ASSIGNMENT

By

SIMRITI KAK

2022A1R004

3RD SEM (A2)

CSE

GROUP A

Model Institute of Engineering & Technology (Autonomous)

(Permanently Affiliated to the University of Jammu, Accredited by NAAC with "A" Grade)

Jammu, India

2023

Assignment

Subject Code: COM [302]

Due Date:4 December 2023

Question Number	**Course Outcomes**	**Blooms' Level**	**Maximum Marks**	**Marks Obtain**
Q1	CO 4	3-4	10	
Q2	CO 5	3-4	10	
Total Marks			20	
Faculty Signature: Dr Mekhla Sharma (Assistant Professor) Email: Mekhla.cse@gmail.com				

TABLE OF CONTENT:

Task – 1

Design a program that implements priority-based process scheduling. Create a set of processes with different priorities and demonstrate how the operating system schedules these processes based on their priorities. Implement and analyze both preemptive and non-preemptive versions of the priority scheduling algorithm.

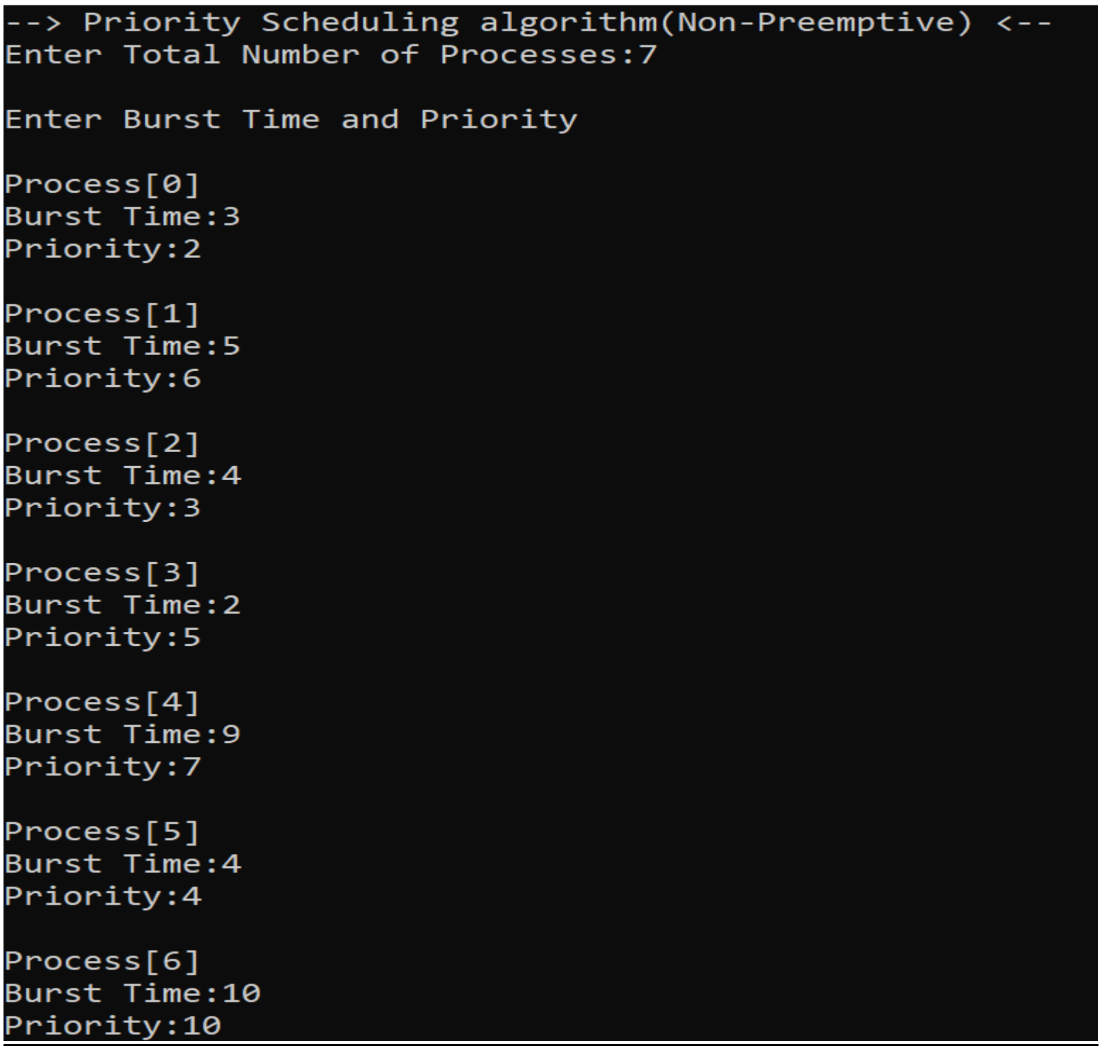
Efficient Task Management: A Priority-Based Non-Preemptive Scheduling Algorithm Implementation (Assuming all processes arrived at time 0):

```
#include <stdio.h>


void main() {
   int burstTime[20], process[20], waitingTime[20], turnaroundTime[20], priority[20],
arrivalTime[20], i, j, numProcesses, total=0, position, temp, avgWaitingTime,
avgTurnaroundTime;
   printf("--> Priority Scheduling algorithm(Non-Preemptive) <--\n");
   printf("Enter Total Number of Processes:");
   scanf("%d", &numProcesses);


   printf("\nEnter Burst Time and Priority\n");
   for(i = 0; i < numProcesses; i++) {
      printf("\nProcess[%d]\n", i);
      printf("Burst Time:");
      scanf("%d", &burstTime[i]);
      printf("Priority:");
      scanf("%d", &priority[i]);
      process[i] = i; //contains process number
   }


   // Sorting burst time, priority, and process number in ascending order using
selection sort
   for(i = 0; i < numProcesses; i++) {
      position = i;
```

