QS. Write a C programme to simulate the following **non-preemptive** CPU scheduling algorithms to find the turnaround time and waiting time for the above problem.

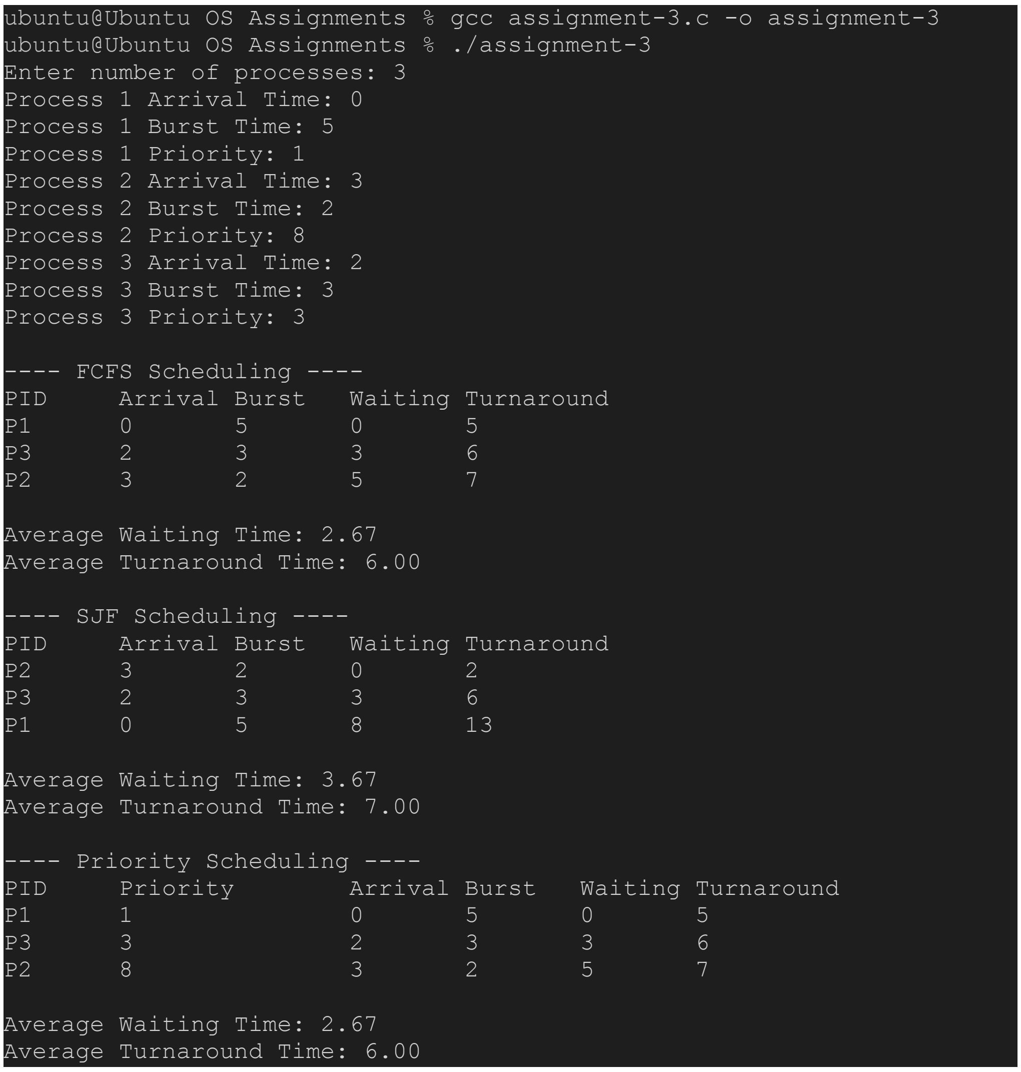
a. FCFS
b. SJF
c. Priority

Input :

```
#include <stdio.h>
struct Process {
   int pid;
   int burst;
   int arrival;
   int priority;
   int waiting;
   int turnaround;
};
void calculateAverage(struct Process p[], int n) {
   float totalWT = 0, totalTAT = 0;
   for (int i = 0; i < n; i++) {
       totalWT += p[i].waiting;
       totalTAT += p[i].turnaround;
   }
   printf("\nAverage Waiting Time: %.2f", totalWT / n);
   printf("\nAverage Turnaround Time: %.2f\n", totalTAT / n);
}

void FCFS(struct Process p[], int n) {
   printf("\n---- FCFS Scheduling ----\n");
   int time = 0;
   for (int i = 0; i < n - 1; i++) {
       for (int j = 0; j < n - i - 1; j++) {
           if (p[j].arrival > p[j + 1].arrival) {
               struct Process temp = p[j];
               p[j] = p[j + 1];
               p[j + 1] = temp;
           }
       }
   }
   time = p[0].arrival;
   for (int i = 0; i < n; i++) {
       if (time < p[i].arrival)
           time = p[i].arrival;
       p[i].waiting = time - p[i].arrival;
       time += p[i].burst;
       p[i].turnaround = p[i].waiting + p[i].burst;
   }
   printf("PID\tArrival\tBurst\tWaiting\tTurnaround\n");
   for (int i = 0; i < n; i++)
       printf("P%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].arrival, p[i].burst,
p[i].waiting, p[i].turnaround);
   calculateAverage(p, n);
}

void SJF(struct Process p[], int n) {
   printf("\n---- SJF Scheduling ----\n");
   for (int i = 0; i < n - 1; i++) {
       for (int j = 0; j < n - i - 1; j++) {
           if (p[j].burst > p[j + 1].burst ||
               (p[j].burst == p[j + 1].burst && p[j].arrival > p[j +
1].arrival)) {
               struct Process temp = p[j];
               p[j] = p[j + 1];
               p[j + 1] = temp;
           }
