Jatiya Kabi Kazi Nazrul Islam University

Trishal,Mymensingh,Bangladesh

Lab Report on

Course Name: Operating Systems Lab

Course Code: CSE-304

Submitted to:

Rubya Shaharin

Assistant Professor

Dept. of Computer Science & Engineering

Jatiya Kabi Kazi Nazrul Islam University

Submitted by:

Name: Md Rabiul Islam Rabbi

Roll: 19102006

Session: 2018-19

Dept. of Computer Science & Engineering

Jatiya Kabi Kazi Nazrul Islam University

Submission Date: 27-09-22

Index

Experiment No. 6

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

Introduction: Memory Management is one of the services provided by OS which is needed for Optimized memory usage of the available memory in a Computer System.

Theory: Best fit uses the best memory block based on the Process memory request. In best fit implementation the algorithm first selects the smallest block which can adequately fulfill the memory request by the respective process. Because of this memory is utilized optimally but as it compares the blocks with the requested memory size it increases the time requirement and hence slower than other methods. It suffers from Internal Fragmentation which simply means that the memory block size is greater than the memory requested by the process, then the free space gets wasted.

Algorithm:

Step-1: Get no. of Processes and no. of blocks.

Step-2: After that get the size of each block and process requests.

Step-3: Then select the best memory block that can be allocated using the above definition.

Step-4: Display the processes with the blocks that are allocated to a respective process.

Step-5: Value of Fragmentation is optional to display to keep track of wasted memory.

Step-6: Stop.

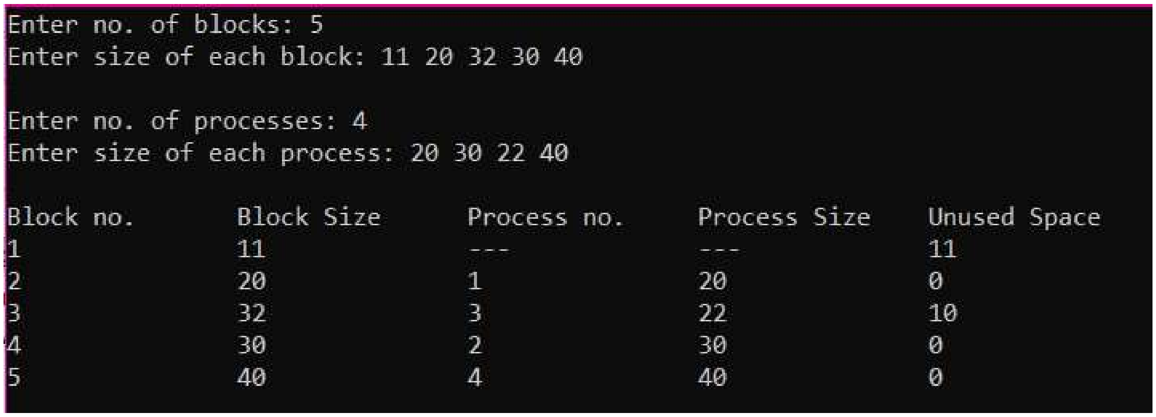
Program Code:

```
#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;            //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){       //flag[j] = 0 -> block must be empty;
            temp = bsize[j] - psize[i];          //temp contains difference of block size and process size
            if(temp >= 0 && temp < lowest){
                lowest = temp;
                id = j;              //storing the process no in id variable, this process has lowest difference
            }
        }
    }
    alloc[id] = i;               //block j contains  i-no. process now
    flags[id] = 1;                           //flag=1, means this block is not empty
```

```
        lowest = 9999;            //set impossible lowest for next process
        id = 9;                   //set impossible id for next process
    }
    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)         //print process info only if block contains process
            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: 5
Enter size of each block: 11 20 32 30 40

Enter no. of processes: 4
Enter size of each process: 20 30 22 40

Block no.       Block Size      Process no.     Process Size    Unused Space
1               11              ---             ---             11
2               20              1               20              0
3               32              3               22              10
4               30              2               30              0
5               40              4               40              0
```

Discussion: This is a successful program running on CodeBlocks IDE that finds the unused space depending on the block size and process size.

