ASSIGNMENT

By

Bhuvan Sharma

2022a1r003

3rd Sem (A-2)

C.S.E

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: Operating System [COM -302]

Due Date: 4/12/23

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	

Faculty Signature: Dr. Mekhla Sharma (Assistant Professor)
Email: mekhla.cse@mietjammu.in

INDEX:

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.

Code for Non-Preemptive Priority Scheduling (Assuming all processes arrived at time 0):

```
#include<stdio.h>

void main() {
    int bt[20], p[20], wt[20], tat[20], pr[20], at[20], i, j, n, total=0, pos,
temp, avg_wt, avg_tat;
    printf("Enter Total Number of Process:");
    scanf("%d", &n);

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

    //sorting burst time, priority, and process number in ascending order using
selection sort
    for(i = 0; i < n; i++) {
        pos = i;
        for(j = i + 1; j < n; j++) {
            if(pr[j] < pr[pos])
                pos = j;
        }

        temp = pr[i];
        pr[i] = pr[pos];
        pr[pos] = temp;
```

```
        temp = bt[i];
        bt[i] = bt[pos];
        bt[pos] = temp;

        temp = p[i];
        p[i] = p[pos];
        p[pos] = temp;
    }

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

    //calculate waiting time
    for(i = 1; i < n; i++) {
        wt[i] = bt[i - 1] + wt[i - 1];
        total += wt[i];
    }

    avg_wt = total / n; //average waiting time
    total = 0;

    printf("\nProcess\t    Burst Time    \tWaiting Time\tTurnaround Time");
    for(i = 0; i < n; i++) {
        tat[i] = bt[i] + wt[i]; //calculate turnaround time
        total += tat[i];
        printf("\nP[%d]\t\t  %d\t\t    %d\t\t\t%d",p[i], bt[i], wt[i], tat[i]);
    }
    avg_tat = total / n; //average turnaround time
    printf("\n\nAverage Waiting Time=%d",avg_wt);
    printf("\nAverage Turnaround Time=%d\n",avg_tat);
}
```

Code Breakdown and Analysis:

Priority-based non-preemptive scheduling is a CPU scheduling algorithm where processes are executed based on their priority without interrupting the currently running process. In contrast to preemptive scheduling, here, once a process starts executing, it continues until it completes its CPU burst, regardless of any higher-priority processes arriving later.

Key Characteristics:

1. **Priority Assignment**:
    - Each process is assigned a priority value that defines its importance or urgency.

2. **Selection of Process:**
    - The scheduler selects the process with the highest priority for execution.
    - Unlike preemptive scheduling, once a process begins execution, it continues until it finishes its burst time.

3. **No Preemption:**
    - Once a process starts its execution, it cannot be interrupted by other processes, even if higher-priority processes arrive later.

4. **Scheduling Decision:**
    - The decision of which process to execute next is based solely on the priority assigned to each process.
    - The scheduler selects the highest-priority process among the ready-to-execute processes only when the current process completes.

Algorithm Steps:

1. **Arrival of Processes:**
   - Record the arrival time of each process.

2. **Assignment of Priorities:**
   - Assign a priority value to each process, indicating its importance.

3. **Process Execution**:
   - Select the process with the highest priority for execution.
   - Execute the chosen process until it completes its CPU burst.

4. **Completion of Processes**:
   - Continue executing processes based on their priorities in a non-preemptive manner, allowing each process to complete its burst before considering the next one.

5. **Calculation of Turnaround and Waiting Times**:
   - Calculate the turnaround time (completion time - arrival time) and waiting time (turnaround time - burst time) for each process.
   - Compute the average turnaround and waiting times for all processes.

Advantages:

- Simpler to implement compared to preemptive scheduling algorithms.
- Avoids the overhead of context switching that occurs in preemptive scheduling.

Limitations:

- May lead to longer waiting times for higher-priority processes arriving after lower-priority processes have started execution.
- It might not be suitable for time-critical tasks that require immediate processing.

Priority-based non-preemptive scheduling optimizes the execution order of processes based on their priorities, allowing higher-priority processes to run without interruption, ensuring they complete their CPU bursts without being preempted by later-arriving higher-priority tasks.

This code is an implementation of a scheduling algorithm Priority Scheduling. It organizes processes based on their priorities and executes them accordingly.

1. Header and Libraries

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

This line includes the standard input-output library in C, which provides functions like printf() and scanf() used for input and output operations.

2. Function Declaration

```
void main() {
    // Code
}
```

The main() function is the starting point of execution in a C program. This program doesn't accept any command-line arguments (void within the parentheses).