       }
```

```c
    }
    int time = p[0].arrival;
    for (int i = 0; i < n; i++) {
        if (time < p[i].arrival)
            time = p[i].arrival;
        p[i].waiting = time - p[i].arrival;
        time += p[i].burst;
        p[i].turnaround = p[i].waiting + p[i].burst;
    }
    printf("PID\tArrival\tBurst\tWaiting\tTurnaround\n");
    for (int i = 0; i < n; i++)
        printf("P%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].arrival, p[i].burst,
p[i].waiting, p[i].turnaround);
    calculateAverage(p, n);
}

void PriorityScheduling(struct Process p[], int n) {
    printf("\n---- Priority Scheduling ----\n");
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (p[j].priority > p[j + 1].priority ||
                (p[j].priority == p[j + 1].priority && p[j].arrival > p[j +
1].arrival)) {
                struct Process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }
    int time = p[0].arrival;
    for (int i = 0; i < n; i++) {
        if (time < p[i].arrival)
            time = p[i].arrival;
        p[i].waiting = time - p[i].arrival;
        time += p[i].burst;
        p[i].turnaround = p[i].waiting + p[i].burst;
    }
    printf("PID\tPriority\tArrival\tBurst\tWaiting\tTurnaround\n");
    for (int i = 0; i < n; i++)
        printf("P%d\t%d\t\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].priority,
p[i].arrival, p[i].burst, p[i].waiting, p[i].turnaround);
    calculateAverage(p, n);
}
int main() {
    struct Process p1[10], p2[10], p3[10];
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        printf("Process %d Arrival Time: ", i + 1);
        scanf("%d", &p1[i].arrival);
        printf("Process %d Burst Time: ", i + 1);
        scanf("%d", &p1[i].burst);
        printf("Process %d Priority: ", i + 1);
        scanf("%d", &p1[i].priority);
        p1[i].pid = i + 1;
        p2[i] = p3[i] = p1[i];
    }
    FCFS(p1, n);
    SJF(p2, n);
    PriorityScheduling(p3, n);
    return 0;
}
```

