Experiment No. 6

Experiment Name: Write a program to implement the best fit memory management algorithm.

Introduction: The process of allocating memory in a way that wastes the least amount of space is called best fit memory management . Though it's optimal from a memory standpoint, implementing it efficiently (from a time complexity perspective) is almost impossible.

Theory: In order to find the best fit we iterate through all holes or empty segments of memory until we find either the perfectly matching memory segment or allocate the empty memory segment with the closest size.

Algorithm:

1. Input the number of blocks and block sizes (bsize[]) along with the number of process and process sizes (psize[])
2. Initialize all blocks to be empty and no process is allocating them,
   flag[i] = false, alloc[i] = -1
3. For all process i, repeat Step-4 to Step-6:
4. Initialize:
   lowest = + Infinite id = invalid_id
5. Do the following for all the blocks, j:

   (i) If flag[j] = 0 AND this block has less unused space than lowest, then:

   a. lowest = unused_space[j]

   b. id = j;

6. Allocate process to the block with lowest unused space

   alloc[id] = i, flag[id] = true;

7. Calculate unused space of blocks:

(i) If block allocates a process (flag[] = true),

a. Unused space = bsize[] – psize[]

(ii) Else

a. Unused space = bsize[]

Program Code:

```
#include<stdio.h>


int main()
{
    int b_no,p_no,bsize[10],psize[10],flags[10],alloc[10],temp,lowest=9999,id=9,i,j;
    printf("Enter no. of blocks: ");
    scanf("%d", &b_no);
    printf("Enter size of each block: ");
    for(i = 0; i < b_no; i++)
    scanf("%d", &bsize[i]);
    printf("\nEnter no. of processes: ");
    scanf("%d", &p_no);
    printf("Enter size of each process: ");
    for(i = 0; i < p_no; i++)
```

```
scanf("%d", &psize[i]);
for(i = 0; i < b_no; i++)
{
flags[i] = 0;
alloc[i] = -1;
}
for(i = 0; i < p_no; i++)
{
for(j = 0; j < b_no; j++)
{
if(flags[j] == 0)
{
        temp = bsize[j] - psize[i];
        if(temp >= 0 && temp < lowest)
        {
        lowest = temp;
        id = j;
        }
}
}
```

```
        alloc[id] = i;
        flags[id] = 1;
        lowest = 9999;
        id = 9;
        }
        printf("\nBlock no.\tBlock Size\tProcess no.\tProcess Size\tUnused Space");
        for(i = 0; i < b_no; i++)
        {
        printf("\n%d\t\t%d\t\t", i+1, bsize[i]);
        if(flags[i] == 1)
        printf("%d \t\t%d \t\t%d",alloc[i]+1, psize[alloc[i]], bsize[i]-psize[alloc[i]]);
        else
        printf("---\t\t---\t\t%d", bsize[i]);
        }
        printf("\n");
}
```

Output:

Enter no. of blocks: 4

Enter size of each block: 20 22 30 35

Enter no. of processes: 3

Enter size of each process: 25 18 33

Block no.	Block Size	Process no.	Process Size	Unused Space
1	20	2	18	2
2	22	---	---	22
3	30	1	25	5
4	35	3	33	2

Discussion: This problem runs successfully on CodeBlocks. The program allocates memory as desired per algorithm . The result is also accurate.

Experiment No. 7

Experiment Name: Write a program to implement the worst fit memory management algorithm**.**

Introduction: Worst fit memory management allocates the current largest available memory block to the process without considering the actual size or necessity . The largest available memory block is searched upon request by the process.

Theory: The memory manager traverses the entire memory to find the current largest segment of available memory or largest hole and allocates the process memory from the starting of that block/memory location. The entire traversal process of memory space slows down the allocation process significantly.

