**实验四 存储管理**

# 姓 名 陈正江 学 号 19001531 成绩

实验时间 2023.11.16 指导教师(签名)

**（诚信声明：本实验报告内容，均由本人亲自上机完成。 签名：\_\_\_\_\_\_\_\_\_\_\_\_\_）**

一．实验目的

1.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

2.掌握请求页式存储管理的页面置换算法。

二．实验工具与设备

装有 Linux 操作系统的计算机。

三．实验内容

计算并输出下面各种算法在不同内存容量下的命中率：

1. FIFO（先进先出算法）
2. LRU （最近最少使用算法）
3. OPT （最优算法）
4. LFU （最少使用页面算法）
5. CLOCK （时钟算法）

**1、实验知识**

（1）命中率

命中率 ＝ 1－页面失效次数 / 页地址流长度

其中：页地址流长度为 320，页面失效次数是指每次访问相应指令时，该指令所对应的页不在内存的次数。

（2）随机数生成方法

Linux 系统提供了函数 srand( )和 rand( )，分别进行初始化和产生随机数。

例如：srand( )语句可初始化一个随机数：

a[0]=10\*rand( )/32767\*319+1; a[1]=10\*rand( )/32767\*a[0]; ……

**（1）程序框架设计**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<math.h>

#define TRUE 1

#define FALSE 0

#define INVALID -1

#define NULL 0

#define total\_instruction 320 // 指令流长

#define total\_vp 32 // 虚页长

#define clear\_period 50 // 清零周期

typedef struct{ // 页面结构体

int pn; // 进程的页号

int pfn; // 物理内存的块号

int counter; // 一个周期内访问该页面的次数

int time; // 访问时间

}pl\_type;

pl\_type pl[total\_vp];

struct pfc\_struct{ // 页面控制结构

int pn; // 页号

int pfn; // 块号

struct pfc\_struct \*next; // 指向下一项的指针

};

typedef struct pfc\_struct pfc\_type;

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

// 用户进程虚页控制结构，空页面头指针，忙页面头指针，忙页面尾指针

int diseffect; // 页面失效次数

int a[total\_instruction]; // 指令序列

int page[total\_instruction]; // 每条指令所在的页号

int offset[total\_instruction]; // 页内偏移量

int total\_pf; // 用户进程的内存页面数

void initialize(int);

void FIFO(int);

void LRU(int);

void OPT(int);

void LFU(int);

void CLOCK(int);

**int main()**{

int S, i;

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

S = (int)( 319.0\*rand()/RAND\_MAX ) + 1; // RAND\_MAX一般为常量32767

for(i = 0; i < total\_instruction; i += 4){ // ①产生指令序列

a[i] = S; // 任选一个指令访问点

a[i+1] = a[i] + 1; // 顺序执行一条指令

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

a[i+3] = a[i+2] + 1; // 执行后地址指令

S = (int)( 1.0\*rand()\*(318-a[i+2])/RAND\_MAX ) + a[i+2] + 2;

}

printf("RAND\_MAX = %d \n", RAND\_MAX);

for(i = 0; i < total\_instruction; i++) { //②记录指令序列

printf("%d ", a[i]);

if((i+1)%4 == 0)

printf("--");

if((i+1)%8 == 0)

printf("\n");

}

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

for(i = 0; i < total\_instruction; i++){ // ③将指令序列变换成页地址流

page[i] = a[i]/10; // 记录页号

offset[i] = a[i]%10; // 记录页内偏移量

}

for(i = 4; i<=32; i++){ // 用户内存工作区从4个页面到32个页面

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

FIFO(i);

LRU(i);

OPT(i);

LFU(i);

CLOCK(i);

printf("\n");

}

return 0;

}

**void initialize(int total\_pf)** {  **// 初始化函数**

int i;

diseffect = 0; // 页面缺失次数置位0

for(i = 0; i<total\_vp; i++) { // ①页面结构数组初始化

pl[i].pn = i;

pl[i].pfn = INVALID; // 置访问页面为空

pl[i].counter = 0; // 访问次数为0

pl[i].time = -1; // 访问时间置为-1

}

for(i = 1; i<total\_pf; i++) { // ②建立pfc[i-1]和pfc[i]之间的链接

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

pfc[i-1].pfn = i-1;

