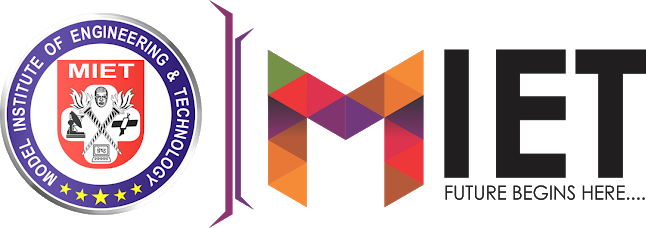
ASSIGNMENT

**By**

***Sukhshum Vaishnavi* 2022A1R002**

# 3rd Semester

# CSE A2 Section



**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 (Operating System)

## Due Date: 04/12/2023

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Question Number** | **Course Outcomes** | **Blooms’ Level** | **Maximum Marks** | **Marks Obtain** |
| Q1 | **CO1, CO2 & CO5** | 3-4 | 10 |  |
| Q2 | **CO3, CO4** | 3-4 | 10 |  |
| **Total Marks** | | | 20 |  |
| Submitted to:  Faculty Signature:  Email: | | | | |

# 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.

**GROUP PHOTO:**

****

# 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 is a method used by operating systems to determine the order in which processes are executed on a CPU. In this system, each process is assigned a priority, which could be based on factors like the process's importance, time sensitivity, or resource requirements.

The CPU scheduler selects the process with the highest priority to execute first. If two processes have the same priority, other factors such as waiting time or arrival time might be considered. The goal is to optimize system performance by giving preference to more critical or time-sensitive tasks.

There are different types of priority-based scheduling algorithms, such as:

1. Preemptive Priority Scheduling: The highest priority process currently in the ready queue is given the CPU, and it can be preempted by a higher priority process arriving or becoming ready.

2. Non-Preemptive Priority Scheduling: The CPU is allocated to a process until it completes or enters a waiting state. In this case, the priority of the running process cannot be changed until it finishes.

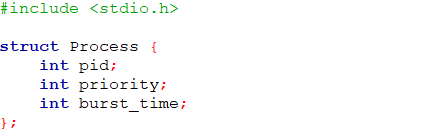
priority scheduling is done through:

1. Scheduling Management where the scheduler opts for the highest priority process in the ready queue for execution.
2. Execution Sequence where Operations are executed according to their designated priorities.
3. Resource Allotment in which the necessary resources are secured for the execution of the vital operations.
4. Completion and Waiting of Operations in which the processes are kept in waiting queue for

the execution of more important tasks.

1. **Process Class**

CODE



* **#include <stdio.h>**: Includes the standard input-output library for input/output operations.
* **struct Process {...};** : Defines a structure **Process** containing attributes such as

1. **pid** (process ID) which provides a distinct identification number to differentiate one process from another.
2. **priority** whichDetermines the order in which processes are scheduled and executed by the operating system.
3. **burst\_time** (time required for execution) which represents the time required for the process to complete its execution.Burst time represents the amount of CPU time required by a process to complete its execution.

**Priority Non-Preemptive**

In Non-Preemptive Priority Scheduling

1. The function iterates through the processes and sorts them based on their priorities in descending order.
2. It executes each process one by one, starting from the highest priority to the lowest.
3. For each process, it prints the process ID, the time it starts executing, and the time it finishes executing.
4. It calculates the average waiting time for all the processes and prints it at the end.

For example, given processes:

