Unit 2 Operating System

Operating Systems (SRM Institute of Science and Technology)

Course Code: 18CSC205J **Course Name: Operating Systems**

Unit 2 – INTRODUCTION

PROCESS SYNCHRONIZATION : Peterson's solution, Synchronization, Hardware - Understanding the two-process solution and the benefits of the synchronization hardware - Process synchronization: Semaphores, usage, implementation - Gaining the knowledge of the usage of the semaphores for the Mutual exclusion mechanisms - Classical Problems of synchronization – Readers writers problem, Bounded Buffer problem - Good understanding of synchronization mechanisms - Classical Problems of synchronization – Dining Philosophers problem (Monitor) - Understanding the synchronization of limited resources among multiple processes - CPU SCHEDULING : FCFS, SJF, Priority - Understanding the scheduling techniques - CPU Scheduling: Round robin, Multilevel queue Scheduling, Multilevel feedback Scheduling - Understanding the scheduling techniques - Real Time scheduling: Rate Monotonic Scheduling and Deadline Scheduling - DEADLOCKS: Necessary conditions, Resource allocation graph, Deadlock prevention methods - Understanding the deadlock scenario -Deadlocks :Deadlock Avoidance, Detection and Recovery - Understanding the deadlock avoidance, detection and recovery mechanisms

PROCESS SYNCHRONIZATION

Peterson's solution

A classic software-based solution to the critical-section problem is known as Peterson's solution.

The structure of process Pi in Peterson's solution is as follows:

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

    critical section

        flag[i] = false;

    remainder section
    } while (true);
```

// when pi wants to enter it sets its flag as true and sets turn = j, gives the chance to other process. /* while is wait loop */

This solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P0 and P1. We use Pj for convenience to denote the other process when Pi is present; 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 denotes whose turn it is to enter its critical section. i.e., if turn == i, then process Pi is allowed to execute in its critical section.
- If a process is ready to enter its critical section, the flag array is used to indicate that. For E.g., if flag[i] is true, this value indicates that Pi is ready to enter its critical section.

We now prove that this solution is correct. We need to show that

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

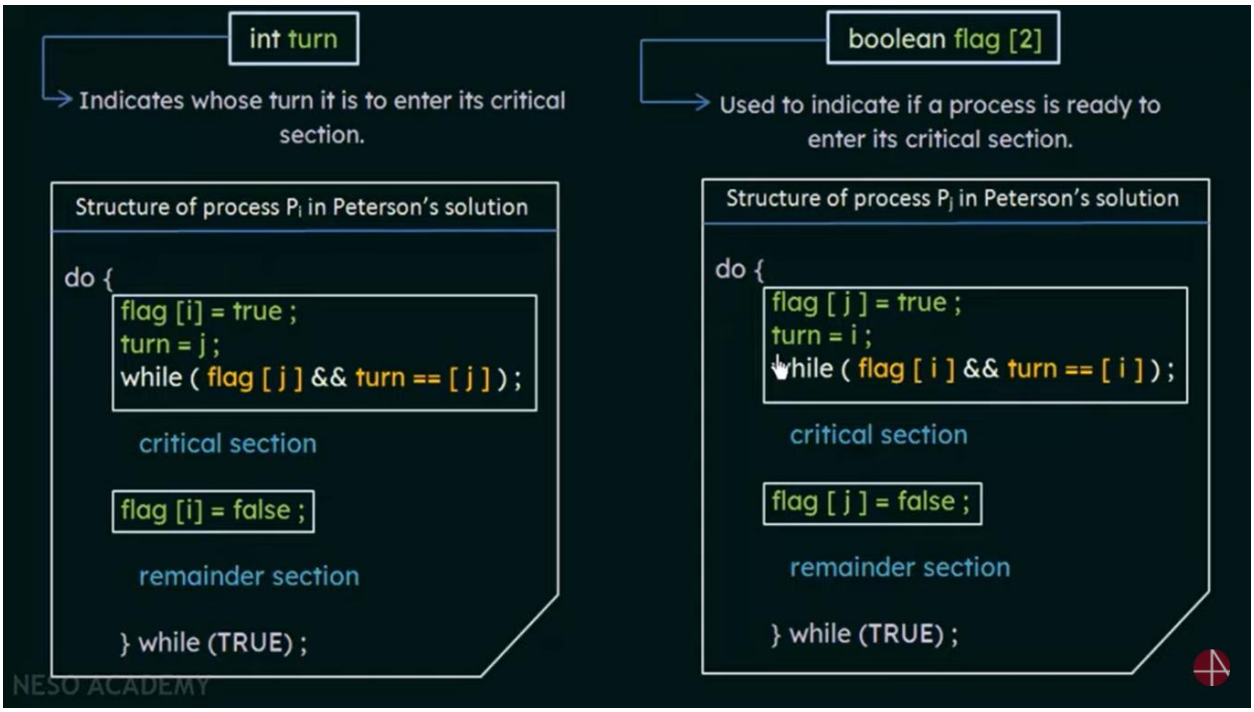
To prove 1, we note that each Pi enters its critical section only if either **flag[j] == false or turn == i.** Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true. These two observations indicate that P0 and P1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes — say, Pj — must have successfully executed the while statement, whereas Pi had to execute at least one additional statement ("turn == j"). However, at that time, flag[j] == true and turn == j, and this condition will persist as long as Pj is in its critical section; as a result, mutual exclusion is preserved.

To prove properties 2 and 3, we note that if a process is stuck in the **while loop with the condition flag[j] == true and turn == j, process Pi can be prevented from entering the critical section only**; this loop is the only one possible. flag[j] will be == false, and Pi can enter its critical section if Pj is not ready to enter the critical section. If Pj has set, flag[j] = true and is also executing in its while statement, then either turn == i or turn == j. If turn == i, Pi will enter the critical section then. Pj will enter the critical section, If turn == j. Although once Pj exits its critical section, it will reset flag[j] to false, allowing Pi to enter its critical section. Pj must also set turn to i, if Pj resets flag[j] to true. Hence, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (progress) after at most one entry by Pj (bounded waiting).

Disadvantage

- Peterson's solution works for two processes. This solution is also a busy waiting solution so CPU time is wasted.

- Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures.



https://www.youtube.com/watch?v=5oZYS5dTrmk

Synchronization Hardware:

Understanding the two-process solution and the benefits of the synchronization hardware

Synchronization hardware is a hardware-based solution to resolve the critical section problem. hardware-based solution for the critical section problem which introduces the hardware instructions that can be used to resolve the critical section problem effectively. Hardware solutions are often easier and also improves the efficiency of the system.

The hardware-based solution to critical section problem is based on a **simple tool i.e. lock**. The solution implies that before entering into the critical section the process must acquire a lock and must release the lock when it exits its critical section. Using of lock also prevent the race condition.

The hardware synchronization provides two kinds of hardware instructions that are TestAndSet and Swap.

Course Code: 18CSC205J **Course Name: Operating Systems**

TestAndSet Hardware Instruction:

The TestAndSet() hardware instruction is atomic instruction. Atomic means both the test operation and set operation are executed in one machine cycle at once. If the two different processes are executing TestAndSet() simultaneously each on different CPU. Still, they will be executed sequentially in some random order.

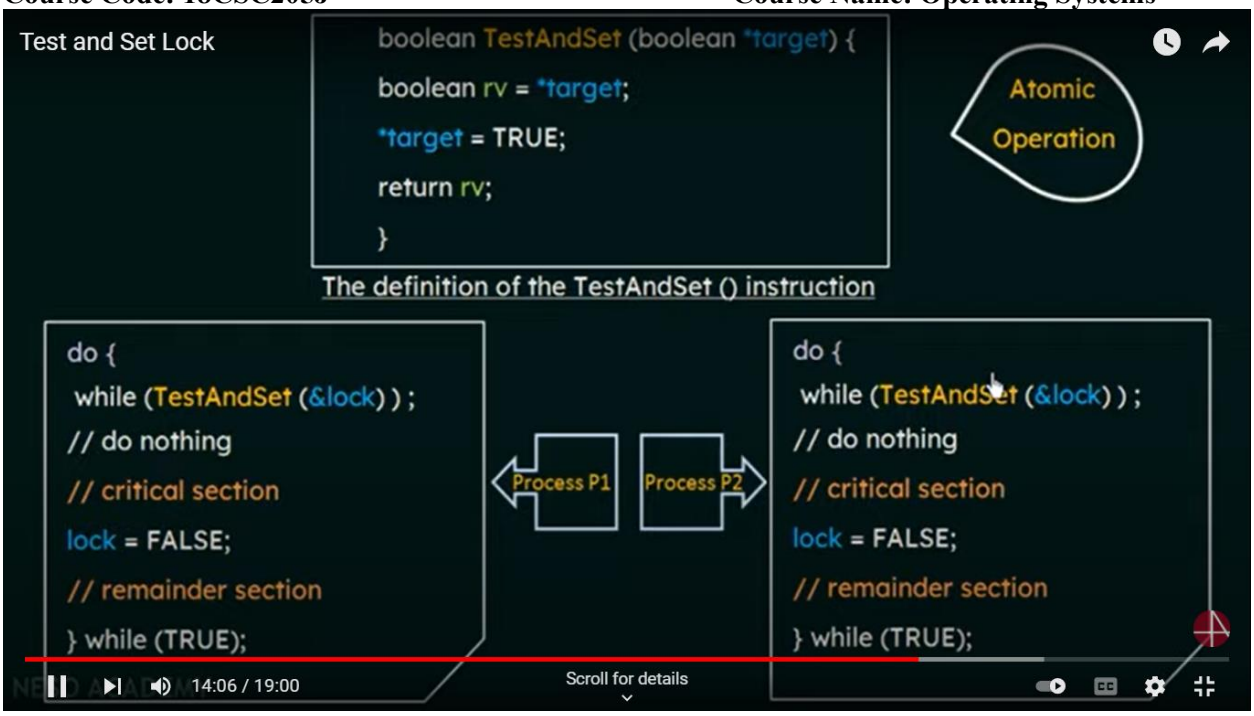
The TestAndSet() instruction can be defined as in the code below:

```
do {
while (TestAndSet(&lock));
// critical section
lock = FALSE;
// remainder section
} while (TRUE);
```

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

Let's say process P0 wants to enter the critical section it executes the code above which let while loop invokes TestAndSet() instruction. Using the TestAndSet() instruction the P0 modifies the lock value to true to acquire the lock and enters the critical section. Now, when P0 is already in its critical section process P1 also wants to enter in its critical section. So it will execute the do-while loop and invoke TestAndSet() instruction only to see that the lock is already set to true which means some process is in the critical section which will make P1 repeat while loop unless P0 turns the lock to false.

Once the process P0 complete executing its critical section its will turn the lock variable to false. Then P1 can modify the lock variable to true using TestAndSet() instruction and enter its critical section. This is how you can achieve mutual exclusion with the do-while structure above i.e. it let only one process to execute its critical section at a time.



Compare and Swap or Swap Hardware Instruction:

Like TestAndSet() instruction the swap() hardware instruction is also an atomic instruction. With a difference that it operates on two variables provided in its parameter. The structure of swap() instruction is:

```
do {
key = TRUE;
while (key == TRUE)
Swap(&lock, &key);
// critical section
lock = FALSE;
// remainder section
} while (TRUE);
```

The structure above operates on one global shared Boolean variable lock and another local Boolean variable key. Both of which are initially set to false. The process P0 interested in executing its critical section execute code above and set lock as true and enter its critical section. Thus refrain other processes from executing their critical section satisfying mutual exclusion.

Advantages of Hardware Instruction

- Hardware instructions are easy to implement and improves the efficiency of the system.
- Supports any number of processes may it be on the single or multiple processor system.
- With hardware instructions, you can implement multiple critical sections each defined with a unique variable.

Disadvantages of Hardware Instruction

- Processes waiting for entering their critical section consumes a lot of processors time which increases busy waiting.
- As the selection of processes to enter their critical section is arbitrary. It may happen that some processes are waiting for the indefinite time which leads to process starvation.
- Deadlock is also possible.

SPIN LOCK : While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to critical section. This type of mutex lock is also called a spinlock because the process "spins" while waiting for the lock to become available.

Process synchronization: Semaphores

- It is a synchronization tool that is used to generalize the solution to the critical section problem in complex situations.

- A Semaphore s is an integer variable that can only be accessed via two indivisible (atomic) operations namely

1. wait or P operation (to test)
2. signal or V operation (to increment)

The wait() operation was originally termed P (from the Dutch proberen, "to test"); signal() was originally called V (from verhogen, "to increment").
The definition of wait() is as follows:

```
wait (s)
{
while(s<=0);
s--;
}
```

The definition of wait() 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.

Semaphore Usage:

The two common kinds of semaphores are Counting semaphores and Binary semaphores.

Binary Semaphores: In Binary semaphores, the value of the semaphore variable will be 0 or 1. Initially, the value of semaphore variable is set to 1 and if some process wants to use some resource then the wait() function is called and the value of the semaphore is changed to 0 from 1. The process then uses the resource and when it releases the resource then the signal() function is called and the value of the semaphore variable is increased to 1. If at a particular instant of time, the value of the semaphore variable is 0 and some other process wants to use the same resource then it has to wait for the release of the resource by the previous process. In this way, process synchronization can be achieved.

Counting Semaphores: In Counting semaphores, firstly, the semaphore variable is initialized with the number of resources available. After that, whenever a process needs some resource, then the wait() function is called and the value of the semaphore variable is decreased by one. The process then uses the resource and after using the resource, the signal() function is called and the value of the semaphore variable is increased by one. So, when the value of the semaphore variable goes to 0 i.e all the resources are taken by the process and there is no resource left to be used, then if some other process wants to use resources then that process has to wait for its turn. In this way, we achieve the process synchronization.

Implementation - Gaining the knowledge of the usage of the semaphores for the Mutual exclusion mechanisms

Semaphore Implementation

- When a process executes the wait operation and finds that the semaphore value is not positive, the process can block itself. The block operation places the process into a waiting queue associated with the semaphore.
- A process that is blocked waiting on a semaphore should be restarted when some other process executes a signal operation. The blocked process should be restarted by a wakeup operation which put that process into ready queue.
- To implemented the semaphore, we define a semaphore as a record as:

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

- Assume two simple operations:
  - **block** suspends the process that invokes it.
  - **wakeup(*P*)** resumes the execution of a blocked process **P**.
  - Semaphore operations now defined as

```
wait(S)
{
S.value--;

if (S.value <= 0) {
        add this process to S.L;
        block;
}
signal(S)
{
S.value++;
remove a process P from S.L;
wakeup(P);
}
```

Classical Problems of synchronization

The classical problems of synchronization are as follows:

- Bound-Buffer problem or Producer-Consumer problem.
- Readers and writers problem
- Dining Philosophers problem.

Readers writers problem, Bounded Buffer problem gives Good understanding of synchronization mechanisms.

Dining Philosophers problem (Monitor) gives understanding the synchronization of limited resources among multiple processes.

The Bounded-Buffer Problem

Bounded-buffer (or Producer-Consumer) Problem:

Bounded Buffer problem is also called producer consumer problem. This problem is generalized in terms of the Producer-Consumer problem. 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. Producers produce a product and consumers consume the product, but both use of one of the containers each time.

Readers and 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 (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as readers and to the latter as writers. Precisely in OS we call this situation as the readers-writers problem. Problem parameters:

One set of data is shared among a number of processes.

- Once a writer is ready, it performs its write. Only one writer may write at a time.
- If a process is writing, no other process can read it.
- If at least one reader is reading, no other process can write.
- Readers may not write and only read.

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 the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

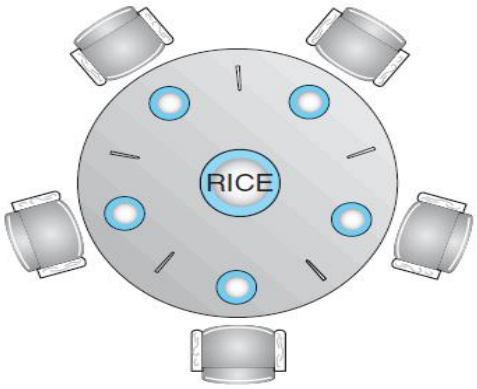


Fig: The dining philosophers Problem

This problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.

Semaphore 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];

The structure of philosopher i is shown

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

Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever. This situation is known as Deadlock.

Several possible remedies to the deadlock problem are:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick

Monitors

- One fundamental high-level synchronization construct
- A monitor is a synchronization construct that supports mutual exclusion and the ability to wait /block until a certain condition becomes true.
- A monitor is an abstract datatype that encapsulates data with a set of functions to operate on the data.

Course Code: 18CSC205J **Course Name: Operating Systems**

Characteristics of Monitor

- The local variables of a monitor can be accessed only by the local functions.
- A function defined within a monitor can only access the local variables of a monitor and its formal parameter.
- Only one process may be active within the monitor at a time.

Syntax of a Monitor

```
monitor monitor-name
{
// shared variable declarations
function P1 ( . . . ) {
. . .
}
function P2 ( . . . ) {
. . .
}
.
.
.
function Pn ( . . . ) {
. . .
}
initialization code ( . . . ) {
. . .
}
}
```

- A monitor type is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor.
- Instead of lock-based protection, monitors use a shared condition variable for synchronization, they are declared as

condition x, y;

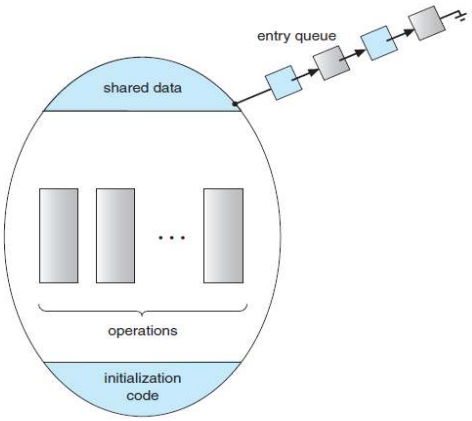


Fig: Schematic view of a monitor.

- Two operations on a condition variable:
  1. x.wait () –a process that invokes the operation is suspended.
  2. x.signal () –resumes one of the suspended processes (if any)

Dining-Philosophers Solution Using Monitors

```
monitor DP

{

enum { THINKING; HUNGRY, EATING) state [5] ; condition self [5];

void pickup (int i) { state[i] = HUNGRY; test(i);

if (state[i] != EATING) self [i].wait;

}

void putdown (int i) { state[i] = THINKING;

// test left and right neighbors test((i + 4) % 5);

test((i + 1) % 5);

}

void test (int i) {

if ( (state[(i + 4) % 5] != EATING) && (state[i] == HUNGRY) &&

(state[(i + 1) % 5] != EATING) ) { state[i] = EATING ;

self[i].signal () ;

}  }

initialization_code() { for (int i = 0; i < 5; i++) state[i] = THINKING;

}

}
```

We also need to declare condition self[5];

Each philosopher, before starting to eat, must invoke the operation pickup() followed by eating and finally invoke putdown().

