Experiment No. 8
Memory Management a. Write a program to demonstrate the concept of dynamic partitioning placement algorithms i.e. Best Fit, First Fit, Worst-Fit
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: To study and implement memory allocation strategy First fit.

Objective:

a. Write a program to demonstrate the concept of dynamic partitioning placement algorithms i.e. Best Fit, First Fit, Worst-Fit etc.

Theory:

The primary role of the memory management system is to satisfy requests for memory allocation. Sometimes this is implicit, as when a new process is created. At other times, processes explicitly request memory. Either way, the system must locate enough unallocated memory and assign it to the process.

Partitioning: The simplest methods of allocating memory are based on dividing memory into areas with fixed partitions.

Selection Policies: If more than one free block can satisfy a request, then which one should we pick? There are several schemes that are frequently studied and are commonly used.

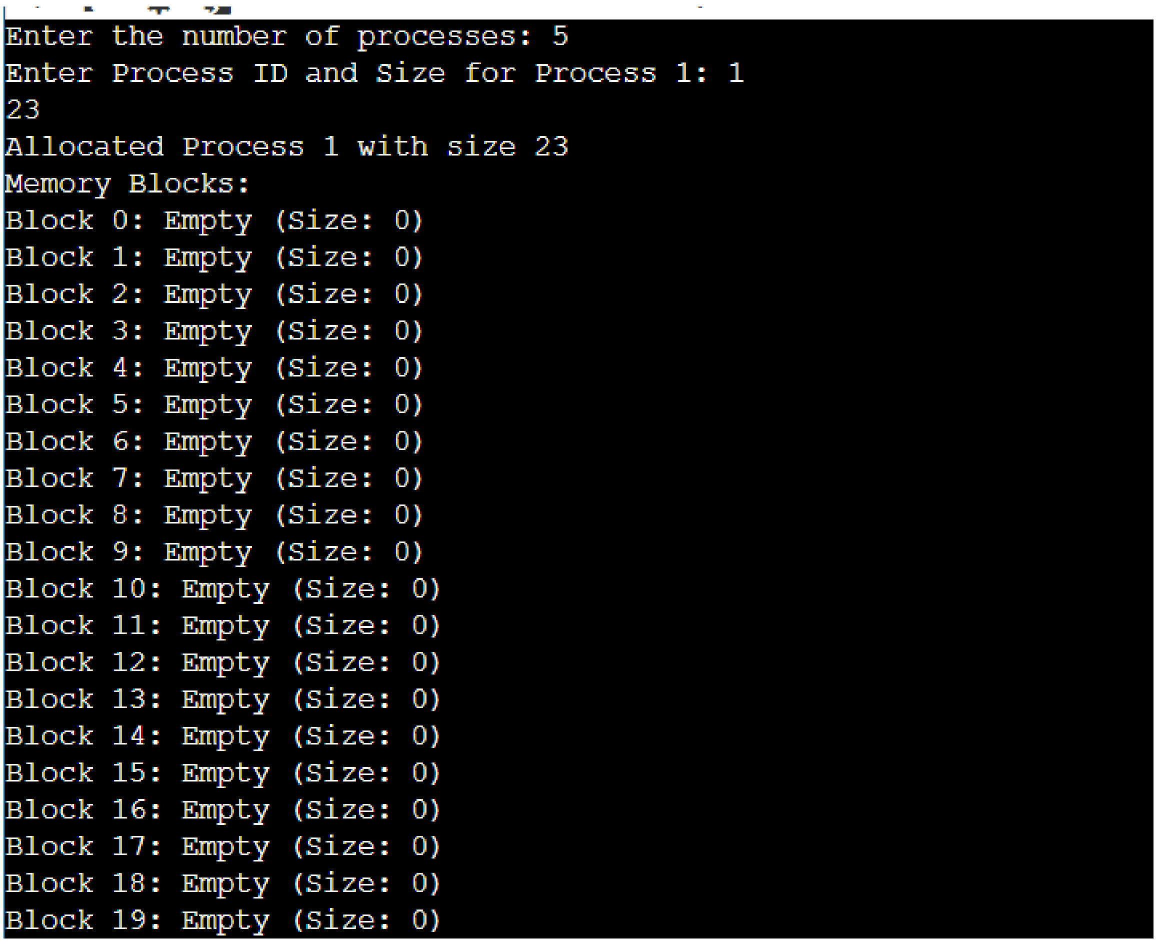
First Fit: In the first fit approach is to allocate the first free partition or hole large enough which can accommodate the process. It finishes after finding the first suitable free partition.

- **Advantage:** Fastest algorithm because it searches as little as possible.
- **Disadvantage:** The remaining unused memory areas left after allocation become waste if it is too smaller. Thus request for larger memory requirement cannot be accomplished
- Best Fit: The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate. It then tries to find a hole which is close to actual process size needed.
- Worst fit: In worst fit approach is to locate largest available free portion so that the portion left will be big enough to be useful. It is the reverse of best fit.

