** 南昌大学实验报告**

学生姓名： 程婧 学 号： 6109117016 专业班级： 计科可视化171班

实验类型：□ 验证 □ 综合 ■ 设计 □ 创新 实验日期： 2019.12.9 实验成绩：

**一、实验目的**

存储管理的主要功能之一是合理地分配空间。请求页式管理是一种常用的虚拟存储管理技术。本实验的目的是通过请求页式存储管理中页面置换算法模拟设计，了解虚拟存储技术的特点，掌握请求页式管理的页面置换算法。

**二、实验内容**

1．过随机数产生一个指令序列，共320条指令。其地址按下述原则生成：

①50%的指令是顺序执行的；

②25%的指令是均匀分布在前地址部分；

③25%的指令是均匀分布在后地址部分；

#具体的实施方法是：

* + 1. 在[0，319]的指令地址之间随机选区一起点M;
    2. 顺序执行一条指令，即执行地址为M+1的指令；
    3. 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’;
    4. 顺序执行一条指令，其地址为M’+1；
    5. 在后地址[M’+2，319]中随机选取一条指令并执行；
    6. 重复A—E，直到执行320次指令。

2．指令序列变换成页地址流

设：（1）页面大小为1K；

* 1. 用户内存容量为4页到32页；
  2. 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。

3. 计算并输出下述各种算法在不同内存容量下的命中率。

* + 1. FIFO先进先出的算法
    2. LRU最近最少使用算法

C．LFU最少访问页面算法

**三、实验要求**

1、需写出设计说明；

2、设计实现代码及说明

3、运行结果；

**四、主要实验步骤**

#include <stdio.h>

#include <stdlib.h>

#include <string.h> //1字符串头文件

#include <unistd.h>

#ifndef \_UNISTD\_H

#define \_UNISTD\_H

#include <IO.H> //1io.h主要定义一些和缓冲区相关的读写函数

#include <PROCESS.H> //1process.h 是包含用于和宏指令的作用声明

#endif

#define TRUE 1

#define FALSE 0

#define INVALID -1

#define total\_instruction 320

#define total\_vp 32

#define clear\_period 50

typedef struct //页面结构

{

int pn, //页面序号

pfn, //页面所在内存区的页框号

counter, time;

} 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;

};

typedef struct pfc\_struct pfc\_type;

pfc\_type pfc[total\_vp], \*freepf\_head, \*busypf\_head, \*busypf\_tail;

int diseffect;

int a[total\_instruction];//指令序号数组

int page[total\_instruction];

int offset[total\_instruction];

int initialize(int);

int FIFO(int);//不同的页面置换算法

int LRU(int);

int OPT(int);

int CLOCK(int);

int main() {

int s;

int i;

srand(10 \*getpid()); // 1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; //第几页

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

{//i作为内存中的页框数

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++) {//对所有的页

pl[i].pn = i;

pl[i].pfn = INVALID;

pl[i].counter = 0;//访问次数置0

pl[i].time = -1;//存在时间设为-1

}

for (i = 0; i < total\_pf - 1; i++) {//所有的页块进行链式链接

pfc[i].next = &pfc[i + 1];

pfc[i].pfn = i;

}//默认选择的是前i个页面共存于内存中

pfc[total\_pf - 1].next = NULL;//最后一页不链接其他页框

pfc[total\_pf - 1].pfn = total\_pf - 1;//最后一个页框，pfn设为编号值，VALID化

freepf\_head = &pfc[0];//指向第一个页框的指针

return 0;

}

int LRU(int total\_pf) {

int MinT;

int MinPn; //最小访问时间对应的叶号

int i, j;

int CurrentTime;

initialize(total\_pf);//初始化

CurrentTime = 0;

diseffect = 0;

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

if (pl[page[i]].pfn == INVALID) {

diseffect++;//产生一次缺页

if (freepf\_head == NULL) {//说明内存中一页都没有空闲

MinT = 100000;//保证这一次读取到的页能够进入内存

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

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

MinT = pl[j].time;//最小访问时间

MinPn = j;

}//通过便利，找出最小访问时间的夜

}

freepf\_head = &pfc[pl[MinPn].pfn];

pl[MinPn].pfn = INVALID;//内存中无该页面，换出l

pl[MinPn].time = -1;

freepf\_head->next = NULL;

}

pl[page[i]].pfn = freepf\_head->pfn;//换入，同时修改信息

pl[page[i]].time = CurrentTime;

freepf\_head = freepf\_head->next;

}

else

pl[page[i]].time = CurrentTime;

CurrentTime++;

