
# OS Assignment

By

Shivam Pandoh

2022a1r006

3rd SEM

A2

**MODEL INSTITUTE OF ENGINEERING AND TECHNOLOGY (AUTONOMOUS)**

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

2023

**Dr. Mekhla Sharma**

# ASSIGNMENT

| QUESTION NUMBERS | COURSE OUTCOMES | BLOOM'S LEVEL | MAXIMUM MARKS | MARKS OBTAIN |
|---|---|---|---|---|
| Q1 | CO 4 | 3-6 | 10 | |
| Q2 | CO 5 | 3-6 | 10 | |
| TOTAL MARKS | | | 20 | |

Faculty signature: Dr. Mekhla Sharma
Email: mekhla.cse@mietjammu.com

## 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 analyse both pre-emptive and non-preemptive versions of the priority scheduling algorithm.

## 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.

**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 analyse both pre-emptive and non-preemptive versions of the priority scheduling algorithm.

Solution: -

# Priority-based Process Scheduling:

Process scheduling in operating systems relies on a priority-based algorithm to determine the order of process execution, considering their assigned priority levels. In this approach:

**Scheduling Decision:**

Processes with higher priority levels are given preference in execution over those with lower priority levels. When the CPU is available or a process moves to a ready state, the scheduler selects the process with the highest priority in the ready queue for execution.

**Execution Sequence:**

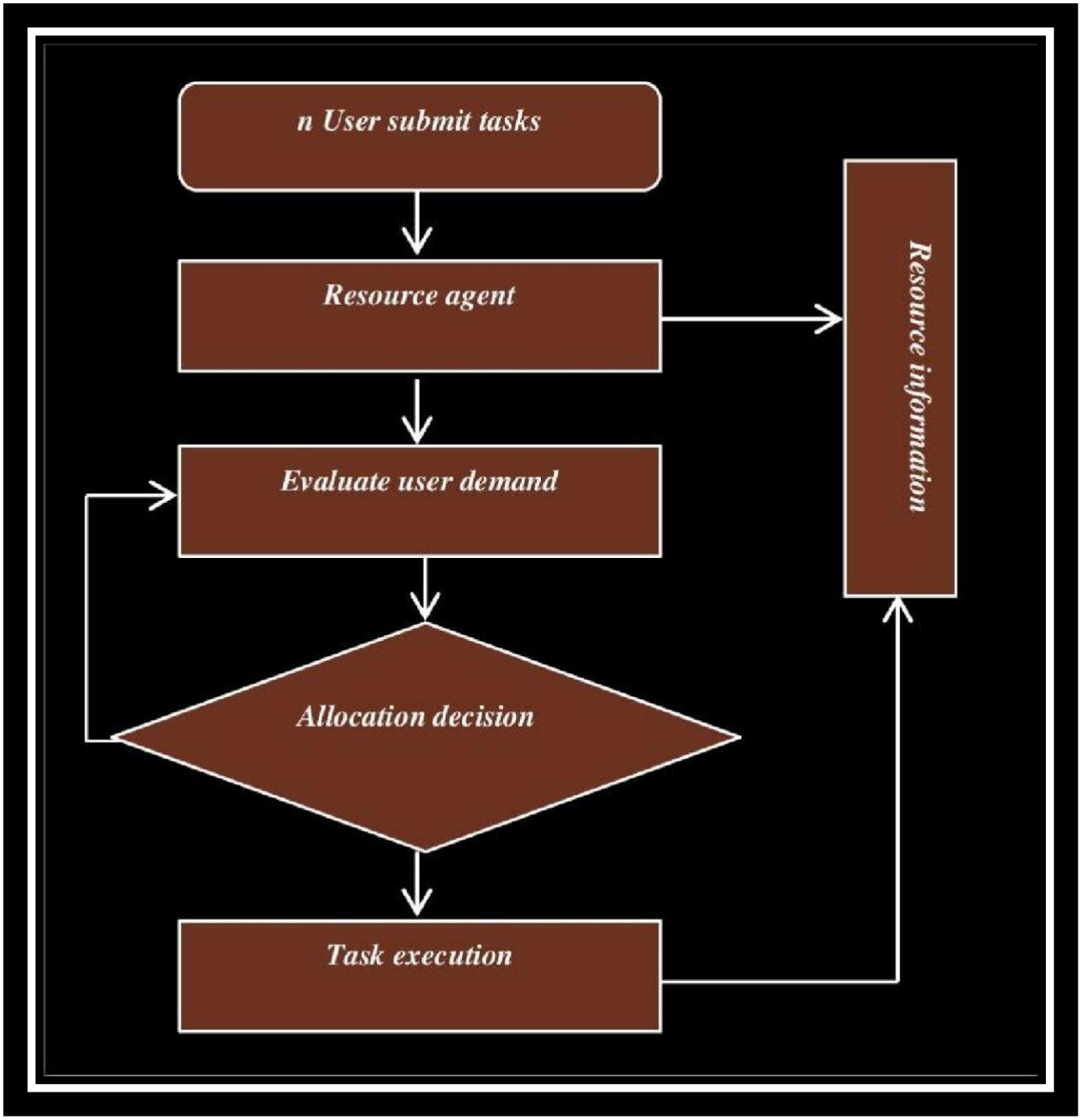
Tasks are executed based on their assigned priorities, ensuring that higher-priority processes take precedence.

