操作系统作业说明文档

——请求调页存储管理方式模拟

学号：1950072

姓名：郑柯凡

指导老师：张惠娟

1. 项目背景

模拟内存管理中请求调页的存储方式，实现页面、页表、地址间的转换；页面的缺页置换；计算缺页数及缺页率等功能，加深对请求调页系统的原理和实现过程的理解。

1.1 基本假设

1. 每个页面可以存放10条指令。
2. 一个作业分配4个内存块，即一个作业在内存中最多只有4个页面。
3. 一个作业共有320条指令。
4. 开始时所有页都还没有调入内存。

1.2 基本需求

1. 模拟一个作业的执行过程，若所访问的指令在内存中，则显示其物理地址，并转到下一条指令；若所访问的指令不在内存中，即发生缺页情况，则记录缺页次数，并将其调入内存中，如遇内存块已满的情况则还需进行页面置换。
2. 按照50%的指令顺序执行，25%的指令均匀分布在前地址部分，25%的指令均匀分布在后地址部分的原则设置指令访问次序。
3. 一个作业执行完成后，计算并显示作业执行过程中发生的缺页率。

2.开发环境

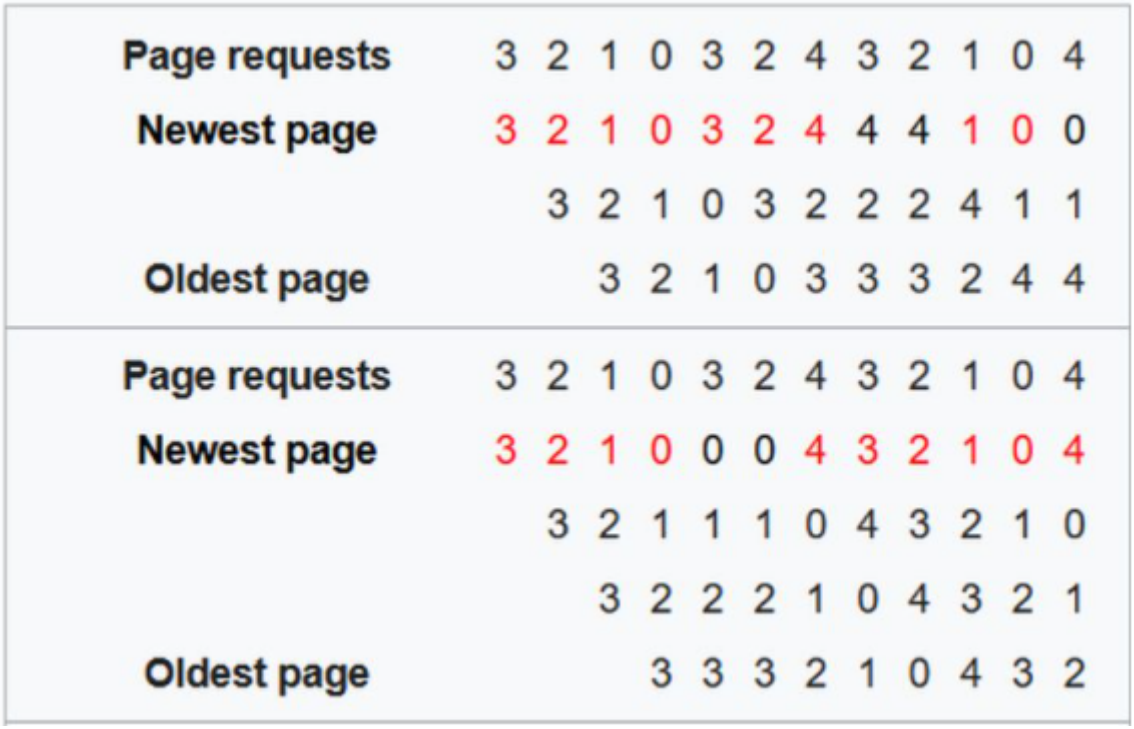
- 操作系统：Windows 10
- 开发软件：Qt5.9.9 + Qt Creator 4.11.0
- 开发语言：C++

