****

操作系统实验报告

姓名： 任杰文

学号： 09013430

东南大学计算机科学与工程学院

School of Computer Science & Engineering

Southeast University

二0 一 六 年 六 月四日

## 实验四

1. **实验内容：**

通过实验，理解LRU页面置换算法的算法思想及其实现方法，比较各种实现算法的复杂度和实现难度，体会LRU算法与各种近似算法间的区别，并进而加深对虚拟内存概念的理解。

1. **实验目的：**

通过实验，理解LRU页面置换算法的算法思想及其实现方法，比较各种实现算法的复杂度和实现难度，体会LRU算法与各种近似算法间的区别，并进而加深对虚拟内存概念的理解。

具体要求：

1. 应实现如下算法：LRU的计数器实现(counter implementation)，LRU的栈实现（stack implementation），Additional-Reference-Bits Algorithm，Second chance Algorithm。

2. 测试程序可参考如下结构：

while(已测页面数<要求测试页面数)

{

1）随机产生新的访问页号

2）调用算法决定是否需要置换页面，如需要，调整内存相关状态，更新页错误数

}

1. **设计思路及流程图**
2. 为比较四个不同的LRU算法的的性能表现，设计了一个LRU类，包含四种不同的LRU算法，LRU类通过四个变量初始化：页码最大编号maxNumOfPages;引用串长度lenOfRS；内容物理帧数int numOfMemFrames;附加引用位算法的时间单元int timeUnitForRefBit;调用bool genRS();生成长度为lenOfRS，最大编号为maxNumOfPages-1的引vector<int> refStr;再调用公共接口函数vector<double> sim\_4\_algorithm\_of\_LRU();返回四个LRU算法对于同一引用串的页错误率。
3. 本实验旨在比较各LRU算法的页错误率，在判断某页是否在内存中时，通过遍历查找，各个函数的时间复杂度都为O（N\*M），N为物理帧的大小，M为引用串长度。实际上可以用堆、二叉平衡树，哈希等方法，以空间换时间，降低时间复杂度。
4. 对于附加引用算法中，可能有多个页表项有相同的最小历史位，此时采用FIFO方法选择。由于算法需要在规定时间间隔内定时产生中断，然后将引用位转移到8位字节的最高位。时间同过模拟的方法，将其规定时间间隔通过一定的测试实例数表示。通过计数模拟系统定时中断。
5. **源程序**

#include <vector>

#include <list>

#include <limits>

#include <ctime>

#include <algorithm>

using namespace std;

//counter算法页表项

struct TabItemForCounter{

int pageID;

int counter;

TabItemForCounter ( int ID, int c){

pageID = ID;

counter = c;

}

};

//附加引用位算法，二次机会算法页表项

struct TabItemForRB{

int pageID;

bool refBit;

unsigned char hisbis;

TabItemForRB ( int ID, bool c){

pageID = ID;

refBit = c;

hisbis = 0;

}

};

class LRU

{

public:

LRU();

LRU( int a, int b, int c,int d = 10);

~LRU();

//返回给定参数下，以同一随机产生的引用串测试的4钟LRU算法的页错误率

vector<double> sim\_4\_algorithm\_of\_LRU();

//用随机数生成引用串

bool genRS();

//LRU计数器实现

int LRU\_Counter();

//LRU栈实现

int LRU\_Stack();

//LRU附加位实现

int AdditionalReferenceBits();

//LRU二次机会实现

int SecondChance();

public:

//最大页号，引用串长度，物理帧数，引用串

int maxNumOfPages;

int lenOfRS ;

int numOfMemFrames;

vector<int> refStr;

//附加引用位算法的时间单元

int timeUnitForRefBit;

};

LRU::LRU(){}

LRU:: ~LRU(){}

LRU::LRU( int a, int b, int c,int d)

{

maxNumOfPages = a;

numOfMemFrames = b;

lenOfRS = c;

timeUnitForRefBit = d;

genRS();

}

bool LRU::genRS()

{

refStr.resize( lenOfRS );

srand( time(0) );

for( vector<int> ::iterator it = refStr.begin(); it != refStr.end(); \*(it++) = rand() % maxNumOfPages);

return true;

}

vector<double> LRU::sim\_4\_algorithm\_of\_LRU()

{

vector<double> res(4,-1);

res[0] = (double)LRU\_Counter() / lenOfRS;

res[1] = (double)LRU\_Stack() / lenOfRS;

res[2] = (double)AdditionalReferenceBits() / lenOfRS;

res[3] = (double)SecondChance() / lenOfRS;

return res;

