System Programming and Operating Systems Lab

ASSIGNMENT 14

Name: Shrirang Mhalgi
Roll No: 322008
Batch: B1

1 Date of Completion:

03/04/2019

2 Aim:

To implement a Java program using OOP features for paging simulation using : Least Frequently Used and Optimal Algorithm.

3 Objectives:

To understand the working of LRU Algorithm and Optimal Algorithm for Page Simulation.

4 Theory

In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

4.1 Least Recently Used Algorithm

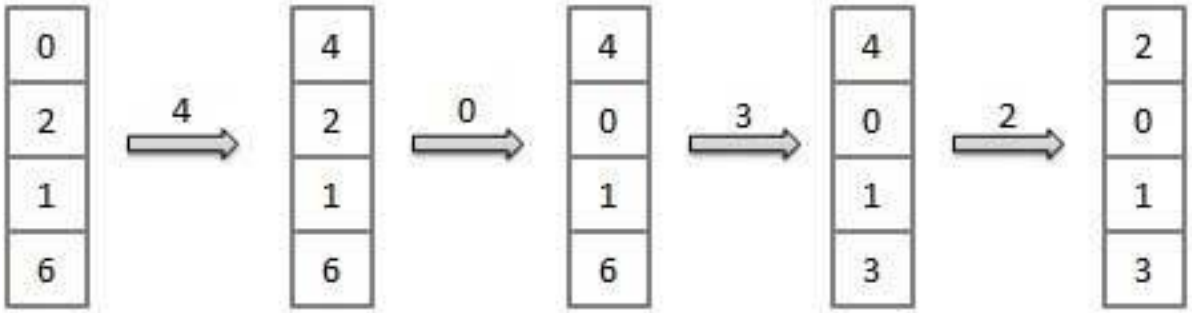
In Least Recently Used (LRU) algorithm is a Greedy algorithm where the page to be replaced is least recently used. The idea is based on locality of reference, the least recently used page is not likely. Each replacement algorithm has its own merits and demerits. In general, the algorithm which maintains a high "hit ratio" is considered to be effective, and is suitable for implementation. The term "hit" means the presence of the same page which is to be inserted. Hence, when a page hit occurs, no replacement of the pages is needed to be carried out, and hence, no time is wasted between processes. The Least Recently Used page replacement algorithm replaces those pages first which are the oldest, and have been the least referred to. To implement this a algorithm, a counter called an "age bit" is maintained, which keeps a track of which page was referred and when it was referred. This ensures that the page which was the least recently used is discarded to make room for the incoming page. Since its implementation requires extra hardware, this is somewhat complex to implement. A good approximation to the optimal algorithm is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few. Conversely, pages that

have not been used for ages will probably remain unused for a long time. This idea suggests a realizable algorithm: when a page fault occurs, throw out the page that has been unused for the longest time. This strategy is called LRU (Least Recently Used) paging. Although LRU is theoretically realizable, it is not cheap. To fully implement LRU, it is necessary to maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear. The difficulty is that the list must be updated on every memory reference. Finding a page in the list, deleting it, and then moving it to the front is a very time consuming operation, even in hardware.

Example:

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x x x



Fault Rate = 8 / 12 = 0.67

4.1.1 Advantages of LRU Page Replacement Algorithm :-

i) It is amenable to full statistical analysis.
ii) Never suffers from Belady's anomaly.

4.1.2 Disadvantages of LRU Page Replacement Algorithm :-

i) Its implementation is not very easy.
ii) Its implementation may require substantial hardware assistance.

4.2 Optimal Algorithm

In this algorithm, OS replaces the page that will not be used for the longest period of time in future.The idea is simple, for every reference we do following :

1. If referred page is already present, increment hit count.
2. If not present, find if a page that is never referenced in future.
3. If such a page exists, replace this page with new page. If no such page exists, find a page that is referenced farthest in future. Replace this page with new page.

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. Despite this limitation, algorithms exist[citation needed] that can offer near-optimal performance — the operating system keeps track of all pages referenced by the program, and it uses those data to decide which pages to swap in and out on subsequent runs. This algorithm can offer near-optimal performance, but not on the first run of a program, and only if the program's memory reference pattern is relatively consistent each time it runs.

Example:

Let us consider page reference string 7 0 1 2 0 3 0 4 2 3 0 3 2 and 4 page slots.
Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> 4 Page faults.
0 is already there so —> 0 Page fault.
when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future.—>1 Page fault.
0 is already there so —> 0 Page fault..
4 will takes place of 1 —> 1 Page Fault.

4.2.1 Advantages of Optimal Page Replacement Algorithm :-

i) Lowest page fault rate.
ii) Never suffers from Belady's anomaly.
iii) Twice as good as FIFO Page Replacement Algorithm.

4.2.2 Disadvantages Optimal Page Replacement Algorithm :-

i) Difficult to implement.
ii) It needs forecast i.e. Future knowledge.