| PID | Priority | Burst Time |

| --- | -------- | ---------- |

| 1 | 2 | 5 |

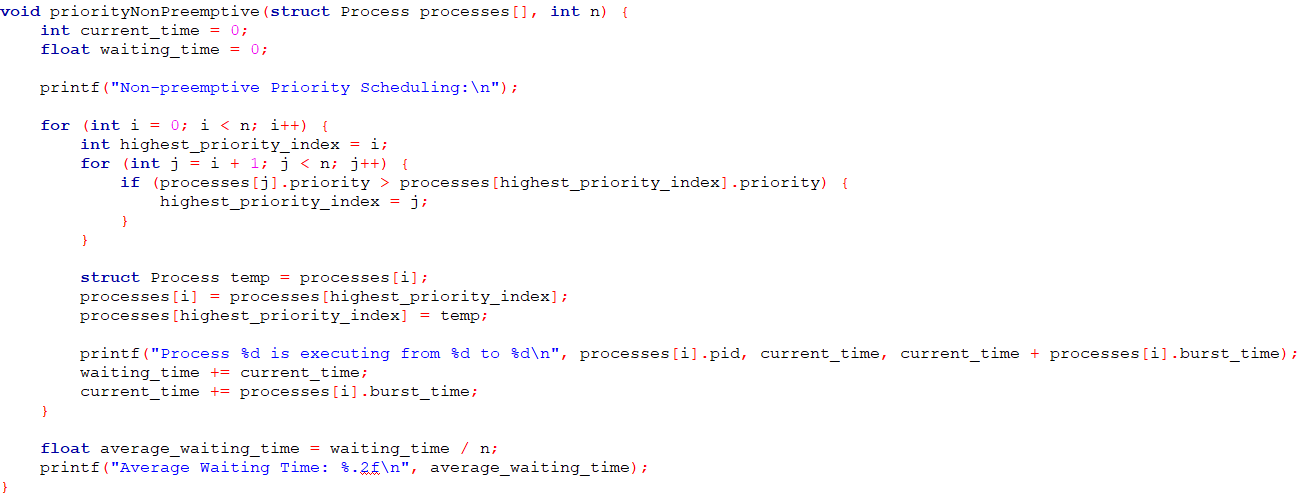
| 2 | 1 | 3 |

| 3 | 3 | 7 |

| 4 | 2 | 2 |

The non-preemptive priority scheduling will execute processes in the order: 3, 1, 4, 2.

**Function Code**

****

#### 

**Inside the function:**

* Initializes current\_time and waiting\_time variables to 0 to track the execution time and waiting time.
* The function implements non-preemptive priority scheduling by sorting processes based on priority and executing them one by one in decreasing order of priority.
* For each process, it prints the execution details (process ID, start, and end times), calculates waiting time, and updates the current time.
* Calculates the average waiting time and prints it at the end.

**Priority Preemptive**

In Preemptive Priority Scheduling

1. The function initializes time variables and calculates the total burst time required by all processes.
2. It repeatedly selects and executes the process with the highest priority among the processes that still have burst time remaining.
3. It prints the process ID, the time it starts executing, and the time it finishes executing for each time unit.
4. It calculates the average waiting time for all the processes and prints it at the end.

Using the same example processes:

| PID | Priority | Burst Time |

| --- | -------- | ---------- |

| 1 | 2 | 5 |

| 2 | 1 | 3 |

| 3 | 3 | 7 |

| 4 | 2 | 2 |

The preemptive priority scheduling will execute processes in a way that allows higher-priority processes to interrupt lower-priority processes if they arrive. The execution will depend on the current time unit, and it might look something like this:

- At time 0: Process 3 starts executing.

- At time 7: Process 1 interrupts and starts executing.

- At time 12: Process 4 interrupts and starts executing.

- At time 14: Process 2 starts executing.