Algorithm:

1. Input the number of blocks and block sizes (bsize[]) along with the number of process and process sizes (psize[]).
2. Initialize all blocks to be empty and no process is allocating them,
   flag[i] = false, alloc[i] = -1
3. For all process i, repeat Step-4 to Step-6:
4. Initialize:
   highest = - Infinite id = invalid_id
5. Do the following for all the blocks, j:

   (i) If flag[j] = 0 AND this block has more unused space than highest, then:

   a. highest = unused_space[j]

   b. id = j;

6. Allocate process to the block with highest unused space

   alloc[id] = i, flag[id] = true;

7. Calculate unused space of blocks:

(i) If block allocates a process (flag[] = true),

a. Unused space = bsize[] – psize[]

(ii) Else

a. Unused space = bsize[]

Program Code:

```
#include<stdio.h>
int main()
{
    int b_no,p_no,bsize[10],psize[10],flags[10],alloc[10],temp,highest=-9999,id = 9,i,j;
    printf("Enter no. of blocks: ");
    scanf("%d", &b_no);
    printf("Enter size of each block: ");
    for(i = 0; i < b_no; i++)
    scanf("%d", &bsize[i]);
    printf("\nEnter no. of processes: ");
    scanf("%d", &p_no);
    printf("Enter size of each process: ");
    for(i = 0; i < p_no; i++)
```

```
scanf("%d", &psize[i]);
for(i = 0; i < b_no; i++)
{
flags[i] = 0;              //All block are empty
alloc[i] = -1;             //Block contains no process
}
for(i = 0; i < p_no; i++)
{
for(j = 0; j < b_no; j++)
{
if(flags[j] == 0)
{
        temp = bsize[j] - psize[i];
        if(temp >= 0 && temp > highest)
        {
        highest = temp;
            id = j;
        }
}
}
```

```
        alloc[id] = i;
        flags[id] = 1;
        highest = -9999;
        id = 9;
        }
        printf("\nBlock no.\tBlock Size\tProcess no.\tProcess Size\tUnused Space");
        for(i = 0; i < b_no; i++)
        {
        printf("\n%d\t\t%d\t\t", i+1, bsize[i]);
        if(flags[i] == 1)
        printf("%d \t\t%d \t\t%d",alloc[i]+1, psize[alloc[i]], bsize[i]-psize[alloc[i]]);
        else
        printf("---\t\t---\t\t%d", bsize[i]);
        }
        printf("\n");
}
```

Output:

Enter no. of blocks: 4

Enter size of each block: 20 22 30 35

Enter no. of processes: 3

Enter size of each process: 25 18 33

Block no.	Block Size	Process no.	Process Size	Unused Space
1	20	---	---	20
2	22	---	---	22
3	30	2	18	12
4	35	1	25	10

Discussion: This program is running successfully generating the correct output following the proper algorithm. The experiment was a success and we are getting our desired output.

Experiment No. 8

Experiment Name: Write a program to implement the first fit memory management algorithm.

Introduction: First fit memory starts to traverse the memory searching for a memory partition large enough to meet the processes needs.

Theory: The memory manager takes two pointers one for the first location of an available memory space, one to track the ending of the processes memory allocation. On the first incident where an available memory segment is larger or equal to the requested or needed memory space size , the necessary amount of space is allocated to the process from that space.

Algorithm:

1. Input the number of blocks and block sizes (bsize[]) along with the number of process and process sizes (psize[]).
2. Initialize all blocks to be empty and no process is allocating them,
   flag[i] = false, alloc[i] = -1
3. For all process i, repeat Step-4:
4. Do the following for all the blocks, j:

   (i) If flag[j] = 0 AND bsize[j] >= psize[i], then:

   a. alloc[j] = i;

   b. flag[j] = true;

   c. Go to Step-3 for next process.