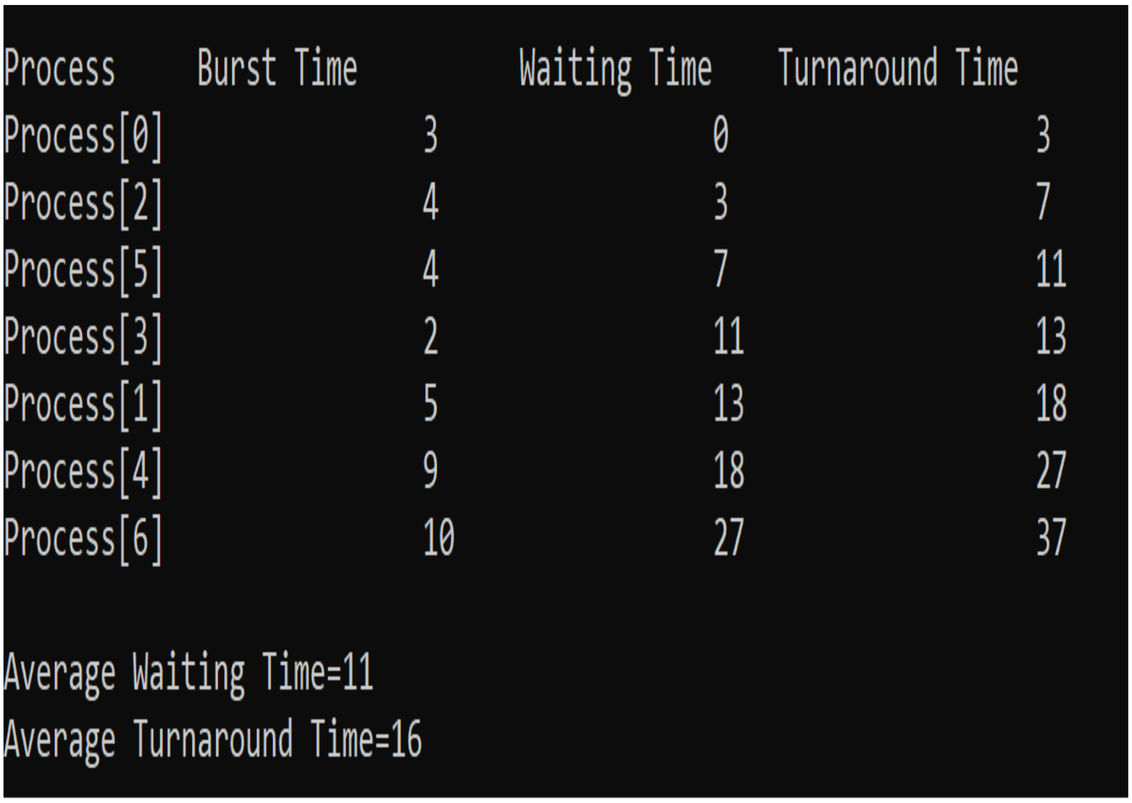
```
    for(j = i + 1; j < numProcesses; j++) {
        if(priority[j] < priority[position])
            position = j;
    }

    temp = priority[i];
    priority[i] = priority[position];
    priority[position] = temp;

    temp = burstTime[i];
    burstTime[i] = burstTime[position];
    burstTime[position] = temp;

    temp = process[i];
    process[i] = process[position];
    process[position] = temp;
}

waitingTime[0] = 0; //waiting time for the first process is zero

// Calculate waiting time
for(i = 1; i < numProcesses; i++) {
    waitingTime[i] = burstTime[i - 1] + waitingTime[i - 1];
    total += waitingTime[i];
}

avgWaitingTime = total / numProcesses; //average waiting time
total = 0;

printf("\nProcess\t    Burst Time    \tWaiting Time\tTurnaround Time");
for(i = 0; i < numProcesses; i++) {
    turnaroundTime[i] = burstTime[i] + waitingTime[i]; //calculate turnaround time
```

```
        total += turnaroundTime[i];

        printf("\nProcess[%d]\t\t  %d\t\t    %d\t\t\t%d", process[i], burstTime[i],
waitingTime[i], turnaroundTime[i]);

    }

    avgTurnaroundTime = total / numProcesses; //average turnaround time

    printf("\n\nAverage Waiting Time=%d", avgWaitingTime);

    printf("\nAverage Turnaround Time=%d\n", avgTurnaroundTime);

}
```

Output:

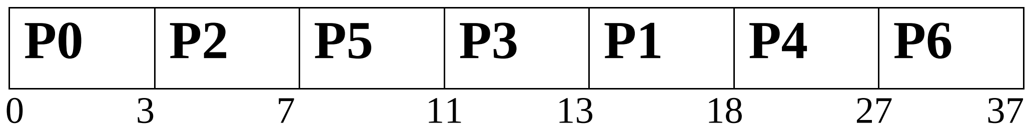


```
--> Priority Scheduling algorithm(Non-Preemptive) <--
Enter Total Number of Processes:7

Enter Burst Time and Priority

Process[0]
Burst Time:3
Priority:2

Process[1]
Burst Time:5
Priority:6

Process[2]
Burst Time:4
Priority:3

Process[3]
Burst Time:2
Priority:5

Process[4]
Burst Time:9
Priority:7

Process[5]
Burst Time:4
Priority:4

Process[6]
Burst Time:10
Priority:10
```



```
Process     Burst Time          Waiting Time    Turnaround Time
Process[0]              3                 0                   3
Process[2]              4                 3                   7
Process[5]              4                 7                  11
Process[3]              2                11                  13
Process[1]              5                13                  18
Process[4]              9                18                  27
Process[6]              10               27                  37

Average Waiting Time=11
Average Turnaround Time=16
```

<u>GYANNT CHART FOR NON PREEMPTIVE SCHEDULING ALGORITHM:</u>



Analyis of Code and its Explaination:

In the context of operating systems and task scheduling, a priority-based non-preemptive algorithm is a scheduling algorithm that assigns priorities to different tasks or processes and allows the task with the highest priority to execute first. Non-preemptive means that once a task starts executing, it cannot be interrupted until it completes its execution or voluntarily yields the CPU.

1. ***Non-Preemptive Scheduling:***

    - Once a task starts executing, it cannot be interrupted until it completes its execution or voluntarily relinquishes control of the CPU.

    - In non-preemptive scheduling, the operating system does not forcibly stop a running task to start or switch to another task. It waits for the currently running task to finish.

Here's a simple example to illustrate the concept:

Suppose you have three tasks with different priorities: Task A with priority 3, Task B with priority 1, and Task C with priority 2. In a priority-based non-preemptive algorithm:

- If Task A arrives and no other task is running, Task A will be scheduled to run.
- If Task B arrives while Task A is running, it will have to wait until Task A completes because the non-preemptive nature does not allow interrupting a running task.
- If Task C arrives after Task A completes, and Task B is still waiting, Task C will be scheduled next because it has a higher priority than Task B.

