	1	4
P4	3	5	5
P5	4	2	5

If the CPU scheduling policy is priority preemptive, calculate the average waiting time and average turn around time. (Higher number represents higher priority)

Solution-

Gantt Chart-



Gantt Chart

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Process Id	Exit time	Turn Around time	Waiting time
P1	15	15 – 0 = 15	15 – 4 = 11
P2	12	12 – 1 = 11	11 – 3 = 8
P3	3	3 – 2 = 1	1 – 1 = 0
P4	8	8 – 3 = 5	5 – 5 = 0
P5	10	10 – 4 = 6	6 – 2 = 4

Now,

- Average Turn Around time = (15 + 11 + 1 + 5 + 6) / 5 = 38 / 5 = 7.6 unit
- Average waiting time = (11 + 8 + 0 + 0 + 4) / 5 = 23 / 5 = 4.6 unit

Round-Robin Scheduling

- The **round-robin (RR) scheduling algorithm** is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.
- A small unit of time, called a **time quantum** or time slice, is defined.
- 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.
- The average waiting time under the RR policy is often long.

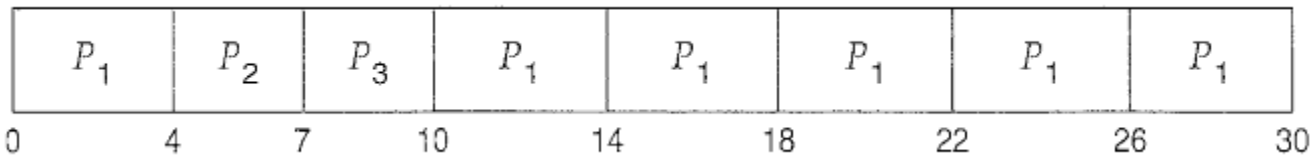
Example:
Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

Use a time quantum of 4 milliseconds.

Process P1 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* . Process *P2* does not need 4 milliseconds, so 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 P1 for an additional time quantum. The resulting RR schedule is as follows:

p1=waiting time= 1st element of last subarray arrival - lastly occured



P1 waits for 6 millisconds (10- 4), *P2* waits for 4 millisconds, and *P3* waits for 7 millisconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds.

Q1. In the following example, there are six processes named as P1, P2, P3, P4, P5 and P6. Their arrival time and burst time are given below in the table. The time quantum of the system is 4 units.

Process ID	Arrival Time	Burst Time
1	0	5
2	1	6
3	2	3
4	3	1
5	4	5
6	6	4

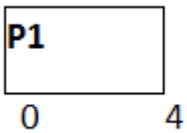
Ready Queue:

Initially, at time 0, process P1 arrives which will be scheduled for the time slice 4 units. Hence in the ready queue, there will be only one process P1 at starting with CPU burst time 5 units.

P1
5

GANTT chart

The P1 will be executed for 4 units first.



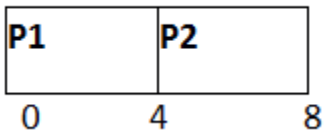
Ready Queue

Meanwhile the execution of P1, four more processes P2, P3, P4 and P5 arrives in the ready queue. P1 has not completed yet, it needs another 1 unit of time hence it will also be added back to the ready queue.

P2	P3	P4	P5	P1
6	3	1	5	1

GANTT chart

After P1, P2 will be executed for 4 units of time which is shown in the Gantt chart.



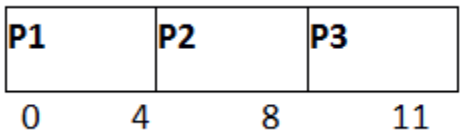
Ready Queue

During the execution of P2, one more process P6 is arrived in the ready queue. Since P2 has not completed yet hence, P2 will also be added back to the ready queue with the remaining burst time 2 units.

P3	P4	P5	P1	P6	P2
3	1	5	1	4	2

GANTT chart

After P1 and P2, P3 will get executed for 3 units of time since its CPU burst time is only 3 seconds.



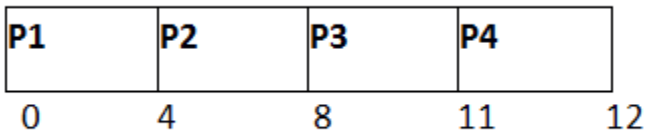
Ready Queue

Since P3 has been completed, hence it will be terminated and not be added to the ready queue. The next process will be executed is P4.