```
DiningPhilosophers.pickup(i);
...
eat
```

...
DiningPhilosophers.putdown(i);

This solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur. However, with this solution it is possible for a philosopher to starve to death.

CPU Scheduling:

- Whenever the CPU becomes idle, the operating system selects one of the processes in the ready queue for execution.
- The selection process is carried out by the short-term scheduler (or CPU scheduler).
- The ready queue is not necessarily a first-in, first-out (FIFO) queue. It may be a FIFO queue, a priority queue, a tree, or simply an unordered linked list.
- CPU scheduling decisions may take place under the following four circumstances:
    1. When a process switches from the running state to the waiting state
    2. When a process switches from the running state to the ready state
    3. When a process switches from the waiting state to the ready state
    4. When a process terminates
- CPU Scheduling can be preemptive or non-preemptive:
- **Non-preemptive Scheduling**: In nonpreemptive scheduling, once the CPU has been allocated a process, the process keeps the CPU until it releases the CPU either by termination or by switching to the waiting state.
- **Preemptive Scheduling**: In preemptive scheduling, the CPU is allocated to the processes for a limited CPU cycle time, When the burst time of the process is greater than CPU cycle, it is moved back to the ready queue and will execute in the next chance. i.e CPU is taken back from the current process before it finishes its execution and can be allocated to other process.
- 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
- **Scheduling Criteria**

    1. **CPU utilization**: The CPU should be kept as busy as possible. CPU utilization may 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).
    2. **Throughput**: Itis the number of processes completed per time unit. For long processes, this rate may be 1 process per hour; for short transactions, throughput might be 10 processes per second.
    3. **Turnaround time**: 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.

4. Waiting time: Waiting time is the sum of the periods spent waiting in the ready queue.
5. Response time: It is the amount of time it takes to start responding, but not the time that it takes to output that response.

- **CPU Scheduling Algorithms**
  1. First-Come, First-Served Scheduling
  2. Shortest Job First Scheduling
  3. Priority Scheduling
  4. Round Robin Scheduling
  5. Multilevel queue Scheduling
  6. Multilevel feedback Scheduling
  7. Real Time scheduling: Rate Monotonic Scheduling and Deadline Scheduling

❖ First-Come, First-Served Scheduling (FCFS)

- The process that requests the CPU first is allocated the CPU first.
- It is a non-preemptive Scheduling technique.
- The implementation of the FCFS policy is easily managed with a FIFO queue

Example:

Process	Burst Time
P1	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

Gantt Chart

Average waiting time = (0+24+27) / 3 = 17 ms
Average Turnaround time = (24+27+30) / 3 = 27 ms

- The FCFS algorithm is particularly troublesome for time – sharing systems, where it is important that each user get a share of the CPU at regular intervals.

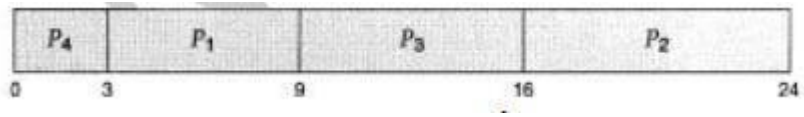
❖ Shortest Job First Scheduling

- The CPU is assigned to the process that has the smallest next CPU burst.
- If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie.

Example:

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Gantt Chart



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

Average turnaround time = (3+9+16+24) / 4 = 13 ms

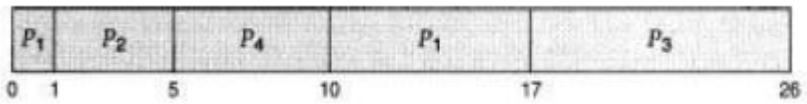
SJF can be Preemptive & non preemptive scheduling

Example 2:

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

❖ **Preemptive SJ F Scheduling or *shortest remaining time first***

- **It is a preemptive scheduling technique.**
- **Preemptive SJF is known as *shortest remaining time first (SRTF)***



Average waiting time :

P1 : 0+ (10 – 1) = 9

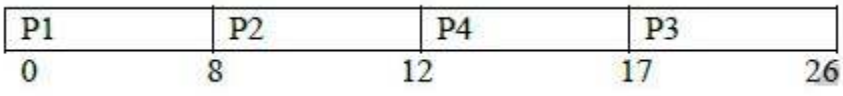
P2 : 1 – 1 = 0

P3 : 17 – 2 = 15

P4 : 5 – 3 = 2

AWT = (9+0+15+2) / 4 = 6.5 ms

❖ **Non-preemptive Scheduling**



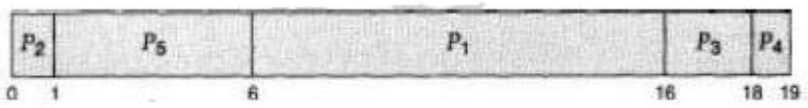
AWT = 0 + (8 – 1) + (12 – 3) + (17 – 2) / 4 = 7.75 ms

❖ Priority Scheduling

A priority is associated with each process, and the CPU is allocated to the process with the highest priority.(smallest integer - highest priority).

Example :

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



AWT=8.2 ms

- Priority Scheduling can be preemptive or non-preemptive.
