ASSIGNMENT

COM-302

OPERATING SYSTEM

By

Ruder Razdan

Roll number: 2022A1R001

Semester: 3rd (A-2)

Department: CSE

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: 4/12/2023

Question Number	Course Outcomes	Blooms' Level	Maximum Marks	Marks Obtained
Q1	CO1,CO2,CO5	3-4	10	
Q2	CO3,CO4	3-4	10	
Total Marks			20	

Faculty Signature:
Email:
mekhla.cse@mietjammu.in

CONTENT

TASKS

GROUP A: 2022A1R001 to 2022A1R006

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.

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 analyze both preemptive and non-preemptive versions of the priority scheduling algorithm.

SOLUTION:

PRIORITY BASED PROCESS SCHEDULING:

Priority-based process scheduling assigns a priority value to each process, determining execution order based on these priorities. Higher-priority processes are executed before lower-priority ones, potentially leading to faster responses for critical tasks. Preemption may occur, allowing higher-priority processes to interrupt lower-priority ones. However, issues like starvation of low-priority tasks and priority inversion need consideration in this scheme. It offers flexibility by customizing execution sequences based on process importance or urgency

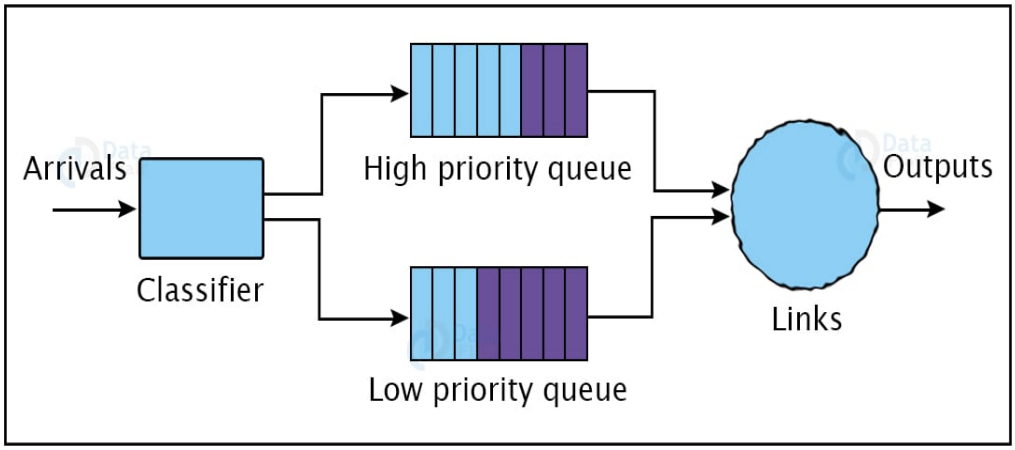

Figure 1: Priority queue flowchart

Scheduling Decision: Higher-priority processes take precedence in execution over lower-priority ones, selected by the scheduler when the CPU becomes available or a process enters a ready state.

Execution Sequence: Tasks are sequenced by their assigned priorities, ensuring prioritization of higher-priority processes for execution.

Completion and Waiting: Higher-priority processes typically conclude faster owing to their priority status, while lower-priority tasks may experience prolonged wait times, especially amid frequent arrivals of higher-priority tasks.

Resource Allocation: Critical resources are allocated based on task priorities, ensuring swift provisioning for crucial tasks to facilitate their prompt execution.