P4	P5	P1	P6	P2
1	5	1	4	2

GANTT chart

After, P1, P2 and P3, P4 will get executed. Its burst time is only 1 unit which is lesser then the time quantum hence it will be completed.



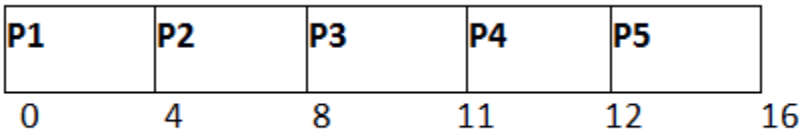
Ready Queue

The next process in the ready queue is P5 with 5 units of burst time. Since P4 is completed hence it will not be added back to the queue.

P5	P1	P6	P2
5	1	4	2

GANTT chart

P5 will be executed for the whole time slice because it requires 5 units of burst time which is higher than the time slice.



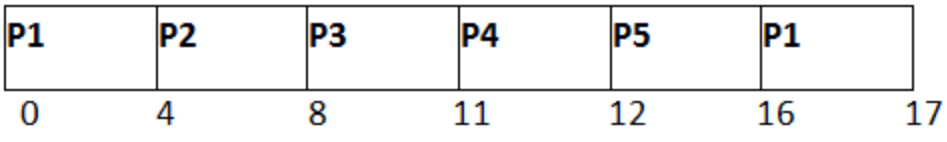
Ready Queue

P5 has not been completed yet; it will be added back to the queue with the remaining burst time of 1 unit.

P1	P6	P2	P5
1	4	2	1

GANTT Chart

The process P1 will be given the next turn to complete its execution. Since it only requires 1 unit of burst time hence it will be completed.



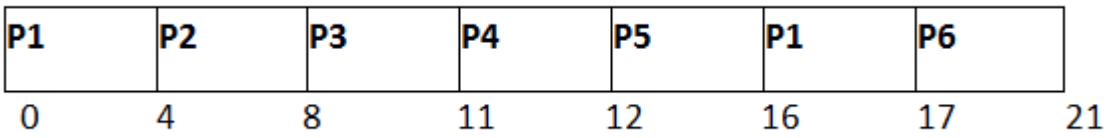
Ready Queue

P1 is completed and will not be added back to the ready queue. The next process P6 requires only 4 units of burst time and it will be executed next.

P6	P2	P5
4	2	1

GANTT chart

P6 will be executed for 4 units of time till completion.



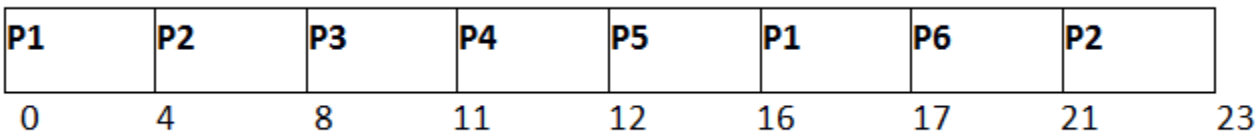
Ready Queue

Since P6 is completed, hence it will not be added again to the queue. There are only two processes present in the ready queue. The Next process P2 requires only 2 units of time.

P2	P5
2	1

GANTT Chart

P2 will get executed again, since it only requires only 2 units of time hence this will be completed.

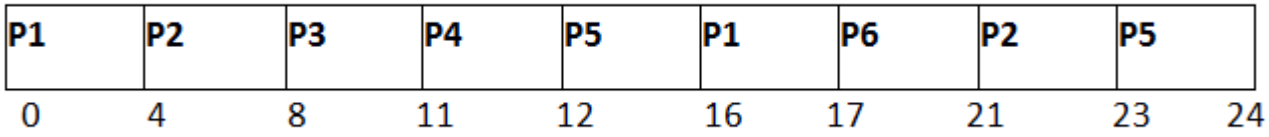


Ready Queue

Now, the only available process in the queue is P5 which requires 1 unit of burst time. Since the time slice is of 4 units hence it will be completed in the next burst.

P5
1

GANTT chart



P5 will get executed till completion.

The completion time, Turnaround time and waiting time will be calculated as shown in the table below.