- **Drawback:** Starvation – low priority processes may never execute.
- **Solution:** Aging – It is a technique of gradually increasing the priority of processes that wait in the system for a long time.

❖ Round-Robin Scheduling

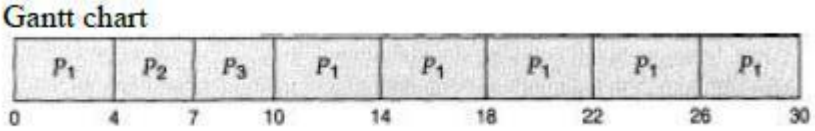
- The round-robin (RR) scheduling algorithm is designed especially for timesharing systems.
- It is similar to FCFS scheduling, but preemption is added to switch between processes.
- A small unit of time, called a time quantum (or time slice), is defined.
- The ready queue is treated as a circular queue.

Example:

Process	Burst Time
P1	24
P2	3
P3	3

Time Quantum = 4 ms.

Gantt chart



Waiting time
P1 = (0-0)+ (10-4) = 6
P2 = 4
P3 = 7 (6+4+7 / 3 = 5.66 ms)

- The average waiting time is 17/3 = 5.66 milliseconds.
- The performance of the RR algorithm depends heavily on the size of the time–quantum.
- If time-quantum is very large(infinite) then RR policy is same as FCFS policy.
- If time quantum is very small, RR approach is called processor sharing and appears to the users as though each of n process has its own processor running at 1/n the speed of real processor.

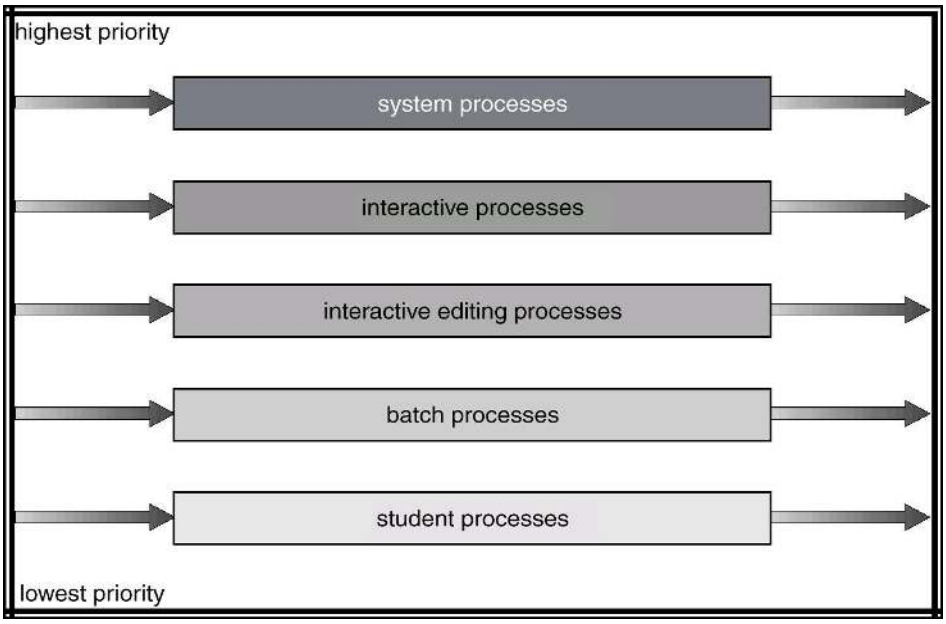
❖ **Multilevel Queue Scheduling**

- It 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.
- There must be scheduling between the queues, which is commonly implemented as a fixed-priority preemptive scheduling.
- For example the foreground queue may have absolute priority over the background queue.

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 queue.

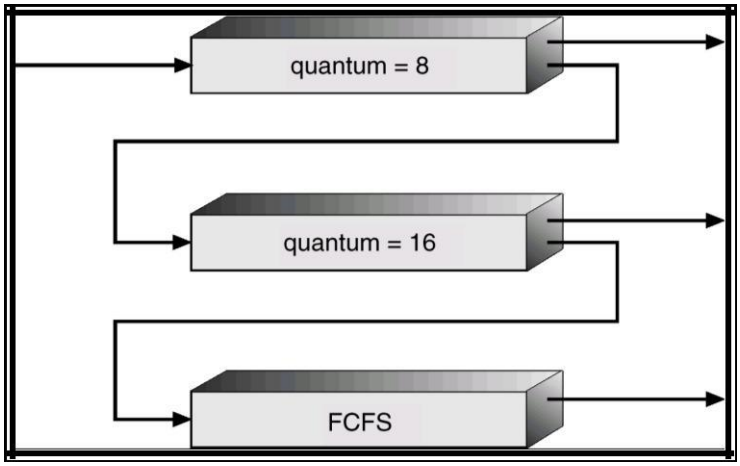


❖ Multilevel Feedback Queue Scheduling

- It 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.
- This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
- 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.

Example:

- Consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 .
- The scheduler first executes all processes in queue 0.
- Only when queue 0 is empty will it execute processes in queue 1.
- Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty.
- A process that arrives for queue 1 will preempt a process in queue 2.
- A process that arrives for queue 0 will, in turn, preempt a process in queue 1.



- A multilevel feedback queue scheduler is defined by the following parameters:
    1. The number of queues
    2. The scheduling algorithm for each queue
    3. The method used to determine when to upgrade a process to a higher priority queue
    4. The method used to determine when to demote a process to a lower-priority queue
    5. The method used to determine which queue a process will enter when that process needs service

Real Time scheduling:

Real Time scheduling: In which Process or job has deadline, execution is to be completed within the deadline, If a result is delayed, huge loss may happen.

Real-time computing is divided into two types.

1. **Hard real-time systems**: Hard Real-Time System must generate accurate responses to the events within the specified time, otherwise there may be huge loss. In Hard real time task must be serviced by its deadline. A hard real-time system is a purely deterministic and time constraint system. Examples: Flight Control Systems, Missile Guidance Systems, Weapons Defense System, Medical System etc

2. **Soft real-time systems**: In a soft real-time system, if the operation is not performed within the time limits, outputs / results are degraded. Soft real time provide no guarantee

as to when a critical real-time process will be scheduled. They guarantee only that the process will be given preference over noncritical processes. Ex: Weather Monitoring Systems, Electronic games, Multimedia system, Web browsing, Online transaction systems

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.
- The shorter the period, the higher the priority; the longer the period, the lower the priority will be assigned.
- Every time a process acquires the CPU, the duration of its CPU burst is the same for all the process.
- **Example:** Consider the following 3 process

Process	Capacity	Period
P1	**3**	**20**
P2	**2**	**5**
P3	**2**	**10**

Solution:
Step1: LCM of period (20,5,10) = 20

For P1: every 20 period need to execute 3 time slots / capacity
For P2: every 5 period need to execute 2 time slots / capacity
For P3: every 10 period need to execute 2 time slots / capacity
Step2: Assigning priority

P1 : 03 because time period 20 high
P2 : 01 because time period 5 less
P3 : 02 because time period 10 medium
Order of execution is p2, p3, p1

P2	P2	P3	P3	P1	P2	P2	P2	P2	P2	P2	P2	P2	P2	P2	P2	P2	P2	P2	P2