**Completion and Waiting:**

Processes assigned higher priority tend to complete more quickly due to their privileged status, while tasks with lower priority may experience extended wait times, especially when higher-priority tasks arise frequently.

**Resource Allocation:**

Assigning resources is in accordance with task priorities, ensuring that essential tasks promptly acquire the required resources for their execution.


n User submit tasks
Resource agent
Evaluate user demand
Allocation decision
Task execution
Resource information

**MAIN ( ):**

```
int main() {
    struct Process processes[] = {
        {1, 2, 5},
        {2, 1, 3},
        {3, 3, 7},
        {4, 2, 2},
    };
    int num_processes = sizeof(processes) / sizeof(processes[0]);

    priorityNonPreemptive(processes, num_processes);
    priorityPreemptive(processes, num_processes);

    return 0;
}
```

**Priority preemptive:**

```
void priorityPreemptive(struct Process processes[], int n) {
    int current_time = 0;
    int total_burst_time = 0;
    float waiting_time = 0;

    printf("\nPreemptive Priority Scheduling:\n");

    for (int i = 0; i < n; i++) {
        total_burst_time += processes[i].burst_time;
    }

    while (current_time < total_burst_time) {
        int highest_priority_index = -1;
        for (int i = 0; i < n; i++) {
            if (processes[i].burst_time > 0 && processes[i].burst_time <
1000) {
                if (highest_priority_index == -1 || processes[i].priority >
processes[highest_priority_index].priority) {
```

```
                highest_priority_index = i;
            }
        }
    }

    if (highest_priority_index != -1) {
        printf("Process %d is executing from %d to %d\n",
processes[highest_priority_index].pid, current_time, current_time +
1);
        waiting_time += current_time;
        processes[highest_priority_index].burst_time -= 1;
        current_time += 1;
    } else {
        current_time += 1;
    }
  }

  float average_waiting_time = waiting_time / n;
  printf("Average Waiting Time: %.2f\n", average_waiting_time);
}
```

**Algorithm:**

- Initializing the variables "timeElapsed" and "completed" to keep track of the overall time elapsed and the count of completed processes, respectively.
- Employing a loop to choose the process with the highest priority from the pool of non-completed processes that still possess burst time.
- Execute the chosen process for a single unit of time, continually updating its burst time after each iteration.
- Complete the process once its burst time reaches zero.
- Display the execution details for each process on the output screen.
- Compute and print the average turnaround time.

## Priority non preemptive:

```
void priorityNonPreemptive(struct Process processes[], int n) {
    int current_time = 0;
    float waiting_time = 0;

    printf("Non-preemptive Priority Scheduling:\n");

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

        struct Process temp = processes[i];
        processes[i] = processes[highest_priority_index];
        processes[highest_priority_index] = temp;

        printf("Process %d is executing from %d to %d\n", processes[i].pid, current_time, current_time +
processes[i].burst_time);
        waiting_time += current_time;
        current_time += processes[i].burst_time;
    }

    float average_waiting_time = waiting_time / n;
    printf("Average Waiting Time: %.2f\n", average_waiting_time);
}
```

## Algorithm:

Organizes processes in any sequence based on their priority levels.

- Sorts the provided processes in descending order according to their priority.

Sequentially executes each process based on its priority.

- Processes each task according to its priority without disrupting ongoing processes.

Provides detailed information about each process execution.

- Generates a report for each process execution, including process ID, priority, and the time required for completion.

Computes and displays the average turnaround time.

- Determines the average turnaround time for all processes executed in this non-preemptive scheduling system.

The average turnaround time reflects the mean duration for a process to complete its execution from initiation to completion.

