山东大学 计算机科学与技术 学院

操作系统 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：201900161140 | 姓名：张文浩 | | 班级： 19智能 |
| 实验题目：内存页面置换算法实验 | | | |
| 实验学时：2 | | 实验日期： 5.24 | |
| 实验目的：  加深对于存储管理的了解，掌握虚拟存储器的实现原理；观察和了解重要的页面置换算法和置换过程。练习模拟算法的编程技巧，锻炼分析试验数据的能力。模拟示例实验中的两种置换算法，比较置换算法在给定条件下的优劣，分析各种页面置换算法适用的内存帧数。 | | | |
| 硬件环境： | | | |
| 软件环境：ubuntu | | | |
| 实验步骤与内容：   1. 根据实验指导书完成实例实验运行结果如下       以上输出报告了FIFO和LRU两种算法的页置换情况。其中每一行数字为当前实存 中的页号，->右边的数字表示当前被淘汰的页号。每种算法最后 3 行输出为：依次 淘汰页号，缺页数，页出错率。  再次执行 vmpr 命令，仍然输入 Belady 串，仅将页帧数改为 4      从以上输出中可以看出 FIFO 置换算法的 Belady 异常现象，即当在相同的引用串下 内存页帧数从 3 帧增加到 4 帧，页出错率反而从 75％增加到了 83.3%。而在相同的  情况下 LUR 置换算法无此异常现象。  2.补充其他算法之后的运行结果：      下列左图是引用页数为12，引用串为1 2 3 4 1 2 5 1 2 3 4 5，内存页帧数为3的结果  右图是引用页数为12，引用串为1 2 3 4 1 2 5 1 2 3 4 5，内存页帧数为4的结果  LRU    在LRU算法中随着内存帧数的增大，缺页率下降了  FIFO    在FIFO算法中随着内存帧数的增大，缺页率反而增加了。  CLOCK    在CLOCK算法中随着内存帧数的增大，缺页率反而增加了。  ECLOCK    在ECLOCK算法中随着内存帧数的增大，缺页率下降了  LFU    在LFU算法中随着内存帧数的增大，缺页率下降了  MFU    在MFU算法中内存帧数由3变为4，缺页率没变  再将内存帧数分别设置为2,3,4,5，观察各个算法的实验结果：  LRU    缺页率变化 100->83.3->66.7->41.7  曲线：  FIFO    缺页率变化 100->75->83.3->41.7  曲线：  CLOCK    缺页率变化 100->75->83.3->41.7  曲线：  ECLOCK    缺页率变化 100->83.3->58.3->41.7  曲线：  LFU    缺页率变化 83.3->75->58.3->41.7  曲线：  MFU    缺页率变化 75->66.7->66.7->41.7  曲线：  实现原理：  二次机会：将帧表设置为循环表,由一个引用位数组记录某一页是否被引用(0表示未被引用,1表示被引用)。当访问一页时,首先从帧表中检查此页是否在实存,若在将其引用位设置为1;若不在,在循环帧表中循序查找,若某页引用位为1,则将其设置为0，然后继续查找；若某页引用位为0，将其替换。若查找一圈，没有替换页（所有引用位均为1），则将全部引用位设置为0。  增强二次机会：将帧表设置为循环表，由引用位数组记录某页是否被引用,修改位数组记录某页是否被修改。当访问一页时,首先从帧表中检查此页是否在实存,若在将引用位和修改位设置为1;若不在,优先寻找引用位为0的页号，再次寻找修改位为0的页号，若有将其替换,并修改引用位和修改位;若没有这种页,将所有引用位置为0,重复第二次查找操作。  最少使用算法：设置count数组记录帧表中每一页被引用次数。当访问一页时,首先从帧表中检查此页是否在实存,若在将其计数加1;若不在,查找计数最小的页将其替换,并将当前页的计数  最多使用算法：设置count数组记录帧表中每一页被引用次数。当访问一页时,首先从帧表中检查此页是否在实存,若在将其计数加1;若不在,查找计数最大的页将其替换,并将当前页的计数置为1  算法分析：  FIFO：这种算法只是在按线性顺序访问地址空间时才是理想的，否则效率不高。因为那些常被访问的页，往往在主存中也停留得最久，结果它们因变“老”而不得不被置换出去。 FIFO的另一个缺点是，它有一种异常现象，即在增加存储块的情况下，反而使缺页中断率增加了。当然，导致这种异常现象的页面走向实际上是很少见的。  LRU：FIFO算法和OPT算法之间的主要差别是，FIFO算法利用页面进入内存后的时间长短作为置换依据，而OPT算法的依据是将来使用页面的时间。如果以最近的过去作为不久将来的近似，那么就可以把过去最长一段时间里不曾被使用的页面置换掉。它的实质是，当需要置换一页时，选择在最近一段时间里最久没有使用过的页面予以置换。这种算法就称为最久未使用算法（Least Recently Used，LRU）。 LRU算法是与每个页面最后使用的时间有关的。当必须置换一个页面时，LRU算法选择过去一段时间里最久未被使用的页面。  CLOCK二次机会算法：第二次机会算法的基本思想是与FIFO相同的，但是有所改进，避免把经常使用的页面置换出去。当选择置换页面时，检查它的访问位。如果是0，就淘汰这页；如果访问位是1，就给它第二次机会，并选择下一个FIFO页面。当一个页面得到第二次机会时，它的访问位就清为0，它的到达时间就置为当前时间。如果该页在此期间被访问过，则访问位置1。这样给了第二次机会的页面将不被淘汰，直至所有其他页面被淘汰过（或者也给了第二次机会）。因此，如果一个页面经常使用，它的访问位总保持为1，它就从来不会被淘汰出去。 第二次机会算法可视为一个环形队列。用一个指针指示哪一页是下面要淘汰的。当需要一个存储块时，指针就前进，直至找到访问位是0的页。随着指针的前进，把访问位就清为0。在最坏的情况下，所有的访问位都是1，指针要通过整个队列一周，每个页都给第二次机会。这时就退化成FIFO算法了。 | | | |
| 结论分析与体会：  通过本次实验，我加深了对于存储管理的了解，掌握了虚拟存储器的实现原理；观察和了解了重要的页面置换算法和置换过程。练习了模拟算法的编程技巧，锻炼了分析试验数据的能力。模拟了示例实验中的两种置换算法，比较了置换算法在给定条件下的优劣，分析了各种页面置换算法适用的内存帧数，并自己实现了二次机会等其他算法，对内存页面置换的知识有了更加深入的了解，收获很大。 | | | |