Priority-based non-preemptive scheduling has advantages such as simplicity and ease of implementation. However, it may lead to potential issues like priority inversion, where a low-priority task holds a resource needed by a high-priority task, causing a delay in the execution of the high-priority task. To address this, more sophisticated scheduling algorithms like priority inheritance or priority ceiling protocols may be used.

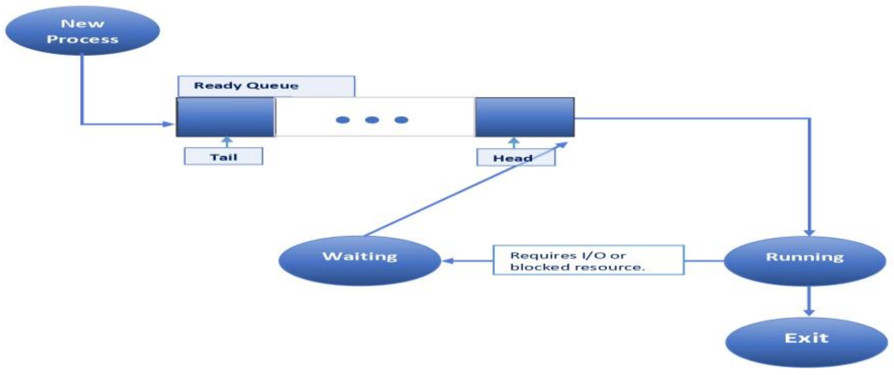


Fig No. 1

This C program implements the Priority Scheduling algorithm for non-preemptive scheduling. Here's a step-by-step explanation:

1. ***Variable Declarations:***

   - **burstTime[20]:** Array to store the burst time of each process.
   - **process[20]:** Array to store the process number.
   - **waitingTime[20]**: Array to store the waiting time of each process.
   - **turnaroundTime[20]**: Array to store the turnaround time of each process.
   - **priority[20]:** Array to store the priority of each process.
   - **arrivalTime[20]**: Array to store the arrival time of each process (though it is declared, it is not used in the provided code).
   - **i, j, numProcesses**: Variables for iteration and storing the number of processes.
   - **total:** Variable to store the total waiting time or turnaround time.
   - **position:** Variable used for finding the position of the process with the minimum priority during sorting.
   - **temp:** Temporary variable used for swapping values during sorting.
   - **avgWaitingTime, avgTurnaroundTime:** Variables to store the average waiting time and average turnaround time.

2. ***Input Section:***
   - The program prompts the user to enter the total number of processes.
   - It then takes input for each process, including burst time and priority.

3. ***Sorting Section:***
   - The program uses the Selection Sort algorithm to sort the processes based on their priority in ascending order.
   - The sorting involves rearranging the ***priority[], burstTime[],*** and ***process[]*** arrays.

4. ***Waiting Time Calculation:***
   - The waiting time for the first process is set to 0.
   - For subsequent processes, the waiting time is calculated as the sum of the burst times of all previous processes.

5. ***Turnaround Time Calculation:***

- The turnaround time for each process is calculated as the sum of its burst time and waiting time.

6. ***Output Section:***
   - The program then prints a table showing the process number, burst time, waiting time, and turnaround time for each process.
   - It also calculates and prints the average waiting time and average turnaround time

7. ***Note:***
   - The **main()** function should return an integer **(int**) instead of void to conform to C standards (**int main()).**
   - The **arrivalTime** array is declared but not used in the code.

Algorithm Steps:

A non-preemptive priority-based scheduling algorithm selects the next task to run based on the priority assigned to each task. Here are the general steps for a non-preemptive priority-based scheduling algorithm:

1. ***Initialization:***
   - Assign priorities to each task. Lower numeric values usually indicate higher priorities.
2. ***Task Arrival:***
   - When a new task arrives, assign it a priority based on its characteristics or the system's policies.
3. ***Task Selection:***
   - Select the task with the highest priority that is ready to execute. This is the task that will be scheduled to run next.
4. ***Execution:***
   - Execute the selected task until it completes or voluntarily yields the CPU.
5. ***Task Completion**:*
   - If the task completes its execution, remove it from the ready queue.
6. ***Task Waiting:***
   - If a task is waiting for an event (e.g., I/O completion), it remains in the ready queue but is not scheduled for execution until the event occurs.
7. ***Task Arrival During Execution:***
   - If a new task arrives while another task is executing, compare its priority with the priority of the running task.
     - If the arriving task has a higher priority, let it wait until the current task completes.
     - If the arriving task has a lower or equal priority, let it wait in the ready queue.
8. ***Repeat:***
   - Repeat the process from step 3 until all tasks are completed.

Advantages of Priority-Based Non-Preemptive Scheduling:

1. ***Simplicity:***
   - Priority-based non-preemptive scheduling algorithms are relatively simple to implement compared to more complex preemptive algorithms.
2. ***Deterministic Execution:***
   - The deterministic nature of non-preemptive scheduling can simplify system analysis and debugging since the execution order is more predictable.
3. ***Low Overhead:***
   - Non-preemptive scheduling tends to have lower overhead compared to preemptive scheduling since there is no need for frequent context switching.
4. ***Efficient for Certain Workloads:***
   - In scenarios where tasks have well-defined priorities and do not require frequent preemption, non-preemptive priority scheduling can be efficient.