3. Variable Declaration

```
int bt[20], p[20], wt[20], tat[20], pr[20], at[20], i, j, n, total=0, pos, temp, avg_wt, avg_tat;
```

Here, various arrays and integer variables are declared to store burst times (bt[]), priorities (pr[]), waiting times (wt[]), turnaround times (tat[]), process numbers (p[]), arrival times (at[]), and loop control variables (i, j). n represents the total number of processes, and total stores the sum of waiting times and turnaround times.

4. Input Gathering

```
printf("Enter Total Number of Process:");
scanf("%d", &n);
```

The user is prompted to input the total number of processes, which is stored in the variable n.

5. Burst Time and Priority Input

```
for(i = 0; i < n; i++) {
   // Prompt for burst time and priority for each process
   printf("\nP[%d]\n",i);
   printf("Burst Time:");
   scanf("%d",&bt[i]);
   printf("Priority:");
   scanf("%d",&pr[i]);
   p[i]=i; //contains process number
}
```

This loop gathers the burst time and priority for each process using a for loop. The process number is stored in p[].

6. Sorting Processes

```
    for(i = 0; i < n; i++) {
    pos = i;
    for(j = i + 1; j < n; j++) {
       if(pr[j] < pr[pos])
          pos = j;
    }
    temp = pr[i];
    pr[i] = pr[pos];
    pr[pos] = temp;
    temp = bt[i];
    bt[i] = bt[pos];
    bt[pos] = temp;
    temp = p[i];
    p[i] = p[pos];
    p[pos] = temp;
  }
```

The code sorts the processes based on their priority in ascending order using the Selection Sort algorithm. It rearranges the burst time, priority, and process number accordingly.

7. Calculating Waiting Time

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

   //calculate waiting time

   for(i = 1; i < n; i++) {

       wt[i] = bt[i - 1] + wt[i - 1];

       total += wt[i];

   }
```

The waiting time for the first process is set to zero (wt[0]). Then, the loop calculates the waiting time for each subsequent process.

8. Calculating Turnaround Time and Output

```
      for(i = 0; i < n; i++) {

       tat[i] = bt[i] + wt[i]; //calculate turnaround time

       total += tat[i];
```

This loop calculates the turnaround time for each process and outputs details like process number, burst time, waiting time, and turnaround time.

9. Calculating Average Waiting Time and Turnaround Time

```
avg_wt = total / n; // Average waiting time

avg_tat = total / n; // Average turnaround time
```

Finally, the code calculates and displays the average waiting time and average turnaround time for all processes.

Output:

```
Enter Total Number of Process:5

Enter Burst Time and Priority

P[0]
Burst Time:10
Priority:0

P[1]
Burst Time:3
Priority:6

P[2]
Burst Time:7
Priority:2

P[3]
Burst Time:11
Priority:1

P[4]
Burst Time:9
Priority:3
```

```
Process     Burst Time          Waiting Time    Turnaround Time
P[0]              10                    0                10
P[3]              11                   10                21
P[2]              7                    21                28
P[4]              9                    28                37
P[1]              3                    37                40

Average Waiting Time=19
Average Turnaround Time=27
```

Gantt Chart:

(Lower to Higher)

P[0]	P[3]	P[2]	P[4]	P[1]

0 10 21 28 37 40

Code for Preemptive Priority Scheduling:

```
#include<stdio.h>
#define MIN -9999;
struct proc
{
    int no,at,bt,rt,ct,wt,tat,pri,temp;
};
struct proc read(int i)
{
    struct proc p;
    printf("\nProcess No: %d\n",i);
    p.no=i;
    printf("Enter Arrival Time: ");
    scanf("%d",&p.at);
    printf("Enter Burst Time: ");
    scanf("%d",&p.bt);
    p.rt=p.bt;
    printf("Enter Priority: ");
    scanf("%d",&p.pri);
    p.temp=p.pri;
    return p;
}
void main()
{
    int i,n,c,remaining,max_val,max_index;
    struct proc p[10],temp;
    float avgtat=0,avgwt=0;
    printf("<--Highest Priority First Scheduling Algorithm (Preemptive)-->\n");
    printf("Enter Number of Processes: ");
    scanf("%d",&n);
    for(int i=0;i<n;i++)
        p[i]=read(i+1);
    remaining=n;
    for(int i=0;i<n-1;i++)
        for(int j=0;j<n-i-1;j++)
            if(p[j].at>p[j+1].at)
            {
            temp=p[j];
            p[j]=p[j+1];
            p[j+1]=temp;
            }
    max_val=p[0].temp,max_index=0;
    for(int j=0;j<n&&p[j].at<=p[0].at;j++)
        if(p[j].temp>max_val)
            max_val=p[j].temp,max_index=j;
    i=max_index;
    c=p[i].ct=p[i].at+1;
    p[i].rt--;
```

