《实验四-存储管理》实验报告

一、实验目的

在 openEuler/Linux 操作系统上，通过模拟实现按需调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，掌握虚拟存储按需调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，并比较他们的效率。

二、实验内容

2.1 实验内容

1.生成内容访问串

首先用 srand()和 rand()函数定义和产生指令地址序列，然后将指令地址序列变换成相应的页地址流。

比如：通过随机数产生一个内存地址，共 100(可自己定义)个地址，地址按下述原则产生：

1) 70%的指令是顺序执行的
2) 10%的指令是均匀分布在前地址部分
3) 20%的指令是均匀分布在后地址部分

具体的实施方法是：

a) 从地址 0 开始；

b) 若当前指令地址为 m，按上面的概率确定要执行的下一条指令地址，分别为顺序、在前和在后：

- 顺序执行：地址为 m+1 的指令；
- 在前地址：[0，m-1]中依前面说明的概率随机选取地址；
- 在后地址：[m+1，99]中依前面说明的概率随机选取地址；

c)重复 b)直至生成 100 个指令地址。

假设每个页面可以存放 10 (可以自己定义)条指令，将指令地址映射到页面，生成内存访问串。

2.设计并实现下述算法，计算访问缺页率，并对算法的性能加以比较。

1) 最优置换算法（Optimal）
2) 最近最少使用（Least Recently Used）
3) 先进先出法（Fisrt In First Out）

其中，缺页率＝页面失效次数 / 页地址流长度

2.2 实验要求

分析在同样的内存访问串上执行，分配的物理内存块数量和缺页率之间的关系；并在同样情况下，对不同置换算法的缺页率比较。

三、实验原理

3.1 生成指令地址序列及内存访问串

1. 数据结构定义

```
#define total_ins 100 //总指令数

#define total_vp 10 //总页数

#define clar_period 50 //清 0 周期

#define INVALID -1

int diseffect;//丢页数

int a[total_ins+1];//指令地址数组

int page[total_ins];//每条指令对应的页表号

int offset[total_ins];

int rd[total_ins+1];//存放 1-total_ins 的随机整数序列，用于索引按比例产生指令地址
```

2. 算法实现

```
rd[1]=(float)total_ins*rand()/32767+1;
for(i=2;i<=total_ins;i++){
    s=(float)total_ins*rand()/32767+1;
    while(isExist(s,rd,i)==true){
        s=(float)total_ins*rand()/32767+1;
    }
    rd[i]=s;
}//产生 1-total_ins 的随机整数序列

printf("生成的地址序列为:\n%d ",a[0]);
for(i=1;i<total_ins;i++){
    s=rd[i];
    if(s<=0.7*total_ins-1){ //a[0]是属于 70%的指令
        a[i]=a[i-1]+1;
        if(a[i]>=total_ins)
            a[i]=0;
    }
    else if(s>0.7*total_ins&&s<=0.8*total_ins){
        a[i]=(float)a[i-1]*rand()/32767;//[0,m-1]的随机地址
    }
    else if(s>0.8*total_ins&&s<=total_ins){
        a[i]=(float)rand()/32767*(total_ins-2-a[i-1])+a[i-
        1]+1; //[m+1,total_ins-1]的随机地址
    }
    printf("%d ",a[i]);
    if(i%9==0)
        printf("\n");
```

```
27. }//生成 total_ins 个指令地址
```

由此可以产生共 total_ins 个指令地址

页面映射原则：

指令地址数：设定为 total_ins

页面存储数：10

从 0 开始，连续的 10 个地址存储在一个页面上，因此算法实现为：

```
1.  for(i=0;i<total_ins;i++) /*将指令序列变换为页地址流*/{
2.          page[i]=a[i]/10;
3.          offset[i]=a[i]%10;
4.  }
```

3.2 页面置换算法设计

3.2.1 三种算法共同点

1.数据结构定义

```
struct pl_type{
    int pn,pfn,counter,time;
};//页表结构
pl_type pl[total_vp];//页表数组
struct pfc_type{
    int pn,pfn;
    pfc_type *next;
};//内存表结构
pfc_type pfc[total_vp],*freepf_head,*busypf_head,*busypf_tail;
int  initialize(int); //初始化页表函数，建立链表
```

2.算法实现

初始化函数首先将每个页表结构体赋值，将所有页表链接生成内存表，并设置内存空页面的头指针为 pfc[0]