As, we know,

1. Turn Around Time = Completion Time - Arrival Time
2. Waiting Time = Turn Around Time - Burst Time

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
1	0	5	17	17	12
2	1	6	23	22	16
3	2	3	11	9	6
4	3	1	12	9	8
5	4	5	24	20	15
6	6	4	21	15	11

Avg Waiting Time = (12+16+6+8+15+11)/6 = 76/6 units

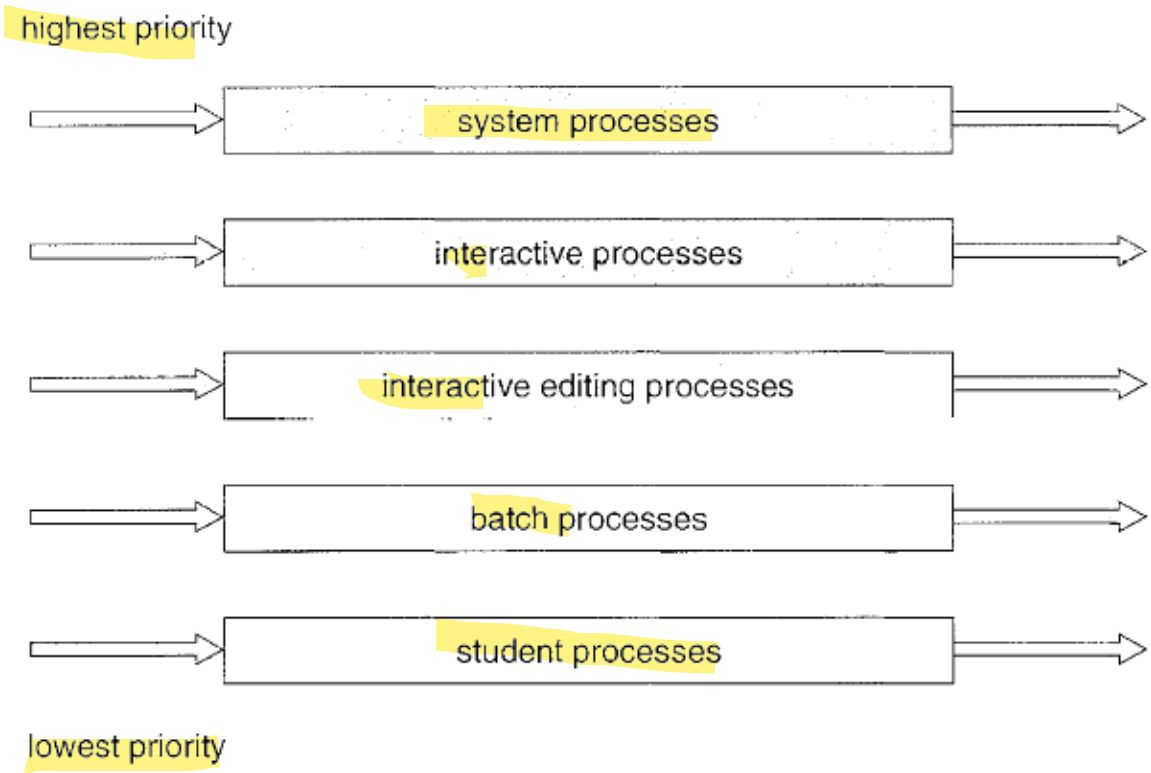
Problem-01: Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3

If the CPU scheduling policy is Round Robin with time quantum = 2 unit, calculate the average waiting time and average turn around time.

Multilevel Queue Scheduling

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



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

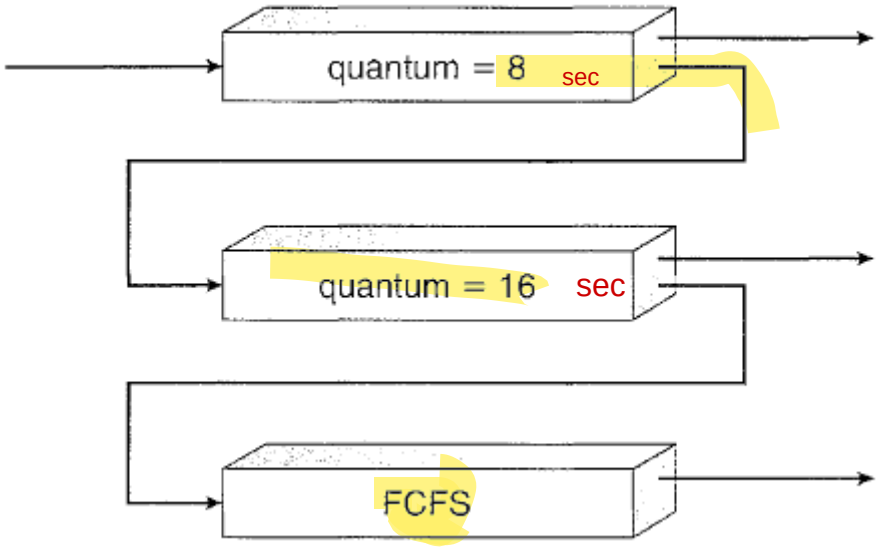
1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