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Explanation:
Time slot 1, 2 -- p2 will execute 2 capacity
Time slot 3, 4 -- p3 will execute 2 capacity

Time slot 5 -- p1 will execute 1 capacity, after that next slot starts, so high priority job p2 have to continue, Therefore it preempts p1

Time slot 6, 7 -- p2 will execute again 2 capacity
The p3 does not required any slot, since for total 10 slots – it has to execute only 2 capacity
Time slot 8, 9 -- p1 will execute 2 capacity
Time slot 10 -- idle
Again next 5 mins slots start for p2,
Time slot 11, 12 -- p2 will execute again 2 capacity
Time slot 13, 14 -- p3 will execute again 2 capacity
Time slot 15 -- idle, since p1 have completed 3 capacity for 20 periods already
Time slot 16, 17 -- p2 will execute 2 capacity
Time slot 18,19,20 -- idle

Link : https://www.youtube.com/watch?v=xgW8VhEOpFg

Deadline Monotonic Scheduling

It is a fixed priority based algorithm in which priorities are assigned to each task based on their relative deadline. Task with shortest deadline is assigned highest priority. It is a Preemptive Scheduling Algorithm that means if any task of higher priority comes then, running task is preempted and higher priority task is assigned to CPU.

Priority of task is inversely proportional to deadline i.e., task with shortest deadline is assigned highest priority. Deadline is time limit in which task has to be completed.

Example –

Process	Capacity	Deadline	Period
P1	3	7	20
P2	2	4	5
P3	2	9	10

Solution:

Step1: LCM of period (20,5,10) = 20

For P1: every 20 period need to execute 3 time slots / capacity

For P2: every 5 period need to execute 2 time slots / capacity

For P3: every 10 period need to execute 2 time slots / capacity

Step2: Assigning priority

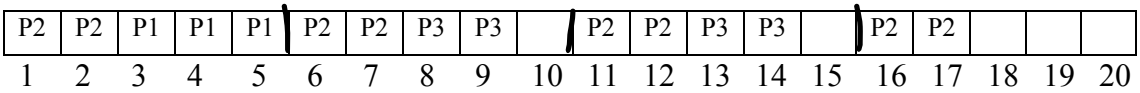
P1 : 02 because deadline is 7 high

P2 : 01 because deadline is 4 - less

P3 : 03 deadline is 9

Complete execution process is shown in figure below –

Order of execution is p2, p1, p3



Advantages:

Optimal for Static Priority Scheduling.

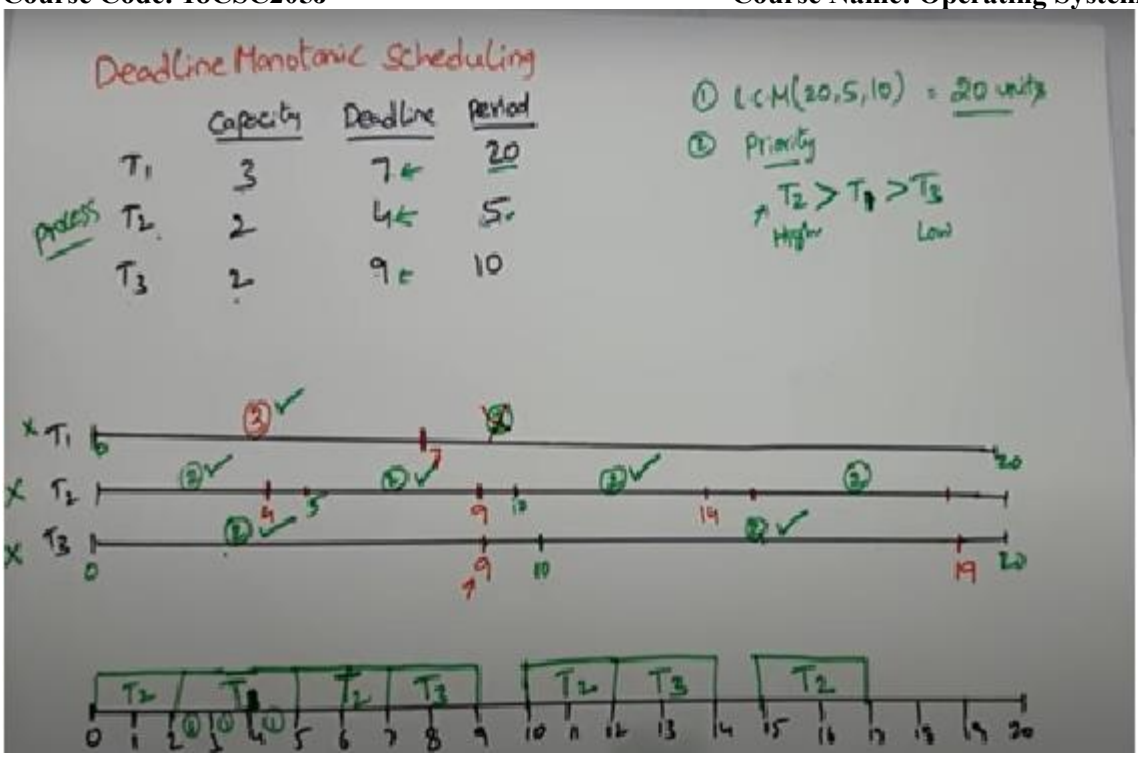
Performs well in case of availability of tasks having longer period but shorter deadline.

Good performance in case of overload.

Disadvantages:

Implementation is complex.

It is a time taking process.



https://www.youtube.com/watch?v=E6KGDpY_XoI

Deadlock

Definition: A process requests resources. If the resources are not available at that time ,the process enters a wait state. Waiting processes may never change state again because the resources they have requested are held by other waiting processes. This situation is called a deadlock.

A process must request a resource before using it, and must release resource after using it.

1. **Request:** If the request cannot be granted immediately then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource
3. **Release:** The process releases the resource.

Deadlock Characterization

Four Necessary conditions for a deadlock

1. **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.

2. **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.

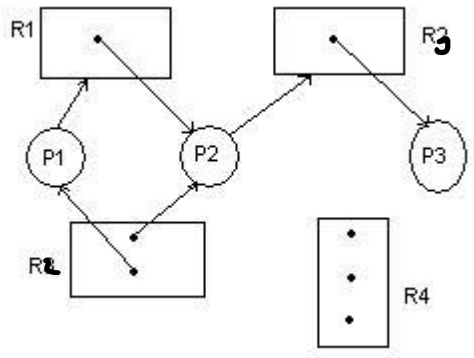
3. **No preemption:** Resources cannot be preempted.

4. **Circular wait:** P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by $P_2 ... P_{n-1}$.

Resource-Allocation Graph

- It is a Directed Graph with a set of vertices V and set of edges E.
- V is partitioned into two types:
    1. nodes P = {p1, p2,..pn}
    2. Resource type R ={R1,R2,...Rm}
- Pi -->Rj - request => request edge
- Rj-->Pi - allocated => assignment edge.
- Pi is denoted as a circle and Rj as a square.
- Rj may have more than one instance represented as a dot with in the square.

Sets P,R and E.
P = { P1,P2,P3}
R = {R1,R2,R3,R4}
E= {P1->R1, P2->R3, R1->P2, R2->P1, R3->P3 }

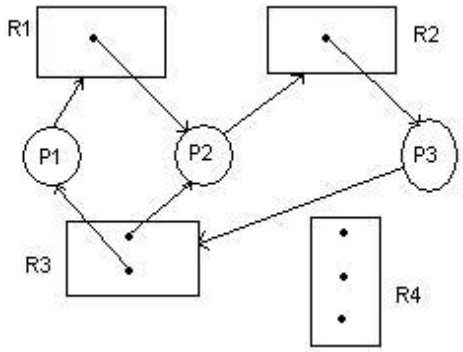


- Resource instances
  One instance of resource type R1, Two instance of resource type R2,One instance of resource type R3,Three instances of resource type R4.

Process states

Process P1 is holding an instance of resource type R2, and is waiting for an instance of resource type R1.

Resource Allocation Graph with a deadlock



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.

P1->R1->P2->R3->P3->R2->P1
P2->R3->P3->R2->P2

Methods for handling Deadlocks

1. Deadlock Prevention
2. Deadlock Avoidance
3. Deadlock Detection and Recovery

2.11.2 Deadlock Prevention:

- This ensures that the system never enters the deadlock state.
- Deadlock prevention is a set of methods for ensuring that at least one of the necessary conditions cannot hold.
