中山大学移动信息工程学院本科生实验报告

（2017 学年春季学期）

课程名称：Operating System　　**任课教师：**饶洋辉　　**批改人(此处为 TA 填写)：**

年级+班级	1506	**专业（方向）**	移动信息工程
学号	15352116	**姓名**	洪子淇
电话	13726205766	**Email**	1102229410@qq.com
开始日期	2017.6.13	**完成日期**	2017.6.15

1. 实验目的

1. 了解页面置换的概念，实现 FIFO、LRU、OPT 三种页面置换的算法。
2. 了解这三种算法的可行性以及优缺点。
3. 复习页面置换的相关内容。

2. 实验过程

(一)实验思路简述

➢ FIFO：

算法描述：最简单的页面置换算法。为每一个也记录该页调入内存的时间，如果需要置换一页的时候就选择最旧的一页。(记录时间的作用只是为了先进先出，所以可以不记录具体时间而通过数据结构来实现)

算法实现思路：

① 每次访问序列的页请求，查找所需页在页表上的位置

② 将请求的页与页表的页表项进行匹配

1. 如果页表中存在命中的页，不进行任何操作

2. 如果页表不存在命中的页

a) 如果页表有空闲的帧，使用压栈的方法将请求的页插入到页表中，此时页错误数加一

b) 如果页表没有空闲的帧，那么就将表尾的帧删掉，再从头插入这次请求的页，此时页错误数加一

③ 判断页请求是否结束，没有结束就回到第一步，结束就按照要求输出相应操作的结果。

- **LRU**

算法描述：全称叫最近最少使用算法，为每个页表项关联一个使用时间域并且为增加一个 counter，对每次访问请求的页的时候，counter++，每次写入页表时，counter 赋值到对应页表项的使用时间域，从而得到了每一页的最近使用时间。当需要置换的时候，只需要置换具有最小时间的页即可（也就是最近最少使用的页）

算法实现思路：

① 每次访问序列的页请求，查找所需页在页表上的位置

② 将请求的页与页表的页表项进行匹配

1. 如果页表中存在命中的页，更新它的时间 count

2. 如果页表不存在命中的页

a) 如果页表有空闲的帧，使用压栈的方法将请求的页插入到页表中，此时页错误数加一，记录它的时间 count

b) 如果页表没有空闲的帧，此时对页表进行遍历找到最小的时间 count 的页（最近最少使用的页），然后请求的页替换掉它，同样需要对请求页的时间进行记录。

③ 判断页请求是否结束，没有结束就回到第一步，结束就按照要求输出相应操作的结果。

- **OPT**

算法描述：最优置换的特点就是可以访问到后面请求的页，当发生页内错误的时候，可以置换掉最长时间不会使用的页。由于不可能知道后来的页请求，该算法在现实中不适用。

算法实现思路：

① 每次访问序列的页请求，查找所需页在页表上的位置

② 将请求的页与页表的页表项进行匹配

1. 如果页表中存在命中的页，更新它的时间 count

2. 如果页表不存在命中的页

a) 如果页表有空闲的帧，使用压栈的方法将请求的页插入到页表中，此时页错误数加一，记录它的时间 count

b）如果页表没有空闲的帧，就遍历整个页表，对于每一个页表项和后面的请求序列进行比较：如果后面都没有出现这个页表项就把这个页表项压入临时的一个容器

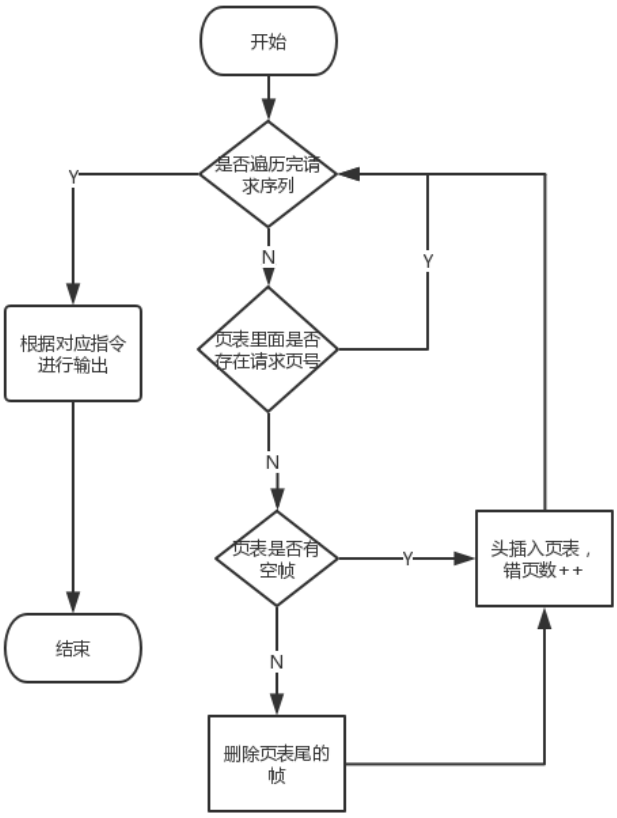
temp 中；如果找到就更新它最晚出现的时间，同时记录到 max 值，方便和页表不同的页表项进行比较，并记录需要被替换的位置（loc）。

