Week No-5 Practical Questions

Objective : Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

Algorithm: Bankers algorithm deadlock.

1. Input memory blocks and their sizes.
2. Input processes and their sizes.
3. Initialize all memory blocks as free.
4. For each process in the list of processes:
    - Traverse the list of memory blocks.
    - If the size of the process is less than or equal to the size of the current memory block:
        - Assign the process to the block.
        - Mark the block as occupied by reducing the block size.
        - Break the loop and move to the next process.
5. If no suitable block is found for a process, mark it as "Not Allocated."
6. Display the allocation results.

INPUT:

Enter number of memory blocks: 5

Enter sizes of blocks: 100 500 200 300 600

Enter number of processes: 4

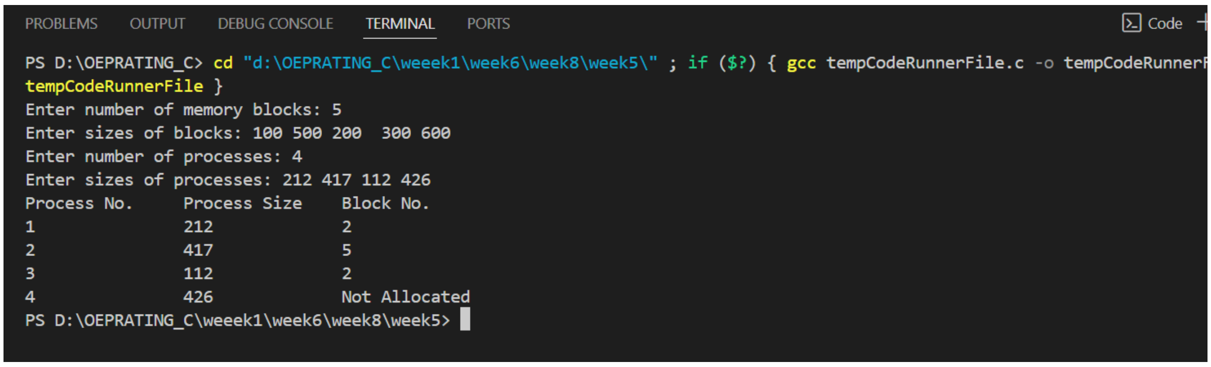
Enter sizes of processes: 212 417 112 426

PROGRAM

```
#include <stdio.h>

void firstFit(int blockSize[], int m, int processSize[], int n) {
  int allocation[n];
  for (int i = 0; i < n; i++) {
    allocation[i] = -1;
  }
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (blockSize[j] >= processSize[i]) {
        allocation[i] = j;
        blockSize[j] -= processSize[i];
        break;
      }
```

```
        } }
    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t", i + 1, processSize[i]);
        if (allocation[i] != -1) {
            printf("%d\n", allocation[i] + 1);
        } else {
            printf("Not Allocated\n");
        }
    }}

int main() {
    int m, n;
    printf("Enter number of memory blocks: ");
    scanf("%d", &m);
    int blockSize[m];
    printf("Enter sizes of blocks: ");
    for (int i = 0; i < m; i++) {
        scanf("%d", &blockSize[i]);}
    printf("Enter number of processes: ");
    scanf("%d", &n);
    int processSize[n];
    printf("Enter sizes of processes: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &processSize[i]);
    }
    firstFit(blockSize, m, processSize, n);
     return 0;
}
```

Output:



```
PS D:\OEPRATING_C> cd "d:\OEPRATING_C\weeek1\week6\week8\week5\" ; if ($?) { gcc tempCodeRunnerFile.c -o tempCodeRunnerF
tempCodeRunnerFile }
Enter number of memory blocks: 5
Enter sizes of blocks: 100 500 200  300 600
Enter number of processes: 4
Enter sizes of processes: 212 417 112 426
Process No.     Process Size    Block No.
1               212             2
2               417             5
3               112             2
4               426             Not Allocated
PS D:\OEPRATING_C\weeek1\week6\week8\week5>
```

Objective : 2. Write a program to implement deadlock detection algorithm.