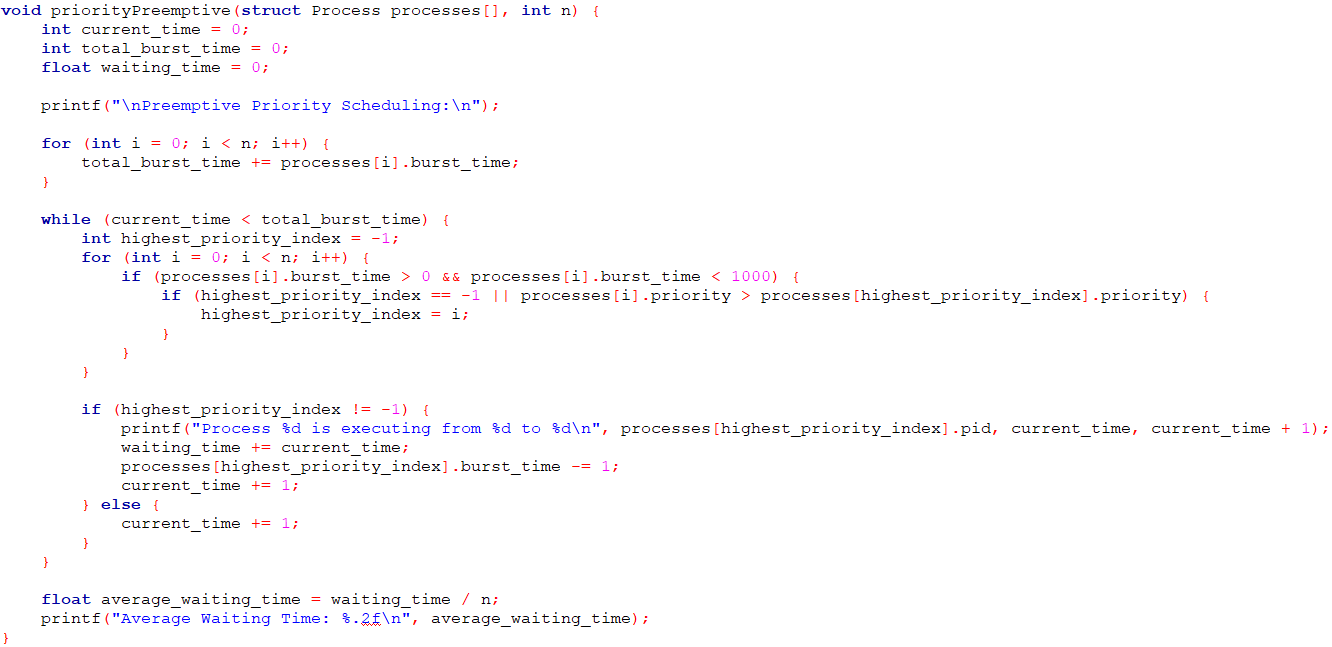
- Process 2 finishes at time 17.

- Process 1 finishes at time 22.

- Process 4 finishes at time 24.

- Process 3 finishes at time 31.

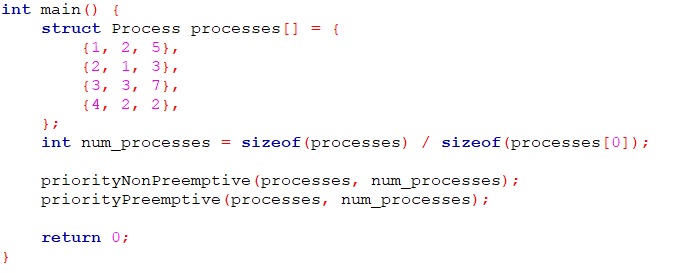
**Function code**

****

**Inside the function:**

* Initializes current\_time, total\_burst\_time, and waiting\_time.
* Calculates the total burst time required by all processes.
* Implements preemptive priority scheduling by iterating through time units and executing the highest priority process at each time unit. It decreases the burst time of the selected process by 1 for each execution.
* Calculates the average waiting time and prints it at the end.