c）根据上面遍历的结果，不出现的是要被替换的（temp 容器获得）；没有不出现的情况下，最晚出现的是要被替换的（利用上面得到的位置 loc）。

③ 判断页请求是否结束，没有结束就回到第一步，结束就按照要求输出相应操作的结果。

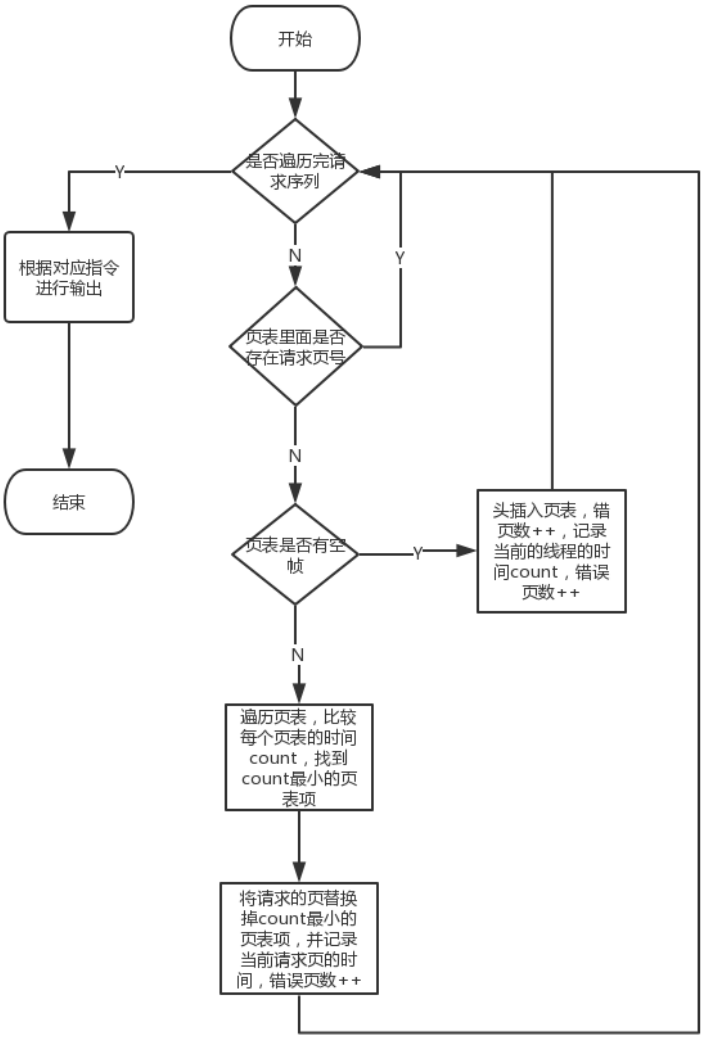
(二)流程图

- **FIFO**



（图一）

- **LRU**

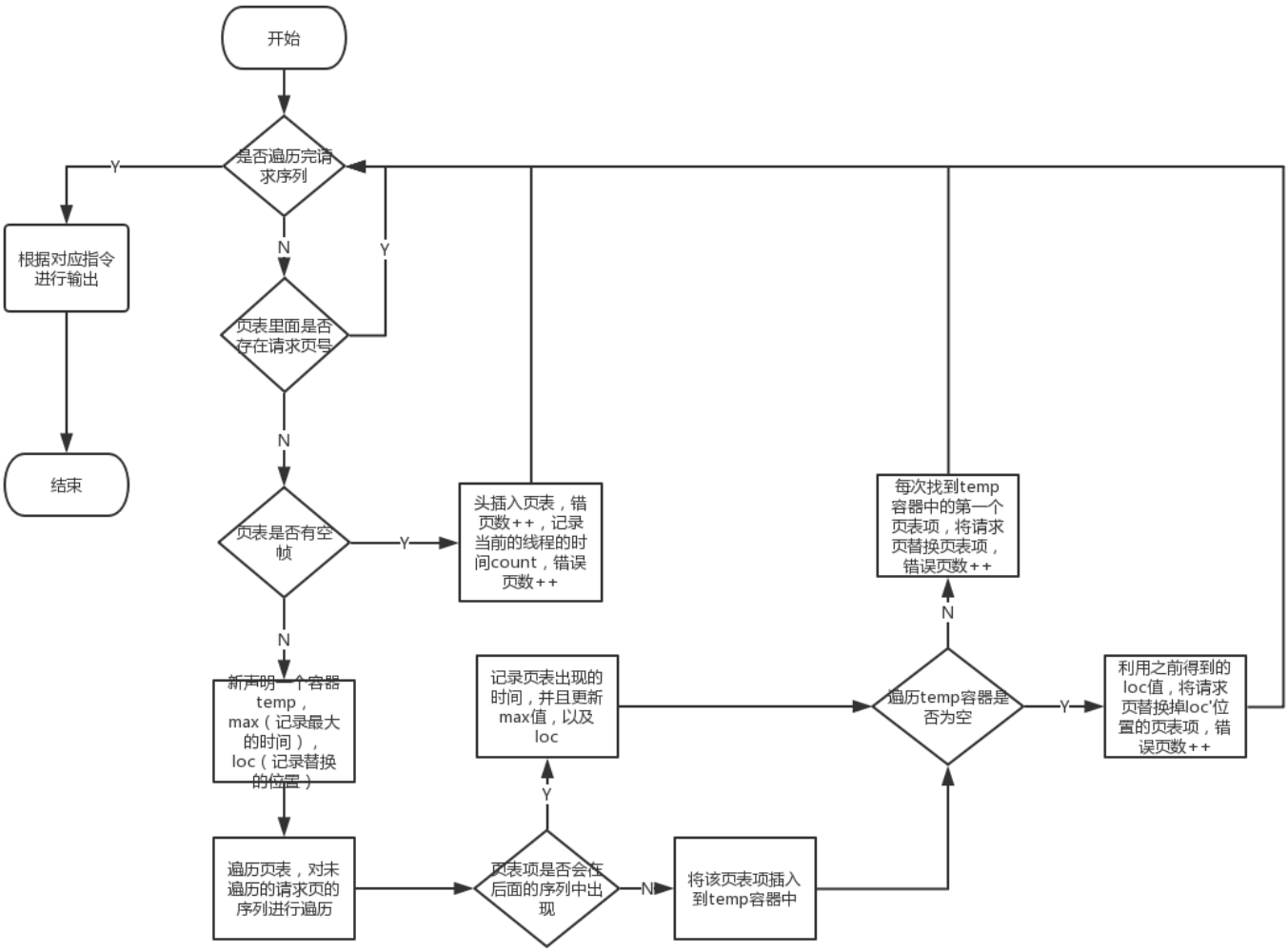


（图二）

（这一切都是为了排版好看啊，太小的流程图好难看啊。。。。。在这里插入一些对于根据命令输出的实现思路把。。其实也没什么，就是写成了一个函数；

而且实验流程图是根据实验思路写的，到我打代码的时候，由于 count 在 LRU 和 OPT 都利用到了，所以我把 count 放在了外面，对于 FIFO 的算法来说，其实也实现了页号长度的时间记录，不过这在 FIFO 下面的判断之中其实没有利用到时间 count。）

- **OPT**



（图三）

（三）具体实现

主要的算法思路其实在实验过程分析已经提到过了，现在就是如果将三个算法有机得联合到一起，我分开了两个函数去实现，一个是 main 函数，一个是_prinft 函数，后者主要是不同页面置换的命令代码的重用性考虑的，所以在全局中定义了容器 frame