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 would be preempted.

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. For instance, in the foreground-background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis

Multilevel Feedback Queue Scheduling

- When the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system.
- If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature.
- **The multilevel feedback queue scheduling algorithm**, in contrast, allows a process to move between queues.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

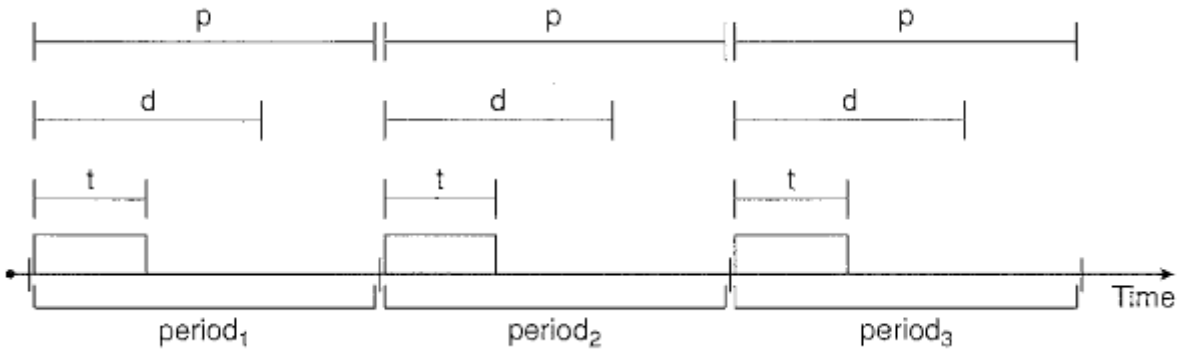


A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

Real Time scheduling

- Coverage of scheduling so far has focused primarily on soft real-time systems. As mentioned, though, scheduling for such systems provides no guarantee on when a critical process will be scheduled; it guarantees only that the process will be given preference over noncritical processes.
- Hard real-time systems have stricter requirements. A task must be serviced by its deadline; service after the deadline has expired is the same as no service at all.

- There must define certain characteristics of the processes that are to be scheduled.
  - ❖ First, the processes are considered periodic. That is, they require the CPU at constant intervals (periods).
  - ❖ Each periodic process has a fixed processing time t once it acquires the CPU.
  - ❖ A deadline *d* by which time it must be serviced by the CPU, and a period *p*.
  - ❖ The relationship of the processing time, the deadline, and the period can be expressed as $0 <= t <= d <= p$.
  - ❖ The rate of a periodic task is $1/p$.



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

Rate-Monotonic Scheduling

- The rate-monotonic scheduling algorithm schedules periodic tasks using a static priority policy with preemption. If a lower-priority process is running and a higher-priority process becomes available to run, it will preempt the lower-priority process.

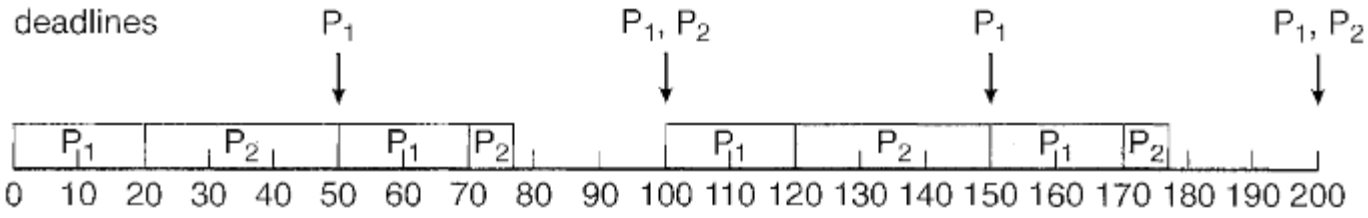
- Upon entering the system, each periodic task is assigned a priority inversely based on its period. The shorter the period, the higher the priority; the longer the period, the lower the priority.