}

//时间O（n^2）,空间O（n）

int LRU::LRU\_Counter()

{

//页表，页错误数，逻辑时钟

vector<TabItemForCounter> pageTab( numOfMemFrames, TabItemForCounter(-1,-1) );

int pageFaultNum = 0;

int logTimCnt = 0;

for( vector<int> ::iterator RSit = refStr.begin();RSit != refStr.end(); ++RSit,++logTimCnt)

{

//查找请求帧号

vector<TabItemForCounter> ::iterator it = pageTab.begin();

//找到最小counter的帧并替换该帧，设置改帧counter为当前logTimCnt

int minCounter = numeric\_limits<int>::max();

int toReplaceFrameID = -1;

for( ; it != pageTab.end() && it->pageID != \*RSit; it++){

if ( it->counter < minCounter ){

toReplaceFrameID = it - pageTab.begin();

minCounter = it->counter;

}

}

//当前请求帧是否在内存中

if( it == pageTab.end() ){

++pageFaultNum;

pageTab[toReplaceFrameID].pageID = \*RSit;

pageTab[toReplaceFrameID].counter = logTimCnt;

}else{

it->counter = logTimCnt;

}

}

return pageFaultNum;

}

//时间O（n^2）,空间O（n）

int LRU::LRU\_Stack()

{

//初始化页码栈

list<int> pageStack( numOfMemFrames, -1);

int pageFaultNum = 0;

//遍历引用串

for( vector<int> ::iterator RSit = refStr.begin();RSit != refStr.end(); RSit++)

{

list<int>::iterator it = find( pageStack.begin(), pageStack.end(), \*RSit );

//当前页是否在内存

if( it != pageStack.end() ){

pageStack.push\_front( \*it);

pageStack.erase( it );

}else{

pageFaultNum++;

pageStack.erase( --it );

pageStack.push\_front( \*RSit);

}

}

return pageFaultNum;

}

//时间O（n^2）,空间O（n）

int LRU::AdditionalReferenceBits( )

{

vector<TabItemForRB> pageTab( numOfMemFrames, TabItemForRB(-1,false));

int pageFaultNum = 0;

int timeCnt = 0;

int nextpos = 0;

//遍历引用串

for( vector<int> ::iterator RSit = refStr.begin();RSit != refStr.end(); RSit++,timeCnt++)

{

//每个时间单元更新历史位

if ( timeCnt == timeUnitForRefBit )

{

timeCnt = 0;

for( vector<TabItemForRB> ::iterator it = pageTab.begin(); it != pageTab.end(); ++it )

{

it->hisbis = it->hisbis >> 1;

it->hisbis = it->refBit == true ? it->hisbis | 0x80 : it->hisbis;

it->refBit = false;

}

}

vector<TabItemForRB> ::iterator it = pageTab.begin();

for( ; it != pageTab.end() && it->pageID != \*RSit; it++){

}

//当前请求帧是否在内存中

if( it == pageTab.end() ){

++pageFaultNum;

//找到最小历史位的页并替换该页

int minhis = numeric\_limits<int>::max();

int toReplaceFrameID = -1;

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

{

int cur = (nextpos+i ) % numOfMemFrames;

//用九位比较

if ( (int)pageTab[ cur ].hisbis + pageTab[ cur ].refBit \* 256 < minhis ){

toReplaceFrameID = cur;

minhis = (int)pageTab[cur ].hisbis + pageTab[ cur ].refBit \* 256 ;

}

}

nextpos = (toReplaceFrameID + 1 ) % numOfMemFrames;

pageTab[toReplaceFrameID] = TabItemForRB(\*RSit,true);

}else{

it->refBit = true;

}

}

return pageFaultNum;

}

//时间O（n^2）,空间O（n）

int LRU::SecondChance()

{

vector<TabItemForRB> pageTab( numOfMemFrames, TabItemForRB(-1,false) );

int pageFaultNum = 0;

int nextpos = 0;

//遍历引用串

for( vector<int> ::iterator RSit = refStr.begin();RSit != refStr.end(); RSit++)

{

vector<TabItemForRB> ::iterator it = pageTab.begin();

for( ; it != pageTab.end() && it->pageID != \*RSit; it++){

if( it->pageID == \*RSit )

break;

}

//当前请求帧是否在内存中

if( it == pageTab.end() ){

++pageFaultNum;

while( pageTab[nextpos % numOfMemFrames].refBit == true )

