Name :- Khan Feisal
Roll no :- 147
Batch :- C12

SAP ID :- 60024240018
Subj :- OS

Experiment. (8)

* Aim :- Implement program on Memory Allocation : First, Best, worst, & next fit.

* Theory :-

Memory allocation refers to the process assigning memory space to programs during execution. The OS must manage memory efficiently to ensure.

- multiple process can run simultaneously
- Memory is utilized optimally
- Fragmentation is minimized.

- Key memory allocation strategies :-

1. Continous memory Allocation
2. Non-Continous memory Allocation

- Memory allocation Algos.

1. First Fit :- Allocates first available size that's large enough.
2. Best Fit :- Allocates smallest suitable space.
3. worst Fit :- Allocates largest available spaces.
4. worst Fit :- Allocates the next available size that's large enough

- Common Issues:-

1] Fragmentation:-
   i] Internal Fragmentation
   ii] External Fragmentation

2] Thrashing

- Why Memory Allocation matters?

   i] Affects system performance.
   ii] Determine how many processes can run simultaneously
   iii] Impact overall system stability.

* Conclusion:-

Thus, I have understood the concept of memory allocation algos. first, next, Best & worst fit algos. I have successfully performed this experiment & implemented in c language.

Program:

```
 #include <stdio.h>
#include <limits.h>

struct MemoryBlock {
   int size;
   int allocated;
   int processID;
   int remaining;
};

void displayResults(struct MemoryBlock partitions[], int
numPartitions, int processes[], int numProcesses) {
   printf("\nProcess Allocation:\n");
   for (int i = 0; i < numPartitions; ++i) {
      if (partitions[i].allocated) {
         printf("Partition %d (Size: %d) -> Process %d
(Size: %d)\n",
               i + 1, partitions[i].size, partitions[i].processID,
processes[partitions[i].processID - 1]);
         if (partitions[i].remaining > 0) {
            printf("  Hole created: %d\n", partitions[i].remaining);
            int holeUsed = 0;
            for (int j = 0; j < numProcesses; ++j) {
               if (processes[j] <= partitions[i].remaining &&
processes[j] != -1) {
                  printf("  * Hole can be utilized by Process %d
(Size: %d)\n", j + 1, processes[j]);
                  holeUsed = 1;
               }
            }
            if (!holeUsed) printf("  * No process fits in the hole.\n");
         }
      } else {
```

```
            printf("Partition %d (Size: %d) -> Unallocated\n", i + 1,
partitions[i].size);
        }
    }
}

void resetAllocations(struct MemoryBlock partitions[], int
numPartitions) {
    for (int i = 0; i < numPartitions; ++i) {
        partitions[i].allocated = 0;
        partitions[i].processID = -1;
        partitions[i].remaining = 0;
    }
}

void firstFit(struct MemoryBlock partitions[], int numPartitions,
int processes[], int numProcesses) {
    resetAllocations(partitions, numPartitions);
    for (int i = 0; i < numProcesses; ++i) {
        int allocated = 0;
        for (int j = 0; j < numPartitions; ++j) {
            if (!partitions[j].allocated && partitions[j].size >=
processes[i]) {
                partitions[j].allocated = 1;
                partitions[j].processID = i + 1;
                partitions[j].remaining = partitions[j].size - processes[i];
                allocated = 1;
                break;
            }
        }
        if (!allocated) {