附源代码

Vmrp.h

/\*

\* Filename

: vmrp.h

\* copyright

: (C) 2021 by zwh

\* Function

: 声明虚拟内存页置换类

\*/

#include<cstdio>

#include<time.h>

#include<stdlib.h>

#include <iostream>

#include <iomanip>

#include <malloc.h>

using namespace std;

class Replace{

public:

Replace();

~Replace();

void InitSpace(char \* MethodName); //初始化页号记录

void Report(void); // 报告算法执行情况

void Fifo(void); //先进先出算法

void Lru(void); //最近最旧未用算法

void Clock(void); //时钟(二次机会)置换算法

void Eclock(void); //增强二次机会置换算法

void Lfu(void); //最不经常使用置换算法

void Mfu(void); //最经常使用置换算法

private:

int \* ReferencePage ; //存放要访问到的页号

int \* EliminatePage ; //存放淘汰页号

int \* PageFrames ;

//存放当前正在实存中的页号

int PageNumber;

//访问页数

int FrameNumber;

//实存帧数

int FaultNumber;

//失败页数

int \*Referencebit;

int \*count;

int \*Modifybit;

};

Vmrp.cc

#include "vmrp.h"

Replace::Replace()

{

int i;

//设定总得访问页数,并分配相应的引用页号和淘汰页号记录数组空间

cout << "Please input page numbers :" ;

cin >> PageNumber;//存放要访问到的页号

ReferencePage = new int[sizeof(int) \* PageNumber];//访问页数

EliminatePage = new int[sizeof(int) \* PageNumber];//存放淘汰页号

cout << "Please input reference page string :";

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

cin >> ReferencePage[i];

cout << "Please input page frames :";

cin >> FrameNumber;//实存帧数

PageFrames = new int[sizeof(int) \* FrameNumber];//存放当前正在实存中的页号

Referencebit=new int[sizeof(int) \* FrameNumber];//引用位

count=new int[sizeof(int)\*FrameNumber];

Modifybit=new int[sizeof(int)\*FrameNumber];//修改位

}

Replace::~Replace()

{

}

void Replace::InitSpace(char \* MethodName)

{

int i;

cout << endl << MethodName << endl;