{

pageTab[nextpos % numOfMemFrames].refBit = false;

nextpos = (nextpos+1) % numOfMemFrames;

}

pageTab[nextpos] = TabItemForRB(\*RSit,true);

nextpos = (nextpos+1) % numOfMemFrames;

}else{

it->refBit = true;

}

}

return pageFaultNum;

}

#include <iostream>

#include <iomanip>

#include "LRU.h"

using namespace std;

int main()

{

//freopen("in.txt","r",stdin);

int maxNumOfPages;

int numOfMemFrames;

int TestNum;

int timeUnitForRefBit;

vector<double> pageFaultNum(4,0);

while (true)

{

cout<<"最大页编号：";

cin>>maxNumOfPages;

cout<<"内存物理帧数：";

cin>>numOfMemFrames;

cout<<"测试序列长度：";

cin>>TestNum;

cout<<"附加引用位算法时间单元（用处理页数表示，默认）：";

cin>>timeUnitForRefBit;

LRU testLRU(maxNumOfPages,numOfMemFrames,TestNum, timeUnitForRefBit);

pageFaultNum = testLRU.sim\_4\_algorithm\_of\_LRU();

//格式对齐输出

cout<<"\n四种算法的页错误率（精度为位小数）：\n"

<<setw(15)<<"计数器"

<<setw(15)<<"栈"

<<setw(15)<<"附加引用位"

<<setw(15)<<"二次机会"<<endl;

cout<<setprecision(9)<<fixed

<<setw(15)<<pageFaultNum[0]

<<setw(15)<<pageFaultNum[1]

<<setw(15)<<pageFaultNum[2]

<<setw(15)<<pageFaultNum[3]<<endl;

}

return 0;

}

1. **时间空间复杂度分析**

四种算法实现如上代码所示：

1. LRU计数器法：页表通过线性表结构定义，页表项中包括页号和计数器值。定义如下：

vector<TabItemForCounter> pageTab(numOfMemFrames,TabItemForCounter(-1,-1) );

查找页表时，通过线性遍历页表项，比较是否有目标页；

综上：空间复杂度为O（N），时间复杂度为O（N\*M），其中N为物理帧的大小，M为引用串长度。

1. LRU栈法：页表通过双向链表结构定义，如下：

list<int> pageStack( numOfMemFrames, -1);

查找页表时，通过线性遍历页表项，比较是否有目标页；

综上：空间复杂度为O（N），时间复杂度为O（N\*M），其中N为物理帧的大小，M为引用串长度。

1. LRU附加引用位法：页表通过线性表结构定义，页表项中包括页号、引用位和历史位。如下：

vector<TabItemForRB> pageTab( numOfMemFrames, TabItemForRB(-1,false));

查找页表时，通过线性遍历页表项，比较是否有目标页；在给定时间单元后，遍历页表，更新历史位

综上：空间复杂度为O（N），时间复杂度为O（N\*M），其中N为物理帧的大小，M为引用串长度。

1. LRU二次机会法：页表通过线性表结构定义，页表项中包括页号、引用位。如下：

vector<TabItemForRB> pageTab( numOfMemFrames, TabItemForRB(-1,false) );

查找页表时，通过线性遍历页表项，比较是否有目标页；

综上：空间复杂度为O（N），时间复杂度为O（N\*M），其中N为物理帧的大小，M为引用串长度。

1. 实现难度：栈法最简单，附加引用位最复杂；
2. **实验结果以及结果分析**
3. 实验结果：

为方便数据统计和分析，添加重定向语句：

freopen("in.txt","r",stdin);freopen("out.txt","w",stdout);

修改输出格式：

while (cin>>maxNumOfPages>>numOfMemFrames>>TestNum>>timeUnitForRefBit){

cout<<"最大页编号："<<maxNumOfPages<<endl;

cout<<"内存物理帧数："<<numOfMemFrames<<endl;

cout<<"测试序列长度："<<TestNum<<endl;

cout<<"附加引用位算法时间单元（用处理页数表示，默认）："<<timeUnitForRefBit<<endl;

…………………………………………………………………………………………………………………

输入文件in.txt，给定最大页编号为1000，引用串长度为100000，附加引用位算法时间单元为处理5个页，通过改变物理帧数，比较四种算法页错误率，in.txt内容如下：