}

printf("%6.3f\t", 1 - (float)diseffect / 320);//用1减去缺页率；

return 0;

}

int OPT(int total\_pf) {//最理想的结果

int i, j;

int MaxD;

int MaxPn;

int dis;

int dist[total\_vp];

initialize(total\_pf);

diseffect = 0;

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

if (pl[page[i]].pfn == INVALID) //页面失效

{

diseffect++;

if (freepf\_head == NULL) {

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

if (pl[j].pfn != INVALID)

dist[j] = 100000;

else

dist[j] = 0;

}

dis = 1;

for (j = i + 1; j < total\_instruction; j++) {

if (pl[page[j]].pfn != INVALID && pl[page[j]].counter == 0) {

dist[page[j]] = dis;

pl[page[j]].counter = 1;

}

dis++;

}

MaxD = -1;

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

pl[j].counter = 0;

if (MaxD < dist[j]) {

MaxD = dist[j];

MaxPn = j;

}

}

freepf\_head = &pfc[pl[MaxPn].pfn];

freepf\_head->next = NULL;

pl[MaxPn].pfn = INVALID;

}

pl[page[i]].pfn = freepf\_head->pfn;

freepf\_head = freepf\_head->next;

} // if

} // for

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

return 0;

}

int CLOCK(int total\_pf) {

int i;

int use[total\_vp];

int swap;

swap = 0;

initialize(total\_pf);

pfc\_type \*pnext;

pfc\_type \*head;

pnext = freepf\_head;

head = freepf\_head;

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

use[i] = 0;

}//都没有使用过

diseffect = 0;

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

if (pl[page[i]].pfn == INVALID) {

diseffect++;//缺页

if (freepf\_head == NULL) {

while (use[pnext->pfn] == 1) {//访问过，就改为0

use[pnext->pfn] = 0;

pnext = pnext->next;//下一页

if (pnext == NULL)

pnext = head;

}

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

swap = 1;

}

if (use[pnext->pfn] == 0) {

pl[page[i]].pfn = pnext->pfn;

pnext->pn = page[i];

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;

}

}

} else { //页面在主存中

use[pl[page[i]].pfn] = 1;

}

}

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

return 0;

}

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++) {//开始对每一条指令进行操作

if (pl[page[i]].pfn == INVALID) {

diseffect++;//缺页

if (freepf\_head == NULL) {//

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

use[pnext->pfn] = 0;

pnext = pnext->next;

if (pnext == NULL)

pnext = head;

}

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

swap = 1;//标记一下，要被换出了

}

if (use[pnext->pfn] == 0) {

pl[page[i]].pfn = pnext->pfn;

pnext->pn = page[i];

use[pnext->pfn] = 1;

pnext = pnext->next;

if (pnext == NULL)

pnext = head;

if (swap == 0) {

freepf\_head = freepf\_head->next;

}

}

}

//访问成功，不进行任何操作

}

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

return 0;

}

FIFO算法流程图：

开始

页面存入数组p[],初始化内存块page[]

i++ P[i]是否已在内存中

是 否

Page[]是否有空

否 是

将最先装入page[]中的页面置换出去 直接将p[i]

装入内存,i++

是

输出当前页面的命中率 i<32

结束

LRU算法流程图：

开始

页面存入数组p[],初始化内存块page[]

i++ P[i]是否已在内存中

是 否

Page[]是否有空

否 是

将最近最久未使用的页面从 直接将p[i]

page[]中的页面置换出去 装入内存,i++

是

输出当前页面的命中率 i<32

结束

OPT算法流程图：

开始

页面存入数组p[],初始化内存块page[]

i++ P[i]是否已在内存中

是 否

Page[]是否有空

否 是

将距离最远的页面从 直接将p[i]

page[]中的页面置换出去 装入内存,i++

是

输出当前页面的命中率 i<32

结束

1. **实验数据及处理结果**

随机产生指令流，并给出不同置换策略的命中率表：









经过多次对比发现，发现了OPT命中率较高。

**六、实验体会或对改进实验的建议**

存储管理子系统是操作系统中最重要的组成部分之一，它的目的是方便用户使用和提高存储器利用率。通过这次实验更加清楚了四种置换算法的实现过程，通过比较了解到了他们的异同之处。先进先出算法主要根据哪个进程块进入的早也就出来的早，最近最少使用算法则是进程块在最近这段时间内最少的使用次数与使用次数，最少访问页面算法则是在整个运行过程中，页面的访问次数最少。

在实验的过程中还遇到了一些问题，在整个代码的实现过程中，四个算法的实现有点困难，最后通过询问同学得以实现。