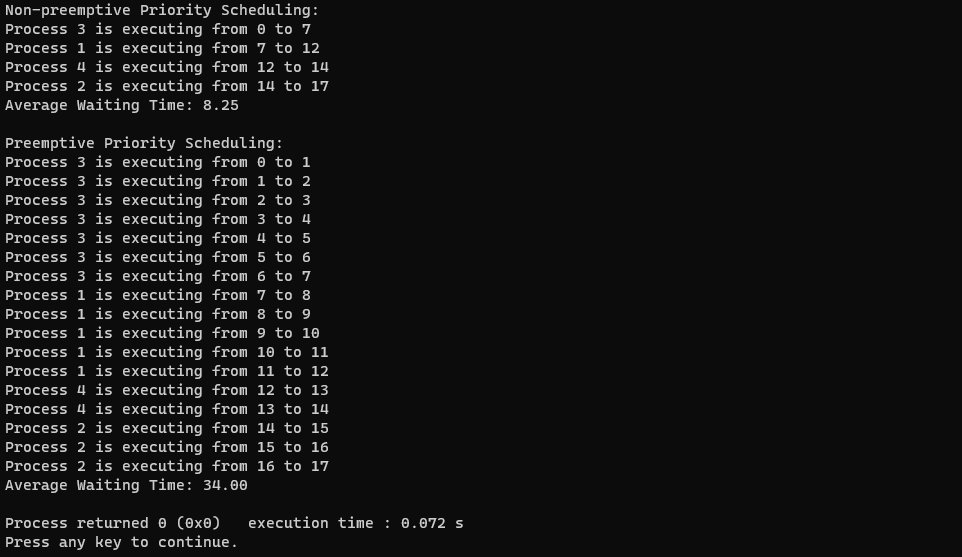
**Main Function with Use Case Example**

****

**Inside the main function:**

* Initializes an array of Process structures representing different processes.
* Computes the number of processes in the array.
* Calls both scheduling functions (priorityNonPreemptive and priorityPreemptive) with the array of processes and the number of processes as arguments. Returns 0 to indicate successful program execution.

**Output**

****

**Non-preemptive Priority Scheduling:**

**Advantages:**

1. Simplicity: Non-preemptive priority scheduling is relatively simple to implement as it executes processes based on their priority without interruption.
2. Avoidance of Starvation: Higher priority processes are executed first, ensuring lower priority processes don't indefinitely wait (avoids starvation).

**Disadvantages:**

1. Possibility of Higher Waiting Time: Processes with lower priority might have to wait for a long time if higher priority processes continuously arrive.
2. Inefficiency in Resource Utilization: If a higher priority process arrives after lower priority processes have already started, it needs to wait until lower priority processes finish, leading to inefficiency.

**Preemptive Priority Scheduling:**

**Advantages:**

1. Higher Responsiveness: Higher priority processes can interrupt the execution of lower priority ones, allowing critical tasks to be handled promptly.
2. Lower Waiting Time for High-Priority Tasks: Ensures that high-priority processes do not wait indefinitely behind lower-priority ones, leading to lower waiting times.

**Disadvantages:**

1. Possibility of Starvation: Lower priority processes might face starvation if higher priority processes continuously arrive.
2. Complexity: Preemptive priority scheduling involves more complex handling due to the need for managing interruptions and context switching.

**Comparison:**

* Waiting Time: Non-preemptive scheduling may result in higher waiting times for lower priority processes compared to preemptive scheduling.
* Responsiveness: Preemptive scheduling provides higher responsiveness to high-priority tasks by allowing them to execute even if lower-priority tasks are running.
* Starvation: Both preemptive and non-preemptive scheduling algorithms have the potential for causing starvation, where lower priority processes might not get a chance to execute if higher priority processes continuously arrive.

**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.

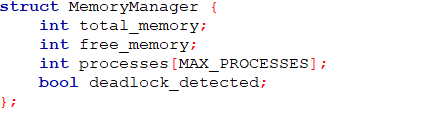
**Solution**:

**Overview:**

1. Memory Manager Structure (struct MemoryManager):

* Tracks the total memory available and the amount of free memory.
* Maintains an array representing memory allocated to different processes.

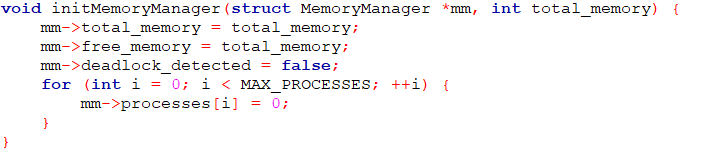
Code:



1. Initialization (initMemoryManager):

* Initializes the memory manager with the total available memory and sets all process memory allocations to zero.

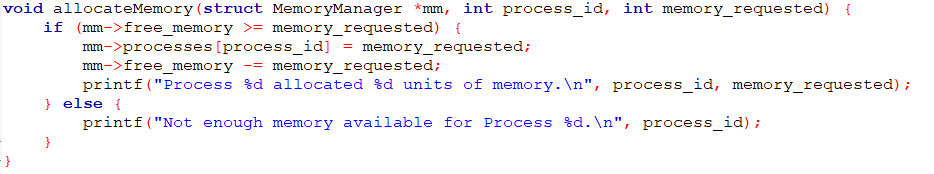
Function Code:



1. Memory Allocation (allocateMemory):

* Allocates memory for a process if enough free memory is available.
* Records the amount of memory allocated to a process and updates the free memory accordingly.