3. 算法分析

在计算机存储结构中，磁盘容量大但访存慢，内存访存快但容量有限。所以在程序的执行过程中，通常会先将所需的数据装载到内存中再交由CPU处理。在请求调页存储方式中，一个作业分配的内存页数通常是比实际物理页数小很多的，所以当程序所要访问的页面不在内存中且内存也没有空闲页面时，就需要替换掉内存中的某个旧页面。这个旧页面的选择也根据不同的置换算法而变化，常见的置换算法有最佳置换算法（OPT）、先进先出置换算法（FIFO）、最近最久未使用算法（LRU）以及时钟置换算法（CLOCK）。本章主要介绍项目中使用到的LRU算法和FIFO算法。

3.1 FIFO算法

FIFO算法，即先进先出置换算法，是最早出现的一种置换算法。其利用一个队列存储内存中的页面，每次需要替换一个内存页时，就直接出队一个元素，也即在当前内存页中最早被调入内存的页面。其算法原理是，最早被调入内存的页面在之后被使用的可能性更小，所以选择将其替换。然而，该原理存在一定的不合理性，页面被调入内存的时间和其之后被访问到的概率之间并没有必然的联系。因此，FIFO算法会产生一种名叫Belady的异常，即随着内存块数的增加，缺页数不降反增的现象。



在上图使用FIFO算法的例子中就产生了Belady异常，在同样的页面访问顺序下，内存页增加了一块，但缺页数反而还增加了一次。

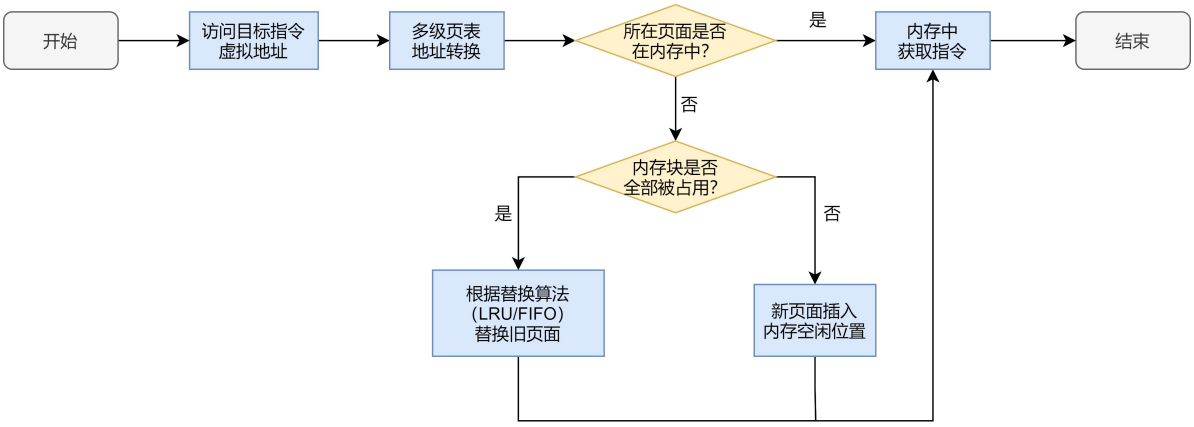
3.2 LRU算法

LRU算法，也叫做最近最久未使用替换算法，是一种常见的为虚拟页式存储管理服务的页面置换算法。其设计原理基于程序的局部性原理，通过一个作业过去的页面访问情况来预测其未来的行为，即假设在最近一段时间内经常被访问的数据在之后也会经常被访问，而在最近一段时间内没有访问过的数据在之后也大概率不会经常访问。因此，当需要替换内存中的一个页面时，总是选择替换最近访问记录最久远的内存页。相比于FIFO算法，LRU算法更稳定，性能更好。

