**课 程 实 验 报 告**

**课程名称： 操作系统**

**实验名称： 请求分页式存储管理模拟实验**

**专业班级： 计算机Ⅱ类2103班**

**学 号： 8008121077**

**姓 名： 陈映融**

**指导教师： 胡勇**

**报告日期： 2023/6/22**

## 1.1 实验概述

实验目的：

1. 理解页式存储管理的基本原理。

2. 掌握几种常见的页面淘汰算法。

实验目标：

1. 理解3中淘汰算法(FIFO、OPT和LRU)
2. 补全3中算法缺失的代码
3. 运行实例检测补全成果

实验要求：

1. 补全附件《操作系统-实验6-file.rar》压缩包中pagemmuhard.c 中的代码

实验语言：c

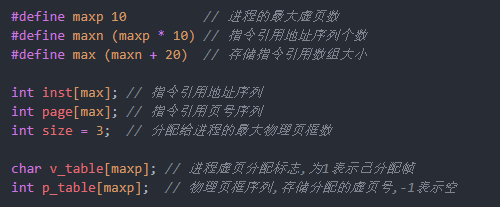
实验环境：linux、gcc

## 1.2 实验过程及结果

**1.2.1 实验代码分析**

从main函数入手，定义的变量choice用于功能的选择，welcome()函数打印本实验的名称和实验的基本内容，然后就是一个死循环用于对功能的选择

* choice == 0退出死循环->实验结束.
* choice == 1调用produce\_inst()函数生成新的引用串
* choice == 2设置空闲页框数
* choice == 6是导入一个引用串
* choice == 3使用FIFO淘汰算法
* choice == 4使用OPT替换算法
* choice == 5使用LRU替换算法



宏定义解释

maxp 和 size maxp是可以分配的最大页框数也就是说可以最大的闲置页框为maxp。size为设置的最大页框数。

maxn是引用串的大小

max是为了能很好的保证存储下maxn的个数所以要+20来保证

主函数代码

int main()

{

    int choice;

    srand(time(NULL));

    welcome();

    while (1)

    {

        input\_hint();

        scanf("%d", &choice);

        printf("\n");

        if (choice == 0)

        {

            printf("OVER!!\n");

            break;

        }

        else if (choice == 1)

        {

            printf("新的引用串:");

            produce\_inst();

            turn\_page\_address();

        }

        else if (choice == 2)

        {

            printf("请输入空闲页框数:");

            scanf("%d", &size);

            if (size < 3 || size > maxp)

                size = 3;

        }

        else if (choice == 6)

        {

            printf("导入的引用串:");

            set\_page\_from\_file();

        }

        else if (choice == 3)

            FIFO\_solve();

        else if (choice == 4)

            OPT\_solve();

        else if (choice == 5)

            LRU\_solve();

    }

    return 0;

}

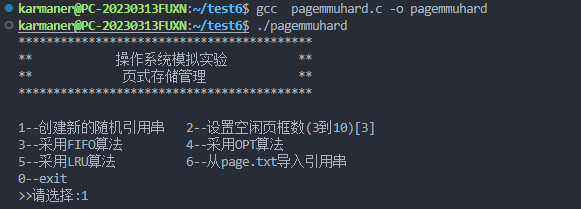
**1.2.2 先来先服务（FIFO）**

基本思想：置换最先调入内存的页面，即置换在内存中驻留时间最久的页面。按照进入内存的先后次序排列成队列，从队尾进入，从队首删除。但是该算法会淘汰经常访问的页面，不适应进程实际运行的规律，目前已经很少使用。

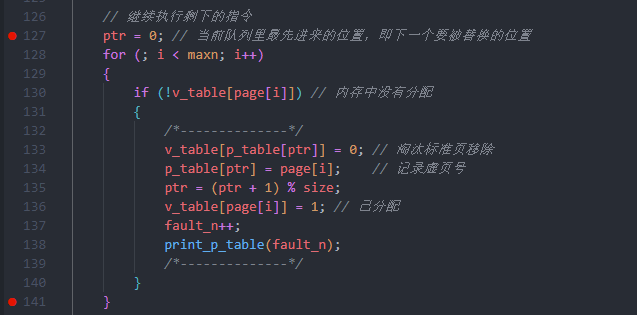
被调换的页是最先调入的页面。所以我们可以使用队列(循环队列)来实现即把ptr作为一个指向最早加入页的索引--每调离一个ptr =( ptr + 1) % size