Limitations of Priority-Based Non-Preemptive Scheduling:

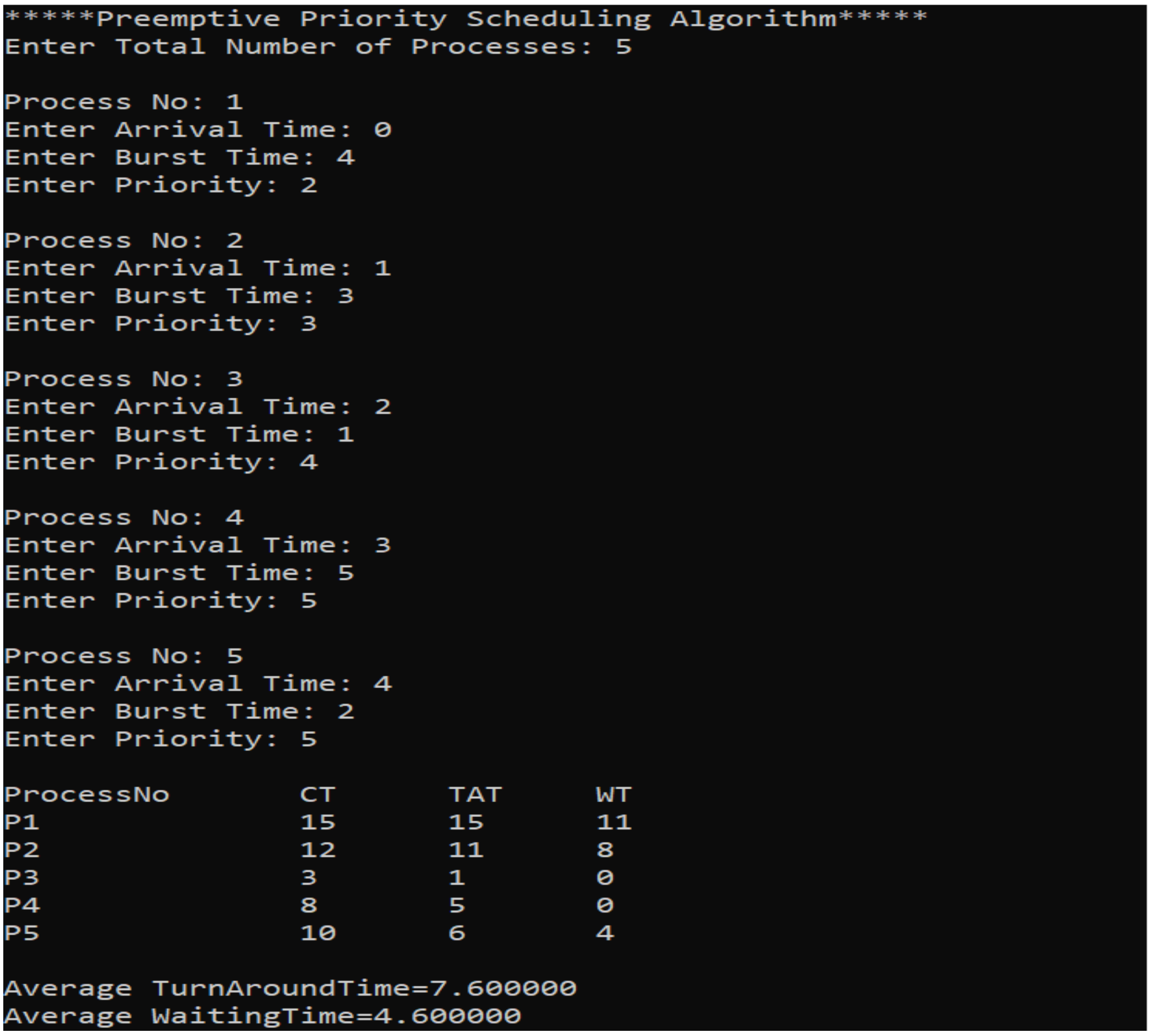
1. ***Inefficiency in Dynamic Environments:***
   - In dynamic environments where task priorities change frequently, non-preemptive scheduling may lead to suboptimal performance as priorities assigned at the start may not be suitable throughout the entire execution.
2. ***Potential for Priority Inversion:***
   - Priority inversion can occur when a low-priority task holds a resource that a high-priority task needs, causing delays in the execution of the high-priority task. This issue can impact system responsiveness.
3. ***Starvation:***
   - Lower priority tasks may experience starvation if higher priority tasks continuously arrive. These lower priority tasks may never get a chance to execute if higher priority tasks keep arriving.
4. ***Difficulty Handling Real-Time Constraints:***
   - Non-preemptive scheduling may struggle to meet real-time constraints, especially if high-priority tasks are delayed due to lower priority tasks currently running.
5. ***Not Suitable for Time-Sharing Systems:***
   - Non-preemptive scheduling is not well-suited for time-sharing systems where multiple users interact with the system simultaneously, as it may result in poor responsiveness.
6. ***Limited Resource Utilization:***
   - The non-preemptive nature of the algorithm may lead to underutilization of resources, especially if higher priority tasks are waiting for lower priority tasks to complete.

Dynamic Task Control: Implementing a Preemptive Scheduling Algorithm

```
#include <stdio.h>
#define MIN_PRIORITY -9999


struct Process {
    int number, arrivalTime, burstTime, remainingTime, completionTime,
waitingTime, turnaroundTime, priority, tempPriority;
};


struct Process readProcess(int i) {
    struct Process p;
    printf("\nProcess No: %d\n", i);
    p.number = i;
    printf("Enter Arrival Time: ");
    scanf("%d", &p.arrivalTime);
    printf("Enter Burst Time: ");
    scanf("%d", &p.burstTime);
    p.remainingTime = p.burstTime;
    printf("Enter Priority: ");
    scanf("%d", &p.priority);
    p.tempPriority = p.priority;
    return p;
}


void main() {
    int i, n, currentTime, remainingProcesses, maxPriority, maxPriorityIndex;
    struct Process processes[10], tempProcess;
    float averageTurnaroundTime = 0, averageWaitingTime = 0;


    printf("*****Preemptive Priority Scheduling Algorithm*****\n");
printf("Enter Total Number of Processes: ");
scanf("%d", &n);

for (i = 0; i < n; i++)
    processes[i] = readProcess(i + 1);

remainingProcesses = n;

for (int i = 0; i < n - 1; i++)
    for (int j = 0; j < n - i - 1; j++)
        if (processes[j].arrivalTime > processes[j + 1].arrivalTime) {
            tempProcess = processes[j];
            processes[j] = processes[j + 1];
            processes[j + 1] = tempProcess;
        }

maxPriority = processes[0].tempPriority;
maxPriorityIndex = 0;

for (int j = 0; j < n && processes[j].arrivalTime <= processes[0].arrivalTime; j++)
    if (processes[j].tempPriority > maxPriority)
        maxPriority = processes[j].tempPriority, maxPriorityIndex = j;

i = maxPriorityIndex;
currentTime = processes[i].completionTime = processes[i].arrivalTime + 1;
processes[i].remainingTime--;

if (processes[i].remainingTime == 0) {
    processes[i].tempPriority = MIN_PRIORITY;
    remainingProcesses--;
    }

    while (remainingProcesses > 0) {
        maxPriority = processes[0].tempPriority;
        maxPriorityIndex = 0;

        for (int j = 0; j < n && processes[j].arrivalTime <= currentTime; j++)
            if (processes[j].tempPriority > maxPriority)
                maxPriority = processes[j].tempPriority, maxPriorityIndex = j;

        i = maxPriorityIndex;
        processes[i].completionTime = currentTime = currentTime + 1;
        processes[i].remainingTime--;

        if (processes[i].remainingTime == 0) {
            processes[i].tempPriority = MIN_PRIORITY;
            remainingProcesses--;
        }
    }

    printf("\nProcessNo\tCT\tTAT\tWT\n");

    for (int i = 0; i < n; i++) {
        processes[i].turnaroundTime = processes[i].completionTime -
processes[i].arrivalTime;
        averageTurnaroundTime += processes[i].turnaroundTime;
        processes[i].waitingTime = processes[i].turnaroundTime -
processes[i].burstTime;
        averageWaitingTime += processes[i].waitingTime;
        printf("P%d\t\t%d\t%d\t%d\n", processes[i].number,
processes[i].completionTime, processes[i].turnaroundTime,
processes[i].waitingTime);
```

```
    }



    averageTurnaroundTime /= n, averageWaitingTime /= n;

    printf("\nAverage TurnAroundTime=%f\nAverage WaitingTime=%f",
averageTurnaroundTime, averageWaitingTime);

}
```