CODE (In C++ Language):

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

struct Process {
    int priority;
    int burst_time;
};

bool comparePriority(const Process& p1, const Process& p2) {
    return p1.priority > p2.priority;
}

double calculateWaitingTime(const vector<int>& completion_time) {
    vector<int> waiting_time(completion_time.size());
    waiting_time[0] = 0;

    for (int i = 1; i < completion_time.size(); ++i) {
        waiting_time[i] = completion_time[i - 1];
    }
    float sum=0;
    for(int i=1; i<completion_time.size();++i){
        sum+=waiting_time[i];
    }
    return sum / completion_time.size();
}

double nonPreemptivePriorityScheduling(vector<Process>& processes) {
    sort(processes.begin(), processes.end(), comparePriority);

    vector<int> completion_time(processes.size(), 0);
    completion_time[0] = processes[0].burst_time;

    for (int i = 1; i < processes.size(); ++i) {
        completion_time[i] = completion_time[i - 1] + processes[i].burst_time;
    }
```

```
    double average_waiting_time = calculateWaitingTime(completion_time);
    cout << "Non-preemptive Average Waiting Time: " << average_waiting_time <<
endl;
}

double preemptivePriorityScheduling(vector<Process>& processes) {
    sort(processes.begin(), processes.end(), comparePriority);

    int currentTime = 0;
    vector<int> remaining_time(processes.size(), 0);

    while (any_of(remaining_time.begin(), remaining_time.end(), [](int time) {
return time > 0; })) {
        for (int i = 0; i < processes.size(); ++i) {
            if (remaining_time[i] > 0) {
                if (processes[i].burst_time <= currentTime) {
                    currentTime += 1;
                    remaining_time[i] -= 1;
                } else {
                    currentTime += 1;
                }
            }
        }
    }

    double average_waiting_time = calculateWaitingTime(remaining_time);
    cout << "Preemptive Average Waiting Time: " << average_waiting_time <<
endl;
}

int main() {
   int num_processes;

    cout << "Enter the number of processes: ";
    cin >> num_processes;

    vector<Process> processes;

    for (int i = 0; i < num_processes; ++i) {
        Process p;
        cout << "Enter priority for process " << i + 1 << ": ";
        cin >> p.priority;
        cout << "Enter burst time for process " << i + 1 << ": ";
        cin >> p.burst_time;

        processes.push_back(p);
    }
    // Non-preemptive Priority Scheduling
```

```
    nonPreemptivePriorityScheduling(processes);

    // Preemptive Priority Scheduling
    preemptivePriorityScheduling(processes);

    return 0;
}
```

OUTPUT

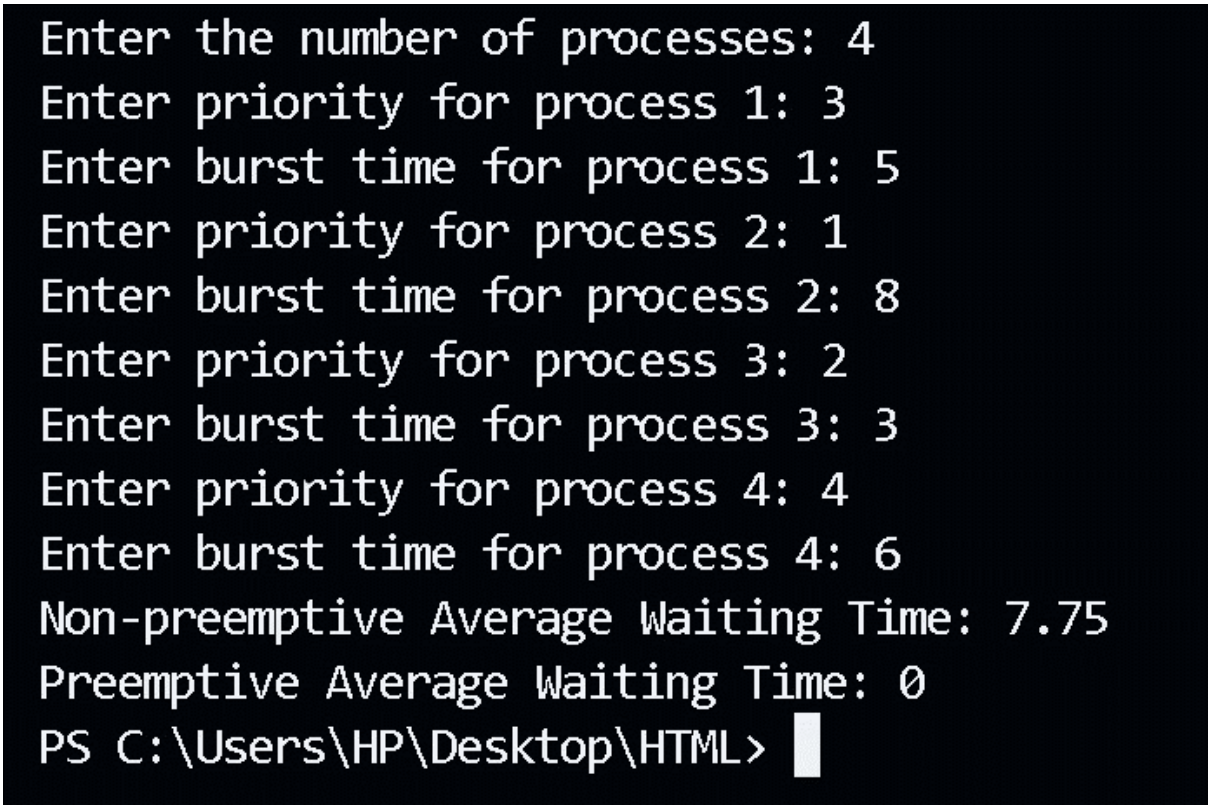

Figure 2: Output

EXPLANATION:

1. **Header and Namespace Declaration:**