（页表），以及 B_inside（是否在页表内），page_fault（错误页数）：

```
vector<char> frame; // 页表
int page_fault; // 页表错误
bool B_inside;  // 1:in; 0: not in
void _printf(string oper)
{
    if (oper == "get")
    {
        char B;
        cin >> B;
        if (find(frame.begin(), frame.end(), B) == frame.end())
            B_inside = 0;   //查找 B，B 在页表内就输出 1，不在就输出 0
        cout << B_inside << endl;
    }
    else if (oper == "pf")
        cout << page_fault << endl; //输出错误页数
    else    //oper == 'seq'
    {
        // cout << '*';
        for (int i = 0; i < frame.size(); ++i)
            cout << frame[i];   //遍历页表，输出页表项
        cout << endl;
    }
    }
```

由于对于一开始的空的页表来说，不需要对页表项进行替换，所以一开始的操作是相同的，下面是相关函数的声明以及一开始插入页表的操作：

```
int n;  //frame 的长度
cin >> n;
string request_page;    //  请求页序列
cin >> request_page;
int k;  //k 次查询操作
cin >> k;
//对于一开始的页表来说，如果三个算法的操作都是一样的
while(frame.size() < n && i < A)
{
    if (find(frame.begin(), frame.end(), request_page[i]) != frame.end())
            i++;    //can find
    else        //can not
    {
            page_fault++;   //没找到就错误页数加一
             frame.push_back(request_page[i++]);
```

```
        }
       //这是为了在 LRU 以及 OPT 算法中使用到 count 而记录的
        count[request_page[i]-'0'] = i;  }
```

➢ FIFO

根据实验思路实现，这个算法比较简单，就不详细分析了：

```
if (choice == 1){
    while (i < A)
    {
        //cout << "--------------" << endl;
        if (find (frame.begin(), frame.end(), request_page[i]) != frame.end())
            i++;    //can find
        else        //can not
        {
            page_fault++;
            frame.erase(frame.begin());
            frame.push_back(request_page[i++]);
        }
    }
    _printf(oper);
    }
```

➢ LRU

在需要进行页表替换的时候，这一个算法就是使用了一个 temp 和 min 的临时变量，每次遍历页表的时候，对页表项的时间 count 进行比较，当 count 较小的时候就更新 min 的值，并且记录下 min 最小的时候的页号是多少，在遍历完页表之后，再次对页表进行一次遍历，主要目的就是为了找到第一次遍历的时候记录下的页号，从而将它替换掉。

在其他情况下，和 fifo 的操作一致，命中的时候就不需要进行其他操作。

【count 定义，由于这次的序号只有 0~9，所以可以开一个大小为 10 的数组存储每个页号的请求时间，页表为空的时候就开始记录到请求页的时间。通过遍历请求序列的 i 值来表示时间。】

```
else if (choice == 2)
{
    while (i < A)
    {
        //找到不用进行任何操作，在后面的时候记录页的时间即可
        if (find (frame.begin(), frame.end(), request_page[i]) != frame.end());
        else        //can not
        {
            int temp;   //保存最近最少请求的页，表现为记录时间小
            int min = 1000;   //初始化 min
            for (int j = 0; j < n; ++j)
                if (min > count[frame[j] - '0'])
                {
                    min = count[frame[j] - '0'];
                    temp = frame[j];    //记录时间小的那个页项
                }
            for (int j = 0; j < n; ++j)
                if (frame[j] == temp){  //找到时间最小的页的位置，将其替换成请求页
                    frame[j] = request_page[i];
                    break;
                }

            page_fault++;
        }
        count[request_page[i]-'0'] = i; //无论找得到找不到都记录页号的时间
        i++;
    }
    _printf(oper);
}
```

➢ OPT

这个算法也是，在命中的时候不需要进行其他操作，只有当需要进行页面置换的时候才开始考虑如何去实现置换。首先这个算法的思路和 LRU 很像，只不过一个是从前出现的时间进行比较，一个是针对后面出现的时间进行比较；由于时间一开始就有记录了，而且对于页表中的页表项在前面肯定是出现，所以在实现上比 OPT 要简单一些。

在需要进行页面置换的时候，需要新定义一个容器 temp，主要是存放不再在后面出现的页表项（也可以理解为它出现的时间是无限长的），还有新定义了一个 max 以及一个 loc；主要