- Rate-monotonic scheduling assumes that the processing time of a periodic process is the same for each CPU burst. That is, every time a process acquires the CPU, the duration of its CPU burst is the same.

time to complete that task is same for all task

Example.
We have two processes P1 and P2. The periods for P1 and P2 are 50 and 100, respectively-that is, *P1* =50 and *P2* = 100. The processing times are t1 = 20 for P1 and *t2* = 35 for P2. The deadline for each process requires that it complete its CPU burst by the start of its next period

We assign P1 a higher priority than P2, since the period of P1 is shorter than that of P2.

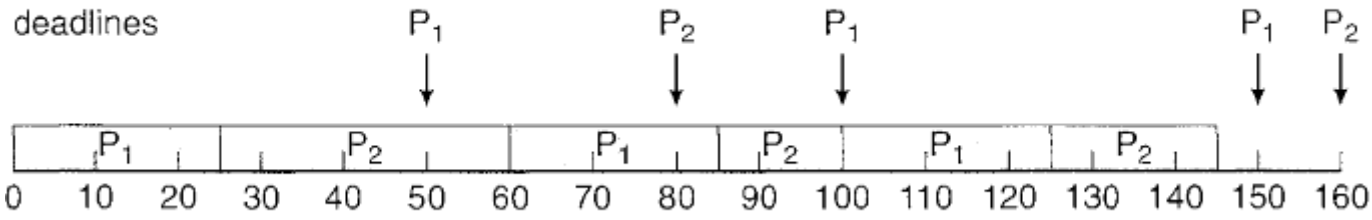


P1 starts first and completes its CPU burst at time 20, thereby meeting its first deadline. *P2* starts running at this point and runs until time 50. At this time, it is preempted by P1, although it still has 5 milliseconds remaining in its CPU burst. P1 completes its CPU burst at time 70, at which point the scheduler resumes P2. *P2* completes its CPU burst at time 75, also meeting its first deadline. The system is idle until time 100, when P1 is scheduled again.

Earliest-Deadline-First Scheduling

- Earliest-deadline-first (EDF) scheduling dynamically assigns priorities according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority.
- Under the EDF policy, when a process becomes runnable, it must announce its deadline requirements to the system.
- Priorities may have to be adjusted to reflect the deadline of the newly runnable process. Note how this differs from rate-monotonic scheduling, where priorities are fixed.

Example: P1 has values of p1 = 50 and t1 = 25 and that P2 has values of *p2* = 80 and *t2* = 35.



Process P1 has the earliest deadline, so its initial priority is higher than that of process P2. Process P2 begins running at the end of the CPU burst for P1. However, whereas rate-monotonic scheduling allows P1 to preempt P2 at the beginning of its next period at time 50, EDF scheduling allows process P2 to continue running. P2 now has a higher priority than P1 because its next deadline (at time 80) is earlier than that of P1 (at time 100).

Deadlock

- In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock

- A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.

Necessary Conditions for Deadlock

must hold all 4

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

Mutual exclusion. At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

Hold and wait. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

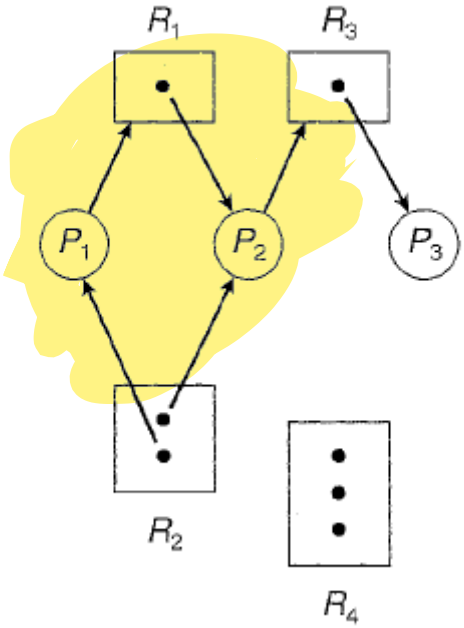
No preemption. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

Circular wait. A set { *P0* , Pl, ... , Pn } of waiting processes must exist such that *Po* is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ... , Pn-1 is waiting for a resource held by *Pn* and Pn is waiting for a resource held by *Po*.