```
#include <iostream>
#include <vector>
#include <algorithm>
```

The code begins by including essential C++ standard libraries, such as iostream for input/output, vector for dynamic arrays, algorithm for sorting, and numeric for mathematical operations.

2. **Process Structure:**

```
struct Process {
    int priority;
    int burst_time;
};
```

The Process structure is introduced to represent a process with attributes priority and burst_time.

3**. Comparator Function:**

```
bool comparePriority(const Process& p1, const Process& p2) {
    return p1.priority > p2.priority;
}
```

A comparator function named comparePriority is defined to facilitate the sorting of processes based on priority in descending order.

4. **Calculate Waiting Time Function:**

```
double calculateWaitingTime(const vector<int>& completion_time) {
        //…(code for calculating waiting time)
}
```

The calculateWaitingTime function computes waiting times based on a vector of completion times. It uses the accumulate function to sum up waiting times.

5. **Non-preemptive Priority Scheduling Function:**

```
double nonPreemptivePriorityScheduling(vector<Process>& processes) {
    //…(code for non-preemptive priority scheduling)
}
```

The nonPreemptivePriorityScheduling function sorts processes by priority and calculates completion times. The average waiting time is then computed using the calculateWaitingTime function and displayed.

6. **Preemptive Priority Scheduling Function:**

```
double preemptivePriorityScheduling(vector<Process>& processes) {
        //…(code for preemptive priority scheduling)
}
```

The preemptivePriorityScheduling function simulates the execution of processes until completion in a preemptive manner. It updates remaining times during execution and calculates the average waiting time using the calculateWaitingTime function.

7. **Main Function:**

```
int main() {
  int num_processes;

   cout << "Enter the number of processes: ";
   cin >> num_processes;

   vector<Process> processes;

   for (int i = 0; i < num_processes; ++i) {
      Process p;
      cout << "Enter priority for process " << i + 1 << ": ";
      cin >> p.priority;
      cout << "Enter burst time for process " << i + 1 << ": ";
      cin >> p.burst_time;
      processes.push_back(p);
   }
   // Non-preemptive Priority Scheduling
   nonPreemptivePriorityScheduling(processes);