## OUTPUT ➔

```
Non-preemptive Priority Scheduling:
Process 3 is executing from 0 to 7
Process 1 is executing from 7 to 12
Process 4 is executing from 12 to 14
Process 2 is executing from 14 to 17
Average Waiting Time: 8.25

Preemptive Priority Scheduling:
Process 3 is executing from 0 to 1
Process 3 is executing from 1 to 2
Process 3 is executing from 2 to 3
Process 3 is executing from 3 to 4
Process 3 is executing from 4 to 5
Process 3 is executing from 5 to 6
Process 3 is executing from 6 to 7
Process 1 is executing from 7 to 8
Process 1 is executing from 8 to 9
Process 1 is executing from 9 to 10
Process 1 is executing from 10 to 11
Process 1 is executing from 11 to 12
Process 4 is executing from 12 to 13
Process 4 is executing from 13 to 14
Process 2 is executing from 14 to 15
Process 2 is executing from 15 to 16
Process 2 is executing from 16 to 17
Average Waiting Time: 34.00

Process returned 0 (0x0)   execution time : 0.072 s
Press any key to continue.
```

## Non-preemptive Priority Scheduling:

• Priority-Based Sorting:

Processes are sorted in descending order of priority before the execution commences.

• Sequential Execution:

Each process is executed in its entirety without any interruptions, following the priority sequence.

• Execution Order:

Gives precedence to tasks with higher priority, disregarding the arrival time of lower priority tasks with shorter burst times.

• Average Turnaround Time:

Prefers higher priority tasks, leading to the possibility of extended waiting times for lower priority tasks.

• Application Scenarios:

Suitable when it is acceptable to postpone the execution of lower priority tasks until higher priority tasks are completed.

## Preemptive Priority Scheduling:

• Continuous Priority Evaluation:

Regularly assesses the most prioritized process among those not yet finished for each time unit.

• Priority-based Interruption:

Permits newly arriving processes with higher priority to preempt and interrupt ongoing processes with lower priority.

• Execution Sequence:

Tasks with higher priority may disrupt those with lower priority, even if the latter have already started execution.

• Average Turnaround Duration:

Tasks with lower priority may experience extended waiting times or frequent interruptions

due to the arrival of tasks with higher priority.

• Practical Applications:

Suitable for environments where critical high-priority tasks emerge frequently, requiring immediate attention and potentially impacting ongoing tasks with lower priority.

# Advantages and Disadvantages of priority scheduling:

## Advantages:

**Prioritization for Urgency:**

Urgent tasks are promptly addressed, guaranteeing swift completion of crucial assignments.

**Faster System Response:**

Efficiently handling crucial tasks enhances system responsiveness, thereby optimizing overall user experience.

**Efficient Resource Use:**

Effectively allocates resources by prioritizing high-importance tasks, ensuring optimal utilization of resources.

**Maximized System Output:**
Ensuring vital tasks are promptly completed enhances the overall efficiency of the system.

## Disadvantages:

**Risk of Task Delay:**

Low-priority tasks may experience prolonged waiting or remain unfinished when constantly faced with incoming high-priority tasks, leading to delays.

**Priority Mix-up:**
When less important tasks impede the availability of resources required for more crucial tasks, it may result in delays to critical activities.

**Challenges in Prioritizing:** Establishing precise priorities can be challenging, resulting in errors

and inefficiencies during task execution.

**Fairness Concerns:** Consistently prioritizing tasks of higher importance can lead to fairness concerns among tasks of varying priority levels.

**Overhead from Changes:** Frequent changes in priorities may lead to additional tasks, affecting system performance as it requires constant adjustments.

