**PAGE REPLACEMENT**

**ACKNOWLEDGEMENT**

I would like to express my special thanks of gratitude to my professor Prof. ANURADHA J who gave me the golden opportunity to do this wonderful project on the topic “ PAGE REPLACEMENT ALGORITHEM“ which also helped me in doing a lot of Research and we came to know about so many new things we are really thankful to them.  
Secondly we would also like to thank our friends and teachers who helped us a lot in finalizing this project within the limited time frame.

**INTRODUCTION**

**What are pages?**

A **page** is a fixed-length contiguous block of virtual memory, described by a single entry in the page table. It is the smallest unit of data for memory management in a virtual memory operating system. Similarly, a **page frame** is the smallest fixed-length contiguous block of physical memory into which memory pages are mapped by the operating system.

A transfer of pages between main memory and an auxiliary store, such as a hard disk drive, is referred to as paging or swapping.

**What is page replacement?**

In a computer operating system that uses paging for virtual memory management, **page replacement algorithms** decide which memory pages to page out (swap out, write to disk) when a page of memory needs to be allocated. Paging happens when a page fault occurs 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.

**Available page replacement techniques**

1. First-in, first-out

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. 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.

### Second chance

A modified form of the FIFO page replacement algorithm, known as the Second-chance page replacement algorithm, fares relatively better than FIFO at little cost for the improvement. It works by looking at the front of the queue as FIFO does, but instead of immediately paging out that page, it checks to see if its referenced bit is set. If it is not set, the page is swapped out. Otherwise, the referenced bit is cleared, the page is inserted at the back of the queue (as if it were a new page) and this process is repeated. This can also be thought of as a circular queue. If all the pages have their referenced bit set, on the second encounter of the first page in the list, that page will be swapped out, as it now has its referenced bit cleared. If all the pages have their reference bit cleared, then second chance algorithm degenerates into pure FIFO.

### Least recently used

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), it is rather expensive to implement in practice.

1. **Optimal**

When a page needs to be swapped in, the 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.

And some other techniques.

IMPLEMENTATION

Our project was driven by the fact that there was a need for an algorithm which could predict the next possible frame to replaced .Our algorithm is different from others available algorithm as we are trying to predict the upcoming frames form the available frames at that instance, whereas leaving optimal all the other page replacement algorithm cannot predict the outcomes and as per optimal page replacement it is not a valid solution for the real time scenario.

**Source code**

#include<iostream>

#include<algorithm>

using namespace std;

int pg[10000],n,fr,pre[10][10][4];

void set\_predict(int,int);

int\* predict(int);

void calibrate();

//void calibrate(int);

main()

{

int fault=0;

cout<<"Enter the no. of pages:";

cin>>n;

cout<<"\nEnter the size of the frame:";

cin>>fr;

cout<<"\n";

cout<<"enter the page sequence -\n";

for(int k=0;k<n;k++)

cin>>pg[k];

calibrate();

int frame[fr];

for(int i=0;i<fr;i++)

{

frame[i]=pg[i];

if(i!=0)

set\_predict(pg[i-1],pg[i]);

fault++;

}

for(int t=fr;t<n;t++)

{

int temp=pg[t],found=0;

set\_predict(pg[t-1],temp);

for(int i=0;i<fr;i++)

if(frame[i]==temp)

found++;

if(found==0)

{

int loc[3];

loc[0]=0;

loc[1]=0;

loc[2]=0;

int \*a=predict(temp);

for(int i=0;i<fr;i++)

if(a[0]==frame[i] || a[1]==frame[i])

{

loc[i]=1;

}

int mini=0;

for(int i=0;i<fr;i++)

if(loc[i]==0)

mini=i;

frame[mini]=temp;

fault++;

}

}

cout<<endl<<fault;

}

void calibrate()

{

for(int i=0;i<9;i++)

for(int j=0;j<9;j++)

{

pre[i][j][0]=i+1;

pre[i][j][1]=j+1;

pre[i][j][2]=0;

}

}

void set\_predict(int a,int b)

{

int temp1,temp2;

for(int i=0;i<9;i++)

for(int j=0;j<9;j++)

{

if(pre[i][j][0]==a && pre[i][j][1]==b)

{

temp1=i;

temp2=j;

break;

}

}

pre[temp1][temp2][2]+=1;

}

int\* predict(int a)

{

int loc[2];

int temp1[9][3],temp2[9];

for(int j=0;j<9;j++)

{

temp1[j][1]=pre[a][j][1];

temp1[j][2]=pre[a][j][2];

temp2[j]=pre[a][j][2];

}

sort(temp2,temp2+9);

for(int j=0;j<9;j++)

{

if(temp2[8]==temp1[j][2])

loc[0]=j;

if(temp2[7]==temp1[j][2])

loc[1]=j;

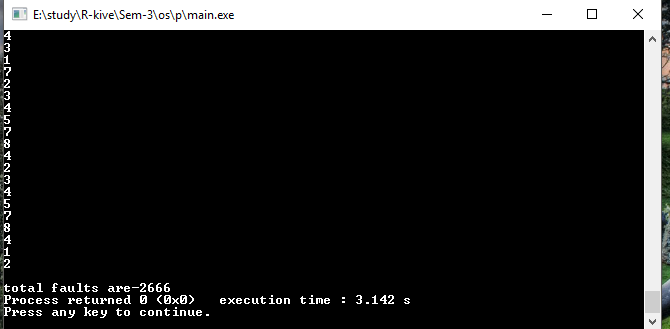
}

return loc;

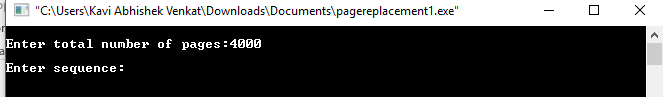
}

Output

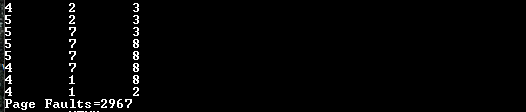




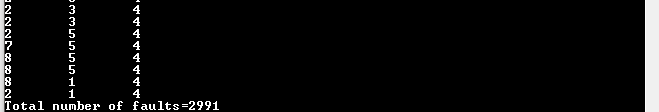
**Comparison with different algorithms**



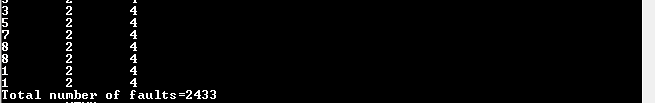
FIFO



LRU



OPTIMAL



**Techniques used**

Our algorithm predicts the pattern form the previous frames and while replacing the page from the frame it predicts the next two numbers to occur and it replaces the frame which is not been predicted and with increase in the number of input our program will learn more about the pattern add will work better with large data.

CONCLUSION

After finishing this project we came to the conclusion that our project was affective and it was working better that previous available algorithms, as our algorithm was able to give less number of page faults and with the increase in the number of pages our algorithms will work more efficient which is a better real time solution for the purpose of reducing page faults.