Experiment No. 7

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

Introduction: The processes need empty memory slots during processing time. This memory is allocated to the processes by the operating system which decides depending on the free memory and the demanded memory by the process in execution.

Theory: Worst fit works in the following way, for any given process P_n. The algorithms searches sequentially starting from first memory block and searches for the memory block that fulfills the following condition –

- Can accommodate the process size
- Leaves the largest wasted space (fragmentation) after the process is allocated to given memory block

Algorithm:

Step-1: *Input memory block with a size.*

Step-2: *Input process with size.*

Step-3: *Initialize by selecting each process to find the maximum block size that can be assigned to the current process.*

Step-4: *If the condition does not fulfill, they leave the process.*

Step-5: *If the condition is not fulfilled, then leave the process and check for the next process.*

Step-6: *Stop.*

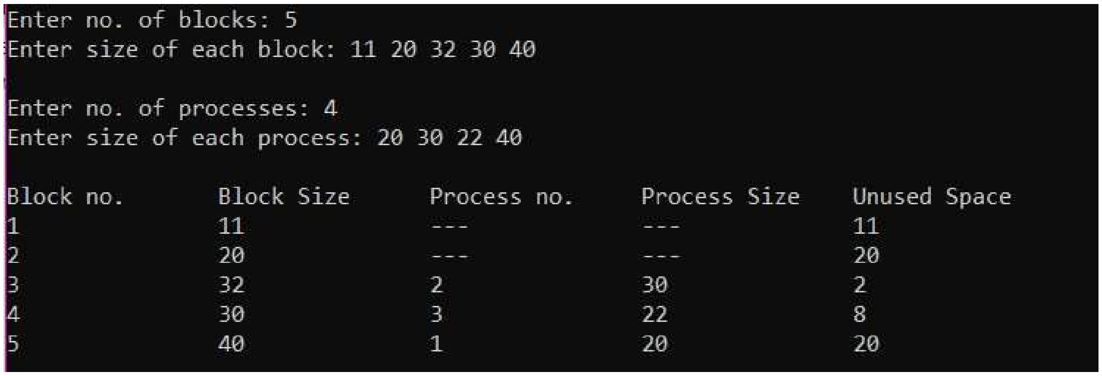
Program Code:

```
#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){       //flag[j] = 0 -> block must be empty;

            temp = bsize[j] - psize[i];         //temp contains difference of block size and process size

            if(temp >= 0 && temp > highest){

                highest = temp;

                id = j;            //storing the process no in id variable, this process has highest difference

            }

        }

    }
```

```
        alloc[id] = i;              //block j contains  i-no. process now
        flags[id] = 1;                         //flag=1, means this block is not empty
        highest = -9999;            //set impossible highest for next process
        id = 9;                              //set impossible id for next process
    }
    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)                    //print process info only if block contains process
            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: 5
Enter size of each block: 11 20 32 30 40

Enter no. of processes: 4
Enter size of each process: 20 30 22 40

Block no.       Block Size      Process no.     Process Size    Unused Space
1               11              ---             ---             11
2               20              ---             ---             20
3               32              2               30              2
4               30              3               22              8
5               40              1               20              20
```

Discussion: This is a successful program running on CodeBlocks IDE that finds the unused space depending on the block size and process size.

Experiment No. 8

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

Introduction: The operating system uses different memory management schemes to optimize memory/resource block allocation to different processes. We will look at one of such memory allocation processes in OS called First Fit in OS.

Theory: Whenever a process (p1) comes with memory allocation request the following happens -

- OS sequentially searches available memory blocks from the first index
- Assigns the first memory block large enough to accommodate process

Whenever a new process P2 comes, it does the same thing. Search from the first index again.

The First Fit memory allocation checks the empty memory blocks in a sequential manner. It means that the memory Block which found empty in the first attempt is checked for size. But if the size is not less than the required size then it is allocated.