4.系统设计

4.1 逻辑设计

一条指令的执行过程如下：



4.2 核心代码设计

- 多级页表地址转换

  本项目采用三级页表的地址映射形式，一个作业有32个页面，编号分别是0至31，因此将三级页表分别划分为4、4、2的形式，即一级页表存储4个地址索引，分别对应4个二级页表；每个二级页表也存储4个地址索引，分别对应4个三级页表；每个三级页表存储2个真实的页表信息。虽然在本题中并未划分明确的除内存页外的内存空间大小，但多级页表在实际中能有效降低用户空间的内存浪费。

```
//虚拟地址页表号初始化
for(int i=0;i<320;i++)
{
    //目标页（目标指令所在的页面）在一级页表中的索引地址
    viraddtable[i].modifypageContentID((i/10)%4);
    //目标页在二级页表中的索引地址
    viraddtable[i].modifypageTableID2(((i/10)%16)/4);
    //目标页在三级页表中的地址，该地址存储的不是索引而是真正的地址信息
    viraddtable[i].modifypageTableID3(((i/10)%32)/16);
    //目标指令在页面中的偏移量
    viraddtable[i].modifypageOffsets(i%10);
}
--------------------------------------------------------------------------
//搜索物理地址函数
int app::searchPhyaddress(VirAddress src,int in)
{
    int i=src.getpageContentID();
    int j=src.getpageTableID2();
    int k=src.getpageTableID3();
    if(in==1)
    {
        return this->pagetable_3[i][j][k].getmainMemoryID();
    }
    else if(in==0)
    {
        return this->pagetable_3[i][j][k].getdiskMemoryID();
    }
    return 0;
}
```

- 置换算法具体实现

  LRU算法：通过比较内存页的最近访问时间获取最近最久未被访问的内存页编号

  FIFO算法：通过比较内存页的调入时间获取最早被调入的内存页编号

```
int app::replace(VirAddress vir)
{
    //找到目标在磁盘中的地址（页号）
    int diskadd=searchPhyaddress(vir,0);
    //找到替换的目标(根据LRU或FIFO)
    int replace=0;
    //LRU
    if(algorithm==0)
    {
        for(int i=1;i<4;i++)
        {
            if(mainmemory[i].getnearesttime()
<mainmemory[replace].getnearesttime())
            {
                replace=i;
            }
        }
    }
    //FIFO
    else if(algorithm==1)
    {
        for(int i=1;i<4;i++)
        {
            if(mainmemory[i].getentertime()
<mainmemory[replace].getentertime())
            {
                replace=i;
            }
        }
    }
    //修改旧页面的页表信息
    int prepageid=mainmemory[replace].getpageid();
    int o=prepageid%4;
    int p=(prepageid%16)/4;
    int q=(prepageid%32)/16;
    pagetable_3[o][p][q].modifymainMemoryID(-1);
    //交换并更新内存信息
    string tmp[10];
    for(int i=0;i<10;i++)
    {
        tmp[i]=disk[diskadd].getinstru(i);
    }
    mainmemory[replace].modifymemo(tmp);
    mainmemory[replace].modifypageid(diskadd);
    return replace;
}
```