Output:



```
*****Preemptive Priority Scheduling Algorithm*****
Enter Total Number of Processes: 5

Process No: 1
Enter Arrival Time: 0
Enter Burst Time: 4
Enter Priority: 2

Process No: 2
Enter Arrival Time: 1
Enter Burst Time: 3
Enter Priority: 3

Process No: 3
Enter Arrival Time: 2
Enter Burst Time: 1
Enter Priority: 4

Process No: 4
Enter Arrival Time: 3
Enter Burst Time: 5
Enter Priority: 5

Process No: 5
Enter Arrival Time: 4
Enter Burst Time: 2
Enter Priority: 5

ProcessNo       CT      TAT     WT
P1              15      15      11
P2              12      11      8
P3              3       1       0
P4              8       5       0
P5              10      6       4

Average TurnAroundTime=7.600000
Average WaitingTime=4.600000
```

<u>Gantt Chart For Preemptive Scheduling Algorithm:</u>

(Priority: Higher to Lower)

P1	**P2**	**P3**	**P4**	**P4**	**P5**	**P2**	**P1**

0 1 2 3 4 8 10 12 15

<u>Analysis of code and its Explaintion:</u>

Preemptive Scheduling:

In preemptive scheduling algorithms, tasks or processes can be forcibly interrupted while executing, and the CPU is given to another task based on priority or other scheduling criteria. Here's an analysis:

1. **Task Interruption:**
   - Unlike non-preemptive scheduling, preemptive scheduling allows a running task to be interrupted before completion if a higher-priority task becomes available or if the running task exceeds its time quantum (in the case of time-sliced scheduling).
2. **Dynamic Task Switching:**
   - The operating system can dynamically switch between tasks based on factors such as priority, time quantum expiration, or external events, leading to more responsive and adaptable system behavior.
3. **Priority-Based Preemption:**
   - In a priority-based preemptive algorithm, the task with the highest priority that arrives or becomes ready to execute is scheduled to run immediately, even if a lower-priority task is currently executing.
4. **Example Scenario:**
   - Suppose Task A with priority 3 is running, and Task B with priority 1 arrives. In a preemptive priority-based algorithm, Task B could preempt Task A and start executing immediately due to its higher priority.
5. **Priority Inversion and Solutions:**
   - Similar to non-preemptive scheduling, priority inversion can still be a concern in preemptive scheduling. It occurs when a low-priority task holds a resource needed by a high-priority task. Solutions like priority inheritance or priority ceiling protocols may be implemented to mitigate this issue.
6. **Time-Sliced Scheduling:**
   - In addition to priority-based preemption, preemptive scheduling can involve time-sliced scheduling, where each task is given a fixed time quantum to execute before the scheduler switches to the next task in the queue.
7. **Common Preemptive Algorithms:**
   - Examples of preemptive scheduling algorithms include Priority Scheduling, Round Robin Scheduling, and Multilevel Queue Scheduling.
8. **Consideration for Real-Time Systems**:

- Preemptive scheduling is often essential in real-time systems where tasks need to meet strict deadlines. It allows higher-priority real-time tasks to preempt lower-priority ones to meet their deadlines.

  In conclusion, preemptive scheduling provides dynamic task switching and responsiveness in the execution of tasks, but it introduces complexities and overhead. The choice between preemptive and non-preemptive scheduling depends on the specific requirements and characteristics of the system.

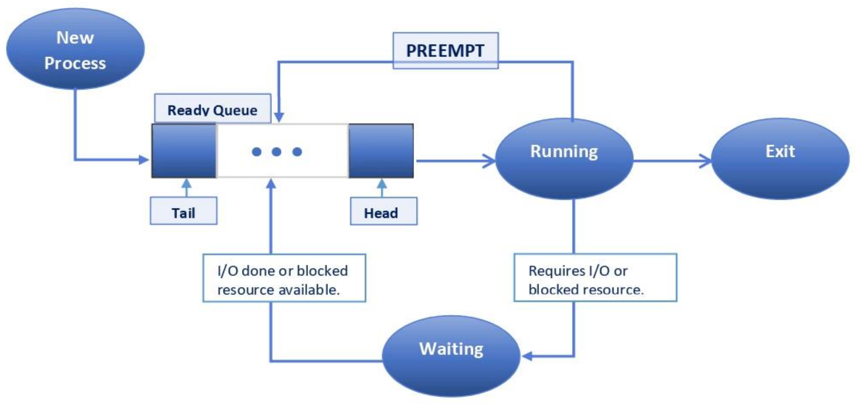

New Process
PREEMPT
Ready Queue
Running
Exit
Tail
Head
I/O done or blocked resource available.
Requires I/O or blocked resource.
Waiting

Fig No. 2

Algorithm Steps:

1. ***Structure Definition:***
   - Define a structure named **Process** to represent information about a process, including arrival time, burst time, remaining time, completion time, waiting time, turnaround time, priority, and temporary priority.
2. ***Input:***
   - Prompt the user to enter the number of processes (n).
   - For each process, read input values such as arrival time, burst time, and priority.
3. ***Sort Processes by Arrival Time:***
   - Sort the array of processes based on arrival time using the Bubble Sort algorithm.
4. ***Initialization:***
   - Initialize variables such as **remainingProcess**, **maxPriority**, **maxPriorityIndex, currentTime, and tempProcess.**
5. ***Main Loop (Scheduling):***
   - Execute a loop until all processes are completed **(remainingProcesses > 0).**
   - Find the process with the highest priority that has arrived and is ready to execute *(***maxPriority).**
   - Update the current time *(***currentTime**) and process completion time **(completionTime**).
   - Decrement the remaining time for the selected process.
6. ***Update Priority and Check Completion:***
   - If the remaining time for a process becomes zero, update its temporary priority to a minimum value **(MIN_PRIORITY**) to exclude it from future selections.

- Decrement the count of remaining processes.

7. ***Calculate Performance Metrics:***
    - Calculate turnaround time and waiting time for each process.
    - Accumulate these values to calculate average turnaround time and average waiting time.

8. ***Output:***
    - Display the process details, including arrival time, burst time, priority, completion time, turnaround time, and waiting time.
    - Print the average turnaround time and average waiting time.
9. ***End of Program:***
    - Terminate the program.

Advantages:

- Preemptive scheduling provides better responsiveness and ensures that higher-priority tasks are executed promptly.
- It can prevent a single long-running task from monopolizing the CPU, improving fairness in task execution.

Limitation:

- Increased complexity: Preemptive scheduling introduces the complexity of managing and handling task interruptions, context switches, and potential race conditions.
- Overhead: Context switching between tasks incurs some overhead, and excessive preemption can lead to increased system overhead.

TASK - 2:

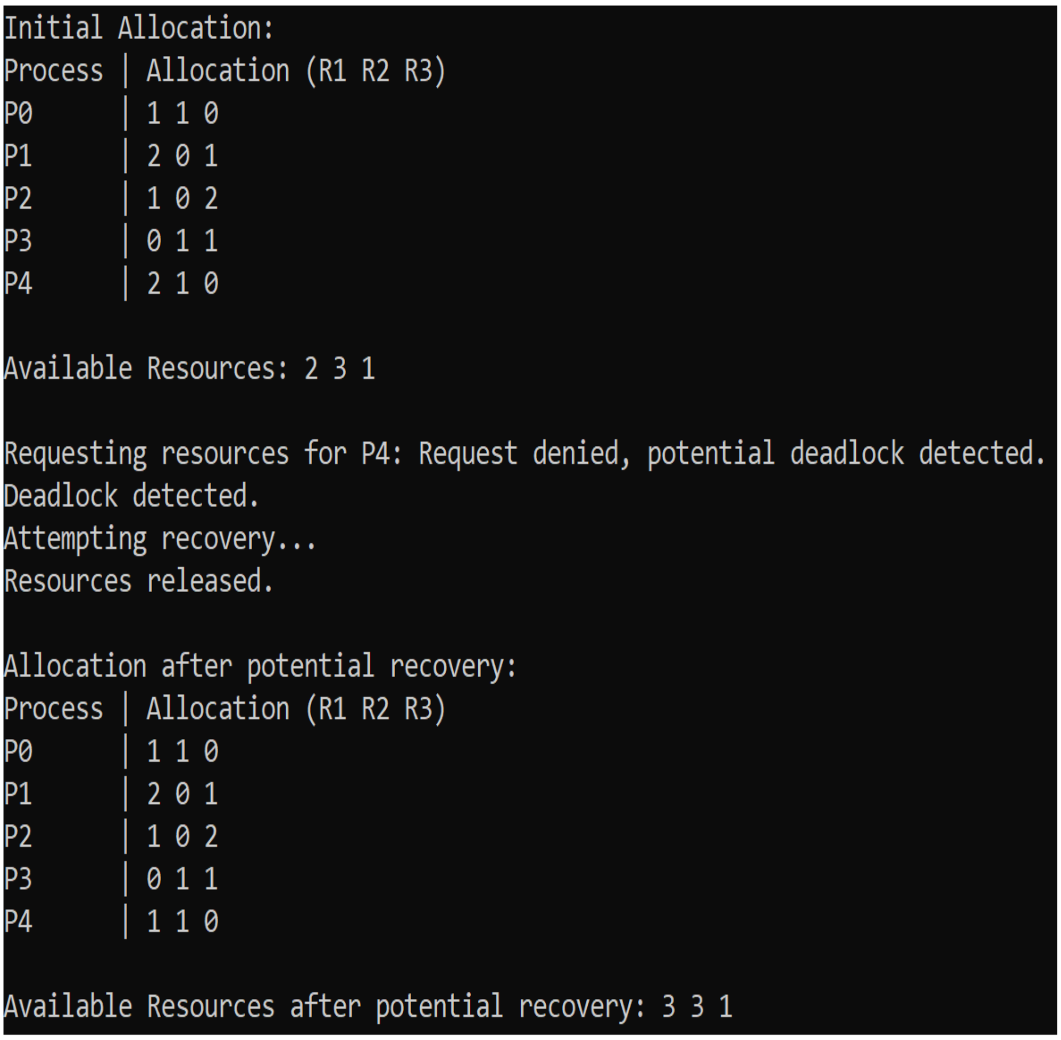
Design a program that simulates a memory allocation system with multiple processes requesting memory blocks. Implement a deadlock detection algorithm within the memory manager that can identify and report when a deadlock occurs. Also demonstrate how to recover from the deadlock by releasing memory resources.

```
#include <stdio.h>
#include <stdbool.h>

#define MAX_PROC 10
#define MAX_RES 10

int alloc[MAX_PROC][MAX_RES];
int max_alloc[MAX_PROC][MAX_RES];
int available[MAX_RES];
bool done[MAX_PROC];

int num_proc, num_res;

void init(int proc, int res, int a[][MAX_RES], int max[][MAX_RES], int avail[]) {
    num_proc = proc;
    num_res = res;

    for (int i = 0; i < num_proc; ++i) {
        for (int j = 0; j < num_res; ++j) {
            alloc[i][j] = a[i][j];
            max_alloc[i][j] = max[i][j];
        }
        done[i] = false;
    }

    for (int i = 0; i < num_res; ++i) {
        available[i] = avail[i];
    }
}

bool request_resources(int pid, int request[]) {
    for (int i = 0; i < num_res; ++i) {
        if (request[i] > available[i] || request[i] > max_alloc[pid][i] - alloc[pid][i]) {
            return false;
        }
    }

    for (int i = 0; i < num_res; ++i) {
        available[i] -= request[i];
        alloc[pid][i] += request[i];
    }

    return true;
}

void release_resources(int pid, int release[]) {
    for (int i = 0; i < num_res; ++i) {
        if (release[i] > alloc[pid][i]) {
            release[i] = alloc[pid][i];
        }

        available[i] += release[i];
        alloc[pid][i] -= release[i];
    }
}

bool is_deadlocked() {
    int work[num_res];
```

```
bool finish[num_proc];

for (int i = 0; i < num_res; ++i) {
    work[i] = available[i];
}

for (int i = 0; i < num_proc; ++i) {
    finish[i] = done[i];
}

bool deadlock = true;
while (deadlock) {
    deadlock = false;
    for (int i = 0; i < num_proc; ++i) {
        if (!finish[i]) {
            bool can_allocate = true;
            for (int j = 0; j < num_res; ++j) {
                if (max_alloc[i][j] - alloc[i][j] > work[j]) {
                    can_allocate = false;
                    break;
                }
            }
            if (can_allocate) {
                deadlock = false;
                finish[i] = true;
                for (int j = 0; j < num_res; ++j) {
                    work[j] += alloc[i][j];
                }
            }
        }
    }
}
```

```
    for (int i = 0; i < num_proc; ++i) {
        if (!finish[i]) {
            return true;
        }
    }

    return false;
}

void print_matrix(const char* name, int matrix[MAX_PROC][MAX_RES], int rows,
int cols) {
    printf("%s:\n", name);
    for (int i = 0; i < rows; ++i) {
        printf("P%d: ", i);
        for (int j = 0; j < cols; ++j) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int proc = 5;
    int res = 3;
    int a[][MAX_RES] = {{1, 1, 0}, {2, 0, 1}, {1, 0, 2}, {0, 1, 1}, {2, 1, 0}};
    int max[][MAX_RES] = {{3, 2, 2}, {3, 2, 3}, {4, 1, 3}, {1, 3, 1}, {4, 3, 3}};
    int avail[] = {2, 3, 1};

    init(proc, res, a, max, avail);

    printf("Initial Allocation:\n");
    printf("Process | Allocation (R1 R2 R3)\n");
    for (int i = 0; i < proc; ++i) {
        printf("P%d      | %d %d %d\n", i, alloc[i][0], alloc[i][1], alloc[i][2]);
    }

    printf("\nAvailable Resources: %d %d %d\n\n", available[0], available[1],
available[2]);

    int pid = 4;
    int request[] = {1, 0, 2};
    printf("Requesting resources for P%d: ", pid);
    if (!request_resources(pid, request)) {
        printf("Request denied, potential deadlock detected.\n");
        if (is_deadlocked()) {
            printf("Deadlock detected.\n");
            printf("Attempting recovery...\n");
            release_resources(pid, request);
            printf("Resources released.\n");
        }
    }

    printf("\nAllocation after potential recovery:\n");
    printf("Process | Allocation (R1 R2 R3)\n");
    for (int i = 0; i < proc; ++i) {
        printf("P%d      | %d %d %d\n", i, alloc[i][0], alloc[i][1], alloc[i][2]);
    }

    printf("\nAvailable Resources after potential recovery: %d %d %d\n", available[0],
available[1], available[2]);

    return 0;
}
```

