**Practical No. 6**

**Aim:** To implement page replacement algorithm for block replacement strategy

**Theory:**

In a [computer](https://en.wikipedia.org/wiki/Computer) [operating system](https://en.wikipedia.org/wiki/Operating_system) that uses [paging](https://en.wikipedia.org/wiki/Paging) for [virtual memory](https://en.wikipedia.org/wiki/Virtual_memory) [management](https://en.wikipedia.org/wiki/Memory_management), page replacement algorithms decide which memory pages to page out, sometimes called swap out, or write to disk, when a [page](https://en.wikipedia.org/wiki/Page_(computer_memory)) of memory needs to be allocated. [Page replacement](https://en.wikipedia.org/wiki/Paging) happens when a requested page is not in memory ([page fault](https://en.wikipedia.org/wiki/Page_fault)) and a free page cannot be used to satisfy the allocation, either because there are none, or because the number of free pages is lower than some threshold.

When the page that was selected for replacement and paged out is referenced again it has to be paged in (read in from disk), and this involves waiting for I/O completion. This determines the *quality* of the page replacement algorithm: the less time waiting for page-ins, the better the algorithm. A page replacement algorithm looks at the limited information about accesses to the pages provided by hardware, and tries to guess which pages should be replaced to minimize the total number of page misses, while balancing this with the costs (primary storage and processor time) of the algorithm itself.

**Optimal Page replacement algorithm**

The optimal page replacement algorithm is an algorithm that works as follows: when a page needs to be swapped in, the [operating system](https://en.wikipedia.org/wiki/Operating_system) swaps out the page whose next use will occur farthest in the future. For example, a page that is not going to be used for the next 6 seconds will be swapped out over a page that is going to be used within the next 0.4 seconds.

This algorithm cannot be implemented in a general purpose operating system because it is impossible to compute reliably how long it will be before a page is going to be used, except when all software that will run on a system is either known beforehand and is amenable to static analysis of its memory reference patterns, or only a class of applications allowing run-time analysis.

**Least Recently Used (LRU) algorithm**

The least recently used (LRU) page replacement algorithm, though similar in name to NRU, differs in the fact that LRU keeps track of page usage over a short period of time, while NRU just looks at the usage in the last clock interval. LRU works on the idea that pages that have been most heavily used in the past few instructions are most likely to be used heavily in the next few instructions too. While LRU can provide near-optimal performance in theory (almost as good as [adaptive replacement cache](https://en.wikipedia.org/wiki/Adaptive_replacement_cache)), it is rather expensive to implement in practice.

**First-in, first-out algorithm**

The simplest page-replacement algorithm is a FIFO algorithm. The first-in, first-out (FIFO) page replacement algorithm is a low-overhead algorithm that requires little bookkeeping on the part of the [operating system](https://en.wikipedia.org/wiki/Operating_system). The idea is obvious from the name – the operating system keeps track of all the pages in memory in a queue, with the most recent arrival at the back, and the oldest arrival in front. When a page needs to be replaced, the page at the front of the queue (the oldest page) is selected. While FIFO is cheap and intuitive, it performs poorly in practical application. Thus, it is rarely used in its unmodified form. This algorithm experiences [Bélády's anomaly](https://en.wikipedia.org/wiki/B%C3%A9l%C3%A1dy%27s_anomaly). In simple words, on a page fault, the frame that has been in memory the longest is replaced.

**Program:**

**Page.C**

#include<stdio.h>

int n,pg[30],fr[10];

void fifo();

void optimal();

void lru();

void main()

{

int i,ch;

printf("\nEnter total number of pages:");

scanf("%d",&n);

printf("\nEnter sequence:");

for(i=0;i<n;i++) //accepting sequence

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

do

{

printf("\n\tMENU\n");

printf("\n1)FIFO");

printf("\n2)OPTIMAL");

printf("\n3)LRU");

printf("\n4)Exit");

printf("\nEnter your choice:");

scanf("%d",&ch);

switch(ch)

{

case 1: fifo();

break;

case 2: optimal();

break;

case 3: lru();

break;

}

}while(ch!=4);

getchar();

}

void fifo()