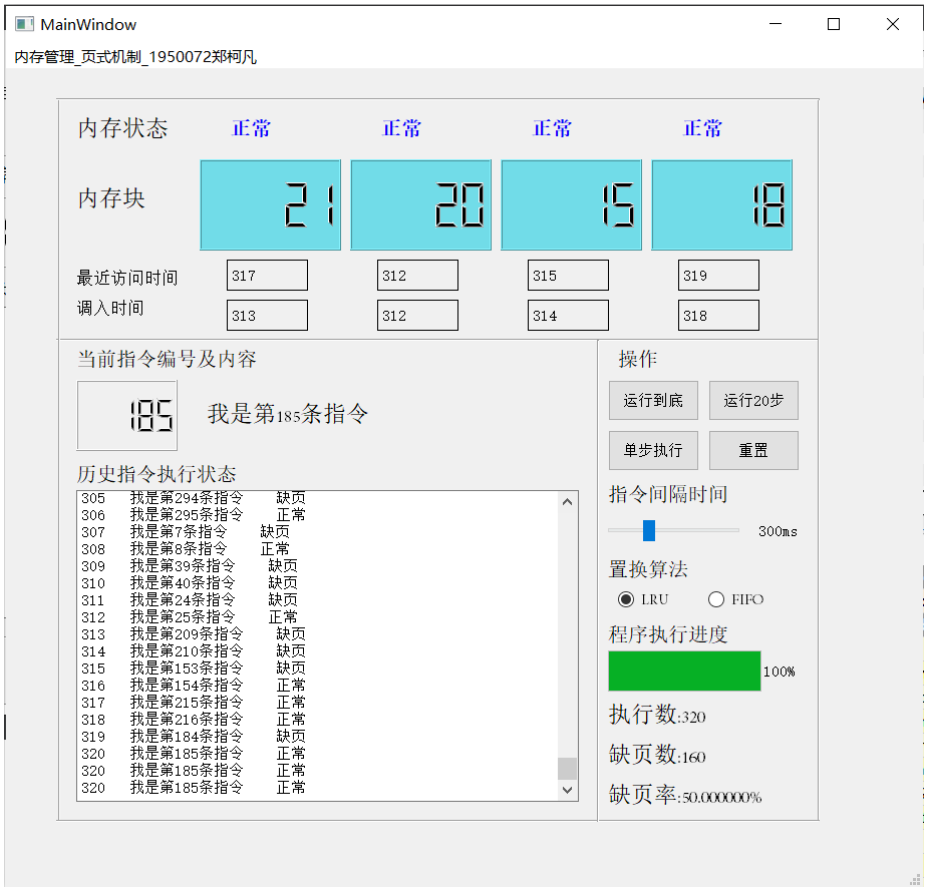
- 指令访问次序设置

本项目的指令次序具体实施方法为：先在0至319条指令之间随机选取一条指令执行，假设其为m，然后顺序执行下一条指令，即$m+1$；再通过随机数跳转到前地址部分，即0至$m-1$中的某一条m_1，再顺序执行m_1+1；再通过随机数跳转到后地址部分，即m_1+2至319中的某一条m_2，再顺序执行m_2+1。通过这样不断地顺序执行再跳转的步骤就能实现规定的指令访问次序原则。

```
//随机地址顺序执行
srand((unsigned)time(NULL));
int start=rand()%320;
runandupdateUI(start);
runandupdateUI(start+1);
while(myApp.getallnum()<320)
{
    //前地址顺序执行
    int addr=rand()%start;
    runandupdateUI(addr);
    runandupdateUI(addr+1);
    //后地址顺序执行
    addr=rand()%(320-myApp.getcurinstruid())+myApp.getcurinstruid();
    start=addr;
    runandupdateUI(addr);
    runandupdateUI(addr+1);
}
```

