项目说明文档

操作系统

——内存管理之请求调页

作 者 姓 名： 杨煜

学 号： 1850217

指 导 教 师： 张惠娟

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

目 录

[1 项目背景 1](#_Toc40428823)

[1.1 项目简介 1](#_Toc40428824)

[1.2 项目目的 1](#_Toc40428825)

[1.3 项目要求 1](#_Toc40428826)

[2 需求分析 2](#_Toc40428827)

[2.1 上下行请求 2](#_Toc40428828)

[2.2 数字按钮请求 2](#_Toc40428829)

[2.3 开关门请求 2](#_Toc40428830)

[2.4 报警请求 2](#_Toc40428831)

[3 算法分析 2](#_Toc40428832)

[3.1 单部电梯 2](#_Toc40428833)

[3.1.1 FCFS算法 2](#_Toc40428834)

[3.1.2 SSTF算法 3](#_Toc40428835)

[3.1.3 SCAN算法 3](#_Toc40428836)

[3.2 多部电梯 3](#_Toc40428837)

[4 设计实现 4](#_Toc40428838)

[4.1 电梯类 4](#_Toc40428839)

[4.2 调度类 5](#_Toc40428840)

[4.2.1 初始化数据 5](#_Toc40428841)

[4.2.2 上行请求 6](#_Toc40428842)

[4.2.3 下行请求 8](#_Toc40428843)

[4.2.4 数字按钮请求 9](#_Toc40428844)

[4.2.5 上行后尚未处理的请求 12](#_Toc40428845)

[4.2.6 下行后尚未处理的请求 13](#_Toc40428846)

[4.2.7 开关门请求 14](#_Toc40428847)

[4.2.8 报警请求 15](#_Toc40428848)

[4.3 界面类 16](#_Toc40428849)

[5 附录 17](#_Toc40428850)

[5.1 文件说明 17](#_Toc40428851)

[5.2 开发环境 18](#_Toc40428852)

# 1 项目背景

## 1.1 项目简介

请求调页建立在基本分页系统基础之上，为了支持虚拟存储器功能而增加了请求调页功能和页面置换功能。请求调页是目前最常用的一种实现虚拟存储器的方法。在请求调页系统中，只要求将当前需要的一部分页面装入内存。在作业执行过程中，当索要访问的页面不在内存中时，再通过调页功能将其调入，同时还可通过置换功能将暂时不用的页面换出到外村上，以便腾出内存空间。在页面置换的有许多置换算法，如FIFO（先进先出），LRU（最近最久未使用）算法等。

## 1.2 项目目的

1、熟悉页面、页表、地址转换的内容和过程

2、加深页面置换过程的理解

3、加深对请求调页系统的原理和实现过程的理解。

## 1.3 项目要求

基本任务：

假设每个页面可存放10条指令，分配给一个作业的内存块为4。模拟一个作业的执行过程，该作业有320条指令，即它的地址空间为32页，目前所有页还没有调入内存。

模拟过程：

1. 在模拟过程中，如果所访问指令在内存中，则显示其物理地址，并转到下一条指令；如果没有在内存中，则发生缺页，此时需要记录缺页次数，并将其调入内存。如果4个内存块中已装入作业，则需进行页面置换。
2. 所有320条指令执行完成后，计算并显示作业执行过程中发生的缺页率。
3. 置换算法可以选用FIFO或者LRU算法
4. 作业中指令访问次序可以按照下面原则形成：50%的指令是顺序执行的，25%是均匀分布在前地址部分，25％是均匀分布在后地址部分。

# 2 需求分析

## 2.1 指令分布情况

50%的指令是顺序执行的，25%是均匀分布在前地址部分，25％是均匀分布在后地址部分。每一条指令都有其所在的逻辑空间所划分的页面。一共有320条指令，逻辑空间有32页。

## 2.2 产生指令序列

通过算法产生指令序列，并计算出指令序列所在的逻辑页。将其展示在界面上。并作为内存管理算法所使用的序列。

## 2.3 描述页面命中情况

通过对于每一页的指令调用两种算法，产生的结果会存入列表中，展示在界面上。表明调用某算法时，需要调入的页面是否命中或缺页。

## 2.4 比较两种算法

比较两种算法对于同一指令序列内存管理过程中的缺页情况的比较。在屏幕上展示当前读取的指令数，缺页的指令数，以及当前缺页情况的百分比。以此来比较两种算法在内存管理中的效果。