{

int i,f,r,s,count,flag,num,psize;

f=0;

r=0;

s=0;

flag=0;

count=0;

printf("\nEnter size of page frame:");

scanf("%d",&psize);

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

{

fr[i]=-1;

}

while(s<n)

{

flag=0;

num=pg[s];

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

{

if(num==fr[i])

{

s++;

flag=1;

break;

}

}

if(flag==0)

{

if(r<psize)

{

fr[r]=pg[s];

r++;

s++;

count++;

}

else

{

if(f<psize)

{

fr[f]=pg[s];

s++;

f++;

count++;

}

else

f=0;

}

}

printf("\n");

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

{

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

}

}

printf("\nPage Faults=%d",count);

getchar();

}

void optimal()

{

int count[10],i,j,k,fault,f,flag,temp,current,c,dist,max,m,cnt,p,x;

fault=0;

dist=0;

k=0;

printf("\nEnter frame size:");

scanf("%d",&f);

//initilizing distance and frame array

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

{

count[i]=0;

fr[i]=-1;

}

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

{

flag=0;

temp=pg[i];

for(j=0;j<f;j++)

{

if(temp==fr[j])

{

flag=1;

break;

}

}

if((flag==0)&&(k<f))

{

fault++;

fr[k]=temp;

k++;

}

else if((flag==0)&&(k==f))

{

fault++;

for(cnt=0;cnt<f;cnt++)

{

current=fr[cnt];

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

{

if(current!=pg[c])

count[cnt]++;

else

break;

}

}

max=0;

for(m=0;m<f;m++)

{

if(count[m]>max)

{

max=count[m];

p=m;

}

}

fr[p]=temp;

}

printf("\n");

for(x=0;x<f;x++)

{

printf("%d\t",fr[x]);

}

}

printf("\nTotal number of faults=%d",fault);

getchar();

}

void lru()

{

int count[10],i,j,k,fault,f,flag,temp,current,c,dist,max,m,cnt,p,x;

fault=0;

dist=0;

k=0;

printf("\nEnter frame size:");

scanf("%d",&f);

//initilizing distance and frame array

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

{

count[i]=0;

fr[i]=-1;

}

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

{

flag=0;

temp=pg[i];

for(j=0;j<f;j++)

{

if(temp==fr[j])

{

flag=1;

break;

}

}

if((flag==0)&&(k<f))

{

fault++;

fr[k]=temp;

k++;

}

else if((flag==0)&&(k==f))

{

fault++;

for(cnt=0;cnt<f;cnt++)

{

current=fr[cnt];

for(c=i;c>0;c--)

{

if(current!=pg[c])

count[cnt]++;

else

break;

}

}

max=0;

for(m=0;m<f;m++)

{

if(count[m]>max)

{

max=count[m];

p=m;

}

}

fr[p]=temp;

}

printf("\n");

for(x=0;x<f;x++)

{

printf("%d\t",fr[x]);