Output :



```
ubuntu@Ubuntu OS Assignments % gcc assignment-3.c -o assignment-3
ubuntu@Ubuntu OS Assignments % ./assignment-3
Enter number of processes: 3
Process 1 Arrival Time: 0
Process 1 Burst Time: 5
Process 1 Priority: 1
Process 2 Arrival Time: 3
Process 2 Burst Time: 2
Process 2 Priority: 8
Process 3 Arrival Time: 2
Process 3 Burst Time: 3
Process 3 Priority: 3

---- FCFS Scheduling ----
PID     Arrival Burst   Waiting Turnaround
P1      0       5       0       5
P3      2       3       3       6
P2      3       2       5       7

Average Waiting Time: 2.67
Average Turnaround Time: 6.00

---- SJF Scheduling ----
PID     Arrival Burst   Waiting Turnaround
P2      3       2       0       2
P3      2       3       3       6
P1      0       5       8       13

Average Waiting Time: 3.67
Average Turnaround Time: 7.00

---- Priority Scheduling ----
PID     Priority        Arrival Burst   Waiting Turnaround
P1      1               0       5       0       5
P3      3               2       3       3       6
P2      8               3       2       5       7

Average Waiting Time: 2.67
Average Turnaround Time: 6.00
```

QS. Write a C program to simulate a multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue.

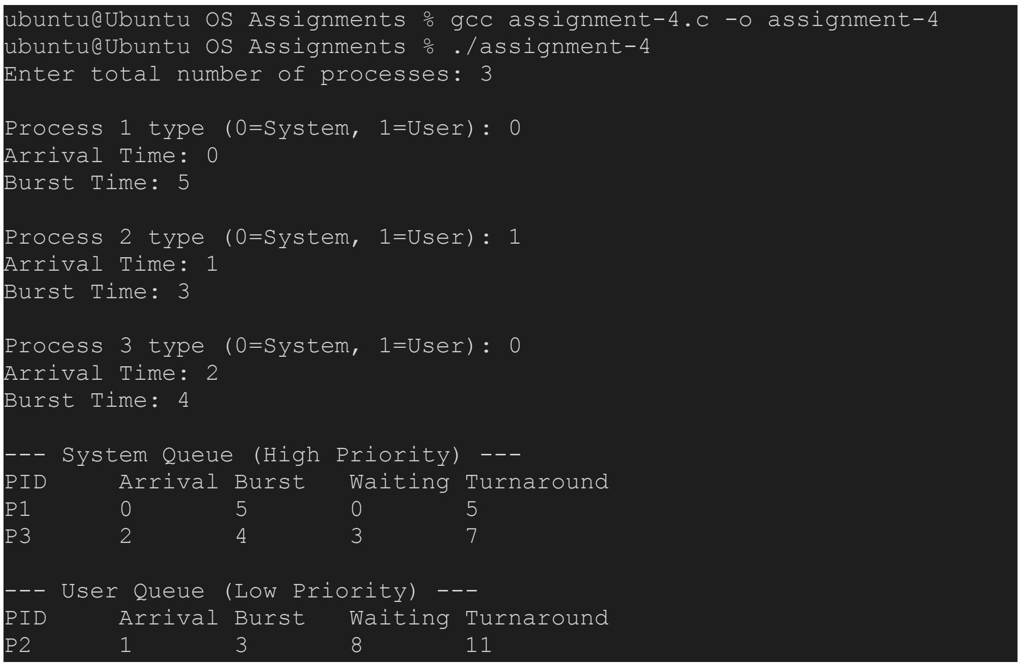
DESCRIPTION

A multi-level queue scheduling algorithm is used in scenarios where the processes can be classified into groups based on properties like process type, CPU time, IO access, memory size, etc. In a multi-level queue scheduling algorithm, there will be 'n' number of queues, where 'n' is the number of groups the processes are classified into. Each queue will be assigned a priority and will have its own scheduling algorithm like round-robin scheduling or FCFS. For the process in a queue to execute, all the queues of priority higher than it should be empty, meaning the process in those high-priority queues should have completed its execution. In this scheduling algorithm, once assigned to a queue, the process will not move to any other queues.