Output:



```
Initial Allocation:
Process | Allocation (R1 R2 R3)
P0      | 1 1 0
P1      | 2 0 1
P2      | 1 0 2
P3      | 0 1 1
P4      | 2 1 0

Available Resources: 2 3 1

Requesting resources for P4: Request denied, potential deadlock detected.
Deadlock detected.
Attempting recovery...
Resources released.

Allocation after potential recovery:
Process | Allocation (R1 R2 R3)
P0      | 1 1 0
P1      | 2 0 1
P2      | 1 0 2
P3      | 0 1 1
P4      | 1 1 0

Available Resources after potential recovery: 3 3 1
```

Analysis of code and Its Explaination:

Memory allocation in a multiprocess system involves managing the allocation and deallocation of memory blocks for various processes running concurrently. The goal is to utilize the available memory efficiently and avoid issues such as deadlock. Let's break down the concepts you've mentioned:

1. **Memory Allocation System with Multiprocess Requesting:**
   - In a multiprocess system, multiple processes run simultaneously, each requiring memory to execute. The memory allocation system is responsible for managing and allocating memory blocks to these processes.

- Various allocation strategies exist, such as first-fit, best-fit, and worst-fit. These strategies determine how the system assigns memory blocks to processes based on their size and the available memory.

2. **Deadlock Detection Algorithm:**
   - Deadlocks can occur in a multiprocess system when each process holds a resource and is waiting for another resource acquired by another process. Deadlock detection algorithms help identify and resolve such situations.
   - One common algorithm is the banker's algorithm. It uses a matrix to represent the current state of resources and processes. It then simulates resource allocation to check for the possibility of deadlock. If a safe sequence is found, the system is in a safe state; otherwise, it's in an unsafe state, and deadlock may occur.