5. Calculate unused space of blocks:

   (i) If block allocates a process (flag[] = true),

   a. Unused space = bsize[] – psize[]

   (ii) Else

   a. Unused space = bsize[]

Program Code:

```
#include<stdio.h>


int main()

{

        int b_no,p_no,bsize[10],psize[10],flags[10],alloc[10],i,j;

        printf("Enter no. of blocks: ");

        scanf("%d", &b_no);

        printf("Enter size of each block: ");

        for(i = 0; i < b_no; i++)

        scanf("%d", &bsize[i]);

        printf("\nEnter no. of processes: ");

        scanf("%d", &p_no);

        printf("Enter size of each process: ");

        for(i = 0; i < p_no; i++)

        scanf("%d", &psize[i]);

        for(i = 0; i < b_no; i++)
```

```
{
flags[i] = 0;
alloc[i] = -1;
}
for(i = 0; i < p_no; i++)
{
for(j = 0; j < b_no; j++)
{
if(flags[j] == 0 && bsize[j] >= psize[i])
{
        alloc[j] = i;
        flags[j] = 1;
        break;
}
}
}
printf("\nBlock no.\tBlock Size\tProcess no.\tProcess Size\tUnused Space");
for(i = 0; i < b_no; i++) {
printf("\n%d\t\t%d\t\t", i+1, bsize[i]);
```

```
        if(flags[i] == 1)
        printf("%d \t\t%d \t\t%d",alloc[i]+1, psize[alloc[i]], bsize[i]-psize[alloc[i]]);
        else
        printf("---\t\t---\t\t%d", bsize[i]);
        }
        printf("\n");
}
```

Output:

Enter no. of blocks: 4

Enter size of each block: 20 22 30 35

Enter no. of processes: 3

Enter size of each process: 25 18 33

Block no.	Block Size	Process no.	Process Size	Unused Space
1	20	2	18	2
2	22	---	---	22
3	30	1	25	5
4	35	3	33	2

Discussion: This program works successfully. As we are getting our desired output according to the algorithm .

Experiment No. 9

Experiment Name: Write a program to simulate First-in First-out (FIFO) Page replacement algorithm.

Introduction: In FIFO page replacement we change the most recent page upon a page fault . The page ordering is tracked using a queue and pages are added at the end of the queue and pages are replaced from the front of the queue.

Theory: It's a very basic page replacement algorithm where the pages are kept track of via a queue . The queue while not too its max size has new elements added to the end on page fault. Once the queue is full the pages are replaced from the front of the queue , replacing the oldest pages first.

Algorithm:

1. Input the reference string (s []) and number of frames.
2. Initialize all frames to be empty,
   frame[i] = -1;
3. For all pages i of the string, repeat Step-4 & Step-5:
4. If page is available in the frame, then:

   (i) Go to Step-3 for next page.

5. Do the following:

   (i) Store this page in the current frame, i.e.
   frame[pos] = s[i];

   (ii) Move position to the next frame, i.e.
   pos = next_frame;

   (iii) Count page fault, i.e.
   count++;

Program Code:

```
#include<stdio.h>

int main()

{

        int l, s[50], frame[10], n, avail, count = 0, i, j, pos=0;

        printf("Enter the length of the string: ");

        scanf("%d",&l);

        printf("Enter the string: ");

        for(i=0; i<l; i++)

        scanf("%d",&s[i]);

        printf("Enter the number of frames: ");

        scanf("%d",&n);

        for(i=0; i<n; i++)

        frame[i]= -1;

        printf("\nString\t\t Page Frames\n");

        for(i=0; i<l; i++)

        {

        printf("%d\t\t",s[i]);

        avail = 0;

        for(j=0; j<n; j++)
```

```
{
if(frame[j] == s[i])
{
        avail = 1;
        break;
}
}
if(avail == 0)
{
frame[pos] = s[i];
pos = (pos+1) % n;
count++;
for(j=0; j<n; j++){
        if(frame[j] != -1)
    printf("%d\t",frame[j]);
}
}
printf("\n");
}
printf("\nPage Fault is = %d\n", count);
```

```
    return 0;
}
```

Output:

```
Enter the length of the string: 5

Enter the string: 10 7 3 5 2

Enter the number of frames: 3
```

String	Page Frames		
10	10		
7	10	7	
3	10	7	3
5	5	7	3
2	5	2	3

```
Page Fault is = 5
```

Discussion: This problem is running successfully on CodeBlocks. The experiment was a success and we are getting our desired output.