            printf("Process %d (Size: %d) -> Unallocated\n", i + 1,
processes[i]);
        }
    }
```

```
    displayResults(partitions, numPartitions, processes,
numProcesses);
}

void bestFit(struct MemoryBlock partitions[], int numPartitions,
int processes[], int numProcesses) {
    resetAllocations(partitions, numPartitions);
    for (int i = 0; i < numProcesses; ++i) {
        int bestIdx = -1;
        int minSize = INT_MAX;
        for (int j = 0; j < numPartitions; ++j) {
            if (!partitions[j].allocated && partitions[j].size >=
processes[i] && partitions[j].size < minSize) {
                bestIdx = j;
                minSize = partitions[j].size;
            }
        }
        if (bestIdx != -1) {
            partitions[bestIdx].allocated = 1;
            partitions[bestIdx].processID = i + 1;
            partitions[bestIdx].remaining = partitions[bestIdx].size -
processes[i];
        } else {
            printf("Process %d (Size: %d) -> Unallocated\n", i + 1,
processes[i]);
        }
    }
    displayResults(partitions, numPartitions, processes,
numProcesses);
}

void worstFit(struct MemoryBlock partitions[], int
numPartitions, int processes[], int numProcesses) {
    resetAllocations(partitions, numPartitions);
    for (int i = 0; i < numProcesses; ++i) {
```

```
        int worstIdx = -1;
        int maxSize = -1;
        for (int j = 0; j < numPartitions; ++j) {
            if (!partitions[j].allocated && partitions[j].size >=
processes[i] && partitions[j].size > maxSize) {
                worstIdx = j;
                maxSize = partitions[j].size;
            }
        }
        if (worstIdx != -1) {
            partitions[worstIdx].allocated = 1;
            partitions[worstIdx].processID = i + 1;
            partitions[worstIdx].remaining = partitions[worstIdx].size
- processes[i];
        } else {
            printf("Process %d (Size: %d) -> Unallocated\n", i + 1,
processes[i]);
        }
    }
    displayResults(partitions, numPartitions, processes,
numProcesses);
}

void nextFit(struct MemoryBlock partitions[], int numPartitions,
int processes[], int numProcesses) {
    resetAllocations(partitions, numPartitions);
    int lastAllocated = 0;
    for (int i = 0; i < numProcesses; ++i) {
        int allocated = 0;
        for (int j = 0; j < numPartitions; ++j) {
            int idx = (lastAllocated + j) % numPartitions;
            if (!partitions[idx].allocated && partitions[idx].size >=
processes[i]) {
                partitions[idx].allocated = 1;
                partitions[idx].processID = i + 1;
```

```
            partitions[idx].remaining = partitions[idx].size -
processes[i];
            allocated = 1;
            lastAllocated = idx;
            break;
         }
      }
      if (!allocated) {
         printf("Process %d (Size: %d) -> Unallocated\n", i + 1,
processes[i]);
      }
   }
   displayResults(partitions, numPartitions, processes,
numProcesses);
}

int main() {
   int numPartitions, numProcesses;
   printf("Enter number of memory partitions: ");
   scanf("%d", &numPartitions);
   struct MemoryBlock partitions[numPartitions];
   printf("Enter sizes of partitions:\n");
   for (int i = 0; i < numPartitions; ++i) {
      scanf("%d", &partitions[i].size);
      partitions[i].allocated = 0;
      partitions[i].processID = -1;
      partitions[i].remaining = 0;
   }

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