Algorithm: Deadlock Detection

1. **Input**:
   - Number of processes and resources.
   - Maximum requirement (Max), allocation (Alloc), and total resources (Total).
2. **Calculate**:
   - Available = Total - ΣAlloc.
   - Need = Max - Alloc.
3. **Detect Deadlock**:
   - Check if a process can run with Available >= Need.
   - Allocate resources, update Available, and mark as finished.
   - Repeat until all processes finish or no progress can be made.
4. **Output**:
   - **No Deadlock** if all processes finish.
   - **Deadlock Detected** otherwise.

INPUT:

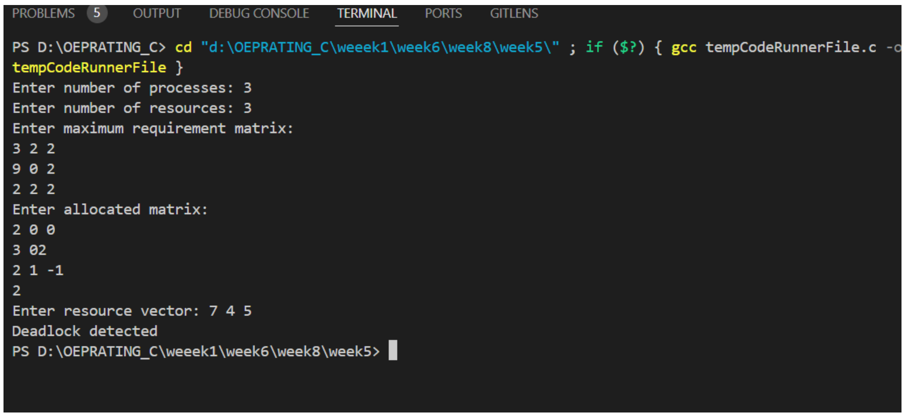
```
Enter number of processes: 3

Enter number of resources: 3

Enter maximum requirement matrix:

3 2 2

9 0 2

2 2 2

Enter allocated matrix:

2 0 0

3 0 2

2 1 1

Enter resource vector: 7 4 5
```

PROGRAM

```
#include <stdio.h>
#include <stdbool.h>
void calculateAvailable(int resources, int processes, int alloc[processes][resources], int
resourceVector[], int available[]) {
  for (int j = 0; j < resources; j++) {
    int sumAllocated = 0;
    for (int i = 0; i < processes; i++) {
            sumAllocated += alloc[i][j];
        }
        available[j] = resourceVector[j] - sumAllocated;
    }
}

bool canAllocate(int need[], int available[], int resources) {
    for (int j = 0; j < resources; j++) {
        if (need[j] > available[j]) {
            return false;
        }
    }
    return true;
}

bool detectDeadlock(int processes, int resources, int max[processes][resources], int
alloc[processes][resources], int available[]) {
    int need[processes][resources];
    bool finish[processes];
    for (int i = 0; i < processes; i++) {
        for (int j = 0; j < resources; j++) {
            need[i][j] = max[i][j] - alloc[i][j];
        }
        finish[i] = false;
    }
    int finishedCount = 0;
    while (finishedCount < processes) {
        bool allocationHappened = false;
        for (int i = 0; i < processes; i++) {
            if (!finish[i] && canAllocate(need[i], available, resources)) {
                for (int j = 0; j < resources; j++) {
                    available[j] += alloc[i][j];
                }
                finish[i] = true;
                finishedCount++;
                allocationHappened = true;
            }
        }
        if (!allocationHappened) {
            return true;
        }
    }
    return false;
}

int main() {
    int processes, resources;
    printf("Enter number of processes: ");
    scanf("%d", &processes);
    printf("Enter number of resources: ");
    scanf("%d", &resources);
```

```c
    int max[processes][resources], alloc[processes][resources], resourceVector[resources],
available[resources];

    printf("Enter maximum requirement matrix:\n");
    for (int i = 0; i < processes; i++) {
        for (int j = 0; j < resources; j++) {
            scanf("%d", &max[i][j]);
        }
    }

    printf("Enter allocated matrix:\n");
    for (int i = 0; i < processes; i++) {
        for (int j = 0; j < resources; j++) {
            scanf("%d", &alloc[i][j]);
        }
    }
    printf("Enter resource vector: ");
    for (int j = 0; j < resources; j++) {
        scanf("%d", &resourceVector[j]);
    }
    calculateAvailable(resources, processes, alloc, resourceVector, available);
    if (detectDeadlock(processes, resources, max, alloc, available)) {
        printf("Deadlock detected\n");
    } else {
        printf("No deadlock detected\n");
    }

    return 0;
}
```