- By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

1. Denying Mutual exclusion

- Mutual exclusion condition must hold for non-sharable resources.
- Printer cannot be shared simultaneously shared by prevent processes.
- sharable resource - example Read-only files.
- 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.

2. Denying Hold and wait

- Whenever a process requests a resource, it does not hold any other resource.
- One technique that can be used requires each process to request and be allocated all its resources before it begins execution.
- Another technique is before it can request any additional resources, it must release all the resources that it is currently allocated.
- These techniques have two main **disadvantages**:
  - First, resource utilization may be low, since many of the resources may be allocated but unused for a long time.
  - We must request all resources at the beginning for both protocols. starvation is possible.

3. Denying No preemption

- 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 currently being held are preempted. (**ALLOW PREEMPTION**)
- These resources are implicitly released.
- The process will be restarted only when it can regain its old resources.

4. Denying Circular wait

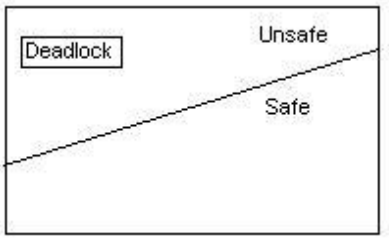
- Impose a total ordering of all resource types and allow each process to request for resources in an increasing order of enumeration.
- Let R = {R1,R2,...Rm} be the set of resource types.
- Assign to each resource type a unique integer number.
- If the set of resource types R includes tapedrives, disk drives and printers.

F(tapedrive)=1,
F(diskdrive)=5,
F(Printer)=12.

- Each process can request resources only in an increasing order of enumeration.

2.11.3 Deadlock Avoidance:

- Deadlock avoidance request that the OS be given in advance additional information concerning which resources a process will request and use during its life time. With this information it can be decided for each request whether or not the process should wait.
- To decide whether the current request can be satisfied or must be delayed, a system must consider the resources currently available, the resources currently allocated to each process and future requests and releases of each process.
- **Safe State**
  A state is safe if the system can allocate resources to each process in some order and still avoid a dead lock.



- A deadlock is an unsafe state.
- Not all unsafe states are dead locks
- An unsafe state may lead to a dead lock

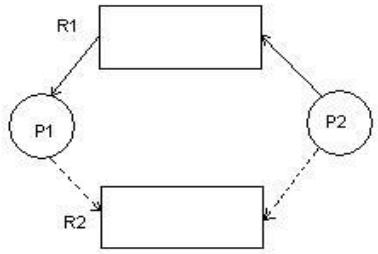
- **Two algorithms are used for deadlock avoidance namely;**
  1. Resource Allocation Graph Algorithm - single instance of a resource type.
  2. Banker's Algorithm – several instances of a resource type.

Resource allocation graph algorithm

- **Claim edge** - Claim edge P_i---> R_j indicates that process Pi may request resource Rj at some time, represented by a dashed directed edge.
- When process Pi request resource R_j, the claim edge P_i -> R_j is converted to a request edge.

Similarly, when a resource R_j is released by P_i the assignment edge R_j -> P_i is reconverted to a claim edge P_i -> R_j

1. The request can be granted only if converting the request edge P_i -> R_j to an assignment edge R_j -> P_i does not form a cycle.



- If no cycle exists, then the allocation of the resource will leave the system in a safe state.
- If a cycle is found, then the allocation will put the system in an unsafe state.

Banker's algorithm

- **Available:** indicates the number of available resources of each type.
- **Max:** Max[i, j]=k then process P_i may request at most k instances of resource type R_j
- **Allocation :** Allocation[i. j]=k, then process P_i is currently allocated K instances of resource type R_j
- **Need :** if Need[i, j]=k then process P_i may need K more instances of resource type R_j

Need [i, j]=Max[i, j]-Allocation[i, j]

Safety algorithm

 Initialize work := available and Finish [i]:=false for i=1,2,3 .. n

 Find an i such that both

i. Finish[i]=false

ii. $Need_i$<= Work

if no such i exists, goto step 4

3. work :=work+ $allocation_i$;

Finish[i]:=true

goto step 2

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

Resource Request Algorithm

Let $Request_i$ be the request from process P_i for resources.

- If $Request_i$<= $Need_i$ goto step2, otherwise raise an error condition, since the process has exceeded its maximum claim.

- If $Request_i$ <= Available, goto step3, otherwise P_i must wait, since the resources are not available.