```
    }

    int choice;
    do {
        printf("\nChoose Memory Allocation Algorithm:\n"
               "1. First Fit\n"
               "2. Best Fit\n"
               "3. Worst Fit\n"
               "4. Next Fit\n"
               "5. Exit\n"
               "Enter choice (1-5): ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("\n--- First Fit Allocation ---\n");
                firstFit(partitions, numPartitions, processes,
numProcesses);
                break;
            case 2:
                printf("\n--- Best Fit Allocation ---\n");
                bestFit(partitions, numPartitions, processes,
numProcesses);
                break;
            case 3:
                printf("\n--- Worst Fit Allocation ---\n");
                worstFit(partitions, numPartitions, processes,
numProcesses);
                break;
            case 4:
                printf("\n--- Next Fit Allocation ---\n");
                nextFit(partitions, numPartitions, processes,
numProcesses);
                break;
            case 5:
```

```
            printf("\nExiting program...\n");
            break;
        default:
            printf("Invalid choice! Please try again.\n");
    }
  } while (choice != 5);

  return 0;
}
```

Output:

```
Enter number of memory partitions: 5
Enter sizes of partitions:
100 200 50 300 150
Enter number of processes: 4
Enter sizes of processes:
90 40 250 120
```

1. First Fit:

```
Choose Memory Allocation Algorithm:
1. First Fit
2. Best Fit
3. Worst Fit
4. Next Fit
5. Exit
Enter choice (1-5): 1

--- First Fit Allocation ---

Process Allocation:
Partition 1 (Size: 100) -> Process 1 (Size: 90)
  Hole created: 10
  * No process fits in the hole.
Partition 2 (Size: 200) -> Process 2 (Size: 40)
  Hole created: 160
  * Hole can be utilized by Process 1 (Size: 90)
  * Hole can be utilized by Process 2 (Size: 40)
  * Hole can be utilized by Process 4 (Size: 120)
Partition 3 (Size: 50) -> Unallocated
Partition 4 (Size: 300) -> Process 3 (Size: 250)
  Hole created: 50
  * Hole can be utilized by Process 2 (Size: 40)
Partition 5 (Size: 150) -> Process 4 (Size: 120)
  Hole created: 30
  * No process fits in the hole.
```

2. Best Fit:

```
Choose Memory Allocation Algorithm:
1. First Fit
2. Best Fit
3. Worst Fit
4. Next Fit
5. Exit
Enter choice (1-5): 2

--- Best Fit Allocation ---

Process Allocation:
Partition 1 (Size: 100) -> Process 1 (Size: 90)
  Hole created: 10
  * No process fits in the hole.
Partition 2 (Size: 200) -> Unallocated
Partition 3 (Size: 50) -> Process 2 (Size: 40)
  Hole created: 10
  * No process fits in the hole.
Partition 4 (Size: 300) -> Process 3 (Size: 250)
  Hole created: 50
  * Hole can be utilized by Process 2 (Size: 40)
Partition 5 (Size: 150) -> Process 4 (Size: 120)
  Hole created: 30
  * No process fits in the hole.
```

3. Worst Fit:

```
Choose Memory Allocation Algorithm:
1. First Fit
2. Best Fit
3. Worst Fit
4. Next Fit
5. Exit
Enter choice (1-5): 3

--- Worst Fit Allocation ---
Process 3 (Size: 250) -> Unallocated

Process Allocation:
Partition 1 (Size: 100) -> Unallocated
Partition 2 (Size: 200) -> Process 2 (Size: 40)
  Hole created: 160
  * Hole can be utilized by Process 1 (Size: 90)
  * Hole can be utilized by Process 2 (Size: 40)
  * Hole can be utilized by Process 4 (Size: 120)
Partition 3 (Size: 50) -> Unallocated
Partition 4 (Size: 300) -> Process 1 (Size: 90)
  Hole created: 210
  * Hole can be utilized by Process 1 (Size: 90)
  * Hole can be utilized by Process 2 (Size: 40)
  * Hole can be utilized by Process 4 (Size: 120)
Partition 5 (Size: 150) -> Process 4 (Size: 120)
  Hole created: 30
  * No process fits in the hole.
```

4. Next Fit:

```
Choose Memory Allocation Algorithm:
1. First Fit
2. Best Fit
3. Worst Fit
4. Next Fit
5. Exit
Enter choice (1-5): 4

--- Next Fit Allocation ---

Process Allocation:
Partition 1 (Size: 100) -> Process 1 (Size: 90)
  Hole created: 10
  * No process fits in the hole.
Partition 2 (Size: 200) -> Process 2 (Size: 40)
  Hole created: 160
  * Hole can be utilized by Process 1 (Size: 90)
  * Hole can be utilized by Process 2 (Size: 40)
  * Hole can be utilized by Process 4 (Size: 120)
Partition 3 (Size: 50) -> Unallocated
Partition 4 (Size: 300) -> Process 3 (Size: 250)
  Hole created: 50
  * Hole can be utilized by Process 2 (Size: 40)
Partition 5 (Size: 150) -> Process 4 (Size: 120)
  Hole created: 30
  * No process fits in the hole.
```

5. Exit:

```
Choose Memory Allocation Algorithm:
1. First Fit
2. Best Fit
3. Worst Fit
4. Next Fit
5. Exit
Enter choice (1-5): 5

Exiting program...
```