Laboratory 8

Title of the Laboratory Exercise: Programs for memory management algorithms

1. Introduction and Purpose of Experiment

   In a multiprogramming system, the user part of memory must be further subdivided to accommodate multiple processes. This task of subdivision is carried out dynamically done by the operating system known as memory management. By solving these problems students will become familiar with the implementations of memory management algorithms in dynamic memory partitioning scheme of operating system.

2. Aim and Objectives

   Aim

   - To develop a simulator for memory management algorithms

   Objectives

   At the end of this lab, the student will be able to

   - Apply memory management algorithms wherever required
   - Develop simulators for the algorithms

3. Experimental Procedure

   - Analyse the problem statement
   - Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   - Implement the algorithm in C language
   - Compile the C program
   - Test the implemented program
   - Document the Results
   - Analyse and discuss the outcomes of your experiment

4. Questions

Implement a simulator for the memory management algorithms with the provision of compaction and garbage collection

a) First fit

b) Best fit

c) Worst fit

5. Calculations/Computations/Algorithms

- `best_fit()`

Calculates the best fit for the process blocks

```
Step 1: Start

Step 2: define variable to store the allocations

Step 3: initialize allocations to -1

Step 4: for all process blocks i, do

      4.1: index = -1

      4.2: for all the blocks j, do

            4.2.1: if process can fit in block, then

                  4.2.1.1: if index is -1, then index = j

                  4.2.1.2: else if there is a better fitting
                  block then, index = j

      4.3: if the best block is found, then

            4.3.1: allocs[i] = index

            4.3.2: reduce available memory in block[index]

Step 5: print results

Step 6: Stop
```

- `first_fit()`

Calculates the first fit for the process blocks

```
Step 1: Start

Step 2: define variable to store the allocations

Step 3: initialize allocations to -1

Step 4: for all process blocks i, do

     4.1: for all the blocks j, do

          4.1.1: if process can fit in block, then

               4.1.1.1: allocs[i] = j

          4.1.1.2: reduce available memory in block[i]

          4.1.1.3: break

Step 5: print results

Step 6: Stop
```

- `worst_fit()`

Calculates the worst fit for the process blocks

```
Step 1: Start

Step 2: define variable to store the allocations

Step 3: initialize allocations to -1

Step 4: for all process blocks i, do

     4.1: index = -1

     4.2: for all the blocks j, do

          4.2.1: if process can fit in block, then

               4.2.1.1: if index is -1, then index = j

               4.2.1.2: else if there is a worse fitting
               block then, index = j

     4.3: if the best block is found, then

          4.3.1: allocs[i] = index
```

4.3.2: reduce available memory in block[index]

Step 5: print results

Step 6: Stop

6. Presentation of Results

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void best_fit(int*, int, int*, int);
void first_fit(int*, int, int*, int);
void worst_fit(int*, int, int*, int);
void disp(int*, int*, int);

int main() {
    // declaring input variables
    int m, n;

    // getting number of blocks from user
    printf("Enter the number of blocks: ");
    scanf("%d", &m);

    // getting block sizes from user
    int blockSizes[m];
    for (int i = 0; i < m; i++) {
        printf("Enter size of block %d: ", i+1);
        scanf("%d", &blockSizes[i]);
        getchar();
    }

    // getting number of processes from user
    printf("\nEnter the number of processes: ");
    scanf("%d", &n);

    // getting process sizes from user
    int processSizes[n];
    for (int i = 0; i < n; i++) {
        printf("Enter size of process %d: ", i+1);
        scanf("%d", &processSizes[i]);
        getchar();
    }

    // creating copy of blockSizes to be passed to the function
    // and calling best fit
    int* blockSizesCopy = malloc(sizeof(*blockSizesCopy) * m);
    memcpy(blockSizesCopy, blockSizes, sizeof(blockSizes));
    best_fit(blockSizesCopy, m, processSizes, n);

    // creating copy of blockSizes to be passed to the function
    // and calling first fit
    blockSizesCopy = malloc(sizeof(*blockSizesCopy) * m);
    memcpy(blockSizesCopy, blockSizes, sizeof(blockSizes));
    first_fit(blockSizesCopy, m, processSizes, n);