Input :

```c
#include <stdio.h>
struct Process {
   int pid, arrival, burst, waiting, turnaround, type;
};
void fcfs(struct Process p[], int n, int startTime) {
   int time = startTime;
   for (int i = 0; i < n; i++) {
       if (time < p[i].arrival)
           time = p[i].arrival;
       p[i].waiting = time - p[i].arrival;
       time += p[i].burst;
       p[i].turnaround = p[i].waiting + p[i].burst;
   }
   for (int i = 0; i < n; i++)
       printf("P%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].arrival, p[i].burst,
p[i].waiting, p[i].turnaround);
}
int main() {
   struct Process sys[10], user[10];
   int n, s = 0, u = 0;
   printf("Enter total number of processes: ");
   scanf("%d", &n);
   for (int i = 0; i < n; i++) {
       int t;
       printf("\nProcess %d type (0=System, 1=User): ", i + 1);
       scanf("%d", &t);
       printf("Arrival Time: ");
       int a; scanf("%d", &a);
       printf("Burst Time: ");
       int b; scanf("%d", &b);
       if (t == 0) sys[s++] = (struct Process){i + 1, a, b, 0, 0, 0};
       else user[u++] = (struct Process){i + 1, a, b, 0, 0, 1};
   }
   printf("\n--- System Queue (High Priority) ---\n");
   printf("PID\tArrival\tBurst\tWaiting\tTurnaround\n");
   fcfs(sys, s, 0);
   int endTime = 0;
   for (int i = 0; i < s; i++)
       if (sys[i].arrival + sys[i].turnaround > endTime)
           endTime = sys[i].arrival + sys[i].turnaround;
   printf("\n--- User Queue (Low Priority) ---\n");
   printf("PID\tArrival\tBurst\tWaiting\tTurnaround\n");
   fcfs(user, u, endTime);
}
```

Output :



```
ubuntu@Ubuntu OS Assignments % gcc assignment-4.c -o assignment-4
ubuntu@Ubuntu OS Assignments % ./assignment-4
Enter total number of processes: 3

Process 1 type (0=System, 1=User): 0
Arrival Time: 0
Burst Time: 5

Process 2 type (0=System, 1=User): 1
Arrival Time: 1
Burst Time: 3

Process 3 type (0=System, 1=User): 0
Arrival Time: 2
Burst Time: 4

--- System Queue (High Priority) ---
PID     Arrival Burst   Waiting Turnaround
P1      0       5       0       5
P3      2       4       3       7

--- User Queue (Low Priority) ---
PID     Arrival Burst   Waiting Turnaround
P2      1       3       8       11
```

QS. Write a C program to simulate the MVT and MFT memory management techniques.