5 Code:

LRU.java:

```
package Replacement;
import java.util.*;
class LRU{
ArrayList<PageReference> pages;
intframes;
PageReference p,p1,f1;
public int pagehit,pagefault;
publicLRU(ArrayList<PageReference> pages,intframes)
{
this.pages=pages;
this.frames=frames;
this.pagehit=0;
this.pagefault=0;
}
publicvoid solve(){
ArrayList<PageReference> f=new ArrayList<>(frames);//frame to store pages
for(int i=0;i<pages.size();i++)
{
p=pages.get(i);//page i
System.out.println("PageReference :"+p.page);


if(!f.contains(p) &&f.size()<frames)
{
/*Condition 1
if frame doesnt contains page i
and frame is not full
increase the backdistance ofevery element in frame by 1
and add page i
*/
for(int j=0;j<f.size();j++)
{
p1=f.remove(j);
p1.distance++;
f.add(p1);
}
f.add(p);

}
else if(!f.contains(p) &&f.size()>=frames)
{
/*Sort ArrayList f so that page with highest backdistance comes first */
Collections.sort(f,new Comparator<PageReference>(){
public intcompare(PageReference p1,PageReference p2)
{
```

```
return p2.distance-p1.distance;
}
});
/*Condition 2
if frame doesnt contains page i
and frame is full
then
remove the page with longest backdistance
increase the backdistance of every other element in frame by 1
and add page i
*/
f.remove(0);
for(int j=0;j<f.size();j++)
{
p1=f.remove(j);
p1.distance++;
f.add(p1);
}
f.add(p);
}

else if(f.contains(p))
{
/*Condition 3
if frame contains page i
then
set the backdistance of page to 0
increase the backdistance of every element in frame by 1
*/
f.remove(p);
for(int j=0;j<f.size();j++)
{
p1=f.remove(j);
p1.distance++;
f.add(p1);
}
p.distance=0;
f.add(p);
pagehit++;
}
System.out.println(f);
System.out.println(" Pagehit : "+pagehit);
}
}
}
```

Optimal.java:

```
package Replacement;
import java.util.*;
class Optimal{
ArrayList<PageReference> pages;
intframes;
Page Reference p, p1, f1;
public int pagehit, pagefault;
publicOptimal(ArrayList<PageReference> pages,intframes)
{
this.pages=pages;
this.frames=frames;
this.pagehit=0;
this.pagefault=0;
}
publicvoid solve(){
ArrayList<PageReference> f=new ArrayList<>(frames);//frame to store pages
for(int i=0;i<pages.size();i++)
{
p=pages.get(i);
System.out.println(" PageReference :"+p.page);


if(!f.contains(p) &&f.size()< frames)
{
/*Condition 1
if frame doesnt contains page i
and frame is not full
then
add page i
*/
f.add(p);

}
else if(!f.contains(p) &&f.size()>=frames)
{
/*Sort ArrayList f so that page with highest forwarddistance comes first */
Collections.sort(f,new Comparator<PageReference>(){
public intcompare(PageReference p1,PageReference p2)
{
return p2.distance-p1.distance;
}
});
/*Condition 2
if frame doesnt contains page i
and frame is full
then
remove the page withlongest forwarddistance
```

```
and add page i
*/
f.remove(0);
f.add(p);
}

else if(f.contains(p))
{
pagehit++;
}
for(int j=0;j<f.size();j++)
{

p=f.remove(0);
int k=i+1;
for(;k<pages.size();k++)
{
p1=pages.get(k);
if(p.page==p1.page)
{
p.distance=k-i;
break;
}
}
if(k==pages.size())
{
p.distance=pages.size()+1;
}
f.add(p);
}

System.out.println(f);
System.out.println("Pagehit : "+pagehit);
}
}
}
```

PageReference.java:

```
package Replacement;
class PageReference{
/*
PageReference: class for storing page references
page:page reference
backdistance: use to calculate least recently used page
*/
public intpage;
public int distance;
public PageReference(int page,int distance)
{
this.page=page;
this.distance=distance;
}
publicboolean equals(Object o)
{
/*
override equals function from Object class
*/
PageReference p=(PageReference)o;// convert Object to PageReference
if(p.page==this.page)// if page attribute is equal then Page Reference are equ
{
return true;
}
else
{
return false;
}

}
public String toString()
{
return Integer.toString(page);
}
}
```

replacement.java:

```
package Replacement ;
import java . u t i l . * ;


class replacement{

public static void main(String args []) throws Exception
{
Scanner s=new Scanner ( System . in ) ;
System . out . println ("Enter number of page reference string\n");
i n t n=s . next Int ( ) ;
System . out . println ("Enter page reference string\n");
Array List<Page Reference> pages=new Array List <>();
for(int i=0; i<n ;i++)
{
pages . add ( new Page Reference ( s . next Int ( ) , 0 ) ) ;
}
System . out . println ("Enter Number of page frames \n");
int frames=s . next Int ( ) ;
System . out . println ("1) Solve using LRU\n2) Solve using Optimal algoritm ");
int choice=s . next Int ( ) ;
switch ( c h o i c e )
{
case 1:
{
LRU l=new LRU( pages , frames ) ;
l . solve ( ) ;
}
case 2:
{
Optimal op=new Optimal ( pages , frames ) ;
op . s o l v e ( ) ;
}
}
}
}
```

6 Output:

```
sarthak@sarthak:~/Documents/SPOS CODES/14/Page_Replacement_Algorithm$ javac -d . replacement.java LRU.java Optimal.java PageReference.java
sarthak@sarthak:~/Documents/SPOS CODES/14/Page_Replacement_Algorithm$ java Replacement.replacement
Enter number of page reference string

6
Enter page reference string

1
3
2
4
2
3
Enter Number of page frames

2
1)Solve using LRU
2)Solve using Optimal algoritm
2
PageReference :1
[1]
Pagehit: 0
PageReference :3
[1, 3]
Pagehit: 0
PageReference :2
[3, 2]
Pagehit: 0
PageReference :4
[2, 4]
Pagehit: 0
PageReference :2
[2, 4]
Pagehit: 1
PageReference :3
[4, 3]
Pagehit: 1
sarthak@sarthak:~/Documents/SPOS CODES/14/Page_Replacement_Algorithm$
```

7 Conclusion:

In this assignment, we learnt in detail the concept of page simulation using both Optimal Algorithm and LRU Algorithm. We solved examples using these algorithms, for the better understanding of the same.