FaultNumber=0;//失败页数

//引用还未开始,-1表示无引用页

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

EliminatePage[i] = -1;//淘汰页号初始都设为-1

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

{

PageFrames[i] = -1;//存放当前正在实存中的页号

Referencebit[i]=0;//未被使用引用位设置为0

count[i]=0;//计数

Modifybit[i]=0;//修改位初始为0

}

}

//分析统计选择的算法对于当前输入的页面走向的性能

void Replace::Report(void)

{

//报告淘汰页顺序

cout << endl << "Eliminate page:";

for(int i=0; EliminatePage[i]!=-1; i++) cout << EliminatePage[i] << " ";

//报告缺页数和缺页率

cout << endl << "Number of page faults = " << FaultNumber <<

endl;

cout << setw(6) << setprecision(3) ;

cout << "Rate of page faults = " << 100\*(float)FaultNumber/(float)PageNumber << "%"<<endl;

}

//最近最旧未用置换算法

void Replace::Lru(void)

{

int i,j,k,l,next;

InitSpace("LRU");//初始化页号记录

//循环装入引用页

for(k=0,l=0; k < PageNumber; k++) //l为淘汰页的标记

{

next=ReferencePage[k];

//检测引用页当前是否已在实存

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

{

if(next == PageFrames[i]) //与当前正在实存中的页号比较

{

//引用页已在实存将其调整到页记录栈顶

next= PageFrames[i];

for(j=i; j>0; j--) PageFrames[j] = PageFrames[j-1];

PageFrames[0]=next;

break;

}

}

if(PageFrames[0] == next)

{

//如果引用页已放栈顶，则为不缺页，报告当前内存页号

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

if(PageFrames[j]>=0) cout << PageFrames[j] << " ";

cout << endl;

continue; //继续装入下一页

}

else

// 如果引用页还未放栈顶，则为缺页，缺页数加1

FaultNumber++;

//栈底页号记入淘汰页数组中

EliminatePage[l] = PageFrames[FrameNumber-1];

//向下压栈

for(j=FrameNumber-1; j>0; j--) PageFrames[j]= PageFrames[j-1];

PageFrames[0]=next; //引用页放栈顶

//报告当前实存中页号

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

if(PageFrames[j]>=0) cout << PageFrames[j] << " ";

//报告当前淘汰的页号

if(EliminatePage[l]>=0)

cout << "->" << EliminatePage[l++] << endl;

else

cout << endl;

}

//分析统计选择的算法对于当前引用的页面走向的性能

Report();

}

//先进先出置换算法

void Replace::Fifo(void)

{

int i,j,k,l,next;

InitSpace("FIFO");

//循环装入引用页

for(k=0,j=l=0; k < PageNumber; k++)

{

next=ReferencePage[k];

//如果引用页已在实存中，报告实存页号

for (i=0; i<FrameNumber; i++) if(next==PageFrames[i]) break;

if (i<FrameNumber)

{

for(i=0; i<FrameNumber; i++) cout << PageFrames[i] << " ";

cout << endl;

continue;// 继续引用下一页

}

//引用页不在实存中，缺页数加1

FaultNumber++;

//最先入页号记入淘汰页数组

EliminatePage[l]= PageFrames[j];

PageFrames[j]=next;//引用页号放最先入页号处

j = (j+1)%FrameNumber;//最先入页号循环下移

//报告当前实存页号和淘汰页号

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

if(PageFrames[i]>=0) cout << PageFrames[i] << " ";

if(EliminatePage[l]>=0)

cout << "->" << EliminatePage[l++] << endl;

else

cout << endl;

}

Report();

}

//时钟(二次机会）置换算法

void Replace::Clock(void)

{

int j,i,k,l,next;

InitSpace("Clock");

for(k=0,j=l=0; k < PageNumber; k++)

{

next=ReferencePage[k];

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

//检测引用页当前是否已在实存

if(next==PageFrames[i]) //如果存在实存，就更新这个页的引用位为1

{

Referencebit[i]=1;

break;

}

if (i<FrameNumber) //存在实存的情况，报告一下当前的实存

{

for(i=0; i<FrameNumber; i++) cout << PageFrames[i] << " ";

cout << endl;

continue;

}

//如果当前页不存在实存，即出现了缺页的情况

while(Referencebit[j]==1){

//如果现在的页号的引用位是1则设置为0，并查找下一个页的引用位，直到找到引用位为0的页号

Referencebit[j]=0;

j=(j+1)%FrameNumber;