Next Fit: If we want to spread the allocations out more evenly across the memory space, we often use a policy called next fit. This scheme is very similar to the first fit approach, except for the place where the search starts.

Program:

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

#define MEMORY_SIZE 100

// Memory block structure
typedef struct MemoryBlock {
    int id; // Process ID occupying the block, -1 if empty
    int size; // Size of the block
    bool allocated; // Flag to indicate if the block is allocated
} MemoryBlock;

// Function to initialize memory blocks
void initializeMemory(MemoryBlock memory[], int size) {
    for (int i = 0; i < size; i++) {
        memory[i].id = -1;
        memory[i].size = 0;
        memory[i].allocated = false;
    }
}

// Function to print memory blocks
void printMemory(MemoryBlock memory[], int size) {
    printf("Memory Blocks:\n");
    for (int i = 0; i < size; i++) {
        if (memory[i].allocated) {
            printf("Block %d: Process %d (Size: %d)\n", i, memory[i].id, memory[i].size);
        } else {
            printf("Block %d: Empty (Size: %d)\n", i, memory[i].size);
        }
    }
    printf("\n");
}

// Function to allocate memory using First Fit algorithm
void firstFit(MemoryBlock memory[], int size, int process_id, int process_size) {
    for (int i = 0; i < size; i++) {
        if (!memory[i].allocated && memory[i].size >= process_size) {
            memory[i].id = process_id;
            memory[i].allocated = true;
            break;
        }
    }
}

// Function to allocate memory using Best Fit algorithm
void bestFit(MemoryBlock memory[], int size, int process_id, int process_size) {
    int best_fit_index = -1;
    int min_remaining_size = MEMORY_SIZE + 1;

    for (int i = 0; i < size; i++) {
        if (!memory[i].allocated && memory[i].size >= process_size) {
            int remaining_size = memory[i].size - process_size;
            if (remaining_size < min_remaining_size) {
                min_remaining_size = remaining_size;
                best_fit_index = i;
            }
        }
    }

    if (best_fit_index != -1) {
        memory[best_fit_index].id = process_id;
        memory[best_fit_index].allocated = true;
    }
}

// Function to allocate memory using Worst Fit algorithm
void worstFit(MemoryBlock memory[], int size, int process_id, int process_size) {
    int worst_fit_index = -1;
    int max_remaining_size = -1;

    for (int i = 0; i < size; i++) {
        if (!memory[i].allocated && memory[i].size >= process_size) {
            int remaining_size = memory[i].size - process_size;
            if (remaining_size > max_remaining_size) {
                max_remaining_size = remaining_size;
                worst_fit_index = i;
            }
        }
    }

    if (worst_fit_index != -1) {
        memory[worst_fit_index].id = process_id;
        memory[worst_fit_index].allocated = true;
    }
}

int main() {
    MemoryBlock memory[MEMORY_SIZE];
    int num_processes;

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

    initializeMemory(memory, MEMORY_SIZE);

    for (int i = 0; i < num_processes; i++) {
        int process_id, process_size;
        printf("Enter Process ID and Size for Process %d: ", i + 1);
        scanf("%d %d", &process_id, &process_size);

        // Choose allocation algorithm
        // Uncomment the desired algorithm
        // firstFit(memory, MEMORY_SIZE, process_id, process_size);
        // bestFit(memory, MEMORY_SIZE, process_id, process_size);
        worstFit(memory, MEMORY_SIZE, process_id, process_size);

        printf("Allocated Process %d with size %d\n", process_id, process_size);
        printMemory(memory, MEMORY_SIZE);
    }

    return 0;
}
```
Output:



```
Enter the number of processes: 5
Enter Process ID and Size for Process 1: 1
23
Allocated Process 1 with size 23
Memory Blocks:
Block 0: Empty (Size: 0)
Block 1: Empty (Size: 0)
Block 2: Empty (Size: 0)
Block 3: Empty (Size: 0)
Block 4: Empty (Size: 0)
Block 5: Empty (Size: 0)
Block 6: Empty (Size: 0)
Block 7: Empty (Size: 0)
Block 8: Empty (Size: 0)
Block 9: Empty (Size: 0)
Block 10: Empty (Size: 0)
Block 11: Empty (Size: 0)
Block 12: Empty (Size: 0)
Block 13: Empty (Size: 0)
Block 14: Empty (Size: 0)
Block 15: Empty (Size: 0)
Block 16: Empty (Size: 0)
Block 17: Empty (Size: 0)
Block 18: Empty (Size: 0)
Block 19: Empty (Size: 0)
```

Conclusion:

Why do we need memory allocation strategies?

Memory allocation strategies are essential for efficient utilization of system resources and effective management of memory. They ensure that processes are allocated memory in a manner that maximizes system performance while minimizing fragmentation and waste. Without these strategies, the system may suffer from inefficient memory usage, leading to degraded system performance and potentially causing issues such as memory exhaustion or fragmentation. Therefore, memory allocation strategies play a crucial role in optimizing system resource utilization and enhancing overall system stability and performance.