**实 验 报 告**

**实验人： 学号： 日期： 2010-12-29**

**院（系）： 信科院计算机系 专业(班级)：**

**实验题目：** 常用页面置换算法模拟实验

**实验二 存储管理**

1. **实验目的**

通过模拟实现请求页式存储管理的几种基本页面置换算法，了解虚拟存储技术的特 点，掌握虚拟存储请求页式存储管理中几种基本页面置换算法的基本思想和实现过程， 并比较它们的效率。

1. **实验内容**

基于一个虚拟存储区和内存工作区，设计下述算法并计算访问命中率。

1、最佳淘汰算法（OPT）

2、先进先出的算法（FIFO）

3、最近最久未使用算法（LRU）

4、简单时钟(钟表)算法（CLOCK）

命中率＝１－页面失效次数／页地址流**(**序列**)**长度

1. **实验原理简述**

UNIX中，为了提高内存利用率，提供了内外存进程对换机制；内存空间的分配和回收均以页为单位进行；一个进程只需将其一部分（段或页）调入内存便可运行；还支持请求调页的存储管理方式。

当进程在运行中需要访问某部分程序和数据时，发现其所在页面不在内存，就立即提出请求（向CPU发出缺中断），由系统将其所需页面调入内存。这种页面调入方式叫请求调页。

为实现请求调页，核心配置了四种数据结构：页表、页帧（框）号、访问位、修改位、有效位、保护位等。

当CPU接收到缺页中断信号，中断处理程序先保存现场，分析中断原因，转入缺页中断处理程序。该程序通过查找页表，得到该页所在外存的物理块号。如果此时内存未满，能容纳新页，则启动磁盘I/O将所缺之页调入内存，然后修改页表。如果内存已满，则须按某种置换算法从内存中选出一页准备换出，是否重新写盘由页表的修改位决定，然后将缺页调入，修改页表。利用修改后的页表，去形成所要访问数据的物理地址，再去访问内存数据。整个页面的调入过程对用户是透明的。

1. **算法描述**

本实验的程序设计基本上按照实验内容进行。即使用srand( )和rand( )函数定 义和产生指令序列，然后将指令序列变换成相应的页地址流，并针对不同的算 法计算出相应的命中率。

（1）通过随机数产生一个指令序列，共320条指令。指令的地址按下述原则生成：

A：50%的指令是顺序执行的

B：25%的指令是均匀分布在前地址部分

C：25%的指令是均匀分布在后地址部分

具体的实施方法是：

A：在[0，319]的指令地址之间随机选取一起点m

B：顺序执行一条指令，即执行地址为m+1的指令

C：在前地址[0,m+1]中随机选取一条指令并执行，该指令的地址为m’

D：顺序执行一条指令，其地址为m’+1

E：在后地址[m’+2，319]中随机选取一条指令并执行

F：重复步骤A-E，直到320次指令

（2）将指令序列变换为页地址流

设：页面大小为1K；

用户内存（页帧）容量为4页~32页；

用户虚存容量为32K。

在用户虚存中，按每K存放10条指令排列虚存地址，即320条指令在虚存中的存放方式为：

第 0 条-第 9 条指令为第0页（对应虚存地址为[0，9]）

第10条-第19条指令为第1页（对应虚存地址为[10，19]）

………………………………

第310条-第319条指令为第31页（对应虚存地址为[310，319]）

按以上方式，用户指令可组成32页。

1. **算法实现与分析**

**1.常量及变量**

#define total\_instruction 320 //指令流长

#define total\_vp 32 //虚页长

#define clear\_period 50 //清周期

pfc\_type pfc[total\_vp], //主存区页面控制结构数组

pfc\_type \*freepf\_head, //主存区页面控制结构的空闲页面头指针

pfc\_type \*busypf\_head, //主存区页面控制结构的忙页面头指针

pfc\_type \*busypf\_tail; //主存区页面控制结构的忙页面尾指针

int diseffect; //页错误计数器，初次把页面载入主存时也当做页错误

pl\_type pl[total\_vp]; //页面结构数组

**2.数据结构**

typedef struct //页面结构

{

int pn, //页面序号

pfn, //页面所在内存区的帧号

counter, //单位时间内访问次数

time; //上次访问的时间

}pl\_type;

struct pfc\_struct{ //页面控制结构，模拟内存中的页集

int pn, //页面号

pfn; //内存区页面的帧号

struct pfc\_struct \*next; //页面指针，用于维护内存缓冲区的链式结构

};