5. 项目演示

5.1 界面展示

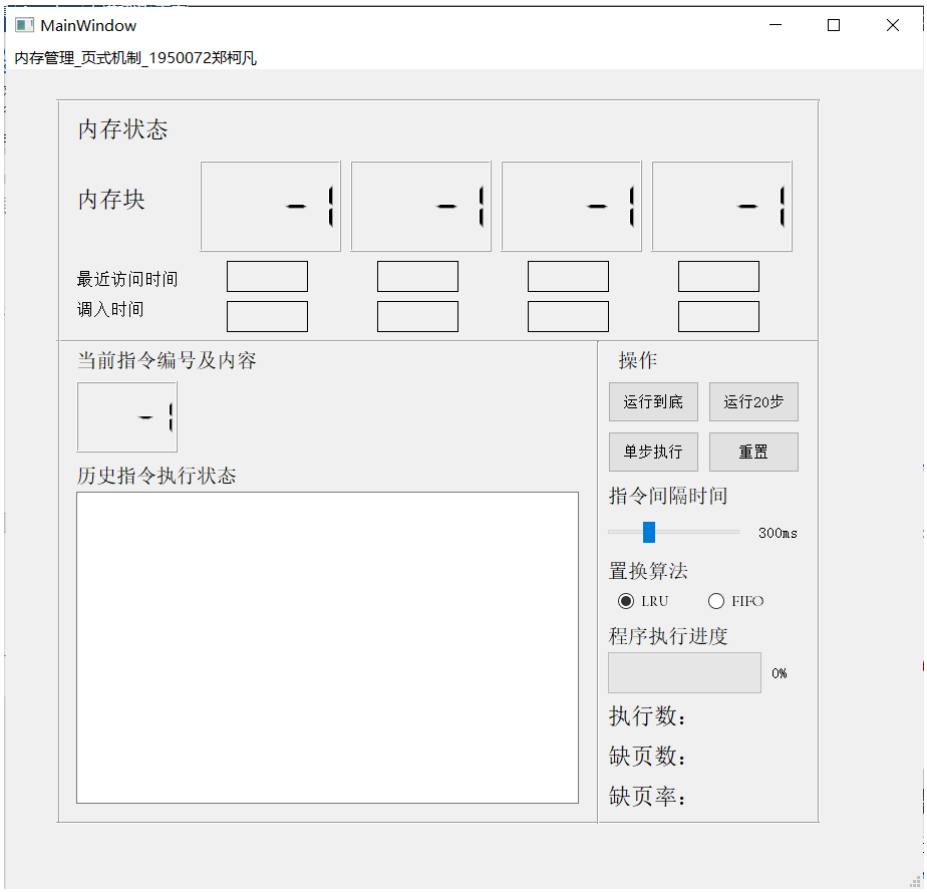


整个项目界面主要划分为3个部分，分别为顶部内存状态显示区、左下部指令展示区以及右下部用户操作区。

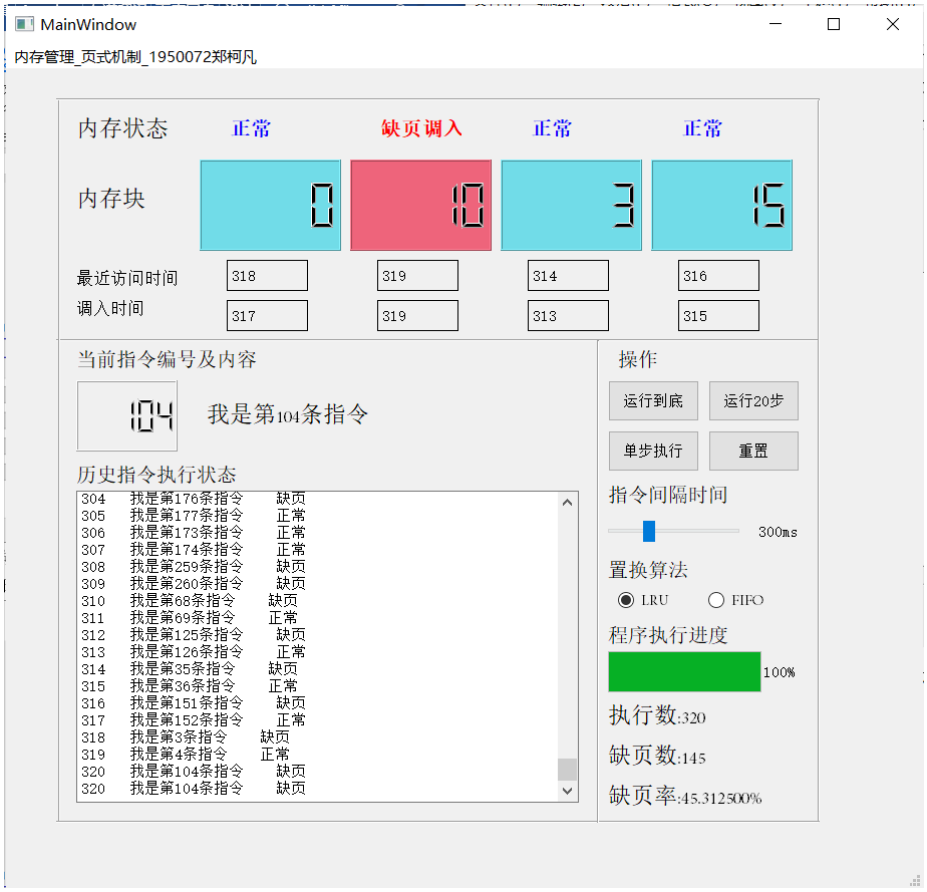
- 内存状态显示区：QLabel+QLCDNumber实现。该部分从左至右分别展示了内存块1、2、3、4的存储页号信息、内存状态信息、页调入信息以及页最近访问信息，供用户实时观察内存状态的信息。
- 指令展示区：QLabel+QLCDNumber+QTextEdit实现。该部分主要显示了当前的指令编号、指令内容以及历史指令信息，使用户对指令的执行状态有直观的了解。
- 用户操作区：QLabel+QPushButton+QSlider+QRadioButton+QProgressBar实现。该部分主要提供给用户选择置换算法、调整指令间隔时间、选择指令运行模式以及观察程序缺页情况的功能。使用户能对不同置换算法的使用情况有一个直观的比较和认识。