运行结果如下

编译运行



补全代码如下



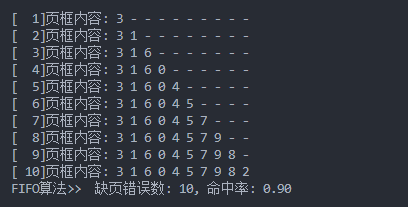
引用串：



3 1 6 3 0 3 4 0 3 6 5 7 4 3 4 9 9 9 6 5 8 5 4 7 7 6 9 6 2 3 0 0 5 2 1 8 6 2 8 4 2 8 4 0 2 2 0 7 7 5 7 9 1 2 1 0 2 1 0 9 2 2 7 7 5 9 4 1 9 9 4 7 8 6 8 5 3 5 7 0 7 4 3 5 6 5 7 7 2 3 1 1 3 0 0 7 9 9 9 2



设置空闲为10运行结果如下

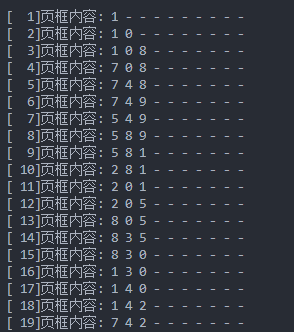
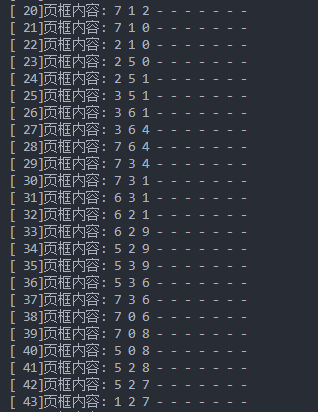


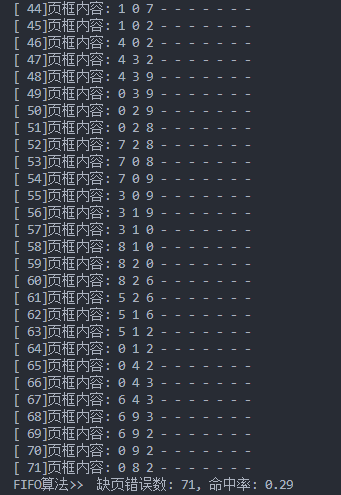
其余全部命中

设置空闲为3运行结果如下

引用串如下：

1 0 8 7 4 8 9 5 8 5 1 2 0 0 5 0 0 8 3 0 1 4 2 7 1 0 2 0 0 2 5 1 3 6 4 7 3 1 6 2 1 9 5 3 6 3 3 6 7 7 7 6 0 8 5 2 2 5 5 7 1 0 2 4 3 9 0 0 2 2 2 8 7 0 9 3 0 1 0 0 8 2 2 6 5 1 2 0 0 1 4 3 6 9 2 9 0 0 2 8



**1.2.3 最佳页面替换法（OPT）**

基本思想：置换以后不再被访问，或者在将来最迟才回被访问的页面，缺页中断率最低。但是该算法需要依据以后各业的使用情况，而当一个进程还未运行完成是，很难估计哪一个页面是以后不再使用或在最长时间以后才会用到的页面。所以该算法是不能实现的。但该算法仍然有意义，作为很亮其他算法优劣的一个标准。

算法的实现基本实现填入前size个记录缺页，在调用OPT淘汰算法(关键就是寻早在p\_table中最晚使用的那个数据(页))然后替换它。

我们需要补充的代码如下

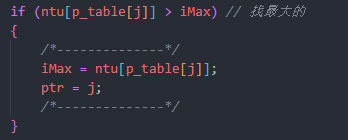


图 2代码补全

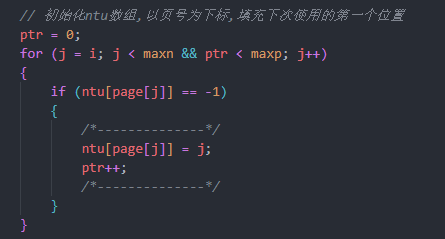
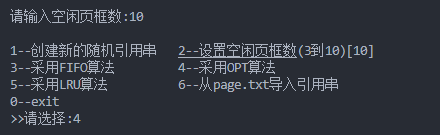