Resource-Allocation Graph

- Deadlocks can be described more precisely in terms of a directed graph called **a System Resource Allocation graph**.
- This graph consists of a set of vertices *V* and a set of edges *E*. The set of vertices V is partitioned into two different types of nodes: *P* = { P1, P2, ... , Pn}, the set consisting of all the active processes in the system, and *R* = {R1, R2, ... , *Rm)* the set consisting of all resource types in the system.
- A directed edge from process Pi to resource type *Rj* is denoted by *P→Rj;* it signifies that process *Pi* has requested an instance of resource type *Rj* and is currently waiting for that resource.
- A directed edge from resource type *Rj* to process *Pi* is denoted by *Rj→Pi* signifies that an instance of resource type *Rj* has been allocated to process *Pi.*
- A directed edge *Pi→Rj* is called a **Request edge.**
- A directed edge *Rj→Pi* is called an **assignment edge.**

- Pictorially we represent each process *Pi* as a circle and each resource type *Rj* as a rectangle.
- Since resource type *Rj* may have more than one instance, we represent each such instance as a dot within the rectangle.
- Note that a request edge points to only the rectangle *R1,* whereas an assignment edge must also designate one of the dots in the rectangle.



The resource-allocation graph shown in Figure depicts the following situation.
The sets *P*, R and *E:*
P = {P1, P2, P3}
R= {R1, R2, R3, ~}
E = {Pl→ Rl, P2→R3, Rl→P2, R2→P2, R2→ Pl, R3→P3}

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

Process states:
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 an instance of R2 and is waiting for an instance of R3.
Process *P3* is holding an instance of R3

If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

Deadlock Handling Strategies

Deadlock Prevention: provides a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made

Deadlock Avoidance: requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait.

Deadlock Prevention

As we noted, for a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can *prevent* the occurrence of a deadlock.

Mutual Exclusion:

- The mutual-exclusion condition must hold for non-sharable resources. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock.
- Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource.
- In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

Hold and Wait

- To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.
- One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.
- An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.

No Preemption

- The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol.
- If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting.

Circular Wait

- The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

- To illustrate, we let *R* = { R1, R2, ... , Rn} be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.

- We can now consider the following protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type -say, *Ri.* After that, the process can request instances of resource type *Rj* if and only if F(Rj) > F(R;).

- Alternatively, we can require that a process requesting an instance of resource type *Rj* must have released any resources *Ri* such that F(Ri) =>F(Rj).

Deadlock Avoidance

- An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.
- For example, in a system with one tape drive and one printer, the system might need to know that process *P* will request first the tape drive and then the printer before releasing both resources, whereas process *Q* will request first the printer and then the tape drive.
- With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.
- Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.
- The resource-allocation ***state*** is defined by the number of available and allocated resources and the maximum demands of the processes.

Safe State

- A state is ***safe*** if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence.

- A sequence of processes <P1, P2, ... , *Pn*> is a safe sequence for the current allocation state if, for each *Pi,* the resource requests that *Pi* can still make can be satisfied by the currently available resources plus the resources held by all *Pj,* with *j* < *i.*

- In this situation, if the resources that *Pi* needs are not immediately available, then *Pi* can wait until all *Pj* have finished. When they have finished, *Pi* can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When *Pi* terminates, *Pi+1* can obtain its needed resources, and so on.

Banker's Algorithm

- The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

- Several data structures must be maintained to implement the banker's algorithm.

  - **Available**. A vector of length *m* indicates the number of available resources of each type. If *Available[j]* equals *k*, then *k* instances of resource type *Rj* are available.

  - **Max.** An *n* x *m* matrix defines the maximum demand of each process. If *Max[i] [j]* equals *k*, then process *Pi* may request at most *k* instances of resource type Rj.

  - **Allocation.** An *n* x *m* matrix defines the number of resources of each type currently allocated to each process. If *Allocation[i][j]* equals k, then process *Pi* is currently allocated k instances of resource type *Rj*.

  - **Need.** An *n* x *m* matrix indicates the remaining resource need of each process. If *Need[i][j]* equals *k*, then process *Pi* may need *k* more instances of resource type *Rj* to complete its task.
    Note that *Need[i][j]* equals *Max[i][j] - Allocation [i][j]*.