**3.函数定义**

int initialize(int); //初始化页面结构数组和页面控制结构数组

int FIFO(int); //先进先出算法

int LRU(int); //最近最久未使用算法

int OPT(int); //最佳置换算法

int CLOCK(int); //简单时钟(钟表)算法

1. **实验结果分析**

**实验数据结果：**

------------随机产生指令流------------

257 258 37 38

226 227 109 110

184 185 164 165

166 167 59 60

310 311 135 136

148 149 105 106

240 241 121 122

124 125 50 51

315 316 308 309

312 313 299 300

315 316 284 285

284 285 272 273

318 319 216 217

310 311 266 267

318 319 127 128

129 130 52 53

53 54 48 49

130 131 62 63

159 160 107 108

206 207 130 131

167 168 123 124

272 273 23 24

123 124 32 33

303 304 163 164

206 207 134 135

269 270 123 124

177 178 124 125

244 245 54 55

68 69 5 6

165 166 144 145

270 271 75 76

88 89 65 66

69 70 31 32

56 57 40 41

189 190 73 74

92 93 50 51

92 93 77 78

88 89 62 63

125 126 71 72

255 256 125 126

289 290 97 98

235 236 163 164

240 241 29 30

158 159 80 81

280 281 263 264

312 313 58 59

226 227 78 79

121 122 108 109

202 203 32 33

42 43 18 19

153 154 67 68

292 293 63 64

264 265 54 55

269 270 40 41

296 297 295 296

318 319 269 270

278 279 214 215

222 223 186 187

220 221 30 31

268 269 33 34

226 227 117 118

211 212 170 171

313 314 77 78

248 249 34 35

232 233 25 26

82 83 59 60

61 62 23 24

168 169 24 25

259 260 239 240

318 319 275 276

283 284 74 75

244 245 144 145

244 245 86 87

120 121 115 116

238 239 209 210

275 276 215 216

284 285 214 215

285 286 186 187

208 209 162 163

238 239 41 42

--------------------------------------

--不同页面工作区各种替换策略的命中率表--

Page FIFO LRU OPT CLOCK

4 0.550 0.559 0.669 0.550

5 0.566 0.572 0.700 0.572

6 0.578 0.594 0.722 0.578

7 0.591 0.603 0.741 0.597

8 0.631 0.628 0.756 0.637

9 0.637 0.656 0.772 0.650

10 0.641 0.669 0.787 0.653

11 0.656 0.678 0.800 0.666

12 0.688 0.684 0.813 0.672

13 0.703 0.697 0.822 0.697

14 0.713 0.713 0.831 0.719

15 0.722 0.728 0.841 0.722

16 0.731 0.747 0.850 0.741

17 0.744 0.772 0.856 0.747

18 0.769 0.778 0.863 0.769

19 0.778 0.787 0.869 0.778

20 0.781 0.797 0.875 0.794

21 0.787 0.800 0.881 0.806

22 0.816 0.809 0.887 0.809

23 0.822 0.822 0.891 0.822

24 0.838 0.831 0.894 0.838

25 0.844 0.847 0.897 0.841

26 0.847 0.856 0.900 0.856

27 0.847 0.869 0.900 0.859

28 0.856 0.878 0.900 0.872

29 0.859 0.884 0.900 0.878

30 0.881 0.891 0.900 0.891

31 0.897 0.897 0.900 0.897

32 0.900 0.900 0.900 0.900

请按任意键继续. . .

**结果分析：**

理论上，四种替换算法的命中率由高到底排列应该是OPT>LRU>CLOCK>FIFO。实际上，从实验数据观测得到，存在这种由高到低的趋势，由page=4时可以观测到，但是效果不是很明显。

效果不明显的原因：

推测与指令流的产生方式有关系。因为指令流的产生方式要能体现局部性原理，所以该指令流产生设计为：50%的指令是顺序执的，25%的指令是均匀分布在前地址部分，25%的指令是均匀分布在后地址部分。但是这样的指令流设计方式能否最佳地体现局部性原理，这还有待验证。

同时，估计和指令数量有关系。因为320条指令太少了，通常一个稍大点的程序都几千行指令了。

而且由于随即数产生具有一定的波动性，该命中率的计算也有一定的波动性。所以会有局部的实验数据与理论不符。改进方法是多次实验取平均值，这样可以减小波动，让实验数据更加平滑。