图 1代码补全

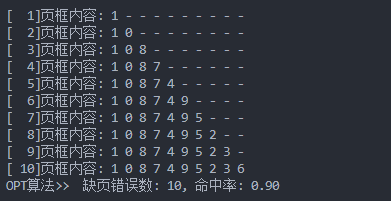
使用引用串：

1 0 8 7 4 8 9 5 8 5 1 2 0 0 5 0 0 8 3 0 1 4 2 7 1 0 2 0 0 2 5 1 3 6 4 7 3 1 6 2 1 9 5 3 6 3 3 6 7 7 7 6 0 8 5 2 2 5 5 7 1 0 2 4 3 9 0 0 2 2 2 8 7 0 9 3 0 1 0 0 8 2 2 6 5 1 2 0 0 1 4 3 6 9 2 9 0 0 2 8

设置空闲页框为10

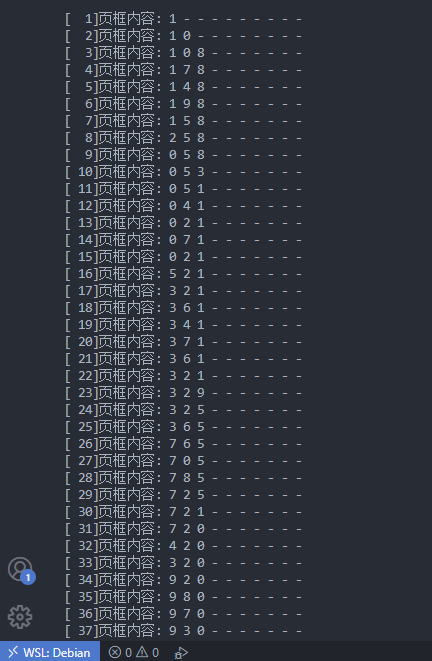


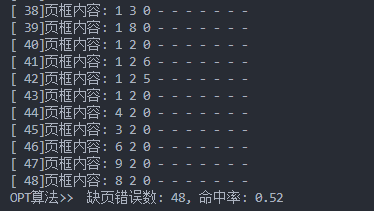
结果如下



设置空闲为3

结果如下：





OPT比较FIFO：从缺页错误率来看在空闲为10时缺页错误数都为10(数据是从0-9决定的)可见在无限大的内存空闲时替换算法是无用的。做出以下对比在空闲为9对算法性能进行比较。

FIFO 9



OPT 9



OPT确实同它的名字一般拥有更高的命中率，在低内存条件下OPT同样拥有更好的命中率，试例中对同一个引用串其命中率是FIFO的两倍。但是命中率提高的同时代码的量和复杂度也更高。

**1.2.4 最近最少使用法（LRU）**

基本思想：置换最近一段时间以来最长时间未访问过的页面。根据程序局部性原理，刚被访问的页面，可能马上又要被访问；而较长时间内没有被访问的页面，可能最近不会被访问。 LRU算法普偏地适用于各种类型的程序，但是系统要时时刻刻对各页的访问历史情况加以记录和更新，开销太大，因此LRU算法必须要有硬件的支持。

LRU的基本思想是寻找到在页框中但是却是最近最少使用的页面如下图2

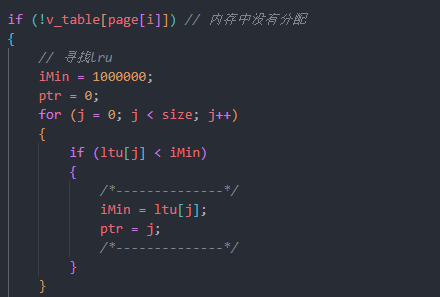


图 2 寻找lru

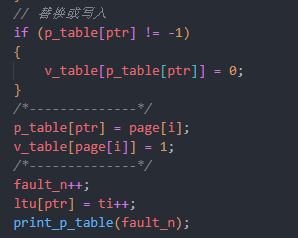
找到最近最少使用的页面接下来对它进行替换 

图 3 替换页

if语句是表示不为空把v\_table标志移除我们要补全的是替换。下面的else语句是对时间进行添加。

完整代码如下

void LRU\_solve()

{

    int ltu[maxp]; *// last\_time\_use*

    int ti = 1;    *// 模拟时间*

    int fault\_n = 0;

    memset(ltu, 0, sizeof(ltu));

    memset(v\_table, 0, sizeof(v\_table));

    memset(p\_table, -1, sizeof(p\_table));

    int iMin, ptr, i, j;

    for (i = 0; i < maxn; i++)