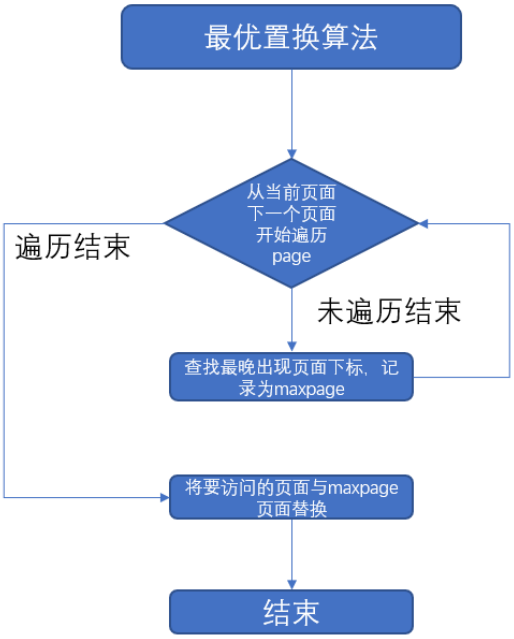
```
1.  int initialize(int total_pf) {              /*初始化相关数据结构*/
2.      diseffect = 0;
3.      // 初始化页表
4.      for (int i = 0; i<total_vp; i++){
5.          pl[i].pn = i;
6.          pl[i].pfn = INVALID;  // 初始值 INVALID、
7.          pl[i].counter = 0;  // 访问次数 0
```

```
            pl[i].time = -1;  // 时间
        }
        // 初始内存表,建立 pfc[i-1]和 pfc[i]之间的链接
        for (int i = 0; i < total_pf - 1; i++){
            pfc[i].next = &pfc[i + 1];
            pfc[i].pfn = i;
        }
        pfc[total_pf - 1].next = NULL;
        pfc[total_pf - 1].pfn = total_pf - 1;
        freepf_head = &pfc[0];  //内存空页面队列的头指针为 pfc[0]

        return 0;
}
```

3.2.2 最优置换算法

1. 算法流程图

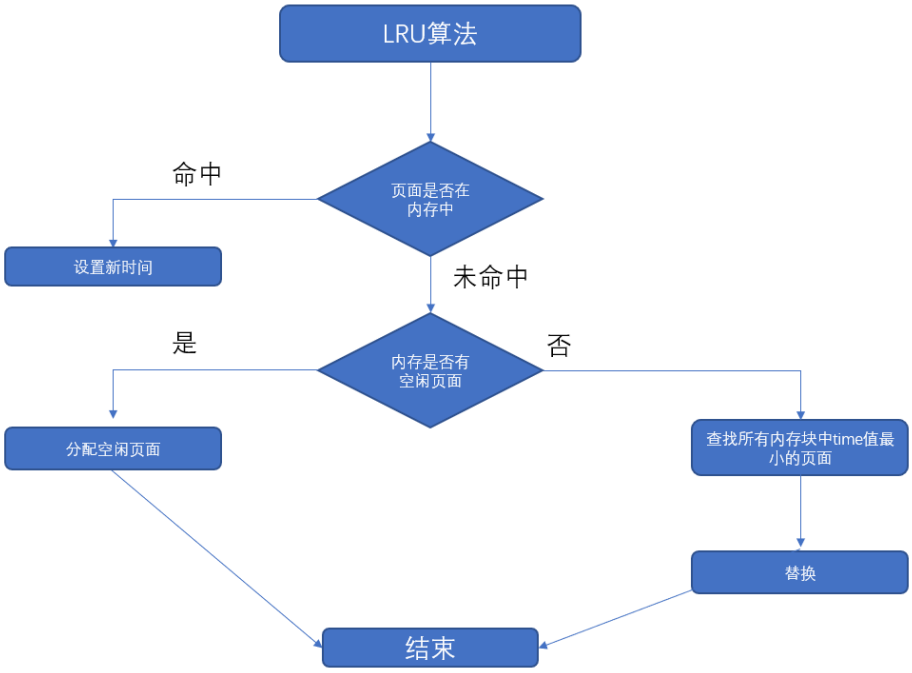


2. 算法步骤

最优置换算法是指替换页面中未来最晚访问的页面，因此需首先遍历整个页表数组，将页表数组中的每个页表在页面访问流中查找，得到每个页表在页面访问流最早出现的时间，将时间中的最大值的页面的下标记为 maxpage，将 maxpage 页面与当前要访问的页面进行替换