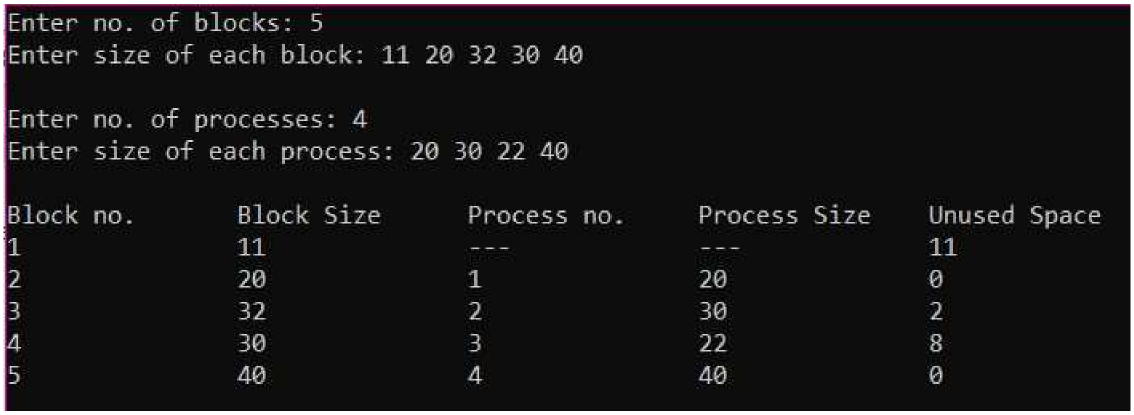
Algorithm:

```
Step-1: Get no. of Processes and no. of blocks.

Step-2: After that get the size of each block and process requests.

Step-3: Now allocate processes

        if(block size >= process size)

            //allocate the process
        else
            //move on to next block

Step-4: Display the processes with the blocks that are allocated to a respective process.

Step-5: Stop.
```

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;           //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 && bsize[j] >= psize[i]){      //flag[j] = 0 -> block must be empty;
                alloc[j] = i;                   //block j contains  i-no. process now
                flags[j] = 1;                        //flag=1, means this block is not empty
            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)                   //print process info only if block contains process
            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: 5
Enter size of each block: 11 20 32 30 40