- Available := Availabe-$Request_i$;

$Allocation_i := Allocation_i + Request_i$

$Need_i := Need_i - Request_i$;

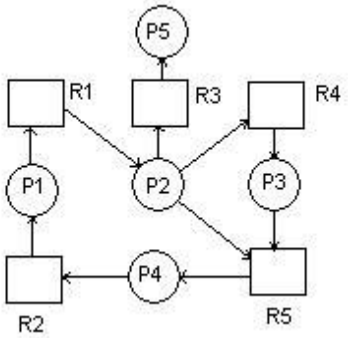
- Now apply the safety algorithm to check whether this new state is safe or not.
- If it is safe then the request from process P_i can be granted.

2.11.4 Deadlock detection

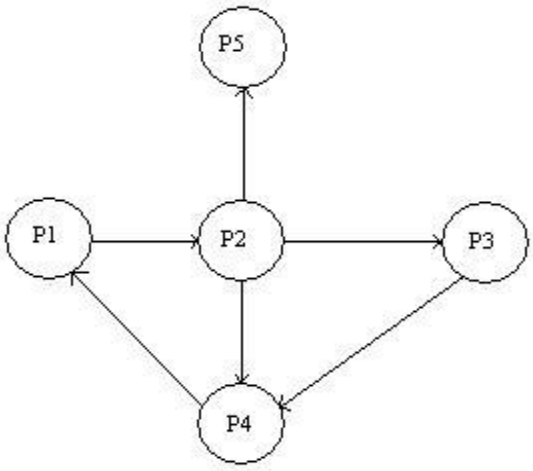
(i) Single instance of each resource type

- If all resources have only a single instance, then we can define a deadlock detection algorithm that use a variant of resource-allocation graph called a wait for graph.

Resource Allocation Graph



Wait for Graph



(ii) Several Instance of a resource type

Available : Number of available resources of each type
Allocation : number of resources of each type currently allocated to each process
Request : Current request of each process
If Request [i,j]=k, then process P_i is requesting K more instances of resource type R_j.

1.Initialize work := available Finish[i]=false,
otherwise finish [i]:=true
2. Find an index i such that both
a. Finish[i]=false
b. $Request_i$<=work
if no such i exists go to step4.
3. Work:=work+$allocation_i$
Finish[i]:=true
goto step2
4. If finish[i]=false
then process P_i is deadlocked

2.11.5 Deadlock Recovery

1. Process Termination

1. Abort all deadlocked processes.
2. Abort one deadlocked process at a time until the deadlock cycle is eliminated.

After each process is aborted , a deadlock detection algorithm must be invoked to determine where any process is still dead locked.

2. Resource Preemption

Preemptive some resources from process and give these resources to other processes until the deadlock cycle is broken.

i. **Selecting a victim:** which resources and which process are to be preempted.

ii. **Rollback:** if we preempt a resource from a process it cannot continue with its normal execution. It is missing some needed resource. We must rollback the process to some safe state, and restart it from that state.

iii. **Starvation:** How can we guarantee that resources will not always be preempted from the same process.

Course Code: 18CSC205J **Course Name: Operating Systems**

Homework:

1. Suppose that the following processes arrive for execution at the times indicated. Each process will run the listed amount of time. In answering the questions, use non-preemptive scheduling and base all decisions on the information you have at the time the decision must be made. (Nov/Dec 2018)

Process	Arrival Time	Burst Time
P1	0.0	8
P2	0.4	4
P3	1.0	1

   a. Find the average turnaround time for these processes with the FCFS scheduling algorithm?

   b. Find the average turnaround time for these processes with the SJF scheduling algorithm?

   c. The SJF algorithm is supposed to improve performance, but notice that we chose to run process P1 at time 0 because we did not know that two shorter processes would arrive soon. Find what is the average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used

2. Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

   The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.

   a. Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, A non pre-emptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1) scheduling.
   b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

3. Explain the FCFS, preemptive and non-preemptive versions of Shortest Job First and Round Robin (time-slice2) scheduling algorithms with Gantt Chart for the four processes given. Compare their average turn around and waiting time

Process	Arrival Time	Burst Time
P1	0.00	8
P2	1.001	4
P3	2 .001	9

Course Code: 18CSC205J **Course Name: Operating Systems**

P4	3.001	5
p5	4.001	3

Example 4: **Evaluate the following snapshot of the system (10Marks)**
Assume that there are three resources, A, B, and C. There are 4 processes P0 to P3. At T0 we have the following snapshot of the system:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	1	0	1	2	1	1	2	1	1
P_1	2	1	2	5	4	4			
P_2	3	0	0	3	1	1			
P_3	1	0	1	1	1	1			

Is the system in a safe state? Why or why not? (10m)

5. Evaluate the following snapshot of the system (10Marks)

	Allocation	**Max**	**Available**
	A B C D	**A B C D**	**A B C D**
			1 5 2 0
Po	**0 0 1 2**	**0 0 1 2**	
P1	**1 0 0 0**	**1 7 5 0**	
P2	**1 3 5 4**	**2 3 5 6**	
P3	**0 6 3 2**	**0 6 5 2**	
P4	**0 0 1 4**	**0 6 5 6**	

Answer the follow based on banker's algorithm.
1.Define safety algorithm
2. What is the content of need matrix?
3. Is the system in a safe state?
4. Is a request from process P1arrives for(0,4,2,0) can the request be granted immediately?