的作用：在页表项会在后面的请求序列出现的情况下，记录了最晚出现的时间，以及对应页号的位置。期间也是利用了 count 进行时间的记录，从而对 max 以及 loc 进行更新。

```
int max = -1;
int loc;
int min = 100;
vector <char> temp; //用于存储之后不出现的请求的页
temp.clear();   //每次请求都需要清空
```

首先是遍历一次页表，遍历请求序列的看是否找到页表项，没找到就压入 temp，找到就记录时间 count，并更新 max 值和 loc 值：

```
for (int j = 0; j < n; ++j) //遍历整个页表
{
    for (int p = i; p < request_page.size(); ++p)   //遍历后面请求的序列
    {
        if (frame[j] != request_page[p] && p == request_page.size() -1) //找不到的页插入 temp 中。
            temp.push_back(frame[j]);
        else if (frame[j] == request_page[p])
        {
            if (max < p)
            {
                max = p;
                loc = j;    //记录页表项需要被替换的位置。
            }
            break;
        }

    }
}
```

遍历完之后根据 temp 的情况进行置换，如果 temp 不为空，就先把 temp 里面的页对应的页表项给置换掉，如果为空，就证明页表中的页表项都在后面的请求序列出现过，这时候就需要把最晚出现的页表项给替换掉，在上一个循环遍历中我们已经得到了最晚出现的页表项的位置 loc，所以直接利用 loc 进行置换即可。

```
if (!temp.empty())  //先替换以后都不出现的请求的页（根据最先进入 frame 的顺序去替换）
{
    char tihuan;
    for (int j = 0; j < temp.size(); ++j)
```

```
        if (min > count[temp[j] - '0']){
            min = count[temp[j] - '0'];
            tihuan = temp[j];
        }
    for (int j = 0; j < n; ++j)
        if (frame[j] == tihuan)
        {
            frame[j] = request_page[i];
            break;
        }
}
else    frame[loc] = request_page[i];   //再替换之后出现最晚的请求的页
page_fault++;
```

最后记录下该页的时间以及进行 i 值的加一：

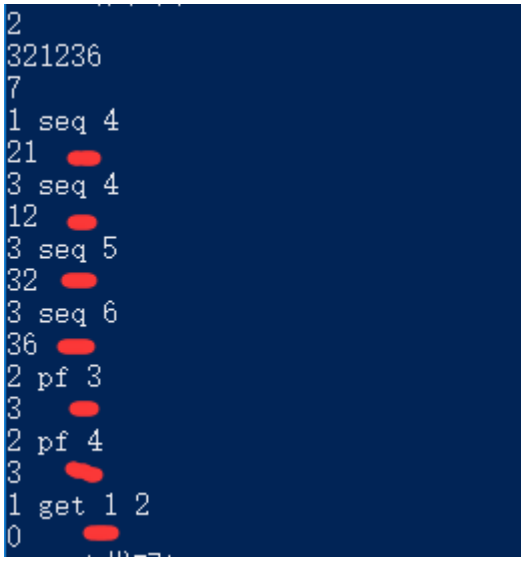
```
count[request_page[i]-'0'] = i;
i++;
```