Output:



```
PS D:\OEPRATING_C> cd "d:\OEPRATING_C\weeek1\week6\week8\week5\" ; if ($?) { gcc tempCodeRunnerFile.c -o
tempCodeRunnerFile }
Enter number of processes: 3
Enter number of resources: 3
Enter maximum requirement matrix:
3 2 2
9 0 2
2 2 2
Enter allocated matrix:
2 0 0
3 02
2 1 -1
2
Enter resource vector: 7 4 5
Deadlock detected
PS D:\OEPRATING_C\weeek1\week6\week8\week5> 
```

Week No-6 Practical Questions

Objective: This lab focuses on process synchronization where multiple processes may need to share resources or data. To ensure data consistency, the synchronized execution of these cooperating processes is necessary.

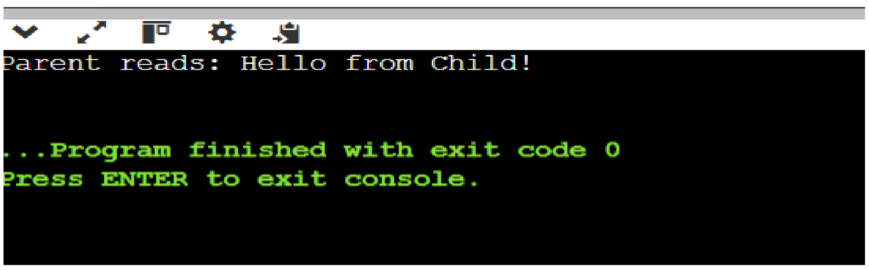
A. Task I: Parent-Child Process Communication.

a)Pipe Communication

PROGRAM

```
#include<stdio.h>
#include<unistd.h>
#include<string.h>
int main() {
  int fd[2];
  pid_t pid;
  char writeMsg[] = "Hello from Child!";
  char readMsg[50];
  if (pipe(fd) == -1) {
    perror("Pipe failed");
    return 1;
  }
  pid = fork();
  if (pid < 0) {
    perror("Fork failed");
    return 1;
  }
  if (pid > 0) {  // Parent Process
    close(fd[1]);  // Close writing end
    read(fd[0], readMsg, sizeof(readMsg));
    printf("Parent reads: %s\n", readMsg);
    close(fd[0]);
  } else {  // Child Process
    close(fd[0]);  // Close reading end
    write(fd[1], writeMsg, strlen(writeMsg) + 1);
    close(fd[1]);
  }
  return 0;
}
```

OUTPUT:



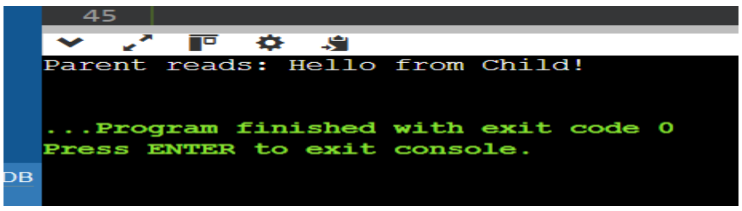
```
Parent reads: Hello from Child!


...Program finished with exit code 0
Press ENTER to exit console.
```

b) Message Passing Using `msgsnd` and `msgrcv`

PROGRAM

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <unistd.h>  // Include this for fork()
#include <sys/wait.h>  // For wait()
struct msg_buffer {
  long msg_type;
  char msg_text[100];
};

int main() {
  key_t key;
  int msgid;
  pid_t pid;
  struct msg_buffer message;
  // Generate a unique key
  key = ftok("progfile", 65);
  // Create a message queue and return its ID
  msgid = msgget(key, 0666 | IPC_CREAT);

  pid = fork();  // Fork the process
  if (pid > 0) {  // Parent Process
    // Wait for the child process to finish writing
    wait(NULL);
    // Parent reads the message from the queue
    msgrcv(msgid, &message, sizeof(message), 1, 0);
    printf("Parent reads: %s\n", message.msg_text);