## 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.

Solution: -

Task 2: Memory Allocation System with Deadlock Detection

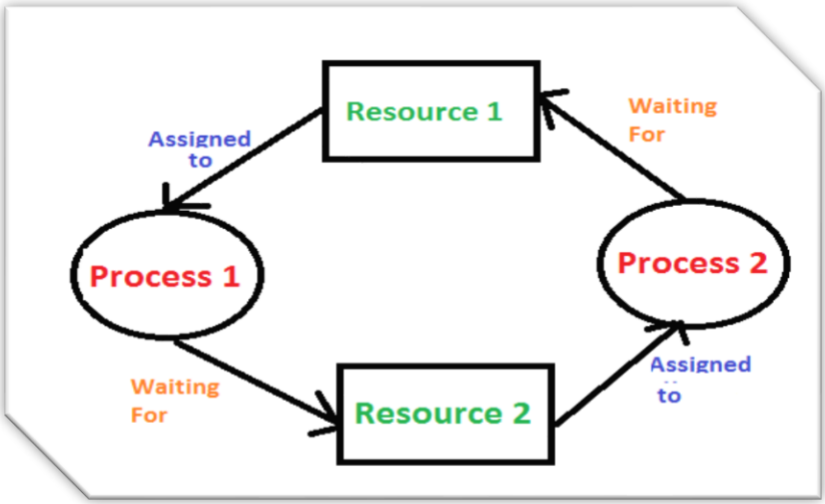
**Program Overview:**

1. The software simulates a resilient memory allocation system.

2. It manages requests for memory blocks from multiple processes.

3. The implementation incorporates an algorithm for detecting deadlocks.

4. The software showcases a recovery mechanism triggered by the release of memory resources.



## Advantages:

- **Efficient Resource Utilization**:
  The program ensures efficient utilization of available memory by allocating it to processes based on their requests.

- **Dynamic Memory Allocation**:
  The memory manager dynamically allocates and releases memory blocks, adapting to the varying needs of processes.

## Disadvantages:

- **Blocking Processes**
  In case of insufficient memory, processes may get blocked while waiting for memory, potentially leading to decreased system performance.

CODE:

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

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

int allocation[MAX_PROCESSES][MAX_RESOURCES];
int max_allocation[MAX_PROCESSES][MAX_RESOURCES];
int available[MAX_RESOURCES];
bool finished[MAX_PROCESSES];
int num_processes, num_resources;
```

INITIALIZE:

```
void initialize(int processes, int resources, int alloc[][MAX_RESOURCES], int max[][MAX_RESOURCES],
        int avail[]) {
 num_processes = processes;
 num_resources = resources;

for (int i = 0; i < num_processes; ++i) {
        for (int j = 0; j < num_resources; ++j) {
                allocation[i][j] = alloc[i][j];
                max_allocation[i][j] = max[i][j];
         }

        finished[i] = false;
 }

for (int i = 0; i < num_resources; ++i) {
        available[i] = avail[i];
```

```
}
 }
```

MEMORY ALLOCATION:

```
bool request_resources(int pid, int request[]) {
        for (int i = 0; i < num_resources; ++i) {
                if (request[i] > available[i] || request[i] > max_allocation[pid][i] - allocation[pid][i]) {
                        return false;  // Request cannot be granted immediately
                        }
                }

for (int i = 0; i < num_resources; ++i) {
        available[i] -= request[i];
        allocation[pid][i] += request[i]; }
        return true;
        // Request granted
}
```

MEMORY RELEASE:

```
void release_resources(int pid, int release[]) {
        for (int i = 0; i < num_resources; ++i) {
                // Ensure release doesn't exceed the currently allocated resources for a process
                if (release[i] > allocation[pid][i]) {
                        release[i] = allocation[pid][i]; // Limit release to currently allocated resources
                }

        available[i] += release[i];
        allocation[pid][i] -= release[i];
        }
}
```

DEADLOCK(DETECTION AND RECOVERY)

```
bool is_deadlocked() {
        int work[num_resources];
        bool finish[num_processes];

        for (int i = 0; i < num_resources; ++i) {
                work[i] = available[i];
                }
        for (int i = 0; i < num_processes; ++i) {
                finish[i] = finished[i];
                }