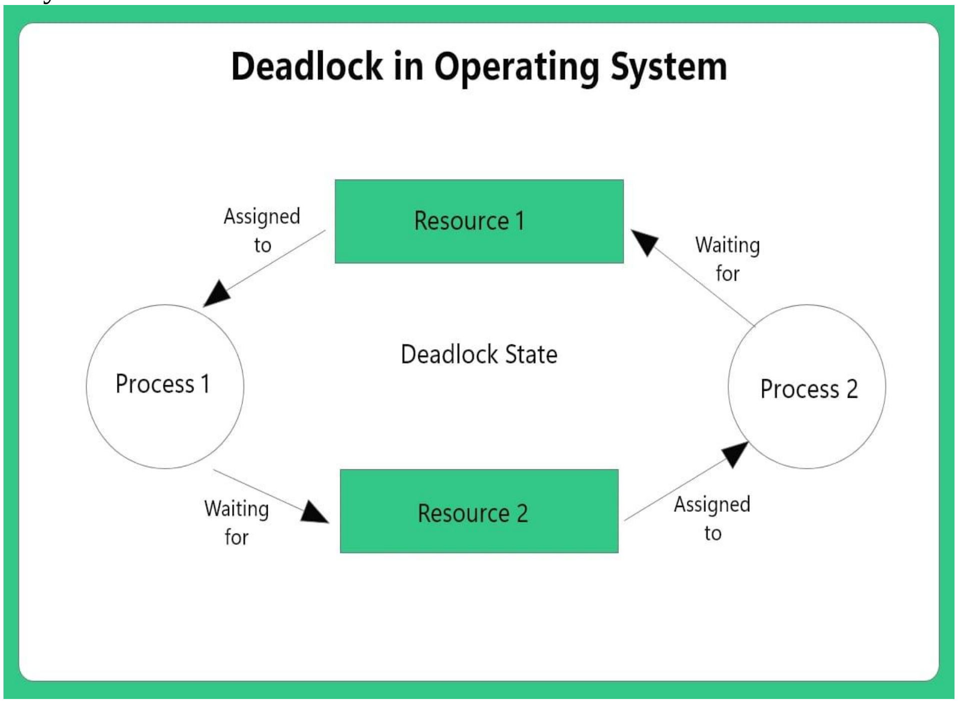


Fig No. 3

3. **Recovering from Deadlock by Releasing Memory Resource:**
   - One approach to recover from deadlock is to release resources held by processes. This can involve terminating one or more processes or preemptively releasing resources.
   - Steps to recover from deadlock by releasing memory resources:
   - a. **Identify Deadlock:** - Use deadlock detection algorithms to identify that a deadlock has occurred.
   - b. **Select a Process to Terminate:** - Choose one or more processes involved in the deadlock to terminate. The selection may be based on various criteria, such as the priority of the process or the amount of resources it holds.
   - c. **Release Resources:** - Release the resources held by the terminated process. This step involves releasing memory blocks and any other resources that the terminated process was holding.
   - d**. Notify Waiting Processes:** - Notify any waiting processes that the resources they were waiting for are now available. This allows them to proceed with their execution.

- e. **Avoid Future Deadlocks:** - Implement strategies to avoid future deadlocks, such as improving resource allocation algorithms or dynamically adjusting resource allocations.

It's important to note that while releasing resources can resolve a deadlock, it may lead to loss of work and potential inefficiencies. Therefore, prevention and avoidance strategies are also crucial in designing robust and efficient multiprocess systems.

1. Data Structures:

The code defines several arrays to manage the state of the resource allocation system:

- **alloc**: Represents the current allocation of resources to processes.
- **max_alloc**: Represents the maximum resources that each process can request.
- **available**: Represents the currently available resources.
- **done:** A boolean array to track whether a process has finished.
- **num_proc** and **num_res**: Store the number of processes and resources, respectively.

2. Initialization:

The **init** function initializes the global variables with the provided values, setting up the initial state of the resource allocation system.

3. Resource Request and Release:

- **request_resources** function checks if the requested resources can be granted to a process and allocates them if possible.
- **release_resources** function releases resources held by a process.

4. Deadlock Detection:

The **is_deadlocked** function implements a simple deadlock detection algorithm based on resource allocation. It iterates through processes and checks if any process can be satisfied with the available resources.

5. Printing Matrix:

The **print_matrix** function prints a matrix, which is used for displaying the allocation and other matrices for debugging or monitoring purposes.

6. Main Function:

The main function demonstrates the usage of the implemented functions with an example scenario. It initializes the system, simulates a scenario where a process requests resources (possibly leading to deadlock), and attempts to recover from potential deadlock situations.

Analysis:

- The code simulates a basic resource allocation system.
- Resource requests and releases are handled, and the system checks for potential deadlocks.
- The deadlock detection algorithm is simplistic and may not cover all possible deadlock scenarios.
- The example in the main function provides a basic demonstration of the implemented functionalities.

Advantages:

1. **Efficient Resource Utilization:**
   - Memory allocation systems help in efficiently utilizing available memory resources among multiple processes, optimizing overall system performance.
2. **Concurrency:**
   - **Multiprocess** systems allow multiple processes to run concurrently, enhancing system throughput and responsiveness.
3. **Isolation:**
   - Each process in a **multiprocess** system operates independently, providing a degree of isolation. This isolation helps in preventing one process from affecting others in case of errors or failures.
4. **Flexibility:**
   - **Multiprocess** systems provide flexibility by allowing different processes to execute concurrently, enabling the development of complex and modular software.
5. **Resource Sharing:**
   - Processes can share resources in a controlled manner, enabling efficient use of shared data structures or other shared resources.

Limitations:

1. **Deadlocks:**
   - Deadlocks can occur in a **multiprocess** system, where processes are unable to proceed due to circular dependencies on resources. Deadlocks can lead to inefficiency and system unresponsiveness.
2. **Complexity:**
   - Managing memory allocation and dealing with concurrent processes add complexity to system design. This complexity can lead to more challenging debugging and maintenance.
3. **Overhead:**
   - Implementing deadlock detection algorithms and recovery mechanisms introduces some overhead, potentially impacting system performance.
4. **Resource Fragmentation:**
   - Over time, as processes allocate and **deallocate** memory, fragmentation may occur, leading to inefficient use of memory. External fragmentation, in particular, can result in available memory being dispersed in small, non-contiguous blocks.
5. **Preemption Challenges:**
   - Preemptively terminating processes to recover from deadlocks can result in the loss of work and potential data corruption. Careful consideration is needed when choosing processes to terminate.
6. **Algorithm Sensitivity:**
   - The effectiveness of deadlock detection algorithms may depend on the specific characteristics of the system. Some algorithms may perform well in certain scenarios but poorly in others.
7. **Increased Latency:**
   - In the process of recovering from a deadlock by releasing resources, there may be an increase in system latency, especially if waiting processes need to restart or recover from the termination of another process.

In summary, while memory allocation in multiprocess systems offers several advantages in terms of resource utilization and concurrency, it also introduces challenges such as deadlocks and increased system complexity. Effective system design and careful consideration of algorithms are crucial to mitigating these limitations and ensuring a robust and efficient multiprocess environment.

Snapshot of Group Discussion :Assignment Insights