唯一显著的是OPT算法的命中率与其他3个调度算法保持了比较大的差距。例如在page=26时，OPT算法就能达到0.9的命中率了。

到后期，由于page越来越大，因此越来越容易命中，因此各替换算法的命中率差距变小了。这由最后几行命中率相似可以看出。

1. **实验总结**

这次实验其实不一定要在linux操作系统下做，在windows操作系统一样可以实现，只要把头文件稍作修改即可。为了保险起见，我在2个操作系统下都编译过，都没问题。在windows操作系统，要屏蔽//#include <unistd.h>这句话，在linux操作系统下则启用。

此次实验借助于老师提供的主函数main模板，只需要写FIFO，LRU，OPT,CLOCK等4个替换算法，所以阻力没那么大。每个替换算法必须弄懂其中的细节，写起来才得心应手。 一开始做这个实验时，首先是看书，先把书上的替换算法知识点弄明白，要明白各种算法的优缺点和相互之间衍生互补关系。这四个算法中，难以实现的是LRU算法，因为它涉及到访问时间的计算，而且它的开销也比较大。OPT算法次难，它需要计算最近访问时间，并替换最近访问时间最大的页。而FIFO和CLOCK实现起来比较容易，FIFO算法的实现和CLOCK算法的实现很相似，FIFO可视为CLOCK的退化版。我先写了CLOCK算法，再删去一些约束条件就退化为FIFO算法。这就是两者的相同之处。理论上，CLOCK算法需要维持一个循环的主存缓冲区，需要一个循环队列去实现，并且，FIFO算法保持先进先出，因此需要一个先进先出队列。但是，我实现这两个算法只用到了单向链表的数据结构，剩下的由其中的指针去把握了。因此，必须对指针使用有敏锐的感觉。

天下无难事，只怕有心人！

1. **程序源码（在两个系统上都通过）**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h> //在window操作系统下要屏蔽此条指令

#include <string.h>

#ifndef \_UNISTD\_H

#define \_UNISTD\_H

#include <IO.H>

#include <PROCESS.H>

#endif

#define TRUE 1

#define FALSE 0

#define INVALID -1

#define total\_instruction 320 //指令流长

#define total\_vp 32 //虚页长

#define clear\_period 50 //清周期

typedef struct //页面结构

{

int pn, //页面序号

pfn, //页面所在内存区的帧号

counter, //单位时间内访问次数

time; //上次访问的时间

}pl\_type;

pl\_type pl[total\_vp]; //页面结构数组

struct pfc\_struct{ //页面控制结构

int pn, //页面号

pfn; //内存区页面的帧号

struct pfc\_struct \*next; //页面指针，用于维护内存缓冲区的链式结构

};

typedef struct pfc\_struct pfc\_type; //主存区页面控制结构别名

pfc\_type pfc[total\_vp], //主存区页面控制结构数组

\*freepf\_head, //主存区页面控制结构的空闲页面头指针

\*busypf\_head, //主存区页面控制结构的忙页面头指针

\*busypf\_tail; //主存区页面控制结构的忙页面尾指针

int diseffect; //页错误计数器，初次把页面载入主存时也当做页错误

int a[total\_instruction]; //随即指令流数组

int page[total\_instruction]; //指令对应的页面号

int offset[total\_instruction]; //指令所在页面中的偏移量

int initialize(int); //初始化页面结构数组和页面控制结构数组

int FIFO(int); //先进先出算法

int LRU(int); //最近最久未使用算法

int OPT(int); //最佳置换算法

int CLOCK(int); //简单时钟(钟表)算法

int main( )

{

int s; //随机数

int i;

srand(10\*getpid()); /\*每次运行时进程号不同，用来作为初始化随机数队列的"种子"\*/

s = (int)((float)(total\_instruction-1)\*(rand()/(RAND\_MAX+1.0)));

printf("\n------------随机产生指令流------------\n");

for (i=0; i<total\_instruction; i+=4) //产生指令队列

{

a[i]=s; //任选一指令访问点m

a[i+1]=a[i]+1; //顺序执行一条指令

a[i+2]=(int)((float)a[i]\*(rand()/(RAND\_MAX+1.0))); //执行前地址指令m'

a[i+3]=a[i+2]+1; //顺序执行一条指令

printf("%6d%6d%6d%6d\n", a[i],a[i+1],a[i+2],a[i+3]);

s = (int)((float)((total\_instruction-1)-a[i+2])\*(rand()/(RAND\_MAX+1.0))) + a[i+2];