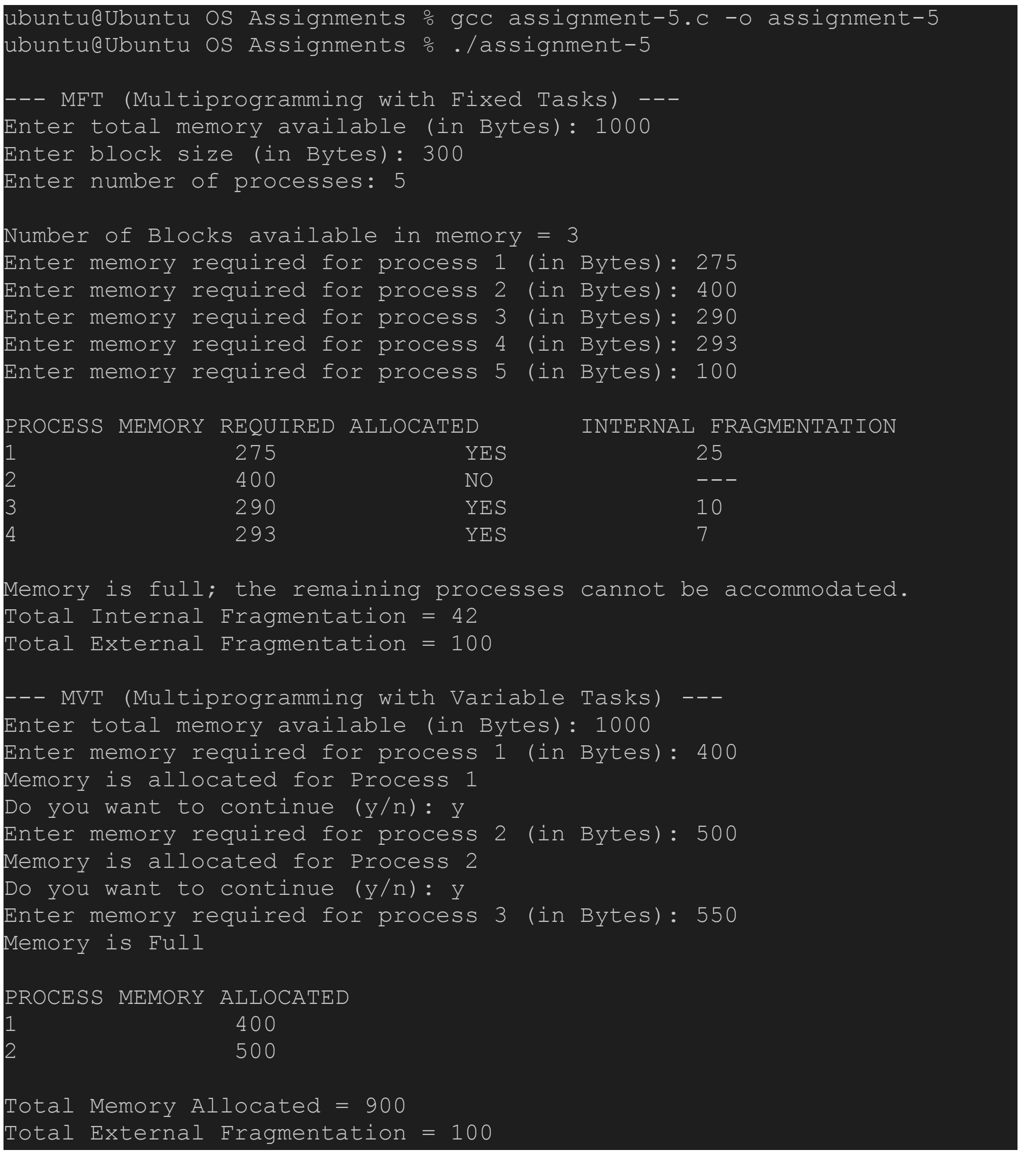
DESCRIPTION:
MFT (Multiprogramming with a Fixed Number of Tasks) is one of the old memory management techniques in which the memory is partitioned into fixed-size partitions and each job is assigned to a partition. The memory assigned to a partition does not change. MVT (Multiprogramming with a Variable Number of Tasks) is a memory management technique in which each job gets just the amount of memory it needs. That is, the partitioning of memory is dynamic and changes as jobs enter and leave the system. MVT is a more ``efficient" user of resources. MFT suffers from the problem of internal fragmentation, and MVT suffers from external fragmentation.

Input :

```
#include <stdio.h>
void MFT() {
   int total_memory, block_size, num_process, memory_required[10];
   int blocks, allocated[10] = {0}, internal_frag[10] = {0};
   int total_internal_frag = 0, external_frag = 0, i, j = 0;

   printf("\n--- MFT (Multiprogramming with Fixed Tasks) ---\n");
   printf("Enter total memory available (in Bytes): ");
   scanf("%d", &total_memory);
   printf("Enter block size (in Bytes): ");
   scanf("%d", &block_size);
   printf("Enter number of processes: ");
   scanf("%d", &num_process);
   blocks = total_memory / block_size;
   printf("\nNumber of Blocks available in memory = %d\n", blocks);
   for (i = 0; i < num_process; i++) {
       printf("Enter memory required for process %d (in Bytes): ", i + 1);
       scanf("%d", &memory_required[i]);
   }
   printf("\nPROCESS\tMEMORY REQUIRED\tALLOCATED\tINTERNAL
FRAGMENTATION\n");
   for (i = 0; i < num_process && j < blocks; i++) {
       if (memory_required[i] > block_size) {
           printf("%d\t\t%d\t\tNO\t\t---\n", i + 1, memory_required[i]);
           allocated[i] = 0;
       } else {
           allocated[i] = 1;
           internal_frag[i] = block_size - memory_required[i];
           total_internal_frag += internal_frag[i];
           printf("%d\t\t%d\t\tYES\t\t%d\n", i + 1, memory_required[i],
internal_frag[i]);
           j++;
       }
   }
   if (i < num_process)
       printf("\nMemory is full; the remaining processes cannot be
accommodated.\n");

   external_frag = total_memory - (blocks * block_size);
   printf("Total Internal Fragmentation = %d\n", total_internal_frag);
   printf("Total External Fragmentation = %d\n", external_frag);
}
void MVT() {
   int total_memory, allocated[10], memory_required, total_allocated = 0;
   int i = 0;
   char choice;

   printf("\n--- MVT (Multiprogramming with Variable Tasks) ---\n");
   printf("Enter total memory available (in Bytes): ");
```

```
    scanf("%d", &total_memory);

    do {
        printf("Enter memory required for process %d (in Bytes): ", i + 1);
        scanf("%d", &memory_required);

        if (memory_required <= (total_memory - total_allocated)) {
            allocated[i] = memory_required;
            total_allocated += memory_required;
            printf("Memory is allocated for Process %d\n", i + 1);
            i++;
        } else {
            printf("Memory is Full\n");
            break;
        }
        printf("Do you want to continue (y/n): ");
        scanf(" %c", &choice);
    } while (choice == 'y' || choice == 'Y');

    printf("\nPROCESS\tMEMORY ALLOCATED\n");
    for (int k = 0; k < i; k++)
        printf("%d\t\t%d\n", k + 1, allocated[k]);

    printf("\nTotal Memory Allocated = %d\n", total_allocated);
    printf("Total External Fragmentation = %d\n", total_memory -
total_allocated);
}
int main() {
    MFT();
    MVT();
}
```

