READER WRITER’S PROBLEM

The reader Writer's problem is a common concurrency problem in which many threads simultaneously try to access the contents of a file/ a shared resource. Here multiple users can read simultaneously since the resource is not getting modified. But while one user is writing into a file, none of the others can access the file (read / write) Here, the access to the file is controlled by semaphores.

void \*readers(void \*arg)  
{

int temp;  
temp = \*(int \*)arg;  
sem\_wait(&mutex);  
printf("reader %d is reading\n",temp);  
readcount++;  
if(readcount==1)  
{

printf("writer blocked\n");  
sem\_wait(&rw);

}  
sem\_post(&mutex);  
sem\_wait(&mutex);  
readcount--;  
printf("reader %d leaving\n",temp);  
if(readcount==0)  
 {printf("writer free\n");  
 sem\_post(&rw);}  
sem\_post(&mutex);  
}  
void \*writers(void \*arg)  
{

int temp;  
temp = \*(int \*)arg;  
printf("writer %d is making changes\n",temp);  
sem\_wait(&rw);  
printf("writing\n");  
printf("writer leaving\n");  
sem\_post(&rw);

}

PAGE REPLACEMENT

The Operating System performs demand paging. This means that only when a page fault occurs, the required page is brought in from the secondary memory. Demand paging causes the cache to fill up fast. Hence, we require page replacement strategies to minimize page faults.The following are the frequently used algorithms-

## FIFO (First in First Out)

In this algorithm, the replacement occurs in a cyclic manner independent of the frequency of occurrence of each page.This algorithm is easy to implement but drives pages out independent of their usage.

**PSEUDOCODE:**

int counter=0;

for i=0 to np

{

flag=0;

for j=0 to n

if(Q[i]==cache[j])

{ flag=1;break; }

for j=0 to n

{

if(cache[j]==-1)

printf("- ");

else

printf("%d ",cache[j]);

}

if(flag==1)

{ printf("\t\tHit\n"); tot\_hits++; }

else

{

cache[counter]=Q[i];

Counter = (counter+1)%n;

printf("\t\tMiss\n");

tot\_miss++;

}

}

## LRU (Least Recently Used)

In this algorithm, the replacement occurs in a according the time at which the page was referenced. The page that was used the earliest in the cache is replaced.

**PSEUDOCODE:**

int i,j,flag,used[10],min,pos;

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

used[i]=0;

int time=0;

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

{

flag=0;

++time;

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

{

if(Q[i]==cache[j])

{

used[j]=time;

flag=1;

break;

}

}

if(flag==1)

{ printf("\t\tHit\n"); tot\_hits++; }

else

{

min=999;

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

{

if(min>used[j])

{ min=used[j];pos=j; }

}

cache[pos]=Q[i];

used[pos]=time;

printf("\t\tMiss\n");

tot\_miss++;

}

}

## LFU (Least Frequently Used)

In this algorithm, an array is used to store the frequency of occurrence of each page. The page with the least frequency is replaced. If multiple pages in the cache have the same frequency, then FIFO is used.

**PSEUDOCODE:**

int usedcnt[100],least,repin,sofarcnt=0,bn=0,hitind,flag,pgfaultcnt=0;

int i,j,k;

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

usedcnt[i]=0;

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

{

flag=0;

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

{

if(Q[i]==cache[j])

{

hitind=j;

flag=1;

break;

}

}

if(flag==1)

{

usedcnt[hitind]++;

printf("No page fault!");

}

else

{

pgfaultcnt++;

if(bn<n)

{

cache[bn]=Q[i];

usedcnt[bn]=usedcnt[bn]+1;

bn++;

}

else

{

least=9999;

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

if(usedcnt[k]<least)

{

least=usedcnt[k];

repin=k;

}

cache[repin]=Q[i];

sofarcnt=0;

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

if(Q[i]==Q[k])

sofarcnt=sofarcnt+1;

usedcnt[repin]=sofarcnt;

}

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

{

if(cache[j]==-1)

printf("- ");

else

printf("%d ",cache[j]);

}

}

}

## Optimal Page Replacement

In this algorithm, the page which occurs nearest in the future is retained and the page occuring farthest is replaced. The prerequisite for this algorithm is that the page queue should be known beforehand which is not always possible.

**PSEUDOCODE:**

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

distance[i]=999;

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

{

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

{

if(cache[j]!=-1)

distance[j]=0;

for(k=i+1;k<np;k++)

{

if(Q[k]!=cache[j] && cache[j]!=-1)

distance[j]++;

else

break;

}

}

flag=0;

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

{

if(Q[i]==cache[j])

{

flag=1;

pos=j;

break;

}

}

if(flag==1)

{ printf("\t\tHit\n"); tot\_hits++; }

else

{ max=0;

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

{

if(max<distance[j])

{ max=distance[j]; pos=j; }

}

cache[pos]=Q[i];

printf("\t\tMiss\n");

tot\_miss++;

}

}

## Second Chance

In this algorithm, a page is given 2 chances. After a page fault, if it is referenced again, its reference bit is changed to 1. During replacement pages with reference bit = 0 get more priority and are replaced first. If a page with reference bit 1 is saved from replacement once, its reference bit is changed back to 0.

**PSEUDOCODE:**

int i,j,k,flag,counter=0,ref[10],hitindex;

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

ref[i]=0;

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

{

flag=0;

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

{

if(Q[i]==cache[j])

{ hitindex=j; flag=1; break; }

}

if(flag==1)

{

ref[hitindex]=(ref[hitindex]+1)%2;

printf("\t\tHit\n");

tot\_hits++;

}

else

{

int count=0;

for(k=counter;count<n;k=(k+1)%n)

{ ++count;

if(ref[k]==1);

else

break;

}

counter=k;

cache[counter]=Q[i];

ref[counter]=0;

printf("\t\tMiss\n");

counter= (counter+1)%n;

tot\_miss++;

}

}