}

printf("--------------------------------------\n");

for (i=0;i<total\_instruction;i++) //将指令序列变换成页地址流

{

page[i]=a[i]/10;

offset[i]=a[i]%10;

}

printf("\n--不同页面工作区各种替换策略的命中率表--\n");

printf("Page\t FIFO\t LRU\t OPT\t CLOCK\n");

for(i=4;i<=32;i++) //用户内存工作区从个页面到个页面

{

printf(" %2d \t",i);

FIFO(i);

LRU(i);

OPT(i);

CLOCK(i);

printf("\n");

}

return 0;

}

//初始化页面结构数组和页面控制结构数组

//total\_pf; 用户进程的内存页面数

int initialize(int total\_pf)

{

int i;

diseffect=0;

for(i=0;i<total\_vp;i++)

{

pl[i].pn=i;

pl[i].pfn=INVALID; //置页面所在主存区的帧号为-1.表示该页不在主存中

pl[i].counter=0; //置页面结构中的访问次数为

pl[i].time=-1; //置页面结构中的上次访问的时间为-1

}

for(i=0;i<total\_pf-1;i++)

{

pfc[i].next=&pfc[i+1]; //建立pfc[i-1]和pfc[i]之间的链接

pfc[i].pfn=i; //初始化主存区页面的帧号

}

pfc[total\_pf-1].next=NULL;

pfc[total\_pf-1].pfn=total\_pf-1;

freepf\_head=&pfc[0]; //主存区页面控制结构的空闲页面头指针指向pfc[0]

return 0;

}

//最近最久未使用算法

//int total\_pf; 用户进程的内存页面数

int LRU (int total\_pf)

{

int MinT; //最小的访问时间，即很久没被访问过

int MinPn; //拥有最小的访问时间的页的页号

int i,j;

int CurrentTime; //系统当前时间

initialize(total\_pf); //初始化页面结构数组和页面控制结构数组

CurrentTime=0;

diseffect=0;

for(i=0;i<total\_instruction;i++)

{

if(pl[page[i]].pfn==INVALID) //页面失效

{

diseffect++; //页错误次数加

if(freepf\_head==NULL) //无空闲页面

{

MinT=100000;

for(j=0;j<total\_vp;j++){ //找出time的最小值，表明该页很久没被访问过

if(MinT>pl[j].time&&pl[j].pfn!=INVALID)

{

MinT=pl[j].time;

MinPn=j;

}

}

freepf\_head=&pfc[pl[MinPn].pfn]; //最久没被访问过的页被释放

pl[MinPn].pfn=INVALID; //最久没被访问过的页被换出主存

pl[MinPn].time=-1; //最久没被访问过的页的访问时间置为无效

freepf\_head->next=NULL;

}

pl[page[i]].pfn=freepf\_head->pfn; //有空闲页面,把相应的页面换入主存，并把pfn改为相应的帧号

pl[page[i]].time=CurrentTime; //令访问时间为当前系统时间

freepf\_head=freepf\_head->next; //减少一个空闲页面

}

else

pl[page[i]].time=CurrentTime; //命中则刷新该单元的访问时间

CurrentTime++; //系统当前时间加

}

printf("%6.3f\t",1-(float)diseffect/320);

return 0;

}

//最佳置换算法

//int total\_pf; 用户进程的内存页面数

int OPT(int total\_pf)

{

int i,j;

int MaxD; //将来最近一次访问的距离的最大值（以时间单元度量）

int MaxPn; //将来最近一次访问的距离的最大值的页号

int dis; //距离计数器

int dist[total\_vp]; //距离数组，保存距离上一次访问的时间差距个数

initialize(total\_pf); //初始化页面结构数组和页面控制结构数组

diseffect=0;

for(i=0;i<total\_instruction;i++)

{

if(pl[page[i]].pfn==INVALID) //页面失效

{

diseffect++; //页错误次数加

if(freepf\_head==NULL) //无空闲页面

{

for(j=0;j<total\_vp;j++)