Output :



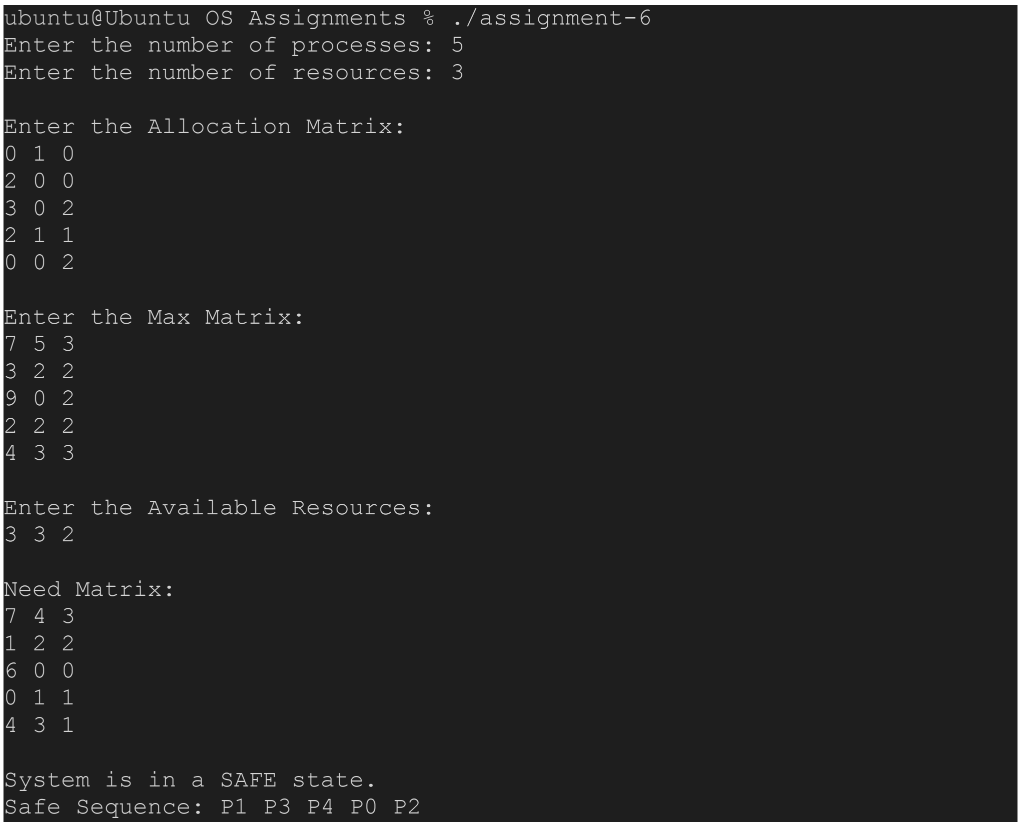
```
ubuntu@Ubuntu OS Assignments % gcc assignment-5.c -o assignment-5
ubuntu@Ubuntu OS Assignments % ./assignment-5

--- MFT (Multiprogramming with Fixed Tasks) ---
Enter total memory available (in Bytes): 1000
Enter block size (in Bytes): 300
Enter number of processes: 5

Number of Blocks available in memory = 3
Enter memory required for process 1 (in Bytes): 275
Enter memory required for process 2 (in Bytes): 400
Enter memory required for process 3 (in Bytes): 290
Enter memory required for process 4 (in Bytes): 293
Enter memory required for process 5 (in Bytes): 100

PROCESS MEMORY REQUIRED ALLOCATED       INTERNAL FRAGMENTATION
1               275             YES             25
2               400             NO              ---
3               290             YES             10
4               293             YES             7

Memory is full; the remaining processes cannot be accommodated.
Total Internal Fragmentation = 42
Total External Fragmentation = 100

--- MVT (Multiprogramming with Variable Tasks) ---
Enter total memory available (in Bytes): 1000
Enter memory required for process 1 (in Bytes): 400
Memory is allocated for Process 1
Do you want to continue (y/n): y
Enter memory required for process 2 (in Bytes): 500
Memory is allocated for Process 2
Do you want to continue (y/n): y
Enter memory required for process 3 (in Bytes): 550
Memory is Full

PROCESS MEMORY ALLOCATED
1               400
2               500

Total Memory Allocated = 900
Total External Fragmentation = 100
```