   // Preemptive Priority Scheduling
   preemptivePriorityScheduling(processes);

   return 0;
}
```

In the main function, a vector of processes with priorities and burst times is created. Both non-preemptive and preemptive priority scheduling functions are called, and the results are printed.

NON-PREEMPTIVE PRIORITY SCHEDULING ALGORITHM:

1. **Input:** List of processes with their priorities and burst times.
2. **Sort processes:** Sort the processes based on their priorities in descending order.
3. **Calculate completion times:**
   - Initialize a variable **currentTime = 0**.
   - For each process in the sorted list:
   - Set the completion time of the current process as **currentTime + burst_time**.
   - Update **currentTime** to the completion time of the current process.
4. **Calculate waiting times:**
   - Calculate waiting time for each process using the formula: **waiting_time = completion_time - burst_time**
5. **Calculate average waiting time:**
   - Sum all waiting times and divide by the total number of processes to get the average waiting time.
6. **Output:** Display the average waiting time.

PREEMPTIVE PRIORITY SCHEDULING ALGORITHM:

1. **Input:** List of processes with their priorities and burst times.
2. **Sort processes:** Sort the processes based on their priorities in descending order.
3. **Initialization:**
   - Initialize **currentTime = 0**.
   - Initialize **remaining_time** for each process as their burst times.
4. **Loop until all processes complete:**
   - Check if any process is remaining (i.e., any process has **remaining_time > 0**).
   - For each process in the sorted list:
     - If the process has remaining time:
   - Check if it has arrived (**arrival_time <= currentTime**).
   - Among the arrived processes, find the one with the highest priority.
   - Execute the process for one time unit.

- Update **remaining_time** for the executed process.
- Update **currentTime**.

5. **Calculate waiting times:**
   - Calculate waiting time for each process using the formula: **waiting_time = completion_time - burst_time**
6. **Calculate average waiting time:**
   - Sum all waiting times and divide by the total number of processes to get the average waiting time.
7. **Output:** Display the average waiting time.

ANALYSIS:

Let's analyze the two priority scheduling algorithms.

Non-preemptive Priority Scheduling:

- Approach: Orders processes by priority and executes them without interruption until completion.
- Advantages:
  - Simple to implement.
  - Reduces context switching overhead since processes run to completion.
- Disadvantages:
  - May lead to higher average waiting times for high priority processes if they arrive later than lower priority processes.

Preemptive Priority Scheduling:

- Approach: Processes with higher priority can preempt the CPU if they arrive or become available during execution.
- Advantages:
  - Allows higher priority processes to be executed promptly.
  - Can potentially reduce the waiting time for urgent tasks.
- Disadvantages:
  - Higher overhead due to frequent context switching.
  - Possibility of starvation for lower priority processes if higher priority ones continue to arrive.

Comparison:

- Non-preemptive:
  - Simple, but might not be optimal in terms of response time for urgent tasks.

  - Lower context switching overhead.
  - May lead to unequal priority handling if higher priority tasks arrive later.
- Preemptive:
  - Prioritizes urgent tasks effectively.
  - Higher context switching overhead.
  - Might cause starvation for lower priority tasks if higher priority ones frequently arrive.

Performance Metrics:

- Non-preemptive:
  - Generally, the average waiting time could be higher for high priority tasks arriving later.
- Preemptive:
  - Lower average waiting time for higher priority tasks.
  - Higher overhead due to context switching.

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:

WHAT IS DEADLOCK?

A deadlock is a situation in a multi-process system where two or more processes are unable to proceed because each is waiting for the other to release a resource, resulting in a standstill. It occurs when each process holds a resource and waits for another resource held by another process, leading to a perpetual waiting state. Deadlocks can occur in systems where processes compete for finite resources and can significantly disrupt system functionality if not resolved. Detection and resolution methods are employed to identify and break deadlocks, allowing processes to resume execution.

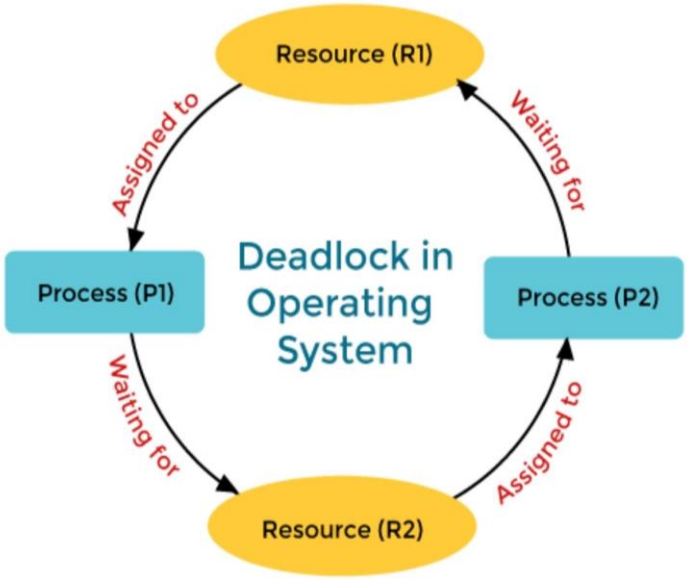

Figure 3: Deadlock Process

EXAMPLE OF DEADLOCK:

Consider two processes, P1 and P2, and two resources, R1 and R2. The processes have the following sequence of events:

P1 acquires R1.

P2 acquires R2.

P1 requests R2 but is blocked because P2 is holding R2.

P2 requests R1 but is blocked because P1 is holding R1.

Now, both processes are waiting for a resource held by the other, creating a deadlock.

PREVENTION AND HANDLING OF DEADLOCK:

To prevent or handle deadlocks, various strategies are employed, including:

- Deadlock Prevention: Ensuring that at least one of the Coffin Conditions is not satisfied.
- Deadlock Avoidance: Dynamically analyzing the resource allocation state to ensure that a circular wait cannot occur.
- Deadlock Detection and Recovery: Identifying deadlocks after they occur and taking corrective actions, such as process termination or resource deallocation.

CODE:

```
#include <iostream>
#include <vector>
#include <unordered_set>

using namespace std;

struct MemoryBlock {
    int processId;
    int blockSize;
};

vector<MemoryBlock> memory;

void allocateMemory(int processId, int blockSize);
void deallocateMemory(int processId, int blockSize);
void displayMemory();
bool isDeadlocked(int processId);
bool isDeadlockedHelper(int processId, unordered_set<int>& visited,
unordered_set<int>& inProgress, unordered_set<int>& allocatedProcesses);
bool recoverFromDeadlock(int processId);

int main() {
    cout << "Memory Allocation Simulation with Deadlock Detection" << endl;

    allocateMemory(1, 50);
    allocateMemory(2, 30);
    displayMemory();
```

```
    deallocateMemory(1, 20);
    displayMemory();

    allocateMemory(3, 10);
    displayMemory();

    int processToCheck = 2;
    if (isDeadlocked(processToCheck)) {
        cout << "Deadlock detected for Process " << processToCheck << "." <<
endl;
        if (recoverFromDeadlock(processToCheck)) {
            cout << "Recovered from deadlock. Trying again..." << endl;
            if (isDeadlocked(processToCheck)) {
                cout << "Deadlock still detected after recovery." << endl;
            } else {
                cout << "No deadlock detected after recovery." << endl;
            }
        } else {
            cout << "Failed to recover from deadlock." << endl;
        }
    } else {
        cout << "No deadlock detected." << endl;
    }

    return 0;
}

void allocateMemory(int processId, int blockSize) {
    for (size_t i = 0; i < memory.size(); ++i) {
        if (memory[i].blockSize >= blockSize) {
            memory[i].blockSize -= blockSize;
            if (memory[i].blockSize == 0) {
                memory.erase(memory.begin() + i);
            }
            cout << "Process " << processId << " allocated " << blockSize << "
units of memory." << endl;
            return;
        }
    }
    cout << "Process " << processId << " cannot be allocated. Not enough
memory available." << endl;
}

void deallocateMemory(int processId, int blockSize) {
    memory.push_back({ processId, blockSize });
    cout << "Memory deallocated for Process " << processId << " with " <<
blockSize << " units." << endl;
}
```