        bool deadlock = true;
        while (deadlock) {
                deadlock = false;
                for (int i = 0; i < num_processes; ++i) {
                        if (!finish[i]) {
```

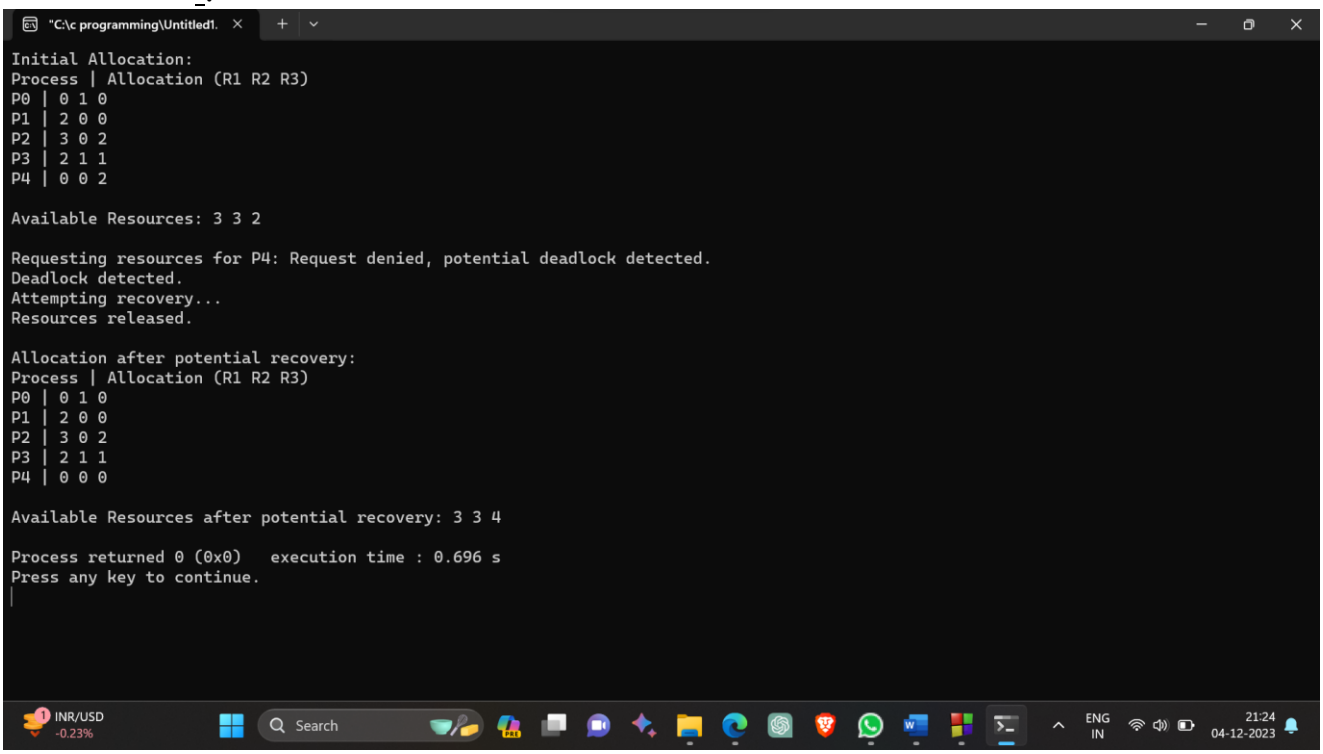
```
                        bool can_allocate = true;
                        for (int j = 0; j < num_resources; ++j) {
                                if (max_allocation[i][j] - allocation[i][j] > work[j]) {
                                        can_allocate = false;
                                        break;
                                        }
                                }
                        if (can_allocate) {
                                deadlock = false;
                                finish[i] = true;
                                for (int j = 0; j < num_resources; ++j) {
                                        work[j] += allocation[i][j];
                                        }
                                }
                        }
                }
        }
for (int i = 0; i < num_processes; ++i) {
        if (!finish[i]) {
                return true; // Deadlock detected
                }
        }

        return false; // No deadlock
}
```

MAIN CODE:

```
int main() {
// Example initialization of resources and processes
        int processes = 5;
        int resources = 3;
        int alloc[][MAX_RESOURCES] = {{0, 1, 0}, {2, 0, 0}, {3, 0, 2}, {2, 1, 1}, {0, 0, 2}};
        int max[][MAX_RESOURCES] = {{7, 5, 3}, {3, 2, 2}, {9, 0, 2}, {2, 2, 2}, {4, 3, 3}};
        int avail[] = {3, 3, 2};

        initialize(processes, resources, alloc, max, avail);

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

        printf("\nAvailable Resources: %d %d %d\n\n", available[0], available[1], available[2]);
        // Example scenario where a request leads to a potential deadlock
        int pid = 4;
        int request[] = {1, 0, 2}; // A request that leads to a deadlock
                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); // Attempt recovery by releasing resources
                                printf("Resources released.\n");
```

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

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

        return 0;
}
```

**OUTPUT →**



Usage:

→The program demonstrates how memory allocation and deadlock detection might work in a simplified system.

→It showcases allocating memory to processes, detecting a deadlock based on a simple condition, and recovering from the deadlock by releasing allocated memory resources.

## Advantages:

- **Efficient Resource Utilization**:
  The program ensures efficient utilization of available memory by allocating it to processes based on their requests.

- **Dynamic Memory Allocation**:
  The memory manager dynamically allocates and releases memory blocks, adapting to the varying needs of processes.

## Disadvantages:

- **Blocking Processes**
  In case of insufficient memory, processes may get blocked while waiting for memory, potentially leading to decreased system performance.

## Conclusion:

This comprehensive program features a sophisticated memory allocation system that incorporates built-in deadlock detection and recovery mechanisms. The interaction among various classes adeptly simulates processes, memory blocks, and oversees the allocation procedure. Notable benefits include efficient resource utilization and dynamic memory allocation, while it's essential to acknowledge a potential drawback that may occur in the form of process blocking during memory shortages.

# GROUP PHOTO