}

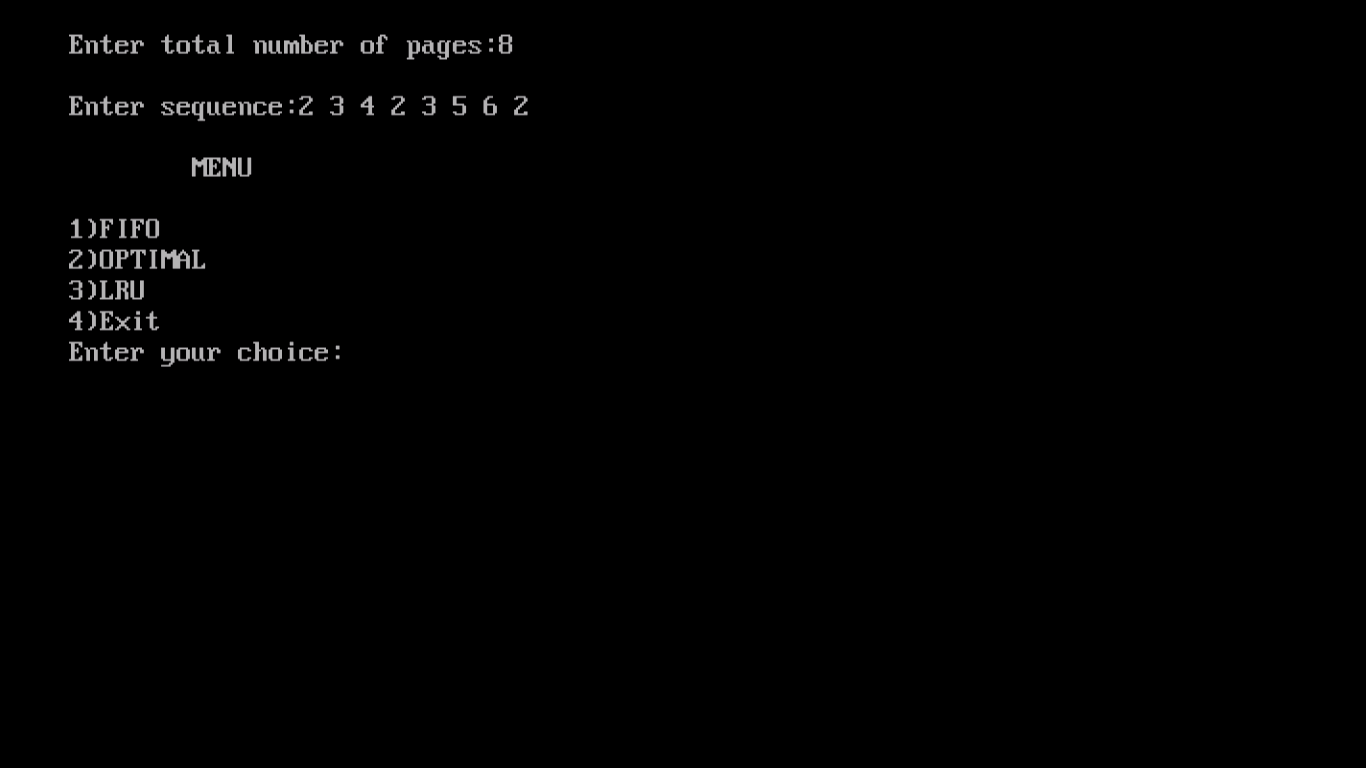
}

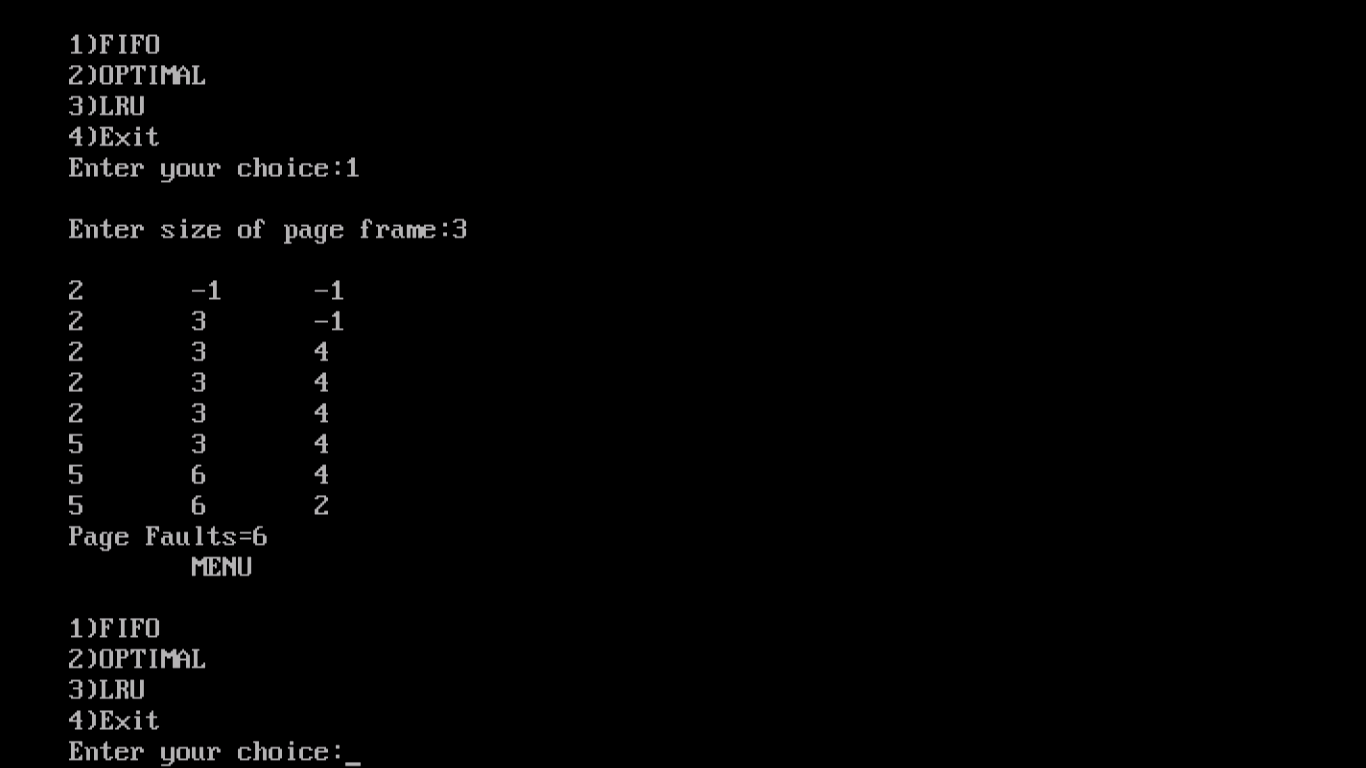
printf("\nTotal number of faults=%d",fault);