}

EliminatePage[l]=PageFrames[j];//最先入页号记入淘汰页数组

PageFrames[j]=next; //引用页号放最先入页号处

Referencebit[j]=1;//引用位设置为1

FaultNumber++;//缺页数加1

j=(j+1)%FrameNumber;//最先入页号循环下移

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

if(PageFrames[i]>=0) cout << PageFrames[i] << " ";

if(EliminatePage[l]>=0)

cout << "->" << EliminatePage[l++] << endl;

else

cout << endl;

}

Report();

}

//增强二次机会置换算法

void Replace::Eclock (void)

{

int j,i,k,l,next;

InitSpace("EClock");

for(k=0,j=l=0; k < PageNumber; k++)

{

next=ReferencePage[k];

//循环装入引用页

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

//检测引用页当前是否已在实存

if(next==PageFrames[i])

{

Referencebit[i]=1;//如果存在实存，引用位和修改位设置为1，退出循环

Modifybit[i]=1;

break;

}

//如果不缺页

if (i<FrameNumber)

{

for(i=0; i<FrameNumber; i++) cout << PageFrames[i] << " ";

cout << endl;

continue;

}

//如果缺页了，寻找被置换出去的页

int min=10\*Referencebit[j]+Modifybit[j];//给引用位更高的权重，优先替换没有被引用的页

int index=j;

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

{

if(10\*Referencebit[i]+Modifybit[i]<min)

{

min=10\*Referencebit[i]+Modifybit[i];

index=i;

}

}

for(i=0;i<(index+FrameNumber-j)%FrameNumber;i++){

Referencebit[j+i]=0;//把给过第二次机会的页的引用位置0

}

//如果所有的页的权重都是一样的，就置换最先进入的页，即index==j

EliminatePage[l]=PageFrames[index];

PageFrames[index]=next;

Referencebit[index]=1;//新加进来的页的引用位设置为1，修改位设置为0

Modifybit[index]=0;

FaultNumber++;//引用页不在实存中，缺页数加1

j=(j+1)%FrameNumber;//最先入页号循环下移

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

if(PageFrames[i]>=0) cout << PageFrames[i] << " ";

if(EliminatePage[l]>=0)

cout << "->" << EliminatePage[l++] << endl;

else

cout << endl;

}

Report();

}

//最不经常使用置换算法

void Replace::Lfu(void)

{

int j,i,k,l,next;

InitSpace("Lfu");

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

count[i]=0;

for(k=0,j=l=0; k < PageNumber; k++)

{

next=ReferencePage[k];

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

if(next==PageFrames[i]) //如果在实存中有这个页，就增加一次访问次数

{

count[i]++;//记录使用次数

break;

}

//如果不缺页

if (i<FrameNumber)

{

for(i=0; i<FrameNumber; i++) cout << PageFrames[i] << " ";

cout << endl;

continue;

}

//如果缺页了，找到访问次数最少的一个页进行替换

FaultNumber++;

int min=count[0];

int index=0;

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

{

if(count[i]<min)

{

min=count[i];

index=i;

}

}

EliminatePage[l]= PageFrames[index];//最不经常使用的页号记入淘汰页数组

PageFrames[index]=next;

count[index]=1;

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

if(PageFrames[i]>=0) cout << PageFrames[i] << " ";

if(EliminatePage[l]>=0)

cout << "->" << EliminatePage[l++] << endl;

else

cout << endl;

}

Report();

}

//最经常使用置换算法

void Replace::Mfu(void)

{

int j,i,k,l,next;

InitSpace("Mfu");

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

count[i]=10000;

for(k=0,j=l=0; k < PageNumber; k++)

{

next=ReferencePage[k];

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

if(next==PageFrames[i])

{

count[i]++;

break;

}

if (i<FrameNumber)

{

for(i=0; i<FrameNumber; i++) cout << PageFrames[i] << " ";

cout << endl;

continue;

}

FaultNumber++;

int max=count[0];

int index=0;

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

{

if(count[i]>max)

{

max=count[i];

index=i;

}

}

EliminatePage[l]= PageFrames[index];

PageFrames[index]=next;

count[index]=1;

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

if(PageFrames[i]>=0) cout << PageFrames[i] << " ";

if(EliminatePage[l]>=0)

cout << "->" << EliminatePage[l++] << endl;

else

cout << endl;

}

Report();

}

int main(int argc,char \*argv[])

{

Replace \* vmpr = new Replace();

vmpr->Lru();

vmpr->Fifo();

vmpr->Clock();

vmpr->Eclock();

vmpr->Lfu();

vmpr->Mfu();

return 0;

}