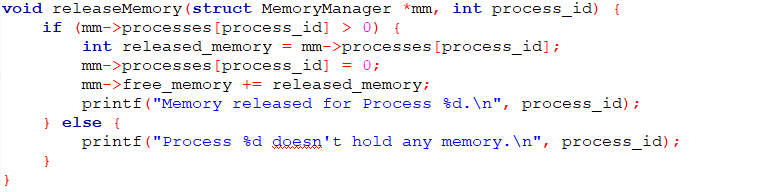
Function Code:



1. Memory Release (releaseMemory):

* Releases memory previously allocated to a process, adding the released memory back to the free memory pool.

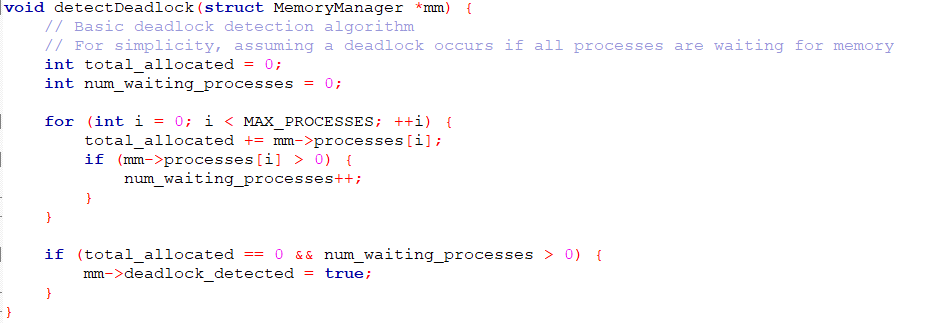
Function Code:



1. Deadlock Detection (detectDeadlock):

* Uses a simple deadlock detection algorithm:
* Checks if any processes are waiting for memory allocation.
* Determines if there is no free memory available.
* If these conditions hold, marks a deadlock as detected.

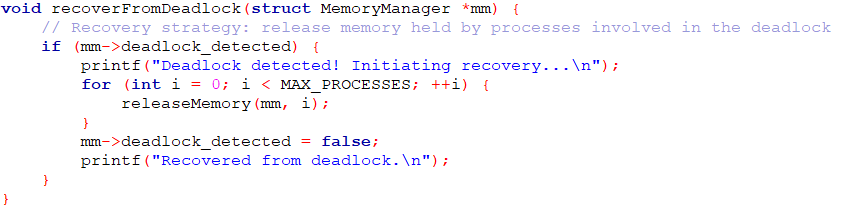
Function Code:



1. Deadlock Recovery (recoverFromDeadlock):

* If a deadlock is detected, releases memory held by all processes involved in the deadlock, effectively breaking the deadlock condition.
* Resets the deadlock flag.

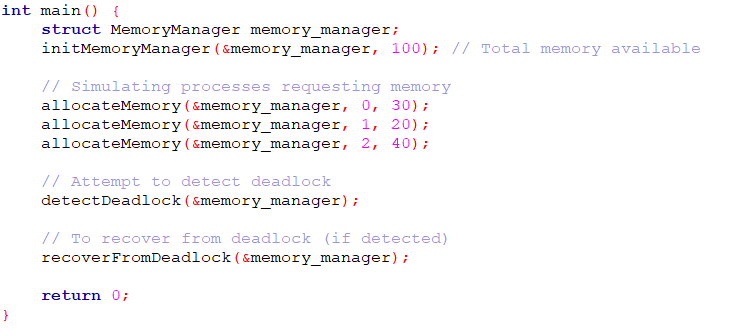
Function Code:



1. Main Function (main):

* Initializes the memory manager with total available memory.
* Simulates processes requesting memory allocation.
* Attempts to detect deadlock.
* If a deadlock is detected, initiates recovery by releasing memory held by processes involved in the deadlock.

Function Code:



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.

**OUTPUT**