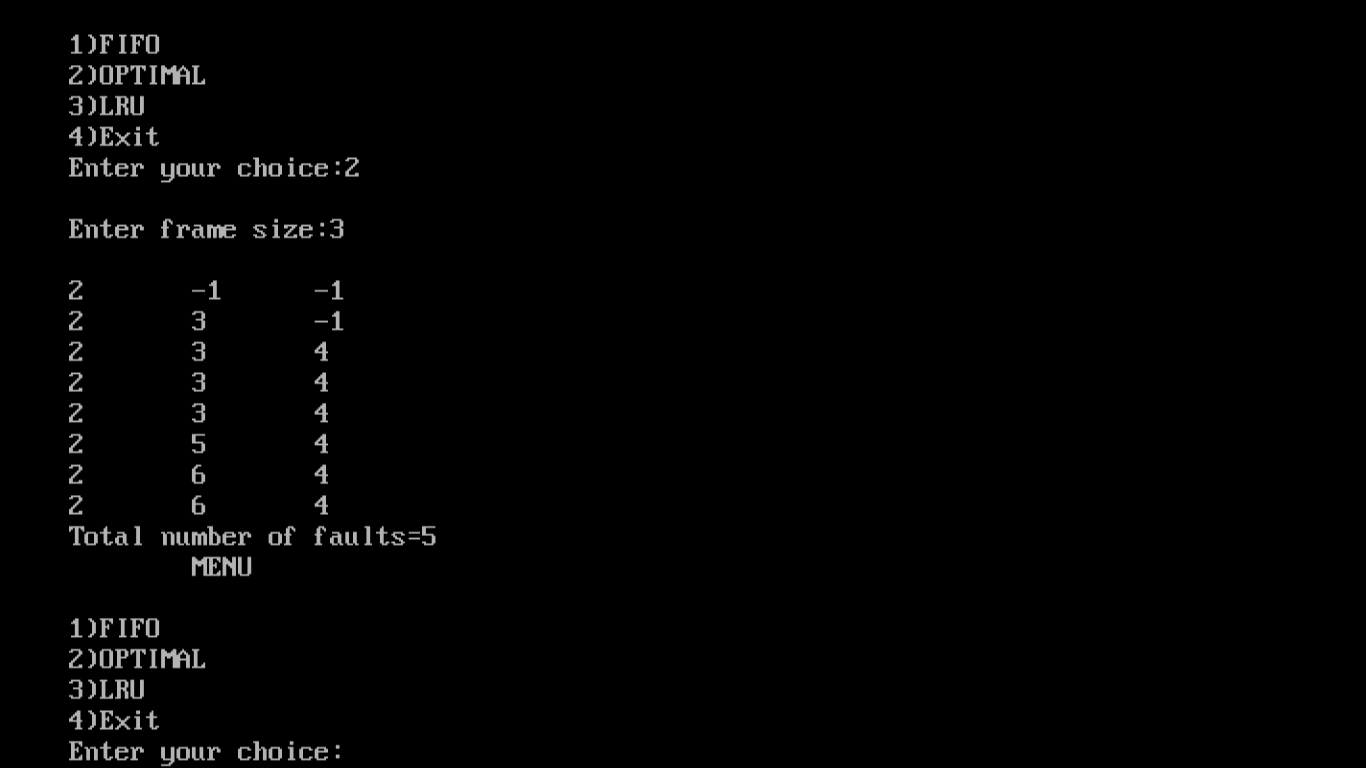
getchar();