```
    if(p[i].rt==0)
    {
        p[i].temp=MIN;
        remaining--;
    }
    while(remaining>0)
    {
        max_val=p[0].temp,max_index=0;
        for(int j=0;j<n&&p[j].at<=c;j++)
            if(p[j].temp>max_val)
                max_val=p[j].temp,max_index=j;
        i=max_index;
        p[i].ct=c=c+1;
        p[i].rt--;
        if(p[i].rt==0)
        {
            p[i].temp=MIN;
            remaining--;
        }
    }
    printf("\nProcessNo\tAT\tBT\tPri\tCT\tTAT\tWT\n");

    for(int i=0;i<n;i++)
    {
        p[i].tat=p[i].ct-p[i].at;
        avgtat+=p[i].tat;
        p[i].wt=p[i].tat-p[i].bt;
        avgwt+=p[i].wt;
        printf("P%d\t\t%d\t%d\t%d\t%d\t%d\t%d\n",p[i].no,p[i].at,p[i].bt,p[i].p
ri,p[i].ct,p[i].tat,p[i].wt);
    }
    avgtat/=n,avgwt/=n;
    printf("\nAverage TurnAroundTime=%f\nAverage WaitingTime=%f",avgtat,avgwt);
}
```

Code Breakdown and Analysis:

Priority-based preemptive scheduling is a CPU scheduling algorithm where processes are executed based on their priority. Each process is assigned a priority, and the scheduler selects the process with the highest priority for execution. In preemptive priority scheduling, a running process can be interrupted by a higher-priority process that arrives later.

Key Characteristics:

1) Priority Assignment:

- Each process is assigned a priority value that determines its importance or urgency.

2) Selection of Process:

- The scheduler selects the process with the highest priority for execution.
- Preemptive nature allows a higher-priority process to interrupt the execution of a lower-priority one, even if the lower-priority process is currently running.

3) Priority Adjustment:

- Priorities might change dynamically based on the behavior or requirements of the processes.
- If a higher-priority process arrives during the execution of a lower-priority process, the lower-priority process may be paused, allowing the higher-priority one to run.

4) Scheduling Decision:

- The decision of which process to execute next is based on the priority assigned to each process.
- The scheduler constantly evaluates and selects the process with the highest priority among the set of ready-to-execute processes.

Algorithm Steps:

1. Arrival of Processes:

- Record the arrival time of each process.

2. Assignment of Priorities:

- Assign a priority value to each process, indicating its importance.

3. Process Execution:

- Select the process with the highest priority for execution.
- If a higher-priority process arrives while another process is running, preempt the current process and execute the higher-priority one.

4. Completion of Processes:

- Continue executing processes based on their priorities until all processes are completed.

5. Calculation of Turnaround and Waiting Times:

- Calculate the turnaround time (completion time - arrival time) and waiting time (turnaround time - burst time) for each process.
- Compute the average turnaround and waiting times for all processes.

Advantages:

- It ensures that high-priority tasks get executed with minimal delay, ensuring critical tasks are completed promptly.
- Helps in meeting deadlines for important processes or tasks.

Limitations:

- May lead to starvation: Lower-priority processes might never get a chance to execute if higher-priority ones continually arrive.
- Process priorities need to be managed effectively to avoid resource monopolization by high-priority processes.

Priority-based preemptive scheduling is an approach aimed at optimizing resource utilization and ensuring efficient execution of processes by constantly prioritizing and allocating CPU time based on process importance.

Priority-based preemptive scheduling is an algorithm where processes are scheduled for execution based on their priority. The higher the priority, the sooner a process gets CPU time. Preemptive means that a higher priority process can interrupt the execution of a lower priority one.

1. Libraries and Macro

```
#include<stdio.h>
#define MIN -9999;
```

The code includes the standard input-output library and defines a macro MIN as -9999.

2. Struct Declaration

```
struct proc {
    int no, at, bt, rt, ct, wt, tat, pri, temp;
};
```

Defines a structure proc to hold process attributes such as process number (no), arrival time (at), burst time (bt), remaining time (rt), completion time (ct), waiting time (wt), turnaround time (tat), priority (pri), and a temporary priority value (temp).

3. Function read()

