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
Name Of Student:-Bhagyashri Kaleni Sutar
Roll No:-75
Date of Performance:
Date of Submission:
Marks:
Sign:

Experiment No. 8

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

Objective:

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>

typedef struct MemoryBlock {
   int start_address;
   int size;
   int allocated;
   int process_id;
   struct MemoryBlock* next;
} MemoryBlock;

MemoryBlock* head = NULL;

void insertMemoryBlock(int start_address, int size) {
   MemoryBlock* newBlock = (MemoryBlock*)malloc(sizeof(MemoryBlock));
   newBlock->start_address = start_address;
   newBlock->size = size;
   newBlock->allocated = 0;
   newBlock->process_id = -1;
   newBlock->next = NULL;

   if (head == NULL) {
      head = newBlock;
   } else {
      MemoryBlock* temp = head;
      while (temp->next != NULL) {
         temp = temp->next;
      }
      temp->next = newBlock;
   }
```

```
}

void displayMemory() {
    MemoryBlock* temp = head;
    printf("Memory Blocks:\n");
    while (temp != NULL) {
        printf("[Start: %d, Size: %d, Allocated: %d, Process: %d]\n",
               temp->start_address, temp->size, temp->allocated, temp->process_id);
        temp = temp->next;
    }
    printf("\n");
}

int firstFit(int process_size, int process_id) {
    MemoryBlock* temp = head;
    while (temp != NULL) {
        if (!temp->allocated && temp->size >= process_size) {
            temp->allocated = 1;
            temp->process_id = process_id;
            if (temp->size > process_size) {
                MemoryBlock* newBlock = (MemoryBlock*)malloc(sizeof(MemoryBlock));
                newBlock->start_address = temp->start_address + process_size;
                newBlock->size = temp->size - process_size;
                newBlock->allocated = 0;
                newBlock->process_id = -1;
                newBlock->next = temp->next;
                temp->size = process_size;
                temp->next = newBlock;
            }
            return 1; // Success
        }
        temp = temp->next;
```

```
    }
    return 0; // Failure
}

int bestFit(int process_size, int process_id) {
    MemoryBlock* temp = head;
    MemoryBlock* bestBlock = NULL;
    while (temp != NULL) {
        if (!temp->allocated && temp->size >= process_size) {
            if (bestBlock == NULL || temp->size < bestBlock->size) {
                bestBlock = temp;
            }
        }
        temp = temp->next;
    }

    if (bestBlock != NULL) {
        bestBlock->allocated = 1;
        bestBlock->process_id = process_id;
        if (bestBlock->size > process_size) {
            MemoryBlock* newBlock = (MemoryBlock*)malloc(sizeof(MemoryBlock));
            newBlock->start_address = bestBlock->start_address + process_size;
            newBlock->size = bestBlock->size - process_size;
            newBlock->allocated = 0;
            newBlock->process_id = -1;
            newBlock->next = bestBlock->next;
            bestBlock->size = process_size;
            bestBlock->next = newBlock;
        }
        return 1;
    }
    return 0;
```

```
}

int worstFit(int process_size, int process_id) {
    MemoryBlock* temp = head;
    MemoryBlock* worstBlock = NULL;
    while (temp != NULL) {
        if (!temp->allocated && temp->size >= process_size) {
            if (worstBlock == NULL || temp->size > worstBlock->size) {
                worstBlock = temp;
            }
        }
        temp = temp->next;
    }

    if (worstBlock != NULL) {
        worstBlock->allocated = 1;
        worstBlock->process_id = process_id;
        if (worstBlock->size > process_size) {
            MemoryBlock* newBlock = (MemoryBlock*)malloc(sizeof(MemoryBlock));
            newBlock->start_address = worstBlock->start_address + process_size;
            newBlock->size = worstBlock->size - process_size;
            newBlock->allocated = 0;
            newBlock->process_id = -1;
            newBlock->next = worstBlock->next;
            worstBlock->size = process_size;
            worstBlock->next = newBlock;
        }
        return 1;
    }
    return 0;
}
```

```
int main() {
    insertMemoryBlock(0, 100);
    insertMemoryBlock(100, 50);
    insertMemoryBlock(150, 200);
    insertMemoryBlock(350, 75);

    int processes[][2] = {{30, 1}, {60, 2}, {100, 3}, {35, 4}};
    int numProcesses = sizeof(processes) / sizeof(processes[0]);

    printf("Initial Memory:\n");
    displayMemory();

    printf("\nFirstFit Allocation:\n");
    MemoryBlock* originalHead = head; //save the first head.

    for (int i = 0; i < numProcesses; i++) {
        if (firstFit(processes[i][0], processes[i][1])) {
            printf("Allocated %d for Process %d\n", processes[i][0], processes[i][1]);
        } else {
            printf("Failed to allocate %d for Process %d\n", processes[i][0], processes[i][1]);
        }
    }
    displayMemory();
    head = originalHead; //reset head for next algorithm.

    printf("\nBestFit Allocation:\n");
    for (int i = 0; i < numProcesses; i++) {
        if (bestFit(processes[i][0], processes[i][1])) {
            printf("Allocated %d for Process %d\n", processes[i][0], processes[i][1]);
        } else {
            printf("Failed to allocate %d for Process %d\n", processes[i][0], processes[i][1]);
        }
```

```
    }
    displayMemory();
    head = originalHead; //reset head for next algorithm.