QS: For deadlock avoidance, write a C program to simulate the Bankers algorithm.

Input :

```c
#include <stdio.h>
int main() {
   int n, m, i, j, k;
   int alloc[10][10], max[10][10], avail[10];
   int need[10][10], finish[10], safeSeq[10];
   int count = 0;

   printf("Enter the number of processes: ");
   scanf("%d", &n);
   printf("Enter the number of resources: ");
   scanf("%d", &m);
   printf("\nEnter the Allocation Matrix:\n");
   for (i = 0; i < n; i++)
       for (j = 0; j < m; j++)
           scanf("%d", &alloc[i][j]);
   printf("\nEnter the Max Matrix:\n");
   for (i = 0; i < n; i++)
       for (j = 0; j < m; j++)
           scanf("%d", &max[i][j]);
   printf("\nEnter the Available Resources:\n");
   for (i = 0; i < m; i++)
       scanf("%d", &avail[i]);
   for (i = 0; i < n; i++)
       for (j = 0; j < m; j++)
           need[i][j] = max[i][j] - alloc[i][j];
   for (i = 0; i < n; i++)
       finish[i] = 0;
   printf("\nNeed Matrix:\n");
   for (i = 0; i < n; i++) {
       for (j = 0; j < m; j++)
           printf("%d ", need[i][j]);
       printf("\n");
   }

   while (count < n) {
       int found = 0;
       for (i = 0; i < n; i++) {
           if (finish[i] == 0) {
               int canAllocate = 1;
               for (j = 0; j < m; j++) {
                   if (need[i][j] > avail[j]) {
                       canAllocate = 0;
                       break;
                   }
               }
               if (canAllocate) {
                   for (k = 0; k < m; k++)
                       avail[k] += alloc[i][k];
                   safeSeq[count++] = i;
                   finish[i] = 1;
                   found = 1;
               }
           }
       }
       if (found == 0) {
           printf("\nSystem is NOT in a safe state!\n");
           return 0;
       }
   }
   printf("\nSystem is in a SAFE state.\nSafe Sequence: ");
   for (i = 0; i < n; i++)
       printf("P%d ", safeSeq[i]);
   printf("\n");
}
```

Output :



```
ubuntu@Ubuntu OS Assignments % ./assignment-6
Enter the number of processes: 5
Enter the number of resources: 3

Enter the Allocation Matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2

Enter the Max Matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3

Enter the Available Resources:
3 3 2

Need Matrix:
7 4 3
1 2 2
6 0 0
0 1 1
4 3 1

System is in a SAFE state.
Safe Sequence: P1 P3 P4 P0 P2
```