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

a) For First Fit:

```
Algorithm main ( ) : integer
Begin
Var flag[max], b[max],f[max], nb, nf, temp, highest = 0 : integer
Var bf[max], ff[max] : static integer

Read nb, nf;

for I in 0 to nb
read b[i];
for I in 1 to nf
read f[i];

for i in 1 to nf
begin
for j in 1 to nb
begin
    if bf[j] != 1
    being
        temp := b[j] - f[i];
        if temp >= 0
        being
                ff[i] = j;
                highest:= temp;
        end
    end
end
flag[i] := highest;
bf[ff[i]] := 1;
highest = 0;
end
for I in 1 to i <= nf and ff[i] != 0
  write i, f[i], ff[i], b[ff[i]], flag[i];
end
```

b) For Best Fit:

```
m Algorithm main ( ) : integer
Begin
Var flag[max], b[max],f[max], nb, nf, temp, lowest = 10000 : integer
Var bf[max], ff[max] : static integer

Read nb, nf;

for I in 0 to nb
read b[i];
for I in 1 to nf
```

```
        read f[i];

        for i in 1 to nf
        begin
        for j in 1 to nb
        begin
            if bf[j] != 1
            being
                temp := b[j] - f[i];
                if temp >= 0
                if lowest > temp
                being
                        ff[i] = j;
                        lowest = temp;
                end
            end
        end
        flag[i] := temp;
        bf[ff[i]] := 1;
        lowest = 1000;
end
for I in 1 to i <= nf and ff[i] != 0
   write i, f[i], ff[i], b[ff[i]], flag[i];
end
```

c) For Worst fit:

```
        Algorithm main ( ) : integer
        Begin
        Var flag[max], b[max],f[max], nb, nf, temp, lowest = 10000 : integer
        Var bf[max], ff[max] : static integer

        Read nb, nf;

        for I in 0 to nb
        read b[i];
        for I in 1 to nf
        read f[i];

        for i in 1 to nf
        begin
        for j in 1 to nb
        begin
            if bf[j] != 1
            being
                temp := b[j] - f[i];
                if temp >= 0
                being
                        ff[i] = j;
                        break;
                end
            end
        end
        flag[i] := temp;
        bf[ff[i]] := 1;
end
for I in 1 to i <= nf and ff[i] != 0
   write i, f[i], ff[i], b[ff[i]], flag[i];
end
```

6. Presentation of Results

Program:

```
#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("P
rocess 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");
}
}
```

Screenshots of output:-

```
PS F:\RUAS\5 sem\Operating system\Os lo l8 } ; if ($?) { .\l8 }
ab\lab_8> cd "f:\RUAS\5 sem\Operating
system\Os lab\lab_8\" ; if ($?) { gcc
l8.c -o l8 } ; if ($?) { .\l8 }
Enter the number of blocks: 3
Enter size of block 1: 110
Enter size of block 2: 210
Enter size of block 3: 310

Enter the number of processes: 3
Enter size of process 1: 160
Enter size of process 2: 60
Enter size of process 3: 110

Best fit results:
Process No.     Process size    Block no.
1               160             2
2               60              1
3               110             3

First fit results:
Process No.     Process size    Block no.
1               160             2
2               60              1
3               110             3
```

```
Worst fit results
Process No.     Process size    Block no.
1               160             3
2               60              2
3               110             2
PS F:\RUAS\5 sem\Operating system\Os lab\lab_8>
```

7. Analysis and Discussions

In addition to the responsibility of managing processes, the operating system must efficiently manage the primary memory of the computer. The part of the operating system which handles this responsibility is called the memory manager. Since every process must have some amount of primary memory in order to execute, the performance of the memory manager is crucial to the performance of the entire system. Nutt explains: "The memory manager is responsible for allocating primary memory to processes and for assisting the programmer in loading and storing the contents of the primary memory. Managing the sharing of primary memory and minimizing memory access time are the basic goals of the memory manager."

The real challenge of efficiently managing memory is seen in the case of a system which has multiple processes running at the same time. Since primary memory can be space-multiplexed, the memory manager can allocate a portion of primary memory to each process for its own use. However, the memory manager must keep track of which processes are running in which memory locations, and it must also determine how to allocate and deallocate available memory when new processes are created and when old processes complete execution. While various different strategies are used to allocate space to processes competing for memory, three of the most popular are Best fit, Worst fit, and First fit. Each of these strategies are described below :
Best fit: The allocator places a process in the smallest block of unallocated memory in which it will fit. For example, suppose a process requests 12KB of memory and the memory manager currently has a list of unallocated blocks of 6KB, 14KB, 19KB, 11KB, and 13KB blocks. The best-fit strategy will allocate 12KB of the 13KB block to the process.

Worst fit: The memory manager places a process in the largest block of unallocated memory available. The idea is that this placement will create the largest hold after the allocations, thus increasing the possibility that, compared to best fit, another process can use the remaining space. Using the same example as above, worst fit will allocate 12KB of the 19KB block to the process, leaving a 7KB block for future use.

First fit: There may be many holes in the memory, so the operating system, to reduce the amount of time it spends analyzing the available spaces, begins at the start of primary memory and allocates memory from the first hole it encounters large enough to satisfy the request. Using the same example as above, first fit will allocate 12KB of the 14KB block to the process.

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.