Experiment No. 10

Experiment Name: Write a program to simulate the Optimal Page replacement algorithm**.**

Introduction: Optimal page replacement algorithm produces the fewest page faults by replacing pages that will not be used for the longest period of time in the future.

Theory: Optimal page replacement algorithm replaces the page that will not be used for the longest period of time in the future. This way only the least used pages in the future is replaced, resulting in the most efficient algorithm. However , in practice it's not possible to know which process will be used in the future efficiently Hence this algorithm is very hard to implement in actual scenarios.

Algorithm:

1. Input the reference string (s[]) and number of frames.
2. Initialize all frames to be empty,
   frame[i] = -1;
3. For all pages i of the string, repeat Step-4 & Step-5:
4. If page is available in the frame, then:

   (i) Go to Step-3 for next page.

5. Do the following:

   (i) If frames are not full, then:

      a. frame[pos] = s[i];

      b. pos = next_frame;

   (ii) Else:

      a. CALL optimal() function.

      b. Get the optimal result for pos.

c. Store page in this frame,
frame[pos] = s[i].

(iii) Count page fault,
count++;

Optimal():

1. Initialize,
   ans = first_frame, farthest = next_page;
2. For all the frames i, repeat Step-3 & Step-4.
3. Search for the frame from the next_page to the last_page of the string.

   (i) If found, i.e. frame[i] = s[j], then:

   a. Compare its position with farthest and if position is greater, then:

   farthest = position, ans = this_frame

4. If frame is not found in the later string, then:

   (i) RETURN this_frame

5. Return the optimal frame position,

   (i) RETURN ans;

Program Code:

```
#include<stdio.h>

int optimal(int s[], int frame[], int l, int n, int idx);

int main()

{

        int l,s[50],frame[10],n,avail,count=0,i,j,pos=0,full=0;

        printf("Enter the length of the string: ");

        scanf("%d",&l);

        printf("Enter the string: ");

        for(i=0; i<l; i++)

        scanf("%d",&s[i]);

        printf("Enter the number of frames: ");

        scanf("%d",&n);

        for(i=0; i<n; i++)

        frame[i]= -1;

        printf("\nString\t\t Page Frames\n");

        for(i=0; i<l; i++)

        {

        printf("%d\t\t",s[i]);

        avail = 0;
```

```
for(j=0; j<n; j++)
{
if(frame[j] == s[i])
{
        avail = 1;
        break;
}
}
if(avail == 0)
{
if(full < n)
{
        frame[pos] = s[i];
        pos++;
        full++;
}
else
{
        pos = optimal(s, frame, l, n, i+1);
        frame[pos] = s[i];
```

```
        }


        count++;
        for(j=0; j<n; j++)
        {
                if(frame[j] != -1)
              printf("%d\t",frame[j]);
        }
        }
        printf("\n");
        }
        printf("\nPage Fault is = %d\n", count);
        return 0;
}


int optimal(int s[], int frame[], int l, int n, int idx)
{
        int ans = 0, farthest = idx, i, j;
        for(i=0; i<n; i++)
        {
```

```
        for(j=idx; j<l; j++)
        {
        if(frame[i] == s[j])
        {
                if(j > farthest)
                {
                farthest = j;
                ans = i;
                }
        break;
              }
        }
        if(j == l)
        return i;
        }
        return ans;
}
```

Output:

Enter the length of the string: 6

Enter the string: 7 4 5 2 9 1

Enter the number of frames: 3

String	Page Frames		
7	7		
4	7	4	
5	7	4	5
2	2	4	5
9	9	4	5
1	1	4	5

Page Fault is = 6

Discussion: This program is working successfully on CodeBlocks and can return the element to replace in case of a page fault.