Safety Algorithm

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 $i = 0, 1, \ldots, n - 1$.
2. Find an index i such that both
   a. $Finish[i] == false$
   b. $Need_i \leq Work$

   If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   Go to step 2.
4. If $Finish[i] == true$ for all i, then the system is in a safe state.

Resource-Request Algorithm

- Next, we describe the algorithm for determining whether requests can be safely granted.
- Let *Request i* be the request vector for process *Pi*. If *Request i [j] = = k*, then process *Pi* wants *k* instances of resource type *Rj*. When a request for resources is made by process *Pj*, the following actions are taken:

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

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

Example

Consider a system that contains five processes P1, P2, P3, P4, P5 and the three resource types A, B and C. Following are the resources types: A has 10, B has 5 and the resource type C has 7 instances.

Process	Allocation A	B	C	Max (total needed amount) A	B	C	Available A	B	C	Need : Max-alloc A B C
P1	0	1	0	7	5	3	3	3	2	7 4 3
P2	2	0	0	3	2	2				1 2 2
P3	3	0	2	9	0	2				6 0 0
P4	2	1	1	2	2	2				0 1 1
P5	0	0	2	4	3	3				4 3 1

Answer the following questions using the banker's algorithm:

1. Determine if the system is safe or not.
2. What will happen if the resource request (1, 0, 0) for process P1 can the system accept this request immediately?

Ans1 : **Apply the Banker's Algorithm:**

Available Resources of A, B and C are 3, 3, and 2.

Now we check if each type of resource request is available for each process.

Step 1: For Process P1:

Need <= Available

7, 4, 3 <= 3, 3, 2 condition is **false**.

So, we examine another process, P2.

Step 2: For Process P2:

Need <= Available

1, 2, 2 <= 3, 3, 2 condition **true**

New available = available + Allocation

(3, 3, 2) + (2, 0, 0) => 5, 3, 2

Similarly, we examine another process P3.

Step 3: For Process P3:

P3 Need <= Available

6, 0, 0 < = 5, 3, 2 condition is **false**.

Similarly, we examine another process, P4.

Step 4: For Process P4:

P4 Need <= Available

0, 1, 1 <= 5, 3, 2 condition is **true**

New Available resource = Available + Allocation

5, 3, 2 + 2, 1, 1 => 7, 4, 3

Similarly, we examine another process P5.

Step 5: For Process P5:

P5 Need <= Available

4, 3, 1 <= 7, 4, 3 condition is **true**

New available resource = Available + Allocation

7, 4, 3 + 0, 0, 2 => 7, 4, 5

Now, we again examine each type of resource request for processes P1 and P3.

Step 6: For Process P1:

P1 Need <= Available

7, 4, 3 <= 7, 4, 5 condition is **true**

New Available Resource = Available + Allocation

7, 4, 5 + 0, 1, 0 => 7, 5, 5

So, we examine another process P2.

Step 7: For Process P3:

P3 Need <= Available

6, 0, 0 <= 7, 5, 5 condition is true

New Available Resource = Available + Allocation

7, 5, 5 + 3, 0, 2 => 10, 5, 7

There can be more than one safe sequence as we can allocate resource to any process whose need is less than availability

Hence, we execute the banker's algorithm to find the safe state and the safe sequence like P2, P4, P5, P1 and P3