3.2.3 LRU(最近最少使用算法)

1.算法流程图

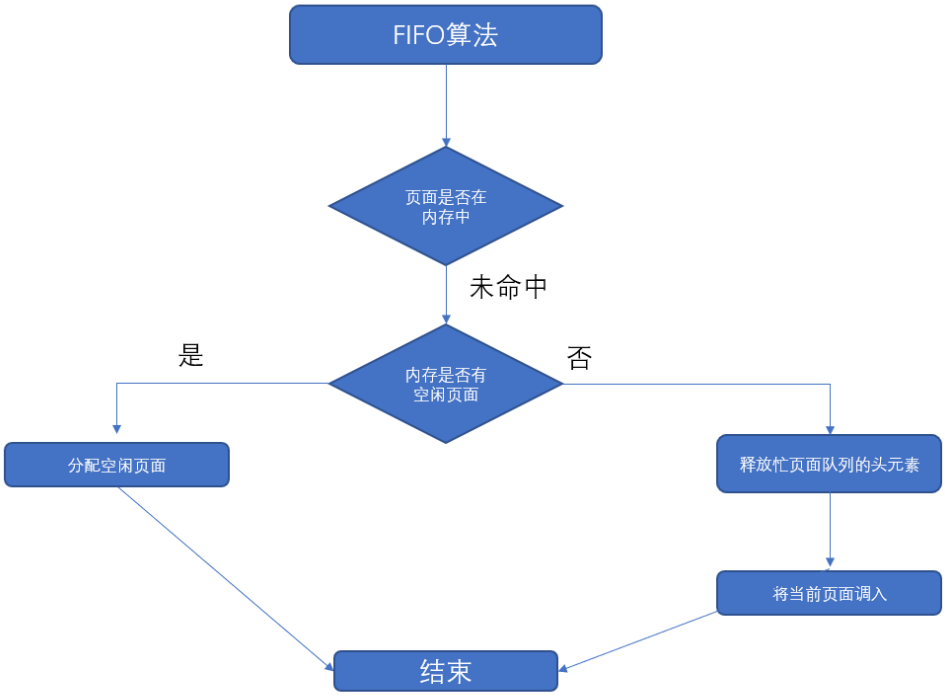


2.算法步骤

首先判断页面是否已经在内存中存在，若命中，则设置其 time 值为当前时间，每一个页面执行完操作后 time 值都会加一，由此来标记页面存在的时间，若未命中，则判断内存汇总是否存在空闲页面位置，若有则将其分配，否则需调用 LRU 算法进行页面置换，查找并记录所有页面中 time 值最小的页面下标，进行置换操作

3.2.4 FIFO(先进先出算法)

1.算法流程图



2.算法步骤

首先判断页面是否已经在内存中存在，若未命中，则判断内存汇总是否存在空闲页面位置，若有则将其分配，否则将忙页面队列中第一个元素释放，再按照FIFO的方式将新页面调入内存页面

四、实验环境

- 操作系统：Ubuntu 16.04 LTS
- 编译环境：g++编译器

五、实验步骤

- 地址序列

```
hexing@ubuntu: ~/Desktop
hexing@ubuntu:~/Desktop$ ./OS-4
生成的地址序列为:
0 0 1 2 3 4 5 6 7 0
0 1 2 3 4 5 6 7 8
9 57 2 3 4 5 6 7 85
10 30 69 70 96 97 98 99 100
100 0 0 1 73 69 96 97 98
99 100 0 1 66 67 76 89 90
91 92 93 94 95 96 97 98 99
100 0 1 2 0 1 97 99 100
0 1 32 39 40 8 9 10 11
12 13 14 15 20 21 22 23 24
25 26 27 24 25 62 47 48 49
```

- 内存访问串

```
生成的内存访问串为:
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
5 0 0 0 0 0 0 8 1 3
6 7 9 9 9 9 10 10 0 0
0 7 6 9 9 9 9 10 0 0
6 6 7 8 9 9 9 9 9 9
9 9 9 9 10 0 0 0 0 0
9 9 10 0 0 3 3 4 0 0
1 1 1 1 1 1 2 2 2 2
2 2 2 2 2 2 6 4 4 4
```

- 九种不同的内存块下的错误率

```
block==2:
FIFO:0.2500
LRU:0.2500
OPT:0.1900

block==3:
FIFO:0.2000
LRU:0.2200
OPT:0.1600

block==4:
FIFO:0.1500
LRU:0.1800
OPT:0.1300

block==5:
FIFO:0.1500
LRU:0.1500
OPT:0.1200

block==6:
FIFO:0.1400
LRU:0.1400
OPT:0.1100

block==7:
FIFO:0.1100
LRU:0.1200
OPT:0.1000

block==8:
FIFO:0.1100
LRU:0.1100
OPT:0.1000

block==9:
FIFO:0.1000
LRU:0.1000
OPT:0.1000

block==10:
FIFO:0.1000
LRU:0.1000
OPT:0.1000
```