    {

        if (!v\_table[page[i]]) *// 内存中没有分配*

        {

*// 寻找lru*

            iMin = 1000000;

            ptr = 0;

            for (j = 0; j < size; j++)

            {

                if (ltu[j] < iMin)

                {

*/\*--------------\*/*

                    iMin = ltu[j];

                    ptr = j;

*/\*--------------\*/*

                }

            }

*// 替换或写入*

            if (p\_table[ptr] != -1)

            {

                v\_table[p\_table[ptr]] = 0;

            }

*/\*--------------\*/*

            p\_table[ptr] = page[i];

            v\_table[page[i]] = 1;

*/\*--------------\*/*

            fault\_n++;

            ltu[ptr] = ti++;

            print\_p\_table(fault\_n);

        }

        else *// 已经在内存里则只需更改最近使用时间*

        {

            for (j = 0; j < size; j++)

            {

                if (p\_table[j] == page[i])

                {

                    ltu[j] = ti++;

                    break;

                }

            }

        }

    }

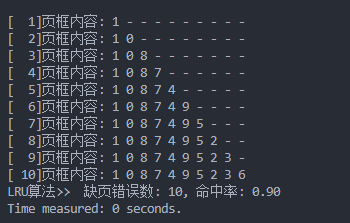
    printf("LRU算法>>  缺页错误数: %d, 命中率: %.2lf\n", fault\_n, (1 - (fault\_n + 0.0) / maxn));

}

引用串：

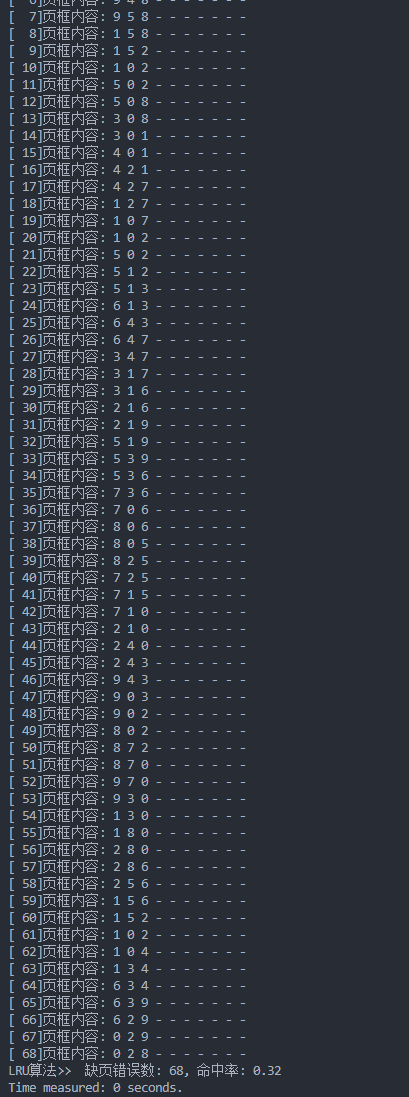
1 0 8 7 4 8 9 5 8 5 1 2 0 0 5 0 0 8 3 0 1 4 2 7 1 0 2 0 0 2 5 1 3 6 4 7 3 1 6 2 1 9 5 3 6 3 3 6 7 7 7 6 0 8 5 2 2 5 5 7 1 0 2 4 3 9 0 0 2 2 2 8 7 0 9 3 0 1 0 0 8 2 2 6 5 1 2 0 0 1 4 3 6 9 2 9 0 0 2 8

空闲框为10



空闲框为3结果如下





对比三个淘汰算法。对于引用串1 0 8 7 4 8 9 5 8 5 1 2 0 0 5 0 0 8 3 0 1 4 2 7 1 0 2 0 0 2 5 1 3 6 4 7 3 1 6 2 1 9 5 3 6 3 3 6 7 7 7 6 0 8 5 2 2 5 5 7 1 0 2 4 3 9 0 0 2 2 2 8 7 0 9 3 0 1 0 0 8 2 2 6 5 1 2 0 0 1 4 3 6 9 2 9 0 0 2 8 而言。OPT > LRU > FIFO算法。

下面有五组数据

引1：



3 1 7 7 1 6 0 0 8 8 2 5 0 0 8 5 1 6 7 7 7 8 2 8 0 0 4 0 0 1 9 8 9 6 2 8 3 0 1 0 0 4 0 0 8 9 1 6 5 2 3 5 4 9 1 1 9 5 4 8 7 4 7 1 1 2 2 0 5 9 5 7 8 3 7 8 3 5 3 0 2 1 1 9 0 0 2 0 0 1 9 9 9 4 2 5 3 0 8 0