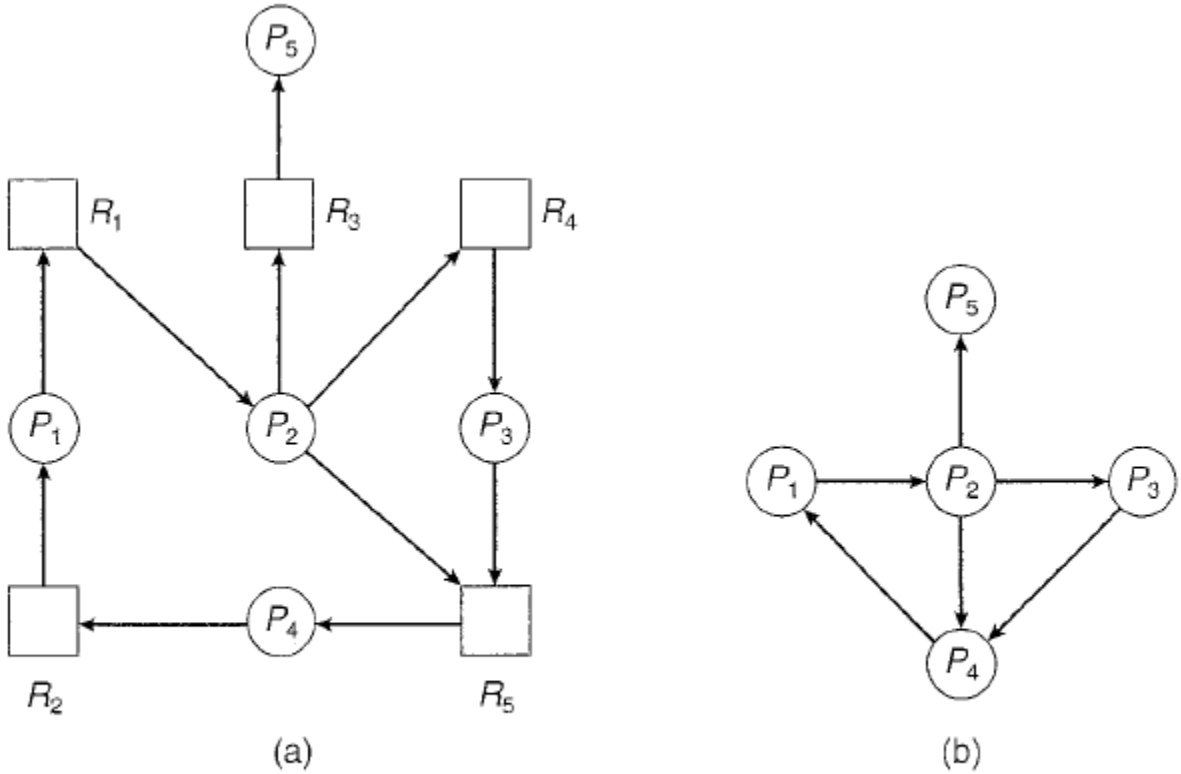
Ans 2 For granting the Request (1, 0, 2), first we have to check that **Request <= Available**, that is (1, 0, 2) <= (3, 3, 2), since the condition is true. So the process P1 gets the request immediately.

Deadlock Detection

- An algorithm has been proposed that examines the state of the system to determine whether a deadlock has occurred

Single Instance of Each Resource Type

- If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a ***wait-for* graph.**
- An edge from *Pi* to *Pj* in a wait-for graph implies that process *Pi* is waiting for process *Pj* 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*.
- 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 *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.



(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]* equals *k*, then process *Pi* is requesting *k* more instances of resource type *Rj*.

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

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

Deadlock Recovery

1. **Process Termination**
   To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

   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 whether any processes are still deadlocked.

2. Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

Selecting a victim. Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed during its execution.

Rollback. If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

Starvation. How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

First make gantt chart then table

In preemtive at each second we minus 1 from the burst time of running process. Then apply whole scheduling at next step with change in burst time of this process.

Response time = 1st time in gantt chartt - arrival time

exit time = completion time = last time in gantt chart

Waiting time = turn around time - burst time

Turn around time = completion -arrival

In case of preemtive also we take the original burst time for calc of waiting time

Rule of duvidha : Select least arrival time & if arrival time is also same then we with respect to process no/id.

FCFS: Rule1. Select least arrival time. If 2 process having arrival time <=gantt time then apply rule for duvidha

SJF: 1. Select least burst time from all process that have arrived. If 2 process are having arrival time <=gantt then apply rule for duvidha

PRIORITY : When arrival time <= gantt chart time , we check whose priority is more/less as per question.(If 2 or more process with same priority then apply rule of duvidha

ROUND ROBIN : [Note : Time Quantum/Slice time= time after which we need to switch to other process.]
Step 1. We make a ready queue along with gantt chart & add all process whose arrival time <=gantt time in ready queue according (if duvidha apply rule of duvidha).
Step2. In ready queue we pop 1st process x & add it in gantt chart with time till slice time or process time of x(whichever is lower) & add all other processes whose arrival time <=gantt time in queue (if same then apply rule of duvidha).
Then if process x time was greater than slice time we add the process with its new time(previous time -slice time) in queue.
In ready queue 1st we cut running process and add process whose arrival time<= gantt chart time in ascending order of their arrival time except the running process.
[Note : There is no non primitive & primitive in this because if slice time =1 it becomes like primitive]