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:

First Fit

```
#include <stdio.h>

void implimentFirstFit(int blockSize[], int blocks, int processSize[], int processes)

{

   int allocate[processes];

   int occupied[blocks];

   for(int i = 0; i < processes; i++) {

                        allocate[i] = -1; }

            for(int i = 0; i < blocks; i++){

      occupied[i] = 0; }

   for (int i = 0; i < processes; i++) {

      for (int j = 0; j < blocks; j++) {

      if (!occupied[j] && blockSize[j] >= processSize[i])  {

               allocate[i] = j;

               occupied[j] = 1;

               break;  }  } }

   printf("\nProcess No.\tProcess Size\tBlock no.\n");

   for (int i = 0; i < processes; i++)

   {
```

```
        printf("%d \t\t\t %d \t\t\t", i+1, processSize[i]);
        if (allocate[i] != -1)
            printf("%d\n",allocate[i] + 1);
        else
            printf("Not Allocated\n"); } }
void main() {
    int blockSize[] = {30, 5, 10};
    int processSize[] = {10, 6, 9};
    int m = sizeof(blockSize)/sizeof(blockSize[0]);
    int n = sizeof(processSize)/sizeof(processSize[0]);
    implimentFirstFit(blockSize, m, processSize, n); }
```

Output:

```
b7@b7-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~/Desktop/AIDS69$ gcc firstfit.c
b7@b7-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~/Desktop/AIDS69$ ./a.out

Process No.     Process Size    Block no.
1                       10                      1
2                       6                       3
3                       9                       Not Allocated
b7@b7-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~/Desktop/AIDS69$ SS
```

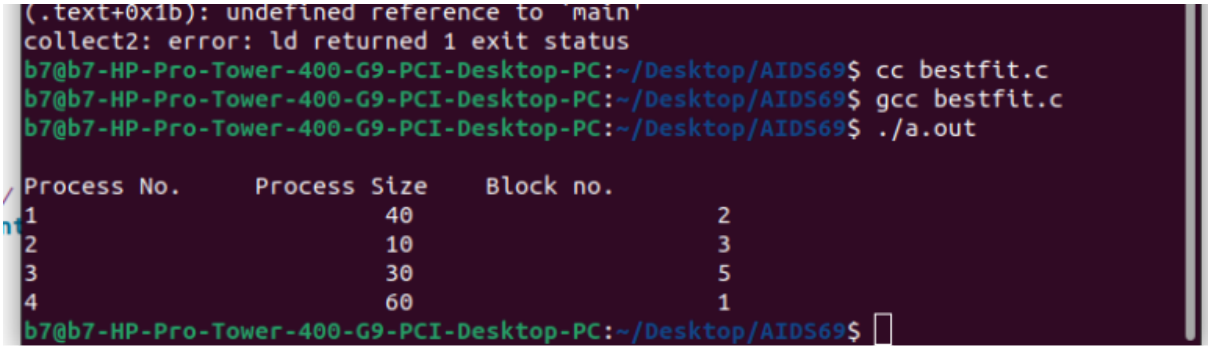
Best fit

```
#include <stdio.h>

void implimentBestFit(int blockSize[], int blocks, int processSize[], int proccesses) {

   int allocation[proccesses];

   int occupied[blocks];

   for(int i = 0; i < proccesses; i++){

      allocation[i] = -1;    }

   for(int i = 0; i < blocks; i++){

      occupied[i] = 0;

   }

   for (int i = 0; i < proccesses; i++)

   {       int indexPlaced = -1;

      for (int j = 0; j < blocks; j++) {

         if (blockSize[j] >= processSize[i] && !occupied[j])

         {

            if (indexPlaced == -1)

               indexPlaced = j;

            else if (blockSize[j] < blockSize[indexPlaced])

               indexPlaced = j;

         }
```

```
        }
        if (indexPlaced != -1)
        {
            allocation[i] = indexPlaced;
            occupied[indexPlaced] = 1;  }  }
     printf("\nProcess No.\tProcess Size\tBlock no.\n");
    for (int i = 0; i < proccesses; i++)
    {
        printf("%d \t\t\t %d \t\t\t", i+1, processSize[i]);
        if (allocation[i] != -1)
            printf("%d\n",allocation[i] + 1);
        else
            printf("Not Allocated\n");   } }
int main() {
    int blockSize[] = {100, 50, 30, 120, 35};
    int processSize[] = {40, 10, 30, 60};
    int blocks = sizeof(blockSize)/sizeof(blockSize[0]);
    int proccesses = sizeof(processSize)/sizeof(processSize[0]);
     implimentBestFit(blockSize, blocks, processSize, proccesses);
```

return 0 ; }

Output:



Worst Fit

```
#include <stdio.h>

void implimentWorstFit(int blockSize[], int blocks, int processSize[], int processes) {

    int allocation[processes];

    int occupied[blocks];

    for(int i = 0; i < processes; i++){

        allocation[i] = -1; }

    for(int i = 0; i < blocks; i++){

        occupied[i] = 0; }

    for (int i=0; i < processes; i++)  {

            int indexPlaced = -1;

            for(int j = 0; j < blocks; j++) {
```

```
            if(blockSize[j] >= processSize[i] && !occupied[j]) {
            if (indexPlaced == -1)
               indexPlaced = j;
            else if (blockSize[indexPlaced] < blockSize[j])
               indexPlaced = j; }   }
        if (indexPlaced != -1)
        {
           allocation[i] = indexPlaced;
           occupied[indexPlaced] = 1
           blockSize[indexPlaced] -= processSize[i]; }  }
    printf("\nProcess No.\tProcess Size\tBlock no.\n");
    for (int i = 0; i < processes; i++) {
       printf("%d \t\t\t %d \t\t\t", i+1, processSize[i]);
       if (allocation[i] != -1)
          printf("%d\n",allocation[i] + 1);
       else
          printf("Not Allocated\n"); } }
int main() {
    int blockSize[] = {100, 50, 30, 120, 35};
```

```
int processSize[] = {40, 10, 30, 60};

    int blocks = sizeof(blockSize)/sizeof(blockSize[0]);

    int processes = sizeof(processSize)/sizeof(processSize[0]);

    implimentWorstFit(blockSize, blocks, processSize, processes);

    return 0;

}
```

Output:

```
b7@b7-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~/Desktop/AIDS69$ touch worstfit.c
b7@b7-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~/Desktop/AIDS69$ gcc worstfit.c
b7@b7-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~/Desktop/AIDS69$ ./a.out

Process No.     Process Size    Block no.
1                       40                      4
2                       10                      1
3                       30                      2
4                       60                      Not Allocated
b7@b7-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~/Desktop/AIDS69$
```

Conclusion:

Why do we need memory allocation strategies?

Memory allocation strategies are essential for efficient utilization of computer memory in any computing system. These strategies govern how memory is allocated and deallocated to processes or programs running on the system. There are several reasons why these strategies are necessary:

1. Optimal Resource Utilization: Memory is a finite resource, and efficient allocation ensures that it is used optimally.

2. Prevention of Fragmentation: Memory fragmentation occurs when memory is allocated and deallocated in a way that leaves small pockets of unused memory scattered throughout the address space.

3. Fairness and Prioritization: Different processes or programs running on a system may have varying memory requirements.

4. Performance Optimization: Efficient memory allocation strategies can significantly impact system performance. For example, strategies that minimize memory overhead or reduce access latency can lead to faster program execution and improved responsiveness of the system as a whole.

5. Memory Protection and Security: Memory allocation strategies play a crucial role in enforcing memory protection and security mechanisms.