Enter no. of processes: 4
Enter size of each process: 20 30 22 40

Block no.       Block Size      Process no.     Process Size    Unused Space
1               11              ---             ---             11
2               20              1               20              0
3               32              2               30              2
4               30              3               22              8
5               40              4               40              0
```

Discussion: This is a successful program running on CodeBlocks IDE that finds the unused space depending on the block size and process size.

Experiment No. 9

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

Introduction: The operating system uses the method of paging for memory management. This method involves page replacement algorithms to make a decision about which pages should be replaced when new pages are demanded. The demand occurs when the operating system needs a page for processing, and it is not present in the main memory. The situation is known as a page fault.

Theory: In this situation, the operating system replaces an existing page from the main memory by bringing a new page from the secondary memory.

In such situations, the FIFO method is used, which is also refers to the First in First Out concept. This is the simplest page replacement method in which the operating system maintains all the pages in a queue. Oldest pages are kept in the front, while the newest is kept at the end. On a page fault, these pages from the front are removed first, and the pages in demand are added.

Algorithm:

Step-1. Start to traverse the pages.

Step-2. If the memory holds fewer pages, then the capacity else goes to step 5.

Step-3. Push pages in the queue one at a time until the queue reaches its maximum capacity or all page requests are fulfilled.

Step-4. If the current page is present in the memory, do nothing.

Step-5. Else, pop the topmost page from the queue as it was inserted first.

Step-6. Replace the topmost page with the current page from the string.

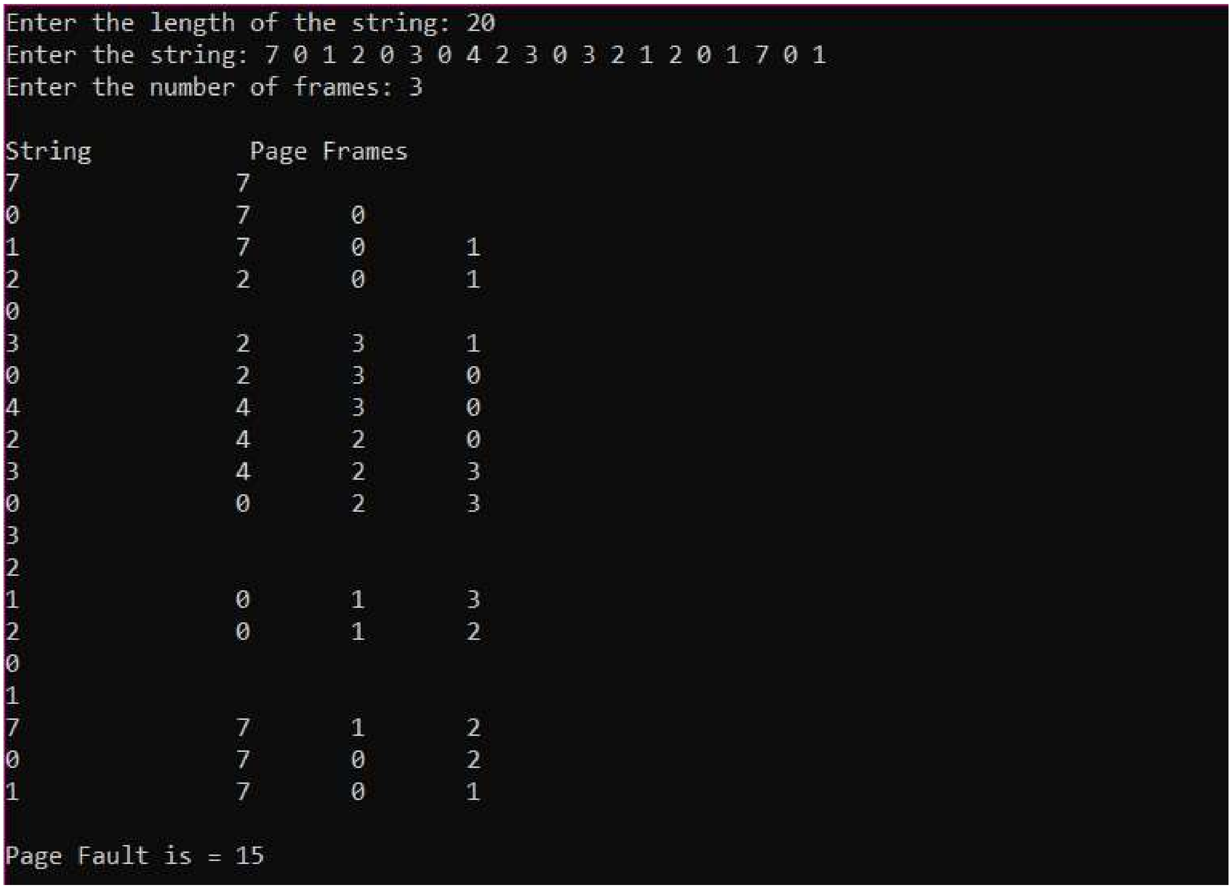
Step-7. Increment the page faults.

Step-8. Stop

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;           //Initially frame is empty, -1 means empty

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

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

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

        avail = 0;          //suppose page is not available in the frame

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

            if(frame[j] == s[i]) {      //page matches with frame

                avail = 1;              //This page is available in frame, no need to page fault

                break;

            }

        }
        if(avail == 0)  {           //Only if page is not in the frame
            frame[pos] = s[i];      //Store page in current frame position
            pos = (pos+1) % n;      //pos is increased circular like a queue, so first in first out will occur
            count++;                //counting the number of page fault
            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: 20
Enter the string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Enter the number of frames: 3

String           Page Frames
7               7
0               7       0
1               7       0       1
2               2       0       1
0
3               2       3       1
0               2       3       0
4               4       3       0
2               4       2       0
3               4       2       3
0               0       2       3
3
2
1               0       1       3
2               0       1       2
0
1
7               7       1       2
0               7       0       2
1               7       0       1

Page Fault is = 15
```

Discussion: This is a successful program running on CodeBlocks IDE that finds the number of page fault.

Experiment No. 10

Experiment Name: Write a C program to simulate Optimal page replacement algorithm.