1000

50

100000

5

1000

100

100000

5

1000

150

100000

5

1000

200

100000

5

1000

250

100000

5

1000

300

100000

5

1000

350

100000

5

1000

400

100000

5

1000

450

100000

5

1000

500

100000

5

1000

550

100000

5

1000

600

100000

5

1000

650

100000

5

1000

700

100000

5

1000

750

100000

5

1000

800

100000

5

1000

850

100000

5

1000

900

100000

5

1000

950

100000

5

1000

1000

100000

5

输出文件out.txt内容如下：

最大页编号：1000

内存物理帧数：50

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.950290000 0.950290000 0.950600000 0.950280000

最大页编号：1000

内存物理帧数：100

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.899210000 0.899210000 0.899330000 0.899590000

最大页编号：1000

内存物理帧数：150

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.848880000 0.848880000 0.848830000 0.848500000

最大页编号：1000

内存物理帧数：200

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.800900000 0.800900000 0.800350000 0.800740000

最大页编号：1000

内存物理帧数：250

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.752650000 0.752650000 0.752850000 0.752370000

最大页编号：1000

内存物理帧数：300

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.700580000 0.700580000 0.701800000 0.700870000

最大页编号：1000

内存物理帧数：350

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.648490000 0.648490000 0.647430000 0.647760000

最大页编号：1000

内存物理帧数：400

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.599970000 0.599970000 0.600030000 0.600020000

最大页编号：1000

内存物理帧数：450

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.551910000 0.551910000 0.552940000 0.552340000

最大页编号：1000

内存物理帧数：500

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.501090000 0.501090000 0.498930000 0.499270000

最大页编号：1000

内存物理帧数：550

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.450890000 0.450890000 0.451470000 0.450640000

最大页编号：1000

内存物理帧数：600

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.403570000 0.403570000 0.404780000 0.403070000

最大页编号：1000

内存物理帧数：650

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.351200000 0.351200000 0.352340000 0.351940000

最大页编号：1000

内存物理帧数：700

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.303130000 0.303130000 0.303430000 0.302770000

最大页编号：1000

内存物理帧数：750

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.254400000 0.254400000 0.255150000 0.253650000

最大页编号：1000

内存物理帧数：800

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.204480000 0.204480000 0.204840000 0.205330000

最大页编号：1000

内存物理帧数：850

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.156770000 0.156770000 0.156400000 0.156260000

最大页编号：1000

内存物理帧数：900

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.107680000 0.107680000 0.107490000 0.105940000

最大页编号：1000

内存物理帧数：950

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.058050000 0.058050000 0.058070000 0.058560000

最大页编号：1000

内存物理帧数：1000

测试序列长度：100000

附加引用位算法时间单元（用处理页数表示，默认10）：5

四种算法的页错误率（精度为9位小数）：

计数器 栈 附加引用位 二次机会

0.010000000 0.010000000 0.010000000 0.010000000

1. 结果统计

通过excel表格统计分析数据：



折线图（横坐标为物理帧数，纵坐标为页错误率）：

1. 结果分析

实验结果发现，在给定最大页编号为1000，引用串长度为100000，附加引用位算法时间单元为处理5个页，通过改变物理帧数，发现随着物理帧数的增加，页错误率呈现线性下降趋势。

LRU计数器与栈有完全一样的性能表现，而LRU近似算法的性能与LRU选法几乎一致，表示这两种近似算法可以很好的替代LRU算法，应用到实际场景中，以降低设计的复杂度和时间的开销。

分析原因：在随机生成的0~999页页号时，随机数产生任意数的概率相等，因而命中页表项的概率为(物理帧数/最大页编号);然而这样随机生成的引用串，不能很好的模拟真实的程序调用过程。程序的局部性无法在随机生成的数字串中体现出来，而计算机缓冲设置的合理前提，就是程序访问的局部特性。因而该随机串只能模拟一个程序的局部。对于没有局部性的调用，缓冲意义有限。

1. 实验改进

该实验可通过生成具有局部性的引用串进行更真实的模拟。

算法的数据结构也可以由更好的替代，以达到更好的时间性能。

1. **实验体会**
2. 通过该实验，更加深入的了解了LRU页置换算法及其近似算法的算法思想及其实现方法，加深了对虚拟内存的理解。
3. 在实际情况中，物理帧的数目与页错误率期望呈现反比例函数关系，本实验用随机数的方法生成引用串，不能很好的模拟计算机中对于页引用的实际情况。更确切的说，只能模拟一个局部的调页情况。因而随着物理帧数的增加，页错误率与其呈现线性关系。
4. 对于程序局部性是设置高速的合理假设前提有了更为深刻的认识。
5. 认识到实验分析对于深入理解和掌握知识的重要性。图表能够直观的呈现出数据间的关系，对于结果的分析大有裨益。