    // creating copy of blockSizes to be passed to the function
    // and calling worst fit
    blockSizesCopy = malloc(sizeof(*blockSizesCopy) * m);
    memcpy(blockSizesCopy, blockSizes, sizeof(blockSizes));
    worst_fit(blockSizesCopy, m, processSizes, n);
}

void best_fit(int* blockSizes, int m, int* processSizes, int n) {
    // defining variable to store the allocation
    int allocs[n];
```

```
    // initializing allocs to be -1
    memset(allocs, -1, sizeof(allocs));

    // going through all the processes
    for (int i=0; i<n; i++) {
        int index = -1;

        // going through all the blocks
        for (int j=0; j<m; j++) {
            // if a process can fit in the block
            if (blockSizes[j] >= processSizes[i]) {
                if (index == -1)
                    index = j;
                // if there is a better fitting block
                else if (blockSizes[index] > blockSizes[j])
                    index = j;
            }
        }

        // if we found the best block
        if (index != -1) {
            // allocate block j to p[i] process
            allocs[i] = index;

            // Reduce available memory in this block.
            blockSizes[index] -= processSizes[i];
        }
    }

    // printing results
    printf("\nBest fit results:\n");
    disp(processSizes, allocs, n);
}

void first_fit(int* blockSizes, int m, int* processSizes, int n) {
    // defining variable to store the allocated processes
    int allocs[n];

    // initializing allocation to -1
    memset(allocs, -1, sizeof(allocs));

    // for all the processes
    for (int i = 0; i < n; i++) {
        // for all the blocks
        for (int j = 0; j < m; j++) {
            // if the process can fit in the block
            if (blockSizes[j] >= processSizes[i]) {
                // allocate block to process
                allocs[i] = j;

                // Reduce available memory in this block.
                blockSizes[j] -= processSizes[i];
                break;
            }
        }
    }

    // printing results
    printf("\nFirst fit results:\n");
    disp(processSizes, allocs, n);
}

void worst_fit (int* blockSizes, int m, int* processSizes, int n) {
    // defining variable to store the allocation
    int allocs[n];

    // initializing allocs to be -1
    memset(allocs, -1, sizeof(allocs));
```

```
    // going through all the processes
    for (int i=0; i<n; i++) {
        int index = -1;

        // going through all the blocks
        for (int j=0; j<m; j++) {
            // if a process can fit in the block
            if (blockSizes[j] >= processSizes[i]) {
                if (index == -1)
                    index = j;
                // if there is a better fitting block
                else if (blockSizes[index] < blockSizes[j])
                    index = j;
            }
        }

        // if we found the best block
        if (index != -1) {
            // allocate block j to p[i] process
            allocs[i] = index;

            // Reduce available memory in this block.
            blockSizes[index] -= processSizes[i];
        }
    }

    // printing results
    printf("\nWorst fit results\n");
    disp(processSizes, allocs, n);
}