    // Destroy the message queue
    msgctl(msgid, IPC_RMID, NULL);
  } else {  // Child Process
    // Prepare the message
    message.msg_type = 1;
    strcpy(message.msg_text, "Hello from Child!");

    // Send the message
    msgsnd(msgid, &message, sizeof(message), 0);
  }
  return 0;
}
```

OUTPUT:



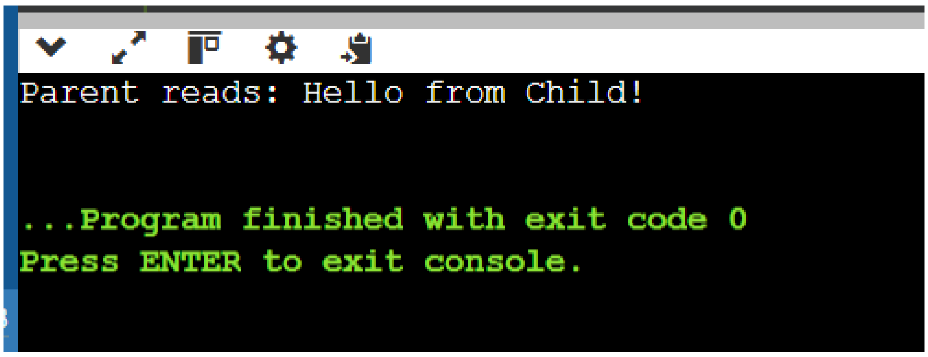
```
Parent reads: Hello from Child!

...Program finished with exit code 0
Press ENTER to exit console.
```

c). Shared Memory Communication

PROGRAM

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include <sys/msg.h>
#include <unistd.h>  // For fork()
#include <sys/wait.h>  // For wait()

int main() {
  // Generate a unique key
  key_t key = ftok("shmfile", 65);

  // Get the shared memory ID
  int shmid = shmget(key, 1024, 0666 | IPC_CREAT);

  // Attach to shared memory
  char *str = (char *) shmat(shmid, (void *)0, 0);  // Corrected: char *str instead of char str

  pid_t pid;
  pid = fork();

  if (pid > 0) {  // Parent Process
    wait(NULL);  // Wait for the child process to finish
    printf("Parent reads: %s\n", str);  // Read the string from shared memory
    shmdt(str);  // Detach from shared memory
    shmctl(shmid, IPC_RMID, NULL);  // Destroy the shared memory
  } else {  // Child Process
    strcpy(str, "Hello from Child!");  // Write a string to shared memory
    shmdt(str);  // Detach from shared memory
  }

  return 0;
}
```

OUTPUT:



```
Parent reads: Hello from Child!


...Program finished with exit code 0
Press ENTER to exit console.
```

Week No-8 Practical Questions

Objective : Write a C program for First Fit Partition Allocation Method

Algorithm: First Fit Partition Allocation Method.

```
1: Input memory blocks with their sizes and processes with their sizes.
2: Initialize all memory blocks as free.
 3: for each process in the list of processes do
 4: for each memory block in the list of memory blocks do
 5: if size of process ≤ size of block then
6: Assign the process to the current block.
 7: Mark the block as occupied.
8: Break the inner loop to check the next process.
 9: end if
10: end for
```

INPUT:

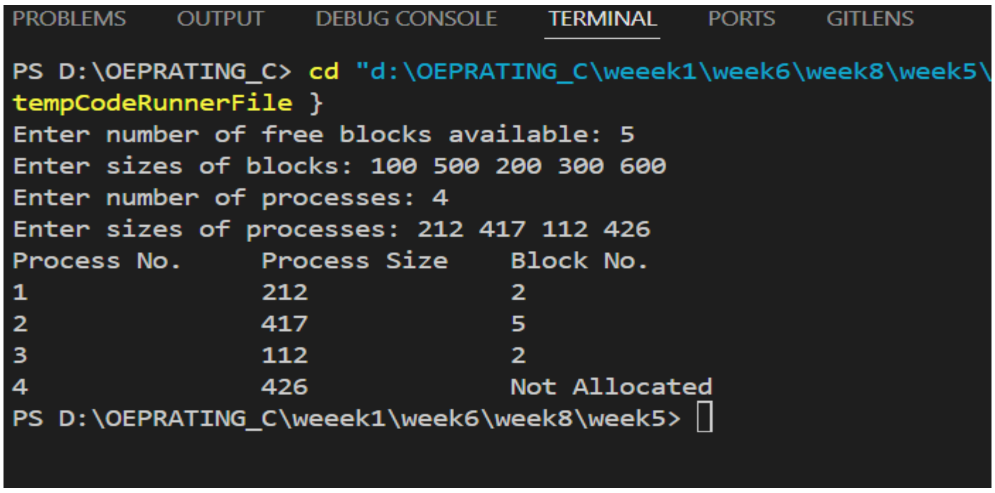
```
Enter number of memory blocks: 5

Enter sizes of blocks: 100 500 200 300 600

Enter number of processes: 4

Enter sizes of processes: 212 417 112 426
```

PROGRAM

```c
#include <stdio.h>
void firstFit(int blockSize[], int m, int processSize[], int n) {
  int allocation[n];
  for (int i = 0; i < n; i++) {
    allocation[i] = -1;
  }
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (blockSize[j] >= processSize[i]) {
        allocation[i] = j;
        blockSize[j] -= processSize[i];
        break;
      }
    }}
  printf("Process No.\tProcess Size\tBlock No.\n");
  for (int i = 0; i < n; i++) {
    printf("%d\t\t%d\t\t", i + 1, processSize[i]);
    if (allocation[i] != -1) {
      printf("%d\n", allocation[i] + 1);
    } else {
      printf("Not Allocated\n");
    }
  }}
```

```
int main() {
  int m, n;
  printf("Enter number of free blocks available: ");
  scanf("%d", &m);

  int blockSize[m];
  printf("Enter sizes of blocks: ");
  for (int i = 0; i < m; i++) {
    scanf("%d", &blockSize[i]);
  }

  printf("Enter number of processes: ");
  scanf("%d", &n);

  int processSize[n];
  printf("Enter sizes of processes: ");
  for (int i = 0; i < n; i++) {
    scanf("%d", &processSize[i]);
  }
  firstFit(blockSize, m, processSize, n);

  return 0;
}
```

Output:



```
PS D:\OEPRATING_C> cd "d:\OEPRATING_C\weeek1\week6\week8\week5\
tempCodeRunnerFile }
Enter number of free blocks available: 5
Enter sizes of blocks: 100 500 200 300 600
Enter number of processes: 4
Enter sizes of processes: 212 417 112 426
Process No.     Process Size    Block No.
1               212             2
2               417             5
3               112             2
4               426             Not Allocated
PS D:\OEPRATING_C\weeek1\week6\week8\week5>
```

Objective : Write a C program for Best Fit Partition Allocation Method

Algorithm: Best Fit Partition Allocation Method.

```
1. Input memory block sizes and process sizes.
2.Initialize all memory blocks as free.
3.For each process:
    • Find the first block that fits the process.
    • Assign the process to the block.
    • Mark the block as occupied.
4. Repeat for all processes.
5.Output allocation or "Not Allocated" for unfit processes.
```

INPUT:

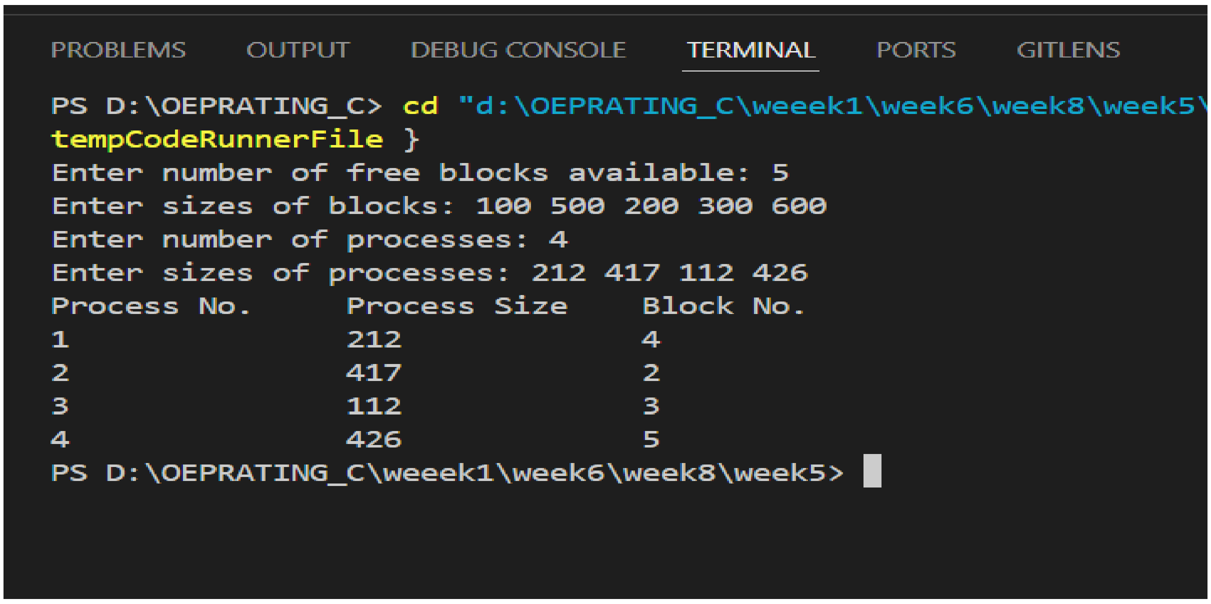
```
Enter number of memory blocks: 5

Enter sizes of blocks: 100 500 200 300 600

Enter number of processes: 4

Enter sizes of processes: 212 417 112 426
```

PROGRAM

```
#include <stdio.h>

void bestFit(int blockSize[], int m, int processSize[], int n) {
  int allocation[n];

  for (int i = 0; i < n; i++) {
    allocation[i] = -1;
  }

  for (int i = 0; i < n; i++) {
    int bestIndex = -1;

    for (int j = 0; j < m; j++) {
      if (blockSize[j] >= processSize[i]) {
        if (bestIndex == -1 || blockSize[j] < blockSize[bestIndex]) {
          bestIndex = j;
        }
      }
    }

    if (bestIndex != -1) {
      allocation[i] = bestIndex;
      blockSize[bestIndex] -= processSize[i];
    }
  }

  printf("Process No.\tProcess Size\tBlock No.\n");
  for (int i = 0; i < n; i++) {
    printf("%d\t\t%d\t\t", i + 1, processSize[i]);
    if (allocation[i] != -1) {
      printf("%d\n", allocation[i] + 1);
```

```
        } else {
            printf("Not Allocated\n");
        }
    }
}

int main() {
    int m, n;

    printf("Enter number of free blocks available: ");
    scanf("%d", &m);

    int blockSize[m];
    printf("Enter sizes of blocks: ");
    for (int i = 0; i < m; i++) {
        scanf("%d", &blockSize[i]);
    }

    printf("Enter number of processes: ");
    scanf("%d", &n);

    int processSize[n];
    printf("Enter sizes of processes: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &processSize[i]);
    }

    bestFit(blockSize, m, processSize, n);

    return 0;
}
```

OUTPUT:



```
PS D:\OEPRATING_C> cd "d:\OEPRATING_C\weeek1\week6\week8\week5\
tempCodeRunnerFile }
Enter number of free blocks available: 5
Enter sizes of blocks: 100 500 200 300 600
Enter number of processes: 4
Enter sizes of processes: 212 417 112 426
Process No.     Process Size    Block No.
1               212             4
2               417             2
3               112             3
4               426             5
PS D:\OEPRATING_C\weeek1\week6\week8\week5>
```

Objective : Write a C program for Worst Fit Partition Allocation Method

Algorithm: Worst Fit Partition Allocation Method.

1. Input memory block sizes and process sizes.
2. Initialize all memory blocks as free.
3. For each process:
   - Find the first block that fits the process.
   - Assign the process to the block.
   - Mark the block as occupied.
4. Repeat for all processes.
5. Output allocation or "Not Allocated" for unfit processes.