统计为表格形式如下：

内存块数	FIFO	LRU	OPT
2	25%	25%	19%
3	20%	22%	16%
4	15%	18%	13%
5	15%	15%	15%
6	14%	14%	14%
7	11%	12%	10%
8	11%	11%	10%
9	10%	10%	10%
10	10%	10%	10%

六、实验结论

6.1 物理内存块数量和缺页率关系

由实验可以看出，在相同的内存访问串上，执行相同的页面置换算法的情况下，物理内存块数量越多，缺页率越少，发生错误的概率越低，当内存块数大于等于一个进程占用的总页数时，由于可以将所有页面都调入内存块中，除刚开始运行阶段，之后便不会产生页错误，因此三种算法的缺页率相同

6.2 不同置换算法的性能

由实验可以看出，在相同的内存块数和相同的内存访问串的条件下，OPT(最优置换算法)的错误率最低，当物理内存块数相对较小时，LRU 算法的错误率大于 FIFO 算法，LRU 调度算法的性能会退化到与 FIFO 调度算法相同，而当内存块数恰当的时候，LRU 算法的效率仍是高于 FIFO 算法的。

《附录》-实验源码

```
#include<stdio.h>
#include<time.h>
#include<stdlib.h>
#define total_ins 100
#define total_vp 10
#define clar_period 50
#define INVALID -1

/*页面结构*/
struct pl_type{
    int pn,pfn,counter,time;
};

pl_type pl[total_vp];//页面结构数组

/*页面控制数组*/
struct pfc_type{
    int pn,pfn;
    pfc_type *next;
};


pfc_type pfc[total_vp],*freepf_head,*busypf_head,*busypf_tail;

int diseffect,  a[total_ins+1];
int page[total_ins],  offset[total_ins];

int  initialize(int);
int  FIFO(int);
int  LRU(int);
int  OPT(int);

bool isExist(int a,int rd[],int length){
    int i=1;
    for(i=1;i<length;i++){
        if(a==rd[i])
            return true;
    }
    return false;
}

int initialize(int total_pf){
```

```
    diseffect = 0;
    int i;
    for(i=0;i<total_vp;i++){
        pl[i].pn = i;
        pl[i].pfn = INVALID;
        pl[i].counter = 0;
        pl[i].time = -1;
    }
    for (int i = 0; i < total_pf - 1; i++){
        pfc[i].next = &pfc[i + 1];
        pfc[i].pfn = i;
    }
    pfc[total_pf - 1].next = NULL;
    pfc[total_pf - 1].pfn = total_pf - 1;
    freepf_head = &pfc[0];  //内存空页面队列的头指针为 pfc[0]
    return 0;
}

int FIFO(int total_pf) {             /*先进先出算法*/
    pfc_type *p;
    initialize(total_pf);  //初始化相关页面控制用数据结构
    busypf_head = busypf_tail = NULL;  //内存页的队列头，队列尾指针接
    for (int i = 0; i<total_ins; i++){
        if (pl[page[i]].pfn == INVALID) {  //页表项不在内存中
            diseffect += 1;                  //失效次数
            if (freepf_head == NULL) {        //内存无空闲页面
                p = busypf_head->next;
                pl[busypf_head->pn].pfn = INVALID;
                freepf_head = busypf_head;  //释放忙页面队列的第一个页面
                freepf_head->next = NULL;
                busypf_head = p;
            }
            // 按 FIFO 方式调新页面入内存页面
            p = freepf_head->next;  // 先保存内存表中当前位置的下一位置
            freepf_head->next = NULL;
            freepf_head->pn = page[i];  // 页表号
            pl[page[i]].pfn = freepf_head->pfn;  // 内存块号

            if (busypf_tail == NULL) {
                // busypf_head 指向最老的，busypf_tail 指向最新的
                busypf_head = busypf_tail = freepf_head;
            }
            else{
                busypf_tail->next = freepf_head;  //free 页面减少一个
```

```
87.                 busypf_tail = freepf_head;
88.             }
89.             freepf_head = p;
90.         }
91.     }
92.     printf("FIFO:%.4f\n", diseffect / (total_ins*1.0));
93.
94.     return 0;
95. }
96.
97. int LRU(int total_pf) {       /*最近最久未使用算法（使用时钟计数器）*/
98.     int min, minj, present_time;
99.     initialize(total_pf);
100.     present_time = 0;
101.     for (int i = 0; i<total_ins; i++){
102.         if (pl[page[i]].pfn == INVALID) {            //页面失效，不在内存中
103.             diseffect++;
104.             if (freepf_head == NULL) {             //内存无空闲页面
105.                 min = 32767;
106.                 for (int j = 0; j < total_vp; j++) {  //找出内存块中 time 的
    最小值
107.                     if (min > pl[j].time && pl[j].pfn != INVALID)  // 查询
    页表
108.                     {
109.                         min = pl[j].time;
110.                         minj = j;  // 记下内存块号
111.                     }