Introduction: Optimal page replacement algorithm says that if page fault occurs then that page should be removed that will not be used for maximum time in future. It is also known as clairvoyant replacement algorithm or Bélády's optimal page replacement policy.

Theory: Optimal Page Replacement algorithm is the best page replacement algorithm as it gives the least number of page faults. It is also known as OPT, clairvoyant replacement algorithm, or Belady's optimal page replacement policy.

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future, i.e., the pages in the memory which are going to be referred farthest in the future are replaced.

This algorithm was introduced long back and is difficult to implement because it requires future knowledge of the program behaviour. However, it is possible to implement optimal page replacement on the second run by using the page reference information collected on the first run.

.

Algorithm:

Step-1: *Push the first page in the stack as per the memory demand.*

Step-2: *Push the second page as per the memory demand.*

Step-3: *Push the third page until the memory is full.*

Step-4: *As the queue is full, the page which is least recently used is popped.*

Step-5: *repeat step 4 until the page demand continues and until the processing is over.*

Step-6: *Terminate the program.*

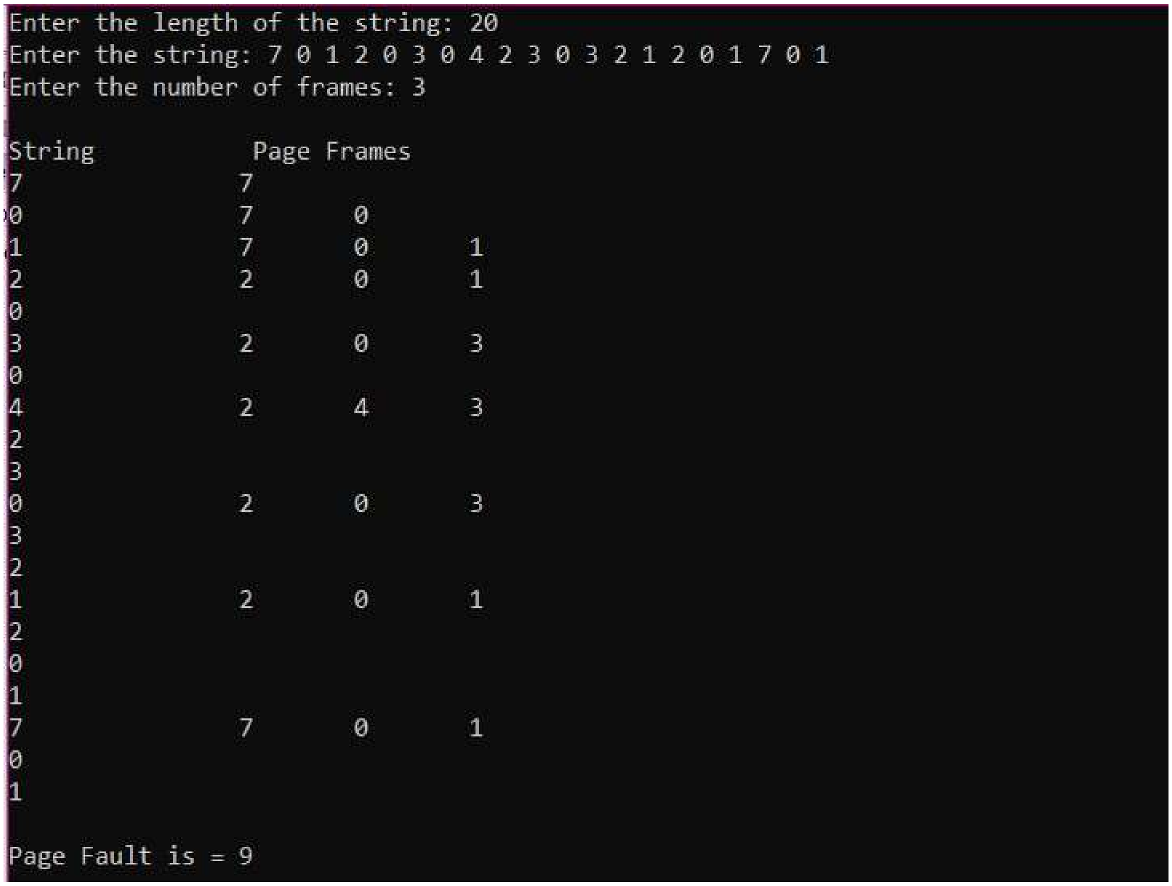
Program Code:

```
#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;          //Initially frame is empty, -1 means empty

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

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

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

      avail = 0;          //suppose page is not available in the frame

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

         if(frame[j] == s[i]){        //page matches with frame

            avail = 1;             //This page is available in frame, no need to page fault

            break;

         }
```

```
        }
        if(avail == 0){             //Only if page is not in the frame
            if(full < n)  {         //Frames are not full, so simple method
                frame[pos] = s[i];      //Store page in current frame position
                pos++;                      //Move to next position
                full++;                 //Full frame increased
            }
            else{
                pos = optimal(s, frame, l, n, i+1);     //search for optimal frame
                frame[pos] = s[i];                          //store page in optimal pos frame
            }

            count++;                //counting the number of page fault
            for(j=0; j<n; j++){
                if(frame[j] != -1)      //printing frame who don't have -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;  //suppose frame-0 is ans, current index is the farthest
    for(i=0; i<n; i++){
        for(j=idx; j<l; j++){
            if(frame[i] == s[j]){        //this frame is found in the string
                if(j > farthest) {     //if this page position is the farthest
                    farthest = j;          //store this page position as farthest
                    ans = i;           //store this frame as ans
                }
               break;                  //break when a frame matches with the string
            }
        }
        if(j == l)                     //This frame is not found in the string
            return i;                      //This is the optimal frame, so return
    }
    return ans;                    //return the optimal frame
}
```