```
void displayMemory() {
    cout << "Memory status:" << endl;
    for (const auto& block : memory) {
        cout << "Process " << block.processId << ": " << block.blockSize << "
units" << endl;
    }
    cout << "---------------------------" << endl;
}

bool isDeadlocked(int processId) {
    unordered_set<int> visited;
    unordered_set<int> inProgress;
    unordered_set<int> allocatedProcesses;

    return isDeadlockedHelper(processId, visited, inProgress,
allocatedProcesses);
}

bool isDeadlockedHelper(int processId, unordered_set<int>& visited,
unordered_set<int>& inProgress, unordered_set<int>& allocatedProcesses) {
    if (visited.count(processId)) {
        return inProgress.count(processId);
    }

    visited.insert(processId);
    inProgress.insert(processId);
    allocatedProcesses.insert(processId);

    for (const auto& block : memory) {
        if (block.processId == processId &&
!isDeadlockedHelper(block.processId, visited, inProgress, allocatedProcesses))
{
            inProgress.erase(processId);
            return false;
        }
    }

    inProgress.erase(processId);
    return true;
}

bool recoverFromDeadlock(int processId) {
    // Here you can implement logic to release resources held by the specified
process.
    // For simplicity, let's assume releasing resources means deallocating all
memory blocks held by the process.
```

```
    for (size_t i = 0; i < memory.size(); ++i) {
        if (memory[i].processId == processId) {
            deallocateMemory(memory[i].processId, memory[i].blockSize);
            i--;  // Decrement i to recheck the current index since memory
vector size has changed.
        }
    }

    cout << "Recovered resources for Process " << processId << " from
deadlock." << endl;

    return true;  // Return true for successful recovery (in a real scenario,
additional checks might be needed).
}
```

EXPLANATION:

1. Header and Namespace Declaration:

```
#include <iostream>
#include <vector>
#include <unordered_set>
```

This code includes essential C++ standard libraries: iostream for input/output operations, vector for dynamic arrays, and unordered_set for an unordered set data structure.

2. Memory Block Structure:

```
struct MemoryBlock {
   int processId;
   int blockSize;
};
```

A structure named MemoryBlock is defined to represent a block of memory allocated to a process. It has two attributes: processId and blockSize.

3. Global Memory Vector and Allocation Functions:

```
vector<MemoryBlock> memory;

void allocateMemory(int processId, int blockSize) {
   // ... (code for memory allocation)
}

void deallocateMemory(int processId, int blockSize) {
   // ... (code for memory deallocation)
}
```

A global vector memory is declared to store memory blocks. Functions allocateMemory and deallocateMemory are defined to handle memory allocation and deallocation, respectively.

4. Display Memory Function:

```
void displayMemory() {
   cout << "Memory status:" << endl;
   for (const auto& block : memory) {
      cout << "Process " << block.processId << ": " << block.blockSize << " units"
    << endl;
   }
   cout << "----------------------------" << endl;
}
```

A function displayMemory is created to display the current status of memory blocks.

5. Deadlock Detection Functions:

```
bool isDeadlocked(int processId) {
   unordered_set<int> visited;
   unordered_set<int> inProgress;
   unordered_set<int> allocatedProcesses;
   return isDeadlockedHelper(processId, visited, inProgress, allocatedProcesses);
```

```
}
bool isDeadlockedHelper(int processId, unordered_set<int>& visited,
    unordered_set<int>& inProgress, unordered_set<int>& allocatedProcesses) {
  // ... (code for deadlock detection using depth-first search)
}
```

Functions isDeadlocked and isDeadlockedHelper are defined for deadlock detection. They use a depth-first search to identify circular waiting conditions among processes.

7. Recovery From Deadlock Function:

```
bool recoverFromDeadlock(int processId) {
  for (size_t i = 0; i < memory.size(); ++i) {
    if (memory[i].processId == processId) {
      deallocateMemory(memory[i].processId, memory[i].blockSize);
      i--;
    }
  }
  cout << "Recovered resources for Process " << processId << " from deadlock." <<
   endl;
  return true;
}
```

This function attempts to recover resources from a potential deadlock by releasing memory blocks held by a specified process. It iterates through the memory blocks, deallocating those belonging to the given process, and adjusts the loop index to recheck modified memory size. Upon completion, it outputs a success message and returns true, assuming successful resource recovery.

8. Main Function:

```
int main() {
  cout << "Memory Allocation Simulation with Deadlock Detection" << endl;

  allocateMemory(1, 50);
  allocateMemory(2, 30);
```

```
    displayMemory();

    deallocateMemory(1, 20);
    displayMemory();

    allocateMemory(3, 10);
    displayMemory();

    int processToCheck = 2;
    if (isDeadlocked(processToCheck)) {
        cout << "Deadlock detected for Process " << processToCheck << "." << endl;
        if (recoverFromDeadlock(processToCheck)) {
            cout << "Recovered from deadlock. Trying again..." << endl;
            if (isDeadlocked(processToCheck)) {
                cout << "Deadlock still detected after recovery." << endl;
            } else {
                cout << "No deadlock detected after recovery." << endl;
            }
        } else {
            cout << "Failed to recover from deadlock." << endl;
        }
    } else {
        cout << "No deadlock detected." << endl;
    }

    return 0;
}
```

The main function is the entry point of the program. It simulates memory allocation, deallocation, and checks for deadlock using the provided functions. The example demonstrates a simple memory allocation system with deadlock detection and attempts to recover from a deadlock scenario. The recovery process is followed by another deadlock check to verify if the recovery was successful.

OUTPUT:

```
Memory Allocation Simulation with Deadlock Detection
Process 1 cannot be allocated. Not enough memory available.
Process 2 cannot be allocated. Not enough memory available.
Memory status:
--------------------------
Memory deallocated for Process 1 with 20 units.
Memory status:
Process 1: 20 units
--------------------------
Process 3 allocated 10 units of memory.
Memory status:
Process 1: 10 units
--------------------------
Deadlock detected for Process 2.
Recovered resources for Process 2 from deadlock.
Recovered from deadlock. Trying again...
Deadlock still detected after recovery.
PS C:\Users\HP\Desktop\HTML>
```

Figure 4: Output

ANALYSIS:

1. Deadlock Detection Algorithm:

- Approach: Utilizes Depth-First Search (DFS).
- Goal: Detect circular waiting conditions among processes, indicating potential deadlocks.
- Implementation: The algorithm marks visited, in-progress, and allocated processes to identify circular dependencies.

2. Deadlock Recovery Algorithm:

- Strategy: Implements a simplified recovery approach.
- Process: Deallocates memory blocks associated with a specified process to break the circular dependency.
- Simplification: Assumes that releasing all memory blocks is sufficient for recovery.

3. Demonstration in Main Function:

- Simulation Steps:
- Simulates a memory allocation system where multiple processes request memory blocks.
- Processes go through memory allocation and deallocation phases.
- Checks for deadlocks using the detection algorithm.
- Attempts recovery by deallocating memory blocks.
- Rechecks for deadlock after recovery.
- Result Communication:
- Prints outcomes, indicating whether deadlocks are detected and if recovery is successful.

4. Additional Considerations:

- Recovery Simplification: Acknowledges the simplified recovery process for illustration purposes.
- Real-World Scenarios: Highlights the potential need for more intricate recovery mechanisms in practical applications.
- Resource Management: Points out the flexibility of the code to extend beyond memory to manage various types of resources.
- Memory Allocation System: Reflects the system's capability to handle multiple processes requesting memory blocks.

5. Conclusion:

- Simulation Overview: Presents a fundamental simulation of a memory allocation system with deadlock detection and simplified recovery.
- Practical Application: Emphasizes the need for tailored recovery mechanisms in real-world scenarios, beyond the simplistic approach demonstrated.
- Multiple Processes: Illustrates the system's ability to manage concurrent memory requests from multiple processes, simulating a more realistic environment.

GROUP PHOTO

Figure 5: Group Photo