void disp(int* processSizes, int* allocs, int numOfProcesses) {
    printf("Process No.\tProcess size\tBlock no.\n");
    for(int i = 0; i < numOfProcesses; i++) {
        printf("%d\t\t%d\t\t", i+1, processSizes[i]);
        if (allocs[i] != -1)
            printf("%d\n", allocs[i]+1);
        else
            printf("Not allocated\n");
    }
}
```

Figure 1: Source code

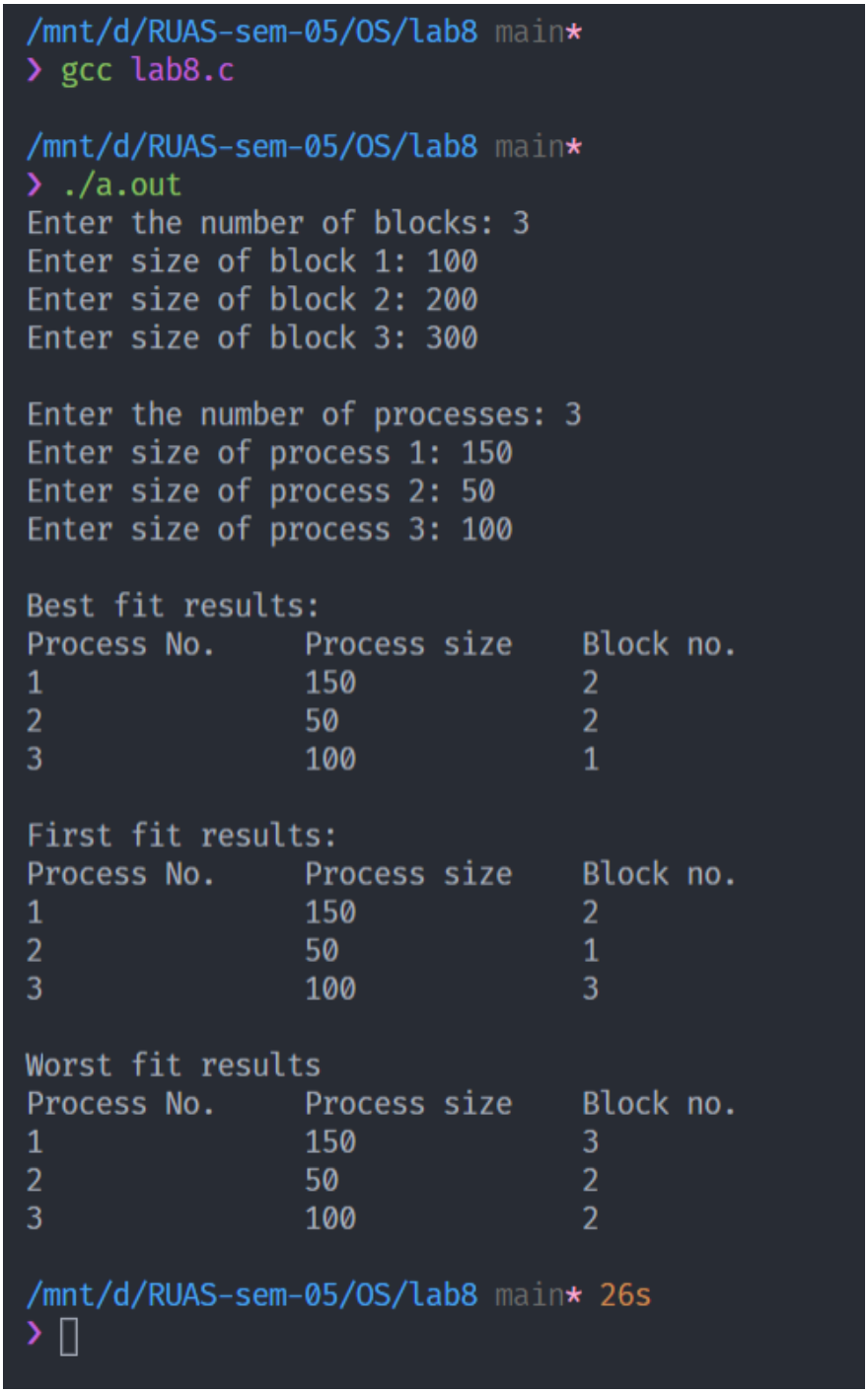

Figure 2: Execution

7. Analysis and Discussions

Memory management is the process of controlling and coordinating computer memory, assigning portions called blocks to various running programs to optimize overall system performance. Memory management resides in hardware, in the OS (operating system), and in programs and applications.

Three algorithms that are used in memory management are best fit, worst fit and first fit

As the name suggests, best fit algorithm assign a process a memory block where the least amount of memory is wasted overall. First fit algorithm assigns a process the first available memory block it finds. Worst fit allocates a process to the partition which is largest sufficient among the freely available partitions available in the main memory. If a large process comes at a later stage, then memory will not have space to accommodate it.

8. Conclusions

The memory management algorithms are understood and implemented in C

9. Comments

1. Limitations of Experiments

The program does not apply the algorithm based on the situation.

2. Limitations of Results

Best fit algorithm consumes a lot of CPU time. First fit algorithm produces a lot of holes.

3. Learning happened

The memory management algorithms were learned.