5.2 操作说明

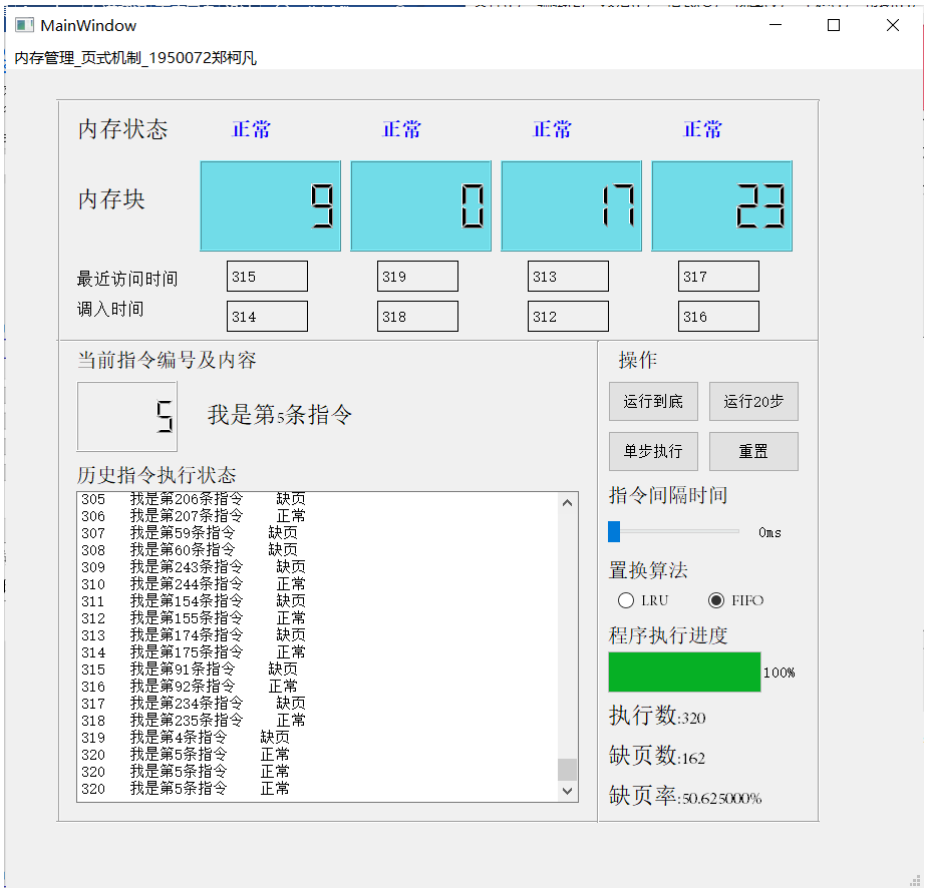
- 以管理员权限运行virtual_paging_mechanism.exe，进入请求调页模拟系统。