FIFO



OPT



LRU



引2：



3 4 5 2 0 6 5 0 7 9 2 8 5 3 6 5 1 4 1 0 5 0 0 5 4 1 9 3 0 1 8 8 8 2 1 4 6 5 6 4 0 1 6 1 5 0 0 2 2 0 2 1 1 2 2 1 8 6 1 7 4 2 5 4 4 4 4 3 3 7 0 5 9 3 9 7 4 8 7 3 9 4 3 9 2 2 6 2 2 8 1 1 5 6 5 8 5 1 1 9

FIFO



OPT



LRU



引3：



6 2 8 0 0 3 7 1 1 0 0 5 5 4 8 2 0 3 4 4 7 1 1 8 7 2 5 8 0 4 7 5 6 8 7 8 9 1 3 6 6 8 8 6 8 7 4 8 7 4 8 8 1 3 7 1 6 8 1 2 2 1 7 0 0 8 5 1 2 4 4 8 9 5 6 6 5 5 0 0 8 6 0 9 5 4 6 8 0 4 8 2 5 5 5 9 2 1 3 1

FIFO



OPT



LRU



引4：



8 5 6 8 6 8 4 3 4 9 0 1 1 0 7 7 2 9 5 2 9 3 3 8 4 3 5 7 3 5 3 3 9 5 0 8 6 1 5 6 4 8 3 0 6 5 0 8 9 9 9 4 3 9 3 1 8 4 0 5 7 0 2 9 4 5 5 1 8 3 1 7 2 0 8 8 7 9 3 1 5 9 4 9 4 2 3 3 1 7 8 0 3 5 2 8 1 0 6 6

FIFO



OPT



LRU



引5：



4 0 0 1 0 9 9 2 3 8 0 8 2 1 3 7 3 6 0 0 5 4 3 5 9 7 9 5 5 9 2 0 1 2 1 7 1 1 4 7 6 8 0 0 6 6 4 5 8 4 5 7 3 6 6 2 6 4 0 4 8 5 6 5 3 9 0 0 7 5 4 9 6 5 6 2 2 8 8 6 8 0 0 4 0 0 7 3 3 4 0 0 7 9 4 5 8 4 9 5

FIFO



OPT



LRU



下面我们对普适行进行验证。设置5个引用串对比淘汰算法优劣。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 引用串 | OPT（1） | LRU（2） | FIFO（3） | 排名情况 | 是否复合结论 |
| 引1 | 0.52 | 0.36 | 0.37 | 1 > 3 > 2 | 否 |
| 引2 | 0.54 | 0.37 | 0.38 | 1 > 3 > 2 | 否 |
| 引3 | 0.54 | 0.33 | 0.34 | 1 > 3 > 2 | 否 |
| 引4 | 0.45 | 0.30 | 0.28 | 1 > 2 > 3 | 是 |
| 引5 | 0.54 | 0.37 | 0.38 | 1 > 3 > 2 | 否 |

表 1 对三种算法的比较

可以看到OPT算法复合他的名字最佳置换。LRU和FIFO性能差不多。但FIFO会比LRU好一点点。但是LRU需要硬件支持这又加高了成本。

## 1.3 实验小结及心得

本次实验难度并不大只要了解了3钟替换算法的原理和仔细阅读pagemmuhard.c的代码完成本次实验的问题是不大的。在第一个实验当中FIFO的实验的第一次忘记了对已经更换的页面进行标志置零即v\_table[p\_table[ptr]] = 0导致FIFO的命中率一直为90%在经过用vscode连接wsl中的debain



图 4 vscode连接wsl

对代码进行调试，不然花费的时间将更多。其他两个函数OPT与LRU函数这两个函数需要补充的代码多但是这两种算法很重要，特别是OPT它的命中率在这3中算法当中最高。

对于请求分页有了新的理解：缺页异常处理程序有可能发生在用户态或者内核态的代码中，在这两种形态下，有可能访问的是内核空间或者用户态空间的内存地址。在不改变内存大小的情况下，请求分页能够提高系统的吞吐量。当进程要访问的页不在内存中的时候，就通过缺页异常处理将所需页调入内存中。这些功能都能通过对系统调用的追踪来实现安装strace工具来实现。