112.                 }
113.                 freepf_head = &pfc[pl[minj].pfn];   //腾出一个单元（对应的内
    存块）
114.                 pl[minj].pfn = INVALID;
115.                 pl[minj].time = -1;
116.                 freepf_head->next = NULL;
117.             }
118.             pl[page[i]].pfn = freepf_head->pfn;   //有空闲页面,改为有效（内存
    块号）
119.             pl[page[i]].time = present_time;
120.             freepf_head = freepf_head->next;      //减少一个 free 页面
121.         }
122.         else {
123.             pl[page[i]].time = present_time;        //命中则设置时间
124.         }
125.         present_time++;
126.     }
```

```
    printf("LRU:%.4f\n", diseffect / (total_ins*1.0));
    return 0;
}


int OPT(int total_pf) {      /*最佳置换算法*/
    int max, maxpage, d, dist[total_vp];
    initialize(total_pf);
    for (int i = 0; i < total_ins; i++) {
        if (pl[page[i]].pfn == INVALID) {      //页面失效，不在内存中
            diseffect++;
            if (freepf_head == NULL) {        //无空闲页面
                for (int j = 0; j < total_vp; j++) {
                    if (pl[j].pfn != INVALID) {
                        dist[j] = 32767;  /* 最大"距离" */
                    }
                    else {
                        dist[j] = 0;
                    }
                }
                d = 1;
                for (int j = i + 1; j < total_ins; j++) {
                    if (pl[page[j]].pfn != INVALID && dist[page[j]] == 3276
7) {
                        dist[page[j]] = d;
                    }
                    d++;
                }
                max = -1;
                for (int j = 0; j < total_vp; j++) {
                    if (max < dist[j]) {
                        max = dist[j];
                        maxpage = j;
                    }
                }
                freepf_head = &pfc[pl[maxpage].pfn];
                freepf_head->next = NULL;
                pl[maxpage].pfn = INVALID;
            }
            pl[page[i]].pfn = freepf_head->pfn;
            freepf_head = freepf_head->next;
        }
    }
    printf("OPT:%.4f\n", diseffect / (total_ins*1.0));
```

```
170.
171.     return 0;
172. }
173.
174.
175. int main(){
176.     int s,i,j;
177.     srand(time(NULL));//使用 time 初始化随机数种子
178.     a[0]=0;
179.     int rd[total_ins+1];
180.     rd[1]=(float)total_ins*rand()/32767/32767/2+1;
181.     for(i=2;i<=total_ins;i++){
182.         s=(float)total_ins*rand()/32767/32767/2+1;
183.         while(isExist(s,rd,i)==true){
184.             s=(float)total_ins*rand()/32767/32767/2+1;
185.         }
186.         rd[i]=s;
187.     }//1 到 100 的数组
188.
189.     printf("生成的地址序列为:\n%d ",a[0]);
190.     for(i=1;i<total_ins;i++){
191.         s=rd[i];
192.         if(s<=0.7*total_ins-1){ //a[0]是属于 70%的指令
193.             a[i]=a[i-1]+1;
194.             if(a[i]>total_ins)
195.                 a[i]=0;
196.         }
197.         else if(s>0.7*total_ins&&s<=0.8*total_ins){
198.             a[i]=(float)a[i-1]*rand()/32767/32767/2;//[0,m-1]的随机地址
199.         }
200.         else if(s>0.8*total_ins&&s<=total_ins){
201.             a[i]=(float)rand()/32767/32767/2*(total_ins-1-a[i-1])+a[i-
    1]+1; //[m+1,99]的随机地址
202.         }
203.         printf("%d ",a[i]);
204.         if(i%9==0)
205.             printf("\n");
206.     }//生成 100 个指令地址
207.
208.
209.     printf("生成的内存访问串为:");
210.     for(i=0;i<total_ins;i++) /*将指令序列变换为页地址流*/{
211.         if(i%10==0)
212.             printf("\n");
```

```
        page[i]=a[i]/10;
        offset[i]=a[i]%10;
        printf("%d ",page[i]);
    }
    for(i=2;i<=total_ins/10;i++){
        printf("\nblock==%d:\n",i);
        FIFO(i);
        LRU(i);
        OPT(i);

    }

}

```