Output:



```
Enter the length of the string: 20
Enter the string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Enter the number of frames: 3

String           Page Frames
7               7
0               7       0
1               7       0       1
2               2       0       1
0
3               2       0       3
0
4               2       4       3
2
3
0               2       0       3
3
2
1               2       0       1
2
0
1
7               7       0       1
0
1

Page Fault is = 9
```

Discussion: This is a successful program running on CodeBlocks IDE that finds the number of page fault.

Experiment No. 11

Experiment Name: Write a C program to simulate Least-Recently-Used (LRU) page replacement algorithm.

Introduction: Least Recently Used (LRU) page replacement algorithm works on the concept that the pages that are heavily used in previous instructions are likely to be used heavily in next instructions. And the page that are used very less are likely to be used less in future. Whenever a page fault occurs, the page that is least recently used is removed from the memory frames. Page fault occurs when a referenced page in not found in the memory frames.

Theory: Least Recently Used page replacement algorithm keeps track of page usage over a short period of time. It works on the idea that the pages that have been most heavily used in the past are most likely to be used heavily in the future too.. Least Recently Used (LRU) algorithm is a page replacement technique used for memory management. According to this method, the page which is least recently used is replaced. Therefore, in memory, any page that has been unused for a longer period of time than the others is replaced.

Algorithm:

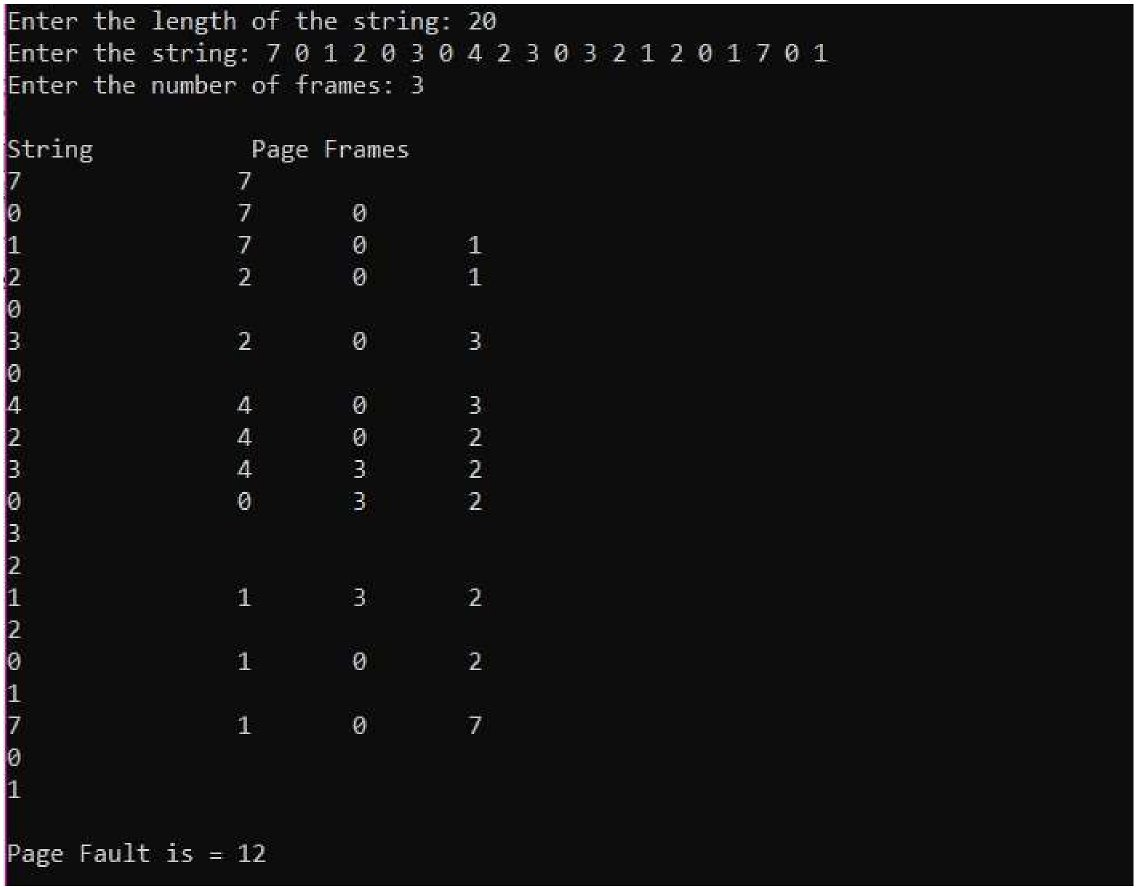
```
Step-1:.  Start the process
Step-2:  Declare the size
Step-3:. Get the number of pages to be inserted
Step-4:  Get the value
Step-5:  Declare counter and stack
Step-6: . Select the least recently used page by counter value
Step-7:  Stack them according the selection.
Step-8:   Display the values
Step-9:  Stop the process
```

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;           //Initially frame is empty, -1 means empty

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

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

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

        avail = 0;          //suppose page is not available in the frame

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

            if(frame[j] == s[i])        //page matches with frame

            {

                avail = 1;              //This page is available in frame, no need to page fault

                break;
            }
        }
        if(avail == 0)  {           //Only if page is not in the frame
            if(full < n) {          //Frames are not full, so simple method
                frame[pos] = s[i];      //Store page in current frame position
                pos++;                  //Move to next position
                full++;                 //Full frame increased
            }
            else{
                pos = LRU(s, frame, l, n, i-1);     //search for least recent frame
                frame[pos] = s[i];                  //store page in least recent pos frame
            }
            count++;                //counting the number of page fault
            for(j=0; j<n; j++){
                if(frame[j] != -1)      //printing frame which don't have -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]){         //this frame is found in the string
                if(j < oldest)        //if this page position is the oldest
                {
                    oldest = j;           //store this page position as oldest
                    ans = i;            //store this frame as ans
                }
               break;                   //break when a frame matches with the string
            }
        }
    }
    return ans;                     //return the Least Recent frame
}
```

Output:



```
Enter the length of the string: 20
Enter the string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Enter the number of frames: 3

String           Page Frames
7               7
0               7       0
1               7       0       1
2               2       0       1
0
3               2       0       3
0
4               4       0       3
2               4       0       2
3               4       3       2
0               0       3       2
3
2
1               1       3       2
2
0               1       0       2
1
7               1       0       7
0
1

Page Fault is = 12
```

Discussion: This is a successful program running on CodeBlocks IDE that finds the number of page fault.