INPUT:

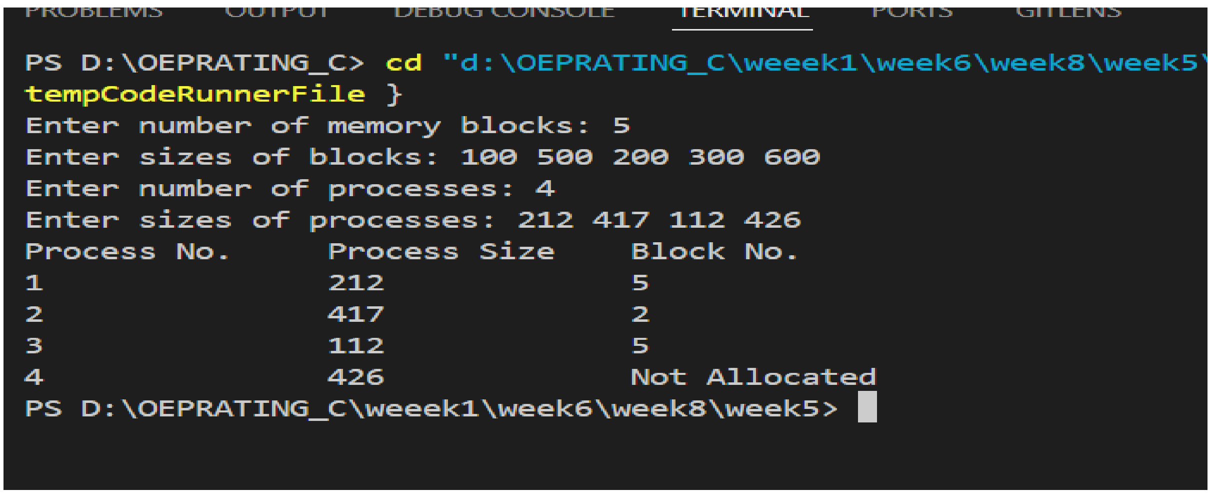
```
Enter number of memory blocks: 5

Enter sizes of blocks: 100 500 200 300 600

Enter number of processes: 4

Enter sizes of processes: 212 417 112 426
```

PROGRAM

```c
#include <stdio.h>

void worstFit(int blockSize[], int m, int processSize[], int n) {
  int allocation[n];

  for (int i = 0; i < n; i++) {
    allocation[i] = -1;
  }

  for (int i = 0; i < n; i++) {
    int worstIndex = -1;

    for (int j = 0; j < m; j++) {
      if (blockSize[j] >= processSize[i]) {
        if (worstIndex == -1 || blockSize[j] > blockSize[worstIndex]) {
          worstIndex = j;
        }
      }
    }

    if (worstIndex != -1) {
      allocation[i] = worstIndex;
      blockSize[worstIndex] -= processSize[i];
    }
  }

  printf("Process No.\tProcess Size\tBlock No.\n");
  for (int i = 0; i < n; i++) {
    printf("%d\t\t%d\t\t", i + 1, processSize[i]);
    if (allocation[i] != -1) {
      printf("%d\n", allocation[i] + 1);
```

```
        } else {
            printf("Not Allocated\n");
        }
    }
}

int main() {
    int m, n;

    printf("Enter number of memory blocks: ");
    scanf("%d", &m);

    int blockSize[m];
    printf("Enter sizes of blocks: ");
    for (int i = 0; i < m; i++) {
        scanf("%d", &blockSize[i]);
    }

    printf("Enter number of processes: ");
    scanf("%d", &n);

    int processSize[n];
    printf("Enter sizes of processes: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &processSize[i]);
    }

    worstFit(blockSize, m, processSize, n);

    return 0;
}
```

OUTPUT:



```
PS D:\OEPRATING_C> cd "d:\OEPRATING_C\weeek1\week6\week8\week5\
tempCodeRunnerFile }
Enter number of memory blocks: 5
Enter sizes of blocks: 100 500 200 300 600
Enter number of processes: 4
Enter sizes of processes: 212 417 112 426
Process No.     Process Size    Block No.
1               212             5
2               417             2
3               112             5
4               426             Not Allocated
PS D:\OEPRATING_C\weeek1\week6\week8\week5>
```