3. 实验结果

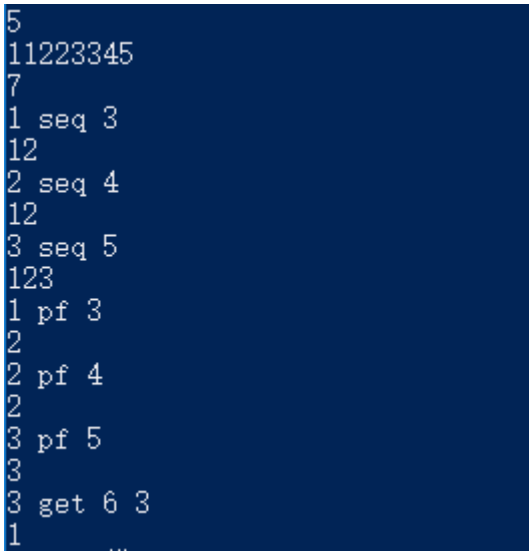
通过情况：

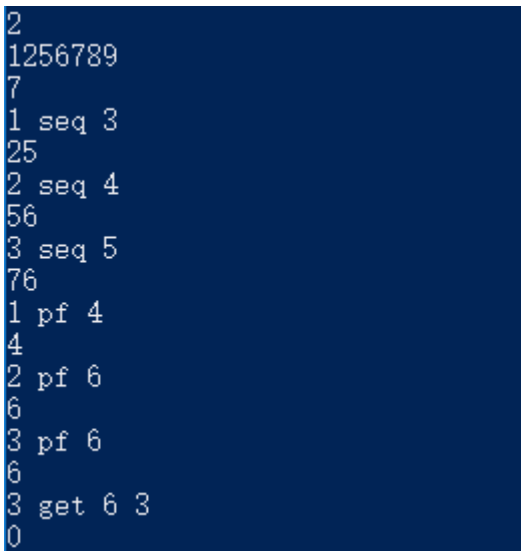
158	15352116	洪子淇	smie15352116	牛奶君	✔Yes	1	9168

对于 sicily 的样例情况：



其他样例测试：





4. 回答问题

- 1. 阅读 PPT 中“基本页面置换过程”部分，回答：有哪些方法可以减少页 面置换过程中的开销？

答：

根据 PPT 的页面置换过程，可以设置脏位来降低消耗。脏位的定义是，每一个页帧都有一个脏位，用于记录页帧是否被修改。当存在脏位，就证明该页的内容被修改过了（已经脏掉了），所以就需要把该页写入到磁盘上。而对于‘干净’的页帧来说，该页没有发生修改，因此就不需要把该页写入到磁盘上面。

这样就可以减少频繁的内存与硬盘中的数据交换，减少页面置换过程的开销。

- 2. 对比三种页面置换算法

答：

FIFO：该算法的优点在于实现代码简单，复杂度低，也不需要开其他的变量或者数据结构去存储时间的值，节省了内存上面的开销。不过这个算法有一个很大的缺点在于没有考虑到页表项的出现的频次，或者说重要性，无论什么页表项，即使是下一个时刻需要用到也会被置换出去，导致了错误页数的增加。

OPT：对于这个算法来说，虽然比 FIFO 增加了一定的内存空间消耗来记录每个页表项的时间，可是由于考虑到页表项的出现的顺序（未来出现的情况），可以有效地降低错误页数。从而实现最优的页面置换。不过由于实际情况中不可能预知到未来出现的请求页。所以对在实际情况中这种页面置换算法不能实现。

LRU：这个算法就是针对了 OPT 在现实生活中不能实现而提出来的，通过最近页号的使用情况来权衡页号的重要（对于最近最少使用的页号进行了替换，表示了那个页号可能不是很重要。不过这也不能完全判断，存在局部思维，是一种统计概率的思维）。对比 OPT 算法出现对于页表项重要性的局部的判别错误，可是对比 FIFO 算法来说，虽然同样增加了内存的消耗（用于记录时间），可是错误页数会有所下降。

- 3. 查阅并介绍其他常用的页面置换算法。并说出它们的使用情况。

答：

最近未使用法（NRU）：

主要思想是找到最久没有使用的页，页被访问的时候设置 R 位，修改设置成 M 位，R 位表示定期清零

页的分类有四种：

0：未访问；M = R = 0

1：未访问；M = 1， R = 0；

2：访问；M = 0， R = 1；

3：访问；M = R = 1；

算法是从组数最小的一组随机选择一个页进行替换。由于使用到脏位，所以减少了页面置换的开销。

5. 实验感想

这一次实验主要是对理论课的页面置换的内容进行了复习以及回顾，加深了理论课知识的印象，对于这三种页面置换有了更深刻的理解。然后就是在进行代码实现的时候，对于一些细节上的思考，在这次实验中由于一个判断语句中没有加上 break 而导致了 bug 的出现，还有一些代码重用性的思考可能也太详细了，虽然对于一开始的空的页表来说，三种页面置换的操作都是一样的，可是由于 FIFO 没有用到 count 值，而一开始我是在在判断页面置换类型之后，在 LRU，OPT 算法里面才定义 count 值，由于一开始没有记录，所以我在后面开始记录时间，由于只是简单的一个 for 循环（j : 0 ->frame.size();count[frame[j]-'0'] = j;）对其进行初始化，这种情况对于一开始重复出现的请求序列是错误的例如 11111222111 的时候，最后将 count 的初始化也拉入到一开始的 while 循环中才通过了这一题。