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.

Code:

```
#include <stdio.h>

#define MEMORY_SIZE 1000
#define NUM_JOBS 5

void best_fit() {
   int memory[MEMORY_SIZE] = {0}; // Represents memory blocks, 0 indicates
free, 1 indicates occupied
   int jobs[NUM_JOBS] = {150, 300, 125, 200, 175}; // Jobs with their sizes
   int num_jobs_allocated = 0;

   printf("Best Fit Placement Algorithm:\n");
   for (int i = 0; i < NUM_JOBS; i++) {
      int job_size = jobs[i];
      int best_fit_index = -1;
      int min_fragmentation = MEMORY_SIZE;

      for (int j = 0; j < MEMORY_SIZE; j++) {
         if (memory[j] == 0) { // If memory block is free
            int fragmentation = 0;
            for (int k = j + 1; k < j + job_size; k++) {
               if (memory[k] == 1) {
                  fragmentation++;
```

```
                }
            }
            if (fragmentation < min_fragmentation && j + job_size <=
MEMORY_SIZE) {
                best_fit_index = j;
                min_fragmentation = fragmentation;
            }
        }
    }

    if (best_fit_index != -1) { // If job can be allocated
        for (int j = best_fit_index; j < best_fit_index + job_size; j++) {
            memory[j] = 1; // Allocate the job
        }
        printf("Job %d of size %d allocated at memory location %d\n", i + 1,
job_size, best_fit_index);
        num_jobs_allocated++;
    } else {
        printf("Unable to allocate Job %d of size %d\n", i + 1, job_size);
    }
  }

  printf("Total number of jobs allocated: %d\n", num_jobs_allocated);
}

void first_fit() {
  int memory[MEMORY_SIZE] = {0}; // Represents memory blocks, 0 indicates
free, 1 indicates occupied
  int jobs[NUM_JOBS] = {150, 300, 125, 200, 175}; // Jobs with their sizes
  int num_jobs_allocated = 0;

  printf("\nFirst Fit Placement Algorithm:\n");
  for (int i = 0; i < NUM_JOBS; i++) {
```

```
    int job_size = jobs[i];
    int allocated = 0;

    for (int j = 0; j < MEMORY_SIZE; j++) {
      if (memory[j] == 0) { // If memory block is free
        int free_space = 1;
        for (int k = j + 1; k < j + job_size; k++) {
          if (memory[k] != 0) { // Check if continuous space is available
            free_space = 0;
            break;
          }
        }
        if (free_space && j + job_size <= MEMORY_SIZE) {
          for (int k = j; k < j + job_size; k++) {
            memory[k] = 1; // Allocate the job
          }
          printf("Job %d of size %d allocated at memory location %d\n", i + 1,
job_size, j);
          num_jobs_allocated++;
          allocated = 1;
          break;
        }
      }
    }

    if (!allocated) {
      printf("Unable to allocate Job %d of size %d\n", i + 1, job_size);
    }
  }

  printf("Total number of jobs allocated: %d\n", num_jobs_allocated);
}
```

```
void worst_fit() {
   int memory[MEMORY_SIZE] = {0}; // Represents memory blocks, 0 indicates
free, 1 indicates occupied
   int jobs[NUM_JOBS] = {150, 300, 125, 200, 175}; // Jobs with their sizes
   int num_jobs_allocated = 0;

   printf("\nWorst Fit Placement Algorithm:\n");
   for (int i = 0; i < NUM_JOBS; i++) {
      int job_size = jobs[i];
      int worst_fit_index = -1;
      int max_fragmentation = -1;

      for (int j = 0; j < MEMORY_SIZE; j++) {
         if (memory[j] == 0) { // If memory block is free
            int fragmentation = 0;
            for (int k = j + 1; k < j + job_size; k++) {
               if (memory[k] == 1) {
                  fragmentation++;
               }
            }
            if (fragmentation > max_fragmentation && j + job_size <=
MEMORY_SIZE) {
               worst_fit_index = j;
               max_fragmentation = fragmentation;
            }
         }
      }

      if (worst_fit_index != -1) { // If job can be allocated
         for (int j = worst_fit_index; j < worst_fit_index + job_size; j++) {
            memory[j] = 1; // Allocate the job
         }
```

```
            printf("Job %d of size %d allocated at memory location %d\n", i + 1,
job_size, worst_fit_index);
            num_jobs_allocated++;
        } else {
            printf("Unable to allocate Job %d of size %d\n", i + 1, job_size);
        }
    }

    printf("Total number of jobs allocated: %d\n", num_jobs_allocated);
}

int main() {
    best_fit();
    first_fit();
    worst_fit();

    return 0;
}
```

Output:

```
Best Fit Placement Algorithm:
Job 1 of size 150 allocated at memory location 0
Job 2 of size 300 allocated at memory location 150
Job 3 of size 125 allocated at memory location 450
Job 4 of size 200 allocated at memory location 575
Job 5 of size 175 allocated at memory location 775
Total number of jobs allocated: 5

First Fit Placement Algorithm:
Job 1 of size 150 allocated at memory location 0
Job 2 of size 300 allocated at memory location 150
Job 3 of size 125 allocated at memory location 450
Job 4 of size 200 allocated at memory location 575
Job 5 of size 175 allocated at memory location 775
Total number of jobs allocated: 5

Worst Fit Placement Algorithm:
Job 1 of size 150 allocated at memory location 0
Job 2 of size 300 allocated at memory location 150
Job 3 of size 125 allocated at memory location 450
Job 4 of size 200 allocated at memory location 575
Job 5 of size 175 allocated at memory location 775
Total number of jobs allocated: 5
PS C:\Users\Lenovo\Downloads\AOA FINAL FOLDER>
```

Conclusion:

In conclusion, the study and implementation of the First Fit memory allocation strategy offer valuable insights into memory management in computing systems. Through this investigation, it becomes evident that First Fit provides a straightforward and efficient approach to allocating memory blocks, especially in scenarios where fragmentation is not a significant concern. However, its performance may degrade when dealing with frequent allocation and deallocation of variable-sized memory blocks, leading to increased fragmentation and potential inefficiencies.

Why do we need memory allocation strategies?

Memory allocation strategies are essential because they govern how a computer's memory is utilized and managed. Here's why they're necessary:

Optimal resource utilization: Memory allocation strategies help ensure that available memory resources are utilized efficiently. By allocating memory appropriately, programs can make the most of the system's resources without wasting memory or causing unnecessary overhead.

Preventing fragmentation: Memory fragmentation occurs when memory is allocated and deallocated in a way that leaves small gaps of unused memory scattered throughout the system. Over time, this fragmentation can lead to inefficient memory usage and hinder performance. Memory allocation strategies help mitigate fragmentation by intelligently managing memory allocation and deallocation.

Performance optimization: Different memory allocation strategies can impact the performance of a program. For example, some strategies may prioritize fast allocation and deallocation times, while others may focus on minimizing memory overhead. By choosing the right allocation strategy for a particular application or scenario, developers can optimize performance and enhance overall system efficiency.

Avoiding memory leaks: Improper memory allocation and deallocation can result in memory leaks, where memory that is no longer needed by a program is not properly released. Over time, memory leaks can consume significant amounts of memory and degrade system performance. Memory allocation strategies help developers implement robust memory management techniques to minimize the risk of memory leaks.

Supporting different application requirements: Different types of applications have varying memory requirements and usage patterns. Memory allocation strategies allow developers to tailor memory management techniques to suit the specific needs of their applications. For example, real-time systems may require deterministic memory allocation strategies to ensure timely response, while other applications may prioritize flexibility and scalability.