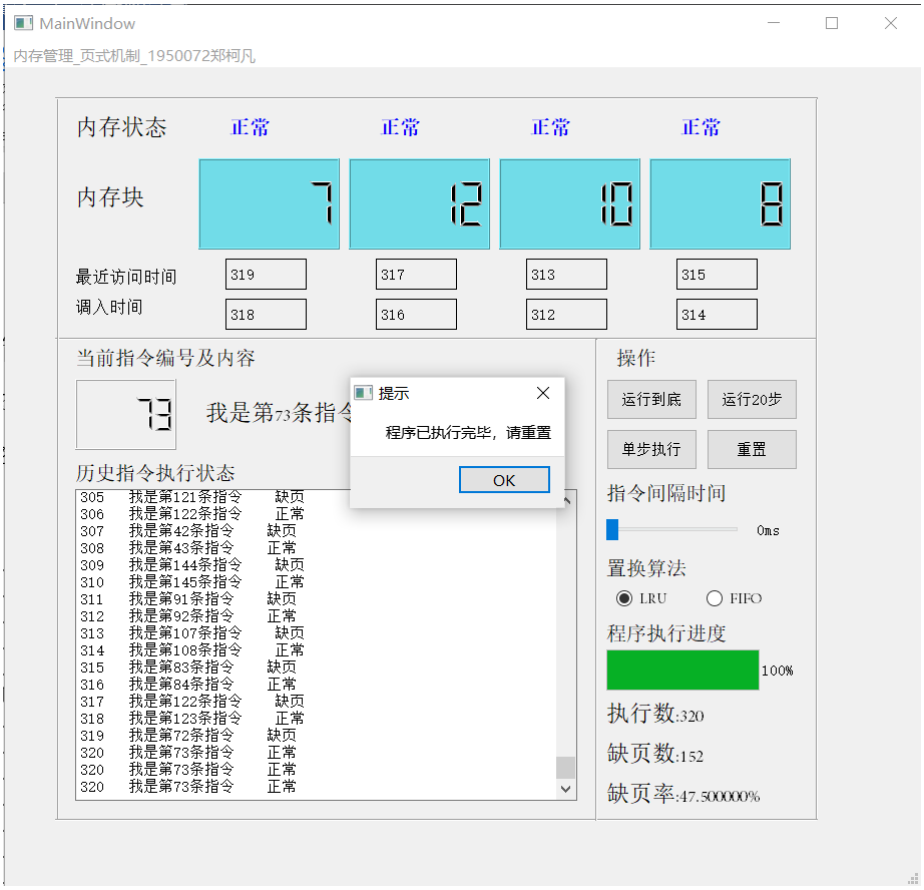
- 选择LRU算法，调整指令间隔时间（根据情况可在0-1000ms内选择）并点击运行到底按钮（也可以选择单步执行或运行20步）观察指令执行状态。在320条指令全部结束后，记录缺页数及缺页率。



- 操作同LRU算法。先选择FIFO算法，并自定义指令间隔时间和运行模式观察指令执行状态。在320条指令全部结束后，记录缺页数及缺页率。



- 比较两种算法的缺页率。虽然由于程序指令执行顺序的原因，两种算法缺页率都比较高，但LRU算法45%左右的缺页率相比FIFO算法50%左右的缺页率还是有5%的提升的，也符合LRU算法性能更优的结论。在程序结束后，也可选择重置重复进行实验。



6. 不足与改进

1. 指令的执行顺序还不能很好地模拟实际中指令执行的状态，不能很好地体现出程序的局部性原理，导致LRU算法相比FIFO算法的优势提升并不明显。后期考虑对指令执行顺序做进一步的改进。
2. 本项目中只使用了LRU和FIFO两种页置换算法，后期考虑加入检测未来指令的功能，便于引进OPT算法（最佳置换算法），保证最低的缺页率。