{

if(pl[j].pfn!=INVALID) //如果该页在主存中

dist[j]=100000; // 该页关联的距离值改为最大值

else

dist[j]=0; //如果不在该页主存中，该页关联的距离值改为

}

dis=1; //初始距离值为

for(j=i+1;j<total\_instruction;j++) //从要替换的指令的下一条算起，

{

if(pl[page[j]].pfn!=INVALID &&pl[page[j]].counter==0) //如果该页在主存中,并且是将要最近访问的页

//if(pl[page[j]].pfn!=INVALID && dist[page[j]]==100000) //此条语句原理与上相同

{ dist[page[j]]=dis; //距离值改为dis

pl[page[j]].counter=1; //使访问次数标志加，区别第一次访问和第二次访问

}

dis++;

}

MaxD=-1;

for(j=0;j<total\_vp;j++)

{

pl[j].counter=0; //重置访问次数为

if(MaxD<dist[j]) //查找将来最近一次访问的距离的最大值及其序号

{

MaxD=dist[j];

MaxPn=j;

}

}

freepf\_head=&pfc[pl[MaxPn].pfn]; //替换将来一段时间最久访问的页

freepf\_head->next=NULL;

pl[MaxPn].pfn=INVALID;

}

pl[page[i]].pfn=freepf\_head->pfn; //把当前页换入主存中，并且把当前页的pfn改为换入页的帧号，

freepf\_head=freepf\_head->next; //减少一个空闲页面

}//if

}//for

printf("%6.3f\t",1-(float)diseffect/320);

return 0;

}

//简单时钟算法

//int total\_pf; 用户进程的内存页面数

int CLOCK(int total\_pf)

{

int i;

int use[total\_vp]; //使用位

int swap;

swap=0; //发生替换

initialize(total\_pf);

pfc\_type \*pnext; //时钟指针

pfc\_type \*head; //队列头指针

pnext=freepf\_head;

head=freepf\_head;

for(i=0;i<total\_vp;i++){use[i]=0;} //初始化使用位为

diseffect=0;

for(i=0;i<total\_instruction;i++)

{

if (pl[page[i]].pfn==INVALID) //页面失效,不在主存中

{

diseffect++; //页错误次数加

if(freepf\_head==NULL) //无空闲页面

{

while(use[pnext->pfn]==1) //若时钟指针指向的页的使用位为，则改为并跳过

{

use[pnext->pfn]=0;

pnext=pnext->next;

if(pnext==NULL) pnext=head; //如果时钟指针到达队列尾部，重新返回头部

}

//换出被替换的页

pl[pnext->pn].pfn=INVALID;

swap=1;

}

if(use[pnext->pfn]==0){ //如果使用位为，则换入相应的页

pl[page[i]].pfn=pnext->pfn; //页面结构中要标记帧号

pnext->pn=page[i]; //页面控制结构中要标记页号

use[pnext->pfn]=1; //重置使用位为

pnext=pnext->next; //时钟指针下移

if(pnext==NULL) pnext=head; //如果时钟指针到达队列尾部，重新返回头部

if(swap==0){ freepf\_head=freepf\_head->next; }

}

}else{//页面在主存中

use[pl[page[i]].pfn]=1; //刷新使用位为

}

}

printf("%6.3f\t",1-(float)diseffect/320);

return 0;

}

//先进先出算法版本

//int total\_pf; 用户进程的内存页面数

//实现细节由CLOCK算法退化而来，与FIFO同效果

int FIFO(int total\_pf)

{

int i;

int use[total\_vp];

int swap=0;

initialize(total\_pf);

pfc\_type \*pnext,\*head;

pnext=freepf\_head;

head=freepf\_head;

for(i=0;i<total\_vp;i++){use[i]=0;}

diseffect=0;

for(i=0;i<total\_instruction;i++)

{

if (pl[page[i]].pfn==INVALID) //页面失效,不在主存中

{

diseffect++;

if(freepf\_head==NULL) //无空闲页面

{

while(use[pnext->pfn]==1)

{

use[pnext->pfn]=0;

pnext=pnext->next;

if(pnext==NULL) pnext=head;

}

//换出被替换的页

pl[pnext->pn].pfn=INVALID;

swap=1;

}

if(use[pnext->pfn]==0){ //如果使用位为，则换入相应的页

pl[page[i]].pfn=pnext->pfn; //页面结构中要标记帧号

pnext->pn=page[i]; //页面控制结构中要标记页号

use[pnext->pfn]=1; //重置使用位为

pnext=pnext->next;

if(pnext==NULL) pnext=head;

if(swap==0){ freepf\_head=freepf\_head->next; }

}

}

}

printf("%6.3f\t",1-(float)diseffect/320);

return 0;

}