}

pfc[total\_pf - 1].next = NULL; // 定义pfc数组尾部元素

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

freepf\_head = &pfc[0]; // 空页面头指针指向pfc[0]

}

**（2）各个算法函数设计**

**1）FIFO（先进先出算法）**

void FIFO(int total\_pf){

int i;

pfc\_type \*p;

initialize(total\_pf); // 初始化相关数据结构

busypf\_head = busypf\_tail = NULL; // 忙页面头尾指针链接

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

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

diseffect += 1; // 页面失效次数+1

if(freepf\_head == NULL) { // 如果没有空闲页面

p = busypf\_head->next;

pl[busypf\_head->pn].pfn = INVALID; // 释放忙页面中的第一个页面

freepf\_head = busypf\_head; // 将忙页面中的第一个页面置为空闲

freepf\_head->next = NULL;

busypf\_head = p;

}

p = freepf\_head->next; // 按照FIFO调新页面放入内存块

freepf\_head->pn = page[i];

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

freepf\_head->next = NULL;

if(busypf\_tail == NULL)

busypf\_head = busypf\_tail = freepf\_head;

else{

busypf\_tail->next = freepf\_head;

busypf\_tail = freepf\_head;

}

freepf\_head = p;

}

}

// 根据公式计算命中率：%6.4表示总共6位数，其中4位小数

printf("FIFO: %6.4f ", 1-(float)diseffect/320);

}

**2）LRU（最近最久未被使用算法）**------赋予每个页面一个访问字段，用来记录相应页面自上次被访问以来所经历的时间time，当淘汰一个页面时，应选择所有页面中其time值最大的页面，即内存中最近一段时间内最长时间未被使用的页面予以淘汰

void LRU(int total\_pf){

// 最久未使用时间、最久未使用页面的页号、访问次数

int iMin, minIndex, i, j, present\_time;

initialize(total\_pf);

present\_time = 0;

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

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

diseffect++;

if(freepf\_head == NULL) { // 如果没有空闲页面

iMin = 32765;

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

// 每次都找到最近最久未使用的那个页面，并且暂存起来

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

iMin = pl[j].time;

minIndex = j;

}

}

// 依照LRU算法置换掉最近最久未使用的页面

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

pl[minIndex].pfn = INVALID;

pl[minIndex].time = -1;

freepf\_head->next = NULL;

}

pl[page[i]].pfn = freepf\_head->pfn; // 有空闲页面则加入内存块

pl[page[i]].time = present\_time; // 更新当前访问时间

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

}

else{

pl[page[i]].time = present\_time;

present\_time++; // 如果命中则增加当前时间

}

}

printf("LRU: %6.4f ", 1-(float)diseffect/320);

}

**3）OPT（最优算法）**------赋予每个页面一个访问距离dist，用来记录相应页面在最长(未来)时间内被访问的距离，当淘汰一个页面时，应选择所有页面中其dist值最大的页面，即内存中在最长(未来)时间内不再被访问的页面

void OPT(int total\_pf){

int i, j, max, maxpage, d, dist[total\_vp];

pfc\_type \*t;

initialize(total\_pf);

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

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

diseffect++;

if(freepf\_head == NULL){ // 如果没有空闲页面

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

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

dist[j] = 32767;

else

dist[j] = 0;

}

max = -1;

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

// 每次都找到距离最大的那个页面并且暂存起来

if(max < dist[j]) {

max = dist[j];

maxpage = j;

}

}

// 依照OPT算法置换掉距离最大的页面

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

freepf\_head->next = NULL;

pl[maxpage].pfn = INVALID;

}

pl[page[i]].pfn = freepf\_head->pfn; // 有空闲页面则加入内存块

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

}

}

printf("OPT: %6.4f ", 1-(float)diseffect/320);

}

**4）LFU（最近最少未被使用页面算法）**------赋予每个页面一个访问次数counter，用来记录相应页面自开始以来的使用次数，当淘汰一个页面时，应选择所有页面中其counter值最小的页面，即内存中使用次数最少的页面

void LFU(int total\_pf){

int i, j, min, minpage;

pfc\_type \*t;

initialize(total\_pf);

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

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

diseffect++;

if(freepf\_head == NULL){ // 如果没有空闲页面

min = 32767;