```
struct proc read(int i)
{
   struct proc p;
   printf("\nProcess No: %d\n",i);
   p.no=i;
   printf("Enter Arrival Time: ");
   scanf("%d",&p.at);
   printf("Enter Burst Time: ");
   scanf("%d",&p.bt);
   p.rt=p.bt;
   printf("Enter Priority: ");
   scanf("%d",&p.pri);
   p.temp=p.pri;
   return p;
}
```

This function reads and returns process attributes for a given process number (i).

4. Main Function

```
void main() {
    // Code for the main function
}
```

The main function where the HPF scheduling algorithm is implemented.

5. Input Gathering and Sorting

```
for(int i=0;i<n;i++)
        p[i]=read(i+1);
    remaining=n;
    for(int i=0;i<n-1;i++)
        for(int j=0;j<n-i-1;j++)
            if(p[j].at>p[j+1].at)
            {
            temp=p[j];
            p[j]=p[j+1];
            p[j+1]=temp;
            }
```

The code collects process attributes by calling the read() function and sorts the processes based on their arrival times using a bubble sort algorithm.

6. Preemptive Priority Scheduling

```
while(remaining>0)
    {
        max_val=p[0].temp,max_index=0;
        for(int j=0;j<n&&p[j].at<=c;j++)
            if(p[j].temp>max_val)
                max_val=p[j].temp,max_index=j;
        i=max_index;
        p[i].ct=c=c+1;
        p[i].rt--;
        if(p[i].rt==0)
        {
```

```
        p[i].temp=MIN;
        remaining--;
      }
   }
```

The algorithm schedules processes based on their priority values in a preemptive manner, updating their completion times and remaining times until all processes are executed.

7. Calculating Turnaround Time and Waiting Time

```
for(int i=0;i<n;i++)
   {
      p[i].tat=p[i].ct-p[i].at;
      avgtat+=p[i].tat;
      p[i].wt=p[i].tat-p[i].bt;
      avgwt+=p[i].wt;
      printf("P%d\t\t%d\t%d\t%d\t%d\t%d\t%d\n",p[i].no,p[i].at,p[i].bt,p[i].pri,p[i].ct,p[i].tat,p[i].wt);
   }
```

This section calculates the turnaround time and waiting time for each process and then computes the average turnaround time and average waiting time for all processes.

8. Output

```
{
…
…
…
avgtat/=n,avgwt/=n;
   printf("\nAverage TurnAroundTime=%f\nAverage WaitingTime=%f",avgtat,avgwt);
}
```

Displays the process details including process number, arrival time, burst time, priority, completion time, turnaround time, and waiting time. It also prints the average turnaround time and average waiting time.

Overall, this code simulates a Preemptive Priority Based Scheduling algorithm and calculates performance metrics for a set of processes provided by the user, such as turnaround time and waiting time.

Output:

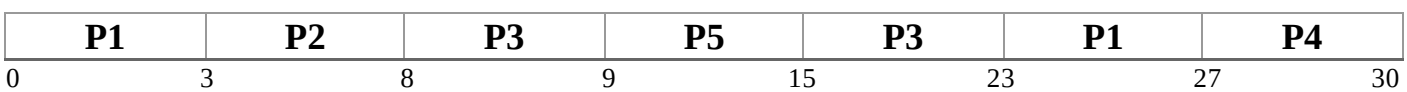
```
<--Highest Priority First Scheduling Algorithm (Preemptive)-->
Enter Number of Processes: 5

Process No: 1
Enter Arrival Time: 0
Enter Burst Time: 7
Enter Priority: 2

Process No: 2
Enter Arrival Time: 3
Enter Burst Time: 5
Enter Priority: 4

Process No: 3
Enter Arrival Time: 5
Enter Burst Time: 9
Enter Priority: 3

Process No: 4
Enter Arrival Time: 10
Enter Burst Time: 3
Enter Priority: 1

Process No: 5
Enter Arrival Time: 9
Enter Burst Time: 6
Enter Priority: 5
```

ProcessNo	AT	BT	Pri	CT	TAT	WT
P1	0	7	2	27	27	20
P2	3	5	4	8	5	0
P3	5	9	3	23	18	9
P5	9	6	5	15	6	0
P4	10	3	1	30	20	17

```
Average TurnAroundTime=15.200000
Average WaitingTime=9.200000
```

Gantt Chart:

(Priority: Higher to Lower)



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_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;

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];
    }
}

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
}

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];
    }
}


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
}
void print_matrix(const char* name, int matrix[MAX_PROCESSES][MAX_RESOURCES],
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() {
    // 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;
}
```

Code Breakdown and Analysis:

1. Memory Allocation System:

- **Processes:** Create a representation for multiple processes, each requesting memory blocks.
- **Memory Blocks:** Represent available memory blocks and track their allocation status.
- **Allocation Methods:** Implement memory allocation methods (like First Fit, Best Fit, or Worst Fit) to allocate memory to processes.