Experiment No. 11

Experiment Name: Write a program to simulate Least-recently-used (LRU) Page replacement algorithm.

Introduction: The LRU Page replacement algorithm replaces the page that has been used the least recently. It gives very close to optimal performance in general .

Theory: The Least Recently Used (LRU) replaces the most recently used page. Hence, any page that has been unused for a longer period of time than the others is replaced. The idea behind the algorithm is that the longer a page has been unused the more likely that it is not going to be used near future or likely will be unused. As a result, replacing that page is much more efficient.

Algorithm:

Step-1 : Input the reference string (s []) and number of frames.

Step-2 : Initialize all frames to be empty, frame[i] = -1;

Step-3 : For all pages i of the string, repeat Step-4 & Step-5:

Step-4 : If page is available in the frame, then:

(i) Go to Step-3 for next page.

Step-5 : Do the following:

(i) If frames are not full, then:

a. frame[pos] = s[i];

b. pos = next_frame;

(ii) Else:

a. CALL LRU() function.

b. Get the least recently used result for pos.

c. Store page in this frame,
frame[pos] = s[i].

(iii) Count page fault,
count++;

LRU()

Step-1 : Initialize,
ans = first_frame, oldest = previous_page;

Step-2 : For all the frames i, repeat Step-3.

Step-3 : Search for the frame from the previous_page to the first_page of the string.

(i) If found, i.e. frame[i] = s[j], then:

a. Compare its position with farthest and if position is greater, then:

lowest = position, ans = this_frame

Step-4 : Return the least recently used frame position,

(i) RETURN ans;

Program Code:

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

```
int LRU(int s[], int frame[], int l, int n, int idx);

int main()

{

        int l,s[50],frame[10],n,avail,count=0,i,j,pos=0,full=0;

        printf("Enter the length of the string: ");

        scanf("%d",&l);

        printf("Enter the string: ");

        for(i=0; i<l; i++)

        scanf("%d",&s[i]);

        printf("Enter the number of frames: ");

        scanf("%d",&n);

        for(i=0; i<n; i++)

        frame[i]= -1;


        printf("\nString\t\t Page Frames\n");

        for(i=0; i<l; i++)

        {

        printf("%d\t\t",s[i]);

        avail = 0;
```

```
for(j=0; j<n; j++)
{
if(frame[j] == s[i])
{
        avail = 1;
        break;
}
}
if(avail == 0)
{
if(full < n)
{
        frame[pos] = s[i];
        pos++;
        full++;
}
else
{
        pos = LRU(s, frame, l, n, i-1);
        frame[pos] = s[i];
```

```
        }
        count++;
        for(j=0; j<n; j++)
        {
                if(frame[j] != -1)
                printf("%d\t",frame[j]);
        }
        }
        printf("\n");
        }
        printf("\nPage Fault is = %d\n", count);
        return 0;
}
int LRU(int s[], int frame[], int l, int n, int idx)
{
        int ans = 0, oldest = idx, i, j;  //suppose frame-0 is ans, current index is the oldest
        for(i=0; i<n; i++)
        {
        for(j=idx; j>=0; j--)
        {
```

```
        if(frame[i] == s[j])
        {
                if(j < oldest)
                {
                oldest = j;
                ans = i;                        }
          break;
        }
        }
        }
        return ans;
}
```

Output:

Enter the length of the string: 6

Enter the string: 2 5 7 9 3 5

Enter the number of frames: 3

String	Page Frames		
2	2		
5	2	5	
7	2	5	7
9	9	5	7
3	9	3	7
5	9	3	5

Page Fault is = 6

Discussion: This program is running successfully. The experiment was a success as we are getting our desired output according to the algorithm. For the same input the fault rate of LRU is usually the closest to the optimum and as we can see LRU is also easily implementable overall.