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

// 每次都找到最近最少使用的那个页面，并且暂存起来

if(min > pl[j].counter && pl[j].pfn != INVALID){

min = pl[j].counter;

minpage = j;

}

pl[j].counter = 0;

}

// 依照LFU算法置换掉最近最少使用的页面

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

pl[minpage].pfn = INVALID;

}

pl[page[i]].pfn = freepf\_head->pfn; // 有空闲页面则加入内存块

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

}

}

printf("LFU: %6.4f ", 1-(float)diseffect/320);

}

**5）CLOCK（时钟算法）-------**为每个页面设置一个访问位cont\_flag，1表示被访问过，0表示未被访问过；如果访问位是0，则选择该页换出，如果访问位是1，则将访问位置为0，暂不换出，设置dp和old\_dp实现页面循环访问，再通过清零周期判断内存中的页面访问次数是否需要归零

void CLOCK(int total\_pf){

int i, j, dp, cont\_flag, old\_dp;

pfc\_type \*t;

initialize(total\_pf);

dp = 0;

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

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

diseffect++;

if(freepf\_head == NULL) { // 如果没有空闲页面

cont\_flag = TRUE;

old\_dp = dp;

while(cont\_flag) {

if(pl[dp].counter == 0 && pl[dp].pfn != INVALID)

cont\_flag = FALSE; // 访问位为0时淘汰

else{

dp++;

if(dp == total\_vp) dp = 0;

if(dp == old\_dp) // 访问位为1时将其更改为0

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

pl[j].counter = 0;

}

}

// 依照CLOCK算法置换掉访问位为0被淘汰的页面

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

pl[dp].pfn = INVALID;

freepf\_head->next = NULL;

}

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

freepf\_head = freepf\_head->next;

}

else

pl[page[i]].counter = 1; // 命中，访问次数置为1

if(i % clear\_period == 0) // 到达清零周期时，将页面次数归零

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

pl[j].counter = 0;

}

printf("CLOCK: %6.4f ", 1-(float)diseffect/320);

}

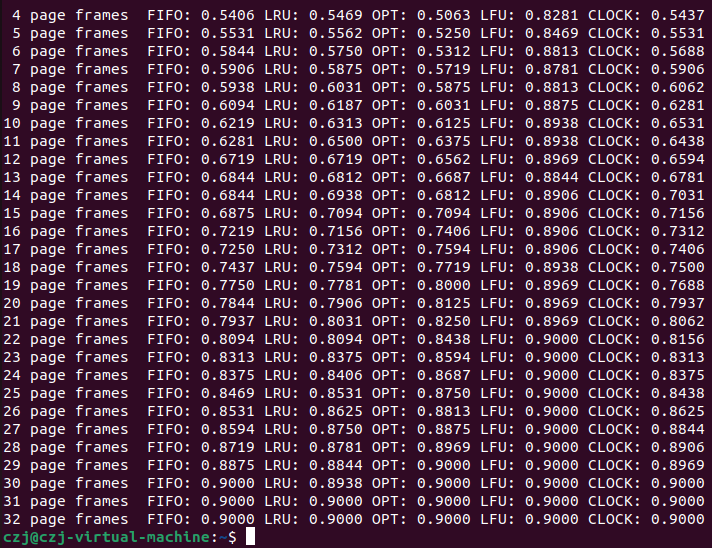
**（3）上机操作与运行结果截图**

·打开终端[czj@czj-virtual-machine~]$，创建c文件，然后进入vim编辑器，编辑该文件；



·按下键盘“i”键，编写 C 语言代码，编辑完成后按 esc 键，输入:wq 保存并退出；

·编译链接test1109\_1.c文件，然后运行该文件，运行截图如下；



# 四．思考题

1. **为什么要进行内存管理，虚拟存储器的特点是什么？**

**（1）**内存管理是计算机系统中非常重要的一种资源管理机制。计算机中的程序在运行时需要占用一定的内存资源，因此需要操作系统来管理内存，确保程序能够正常运行。进行内存管理的主要目的是为了实现内存资源的合理分配、回收和保护；如果操作系统无法有效地进行内存管理，就会导致系统崩溃、程序异常等问题。内存管理的重要性在于：