    printf("\nWorstFit Allocation:\n");
    for (int i = 0; i < numProcesses; i++) {
        if (worstFit(processes[i][0], processes[i][1])) {
            printf("Allocated %d for Process %d\n", processes[i][0], processes[i][1]);
        } else {
            printf("Failed to allocate %d for Process %d\n", processes[i][0], processes[i][1]);
        }
    }
    displayMemory();

    // Free allocated memory (important!)
    MemoryBlock* current = originalHead;
    while (current != NULL) {
        MemoryBlock* next = current->next;
        free(current);
        current = next;
    }

    return 0;
}

//Output
Initial Memory:
Memory Blocks:
[Start: 0, Size: 100, Allocated: 0, Process: -1]
[Start: 100, Size: 50, Allocated: 0, Process: -1]
[Start: 150, Size: 200, Allocated: 0, Process: -1]
[Start: 350, Size: 75, Allocated: 0, Process: -1]
```

FirstFit Allocation:

Allocated 30 for Process 1

Allocated 60 for Process 2

Allocated 100 for Process 3

Allocated 35 for Process 4

Memory Blocks:

[Start: 0, Size: 30, Allocated: 1, Process: 1]

[Start: 30, Size: 60, Allocated: 1, Process: 2]

[Start: 90, Size: 10, Allocated: 0, Process: -1]

[Start: 100, Size: 35, Allocated: 1, Process: 4]

[Start: 135, Size: 15, Allocated: 0, Process: -1]

[Start: 150, Size: 100, Allocated: 1, Process: 3]

[Start: 250, Size: 100, Allocated: 0, Process: -1]

[Start: 350, Size: 75, Allocated: 0, Process: -1]

BestFit Allocation:

Allocated 30 for Process 1

Allocated 60 for Process 2

Failed to allocate 100 for Process 3

Allocated 35 for Process 4

Memory Blocks:

[Start: 0, Size: 30, Allocated: 1, Process: 1]

[Start: 30, Size: 60, Allocated: 1, Process: 2]

[Start: 90, Size: 10, Allocated: 0, Process: -1]

[Start: 100, Size: 35, Allocated: 1, Process: 4]

[Start: 135, Size: 15, Allocated: 0, Process: -1]

[Start: 150, Size: 100, Allocated: 1, Process: 3]

[Start: 250, Size: 60, Allocated: 1, Process: 2]

[Start: 310, Size: 35, Allocated: 1, Process: 4]

[Start: 345, Size: 5, Allocated: 0, Process: -1]

[Start: 350, Size: 30, Allocated: 1, Process: 1]

[Start: 380, Size: 45, Allocated: 0, Process: -1]

WorstFit Allocation:

Allocated 30 for Process 1

Failed to allocate 60 for Process 2

Failed to allocate 100 for Process 3

Failed to allocate 35 for Process 4

Memory Blocks:

[Start: 0, Size: 30, Allocated: 1, Process: 1]

[Start: 30, Size: 60, Allocated: 1, Process: 2]

[Start: 90, Size: 10, Allocated: 0, Process: -1]

[Start: 100, Size: 35, Allocated: 1, Process: 4]

[Start: 135, Size: 15, Allocated: 0, Process: -1]

[Start: 150, Size: 100, Allocated: 1, Process: 3]

[Start: 250, Size: 60, Allocated: 1, Process: 2]

[Start: 310, Size: 35, Allocated: 1, Process: 4]

[Start: 345, Size: 5, Allocated: 0, Process: -1]

[Start: 350, Size: 30, Allocated: 1, Process: 1]

[Start: 380, Size: 30, Allocated: 1, Process: 1]

[Start: 410, Size: 15, Allocated: 0, Process: -1]

Conclusion:

Why do we need memory allocation strategies?

Ans:- Memory allocation strategies are fundamental to efficient computer system operation, particularly in managing the limited resource of memory (RAM). Here's a breakdown of why they're essential:

- **Efficient Resource Management:**

- Memory is a finite resource. Without proper allocation, programs could overwrite each other's data or consume all available memory, leading to system crashes.
-
- Allocation strategies ensure that memory is distributed fairly and effectively among running processes.
-

- **Preventing Memory Leaks:**
  - Memory leaks occur when programs allocate memory but fail to release it when it's no longer needed.
  -
  - Allocation strategies, especially those involving dynamic allocation, help prevent leaks by providing mechanisms for tracking and reclaiming unused memory.
  -
- **Reducing Fragmentation:**
  - Memory fragmentation happens when free memory becomes scattered in small, non-contiguous blocks, making it difficult to allocate larger chunks.
  -
  - Allocation strategies aim to minimize fragmentation by strategically placing data in memory.
  -
- **Supporting Dynamic Memory Allocation:**
  - Many programs require the ability to dynamically allocate memory during runtime, as their memory needs may vary.
  - Allocation strategies provide the necessary mechanisms for requesting and releasing memory on demand.
- **Enhancing System Performance:**
  - Efficient memory allocation can significantly impact system performance. By minimizing fragmentation and reducing the overhead of memory management, allocation strategies contribute to faster program execution and a more responsive user experience.

- **Enabling Multitasking:**
  - Modern operating systems support multitasking, which involves running multiple programs concurrently.
  - Memory allocation strategies are crucial for isolating processes and preventing them from interfering with each other's memory.
- **Safety and Stability:**
  - Proper memory allocation prevents one program from accessing memory that belongs to another program. This increases system stability, and security.

In summary, memory allocation strategies are essential for optimizing memory usage, preventing errors, and ensuring the smooth and reliable operation of computer systems.