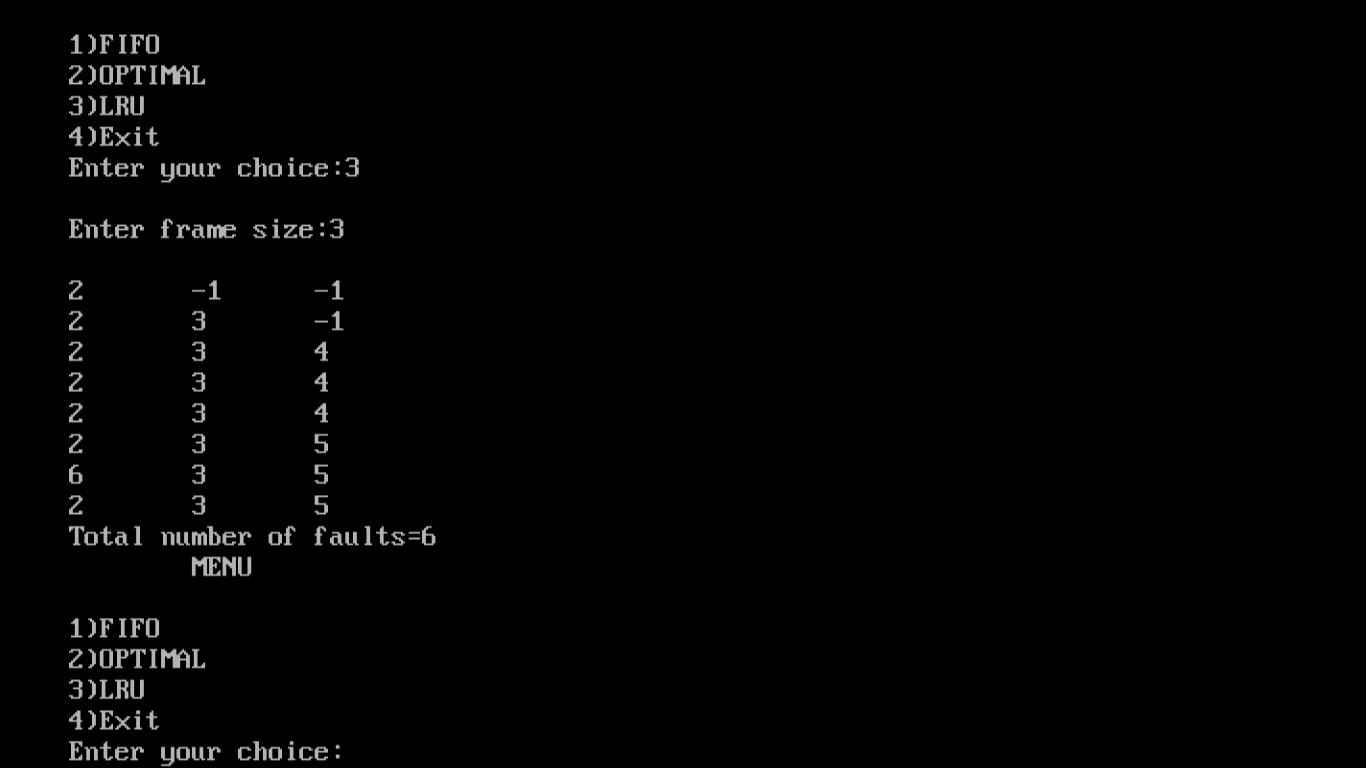
}

**Output:**

****

****

****

****

**Conclusion:** Various page replacement algorithms have been implemented for block replacement strategy.