1）能够提高内存的利用率。内存管理可以实现内存资源的动态分配和释放，使得内存的利用率得到了提高。例如，使用内存池技术和分页技术可以避免内存碎片的产生，节省了内存空间。

2）能够保护系统的稳定性和可靠性。操作系统中运行的程序需要占用一定的内存资源，如果内存管理不当，就可能导致内存泄漏、内存溢出、访问非法内存等问题，从而影响系统的稳定性和可靠性。

3）能够提高系统的安全性。内存管理可以实现内存保护，防止程序之间相互干扰，保障数据的安全性，避免系统被黑客攻击。

4）能够提高系统的性能。合理的内存管理可以减少操作系统的开销，提高内存操作的效率，从而提高系统的性能。

**（2）**虚拟存储器是一种计算机内存管理技术，它能够让程序在运行时感觉到系统提供了比实际内存更大的内存空间。虚拟存储器的特点如下：

1）虚拟存储器屏蔽了物理内存和外部存储器设备之间的差异，使得程序在运行时感觉到系统提供了比实际内存更大的内存空间。这样程序在读写大文件时，可以将文件的部分内容存放在虚拟内存中，减少了对磁盘操作的频率，提高了系统的响应速度和效率。

2）虚拟存储器只有实际需要的内存才会被加载到内存中。这就意味着程序无需运行时一次性加载全部内存到内存中，减少了内存浪费的现象，提高了程序在资源受限的环境中的运行效率。

3）虚拟存储器提高了程序的运行效率，使得程序能够更加顺畅地运行。这是因为虚拟存储器在处理器缓存中维护了多级页面表，使得操作系统可以按需将内存页载入内存中，从而提高了CPU访问内存的速度，减小了CPU空闲时间的占用。

4）虚拟存储器可以提高多任务操作系统下的内存管理效率。多个程序可以共享同一个虚拟地址空间，不会互相干扰，从而提高了程序之间的隔离性和安全性，避免了冲突和相互干扰的现象。

5）虚拟存储器可以实现内存保护机制，保护程序间不会相互干扰，保障程序的稳定性、安全性和数据的完整性。

1. **几种内存管理算法有何区别与联系？**

实验中的五种页面置换算法都是操作系统在运行时对内存中页面进行管理和置换的技术。它们之间的区别和联系如下：

（1）先进先出算法FIFO：该算法是最简单的页面置换算法，使用队列的数据结构将页面进入内存的时间先后顺序记录下来，当内存空间不足时，将队首的页面淘汰。这个算法实现简单，但是不适合内存访问模式比较复杂或者峰值较高的场合。

（2）最近最久未使用算法LRU：该算法依据时间，根据最近一段时间内页面被访问的历史记录进行选择被淘汰的页面。具体实现是根据每个页面被访问的时间戳给页面排序，每次需要置换时淘汰最近没有被访问过的页面。这个算法虽然能够适应较为复杂的内存访问模式，但是需要频繁地更新时间戳，消耗较大。

（3）最优算法OPT：该算法是一种理想的算法，它通过预测未来将要被访问的内存页面来决定淘汰哪些页面。具体实现是根据对内存访问的分析和未来的预测，选择将未来不会用到的页面或者最久不会用到的页面淘汰掉，使得内存中尽可能地留下未来会频繁访问的页面。但是这个算法需要对未来访问模式做出正确的预测，实现难度很大。

（4）时钟算法CLOCK：也称为最近未使用算法NUR，该算法也是依据时间的算法，实现思路是将页面存放到一个环形牵连型缓存中，每个页面都有一个访问位，当被访问时，访问位被置为1；当需要淘汰页面时，从当前指针所指位置开始扫描，把访问位为0的淘汰，访问位为1则将其访问位置为0。这个算法实现相对简单，适用性比较广泛。

（5）最少使用算法LFU：该算法是基于访问频率的算法，它根据页面被访问的频率来选择淘汰哪些页面。具体实现是通过全局访问记录计数，选择访问次数最少的页面进行淘汰。这个算法适用于内存访问分布相对稳定的场合，但是也可能因为一些暂时性的高访问率导致性能下降。

这五种算法都是为了解决内存管理过程中的页面置换问题，适用场景不同，需要根据实际场景选择最合适的算法。