2. Deadlock Detection Algorithm:

- **Resource Allocation Graph (RAG):** Maintain a graph representing processes as nodes and allocated resources as edges.
- **Cycle Detection:** Implement an algorithm to detect cycles in the RAG. A cycle indicates a potential deadlock.
- **Detection Strategy:** Periodically check for cycles in the RAG to detect potential deadlocks.

3. Deadlock Recovery:

- **Resource Preemption:** When a deadlock is detected, use resource preemption to break the cycle.
- **Process Termination:** Terminate one or more processes involved in the deadlock to free up resources.
- **Rollback Mechanism:** Rollback the allocation to a safe state before the deadlock occurred.

Execution Workflow:

1. **Process Creation:** Create multiple processes, each requesting memory blocks. Maintain data structures to track their requests and allocated memory.
2. **Memory Allocation:** Implement memory allocation methods to assign memory blocks to processes based on available blocks and allocation strategies.

3. **Deadlock Detection:** Periodically check for cycles in the RAG (Resource Allocation Graph) representing process-resource relationships.

4. **Deadlock Handling:**
    - If a deadlock is detected:
        - Identify the processes and resources involved in the deadlock.
        - Choose a strategy to break the deadlock (resource preemption or process termination).
    - Free up resources by terminating selected processes or releasing allocated memory blocks.
5. **System Recovery:** After resolving the deadlock, ensure the system returns to a consistent and safe state without further deadlocks.

Considerations:

- **Safety and Avoidance:** Implement resource allocation methods that aim to avoid deadlock scenarios.
- **Efficiency:** Optimize the deadlock detection and recovery process to minimize system downtime.
- **Logging and Reporting:** Maintain logs and reports of deadlock occurrences and recovery actions for system analysis and improvement.

This high-level design outlines the components and strategies needed for a memory allocation system with deadlock detection and recovery mechanisms. The focus should be on implementing robust memory allocation methods and an efficient deadlock detection algorithm to ensure the system's stability and reliability.

1. Initialization of Data Structures and Variables:

```
#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;
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];
    }
}
```

This section defines arrays to represent allocation, maximum allocation, available resources, and a boolean array to track finished processes. The initialize() function sets up these arrays based on the input provided.

2. Resource Request Handling:

```
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
}
```

The request_resources() function checks if a process can be allocated the requested resources without causing a deadlock. If the resources are available, it allocates them to the process and returns true, otherwise returns false.

3. Resource Release Function:

```
void release_resources(int pid, int release[]) {
    for (int i = 0; i < num_resources; ++i) {
        if (release[i] > allocation[pid][i]) {
            release[i] = allocation[pid][i];
        }
        available[i] += release[i];
        allocation[pid][i] -= release[i];
    }
}
```

The release_resources() function releases resources held by a process. It updates the available resources and adjusts the allocation for the specified process.

4. Deadlock Detection Algorithm:

```
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
}
```

The is_deadlocked() function simulates resource allocation and checks for potential deadlocks by analyzing the resource allocation state among processes. It returns true if a deadlock is detected, false otherwise.

5. Printing Utility:

```
void print_matrix(const char* name, int matrix[MAX_PROCESSES][MAX_RESOURCES], 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");
    }
}
```

The print_matrix() function prints matrices representing resource allocation or maximum allocation for processes. It assists in visualizing the allocation state for debugging.

6. Main Function Flow:

```
int main() {
    // Initialization of processes, resources, allocation, and available resources
    // Demonstration of resource allocation and deadlock recovery
    return 0;
}
```

The main() function orchestrates the program flow. It initializes the system, demonstrates resource allocation, and handles potential deadlock scenarios by requesting resources and attempting recovery by releasing them.

Output:

```
Initial Allocation:
Process | Allocation (R1 R2 R3)
P0      | 0 1 0
P1      | 2 0 0
P2      | 3 0 2
P3      | 2 1 1
P4      | 0 0 2

Available Resources: 3 3 2

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      | 0 1 0
P1      | 2 0 0
P2      | 3 0 2
P3      | 2 1 1
P4      | 0 0 0

Available Resources after potential recovery: 3 3 4
```

Group Photo