# 3 算法分析

## 3.1 指令产生算法

根据指令序列的需求和**局部性原理**产生指令序列。首先在0－319条指令之间，随机选取一个起始执行指令，如序号为m顺序执行下一条指令，即序号为m+1的指令。通过随机数，跳转到前地址部分m-60—m-1中的某个指令处，其序号为m1，顺序执行下一条指令，即序号为m1+1的指令。随后通过随机数，跳转到后地址部分m1+2—m1+60中的某条指令处，其序号为m2。再顺序执行下一条指令，即m2+1处的指令。重复跳转到前地址部分、顺序执行、跳转到后地址部分、顺序执行的过程，直到执行完320条指令

## 3.2 FIFO算法

该算法优先淘汰最早进入内存的页面，即在内存中驻留时间最久的页面。该算法实现简单，只需把调入内存的页面根据先后次序链接成队列，设置一个指针总指向最早的页面。但该算法与进程实际运行时的规律不适应，因为在进程中，有的页面经常被访问。该算法还会产生所分配的物理块数增大而页故障数不减反增的异常现象的Belady现象。

## 3.3 LRU算法

选择最近最长时间未访问过的页面予以淘汰，它认为过去一段时间内未访问过的页面，在最近的将来可能页不会被访问。该算法为每个页面设置一个访问字段，来记录页面自上次被访问以来所经历的时间，淘汰页面时选择现有页面中值最大的予以淘汰。LRU算法性能较号，理论上可以证明，不可能出现Belady现象。

# 4 设计实现

## 4.1 内存管理类

内存管理类对于一系列在两种算法中所需要的变量进行初始化。建立一个二维列表用于保存算法运行过程中的物理块所存储的内容。建立一个列表用于模拟FIFO算法中使用到的队列。建立一个列表用于存放LRU算法中增长的时间信号量。使用列表存放两种算法各自产生的总页数和缺页数，和缺页的命中情况。

class MemoryManagementCore:

def \_\_init\_\_(self):

self.blockPage=[[-1 for i in range(4)] for i in range(2)]#用于存放当前在物理内存中的页号

self.blockFIFOQueue=[-1]\*320#FIFO算法所需要使用的队列

self.queueHead=0#队列头部

self.queueCurrent=0#队列尾部

self.blockLRUSignal=[0]\*4#LRU算法所需的列表

self.pageNumber=[0]\*2#存放总的页数

self.pageFaultNumber=[0]\*2#存放缺页数

self.pageStaus1=[]#存放缺页命中情况

self.pageStaus2=[]

## 4.2 FIFO算法

本项目中实现的FIFO算法接受一个新页的变量进入函数。首先遍历四个内存块，如果他们的内部存放着的逻辑页号为-1，则将其替换为新的页号，并将内存块的使用队列添加该块号进去。随后缺页增加，块数增加，缺页情况加入列表中，随后函数直接返回。如果遍历过程中发现物理块内有相同的页号，则总数增加，命中情况加入列表中，随后函数直接返回。遍历结束函数没有返回则发生替换情况，在队列中取出队首，表明需替换该块，将其替换为新的页号，并将内存块的使用队列添加该块号进去。随后缺页增加，块数增加，缺页情况加入列表中。

def algorithmFIFO(self,newPage):#FIFO算法

for i in range(4):

if self.blockPage[0][i]==-1:#首先判断是否为放入任何块

self.blockPage[0][i]=newPage#放入新页

self.blockFIFOQueue[self.queueCurrent]=i#调整队列

self.queueCurrent+=1

self.pageFaultNumber[0]+=1

self.pageNumber[0]+=1

self.pageStaus1.append('Block'+str(i+1)+' Miss')

return

elif self.blockPage[0][i]==newPage:#命中情况

self.pageNumber[0]+=1

self.pageStaus1.append('Block'+str(i+1)+' Hit')

return

replaceBlock= self.blockFIFOQueue[self.queueHead]#没有命中，取出队列头部

self.queueHead+=1

self.blockPage[0][replaceBlock]=newPage#放入新页

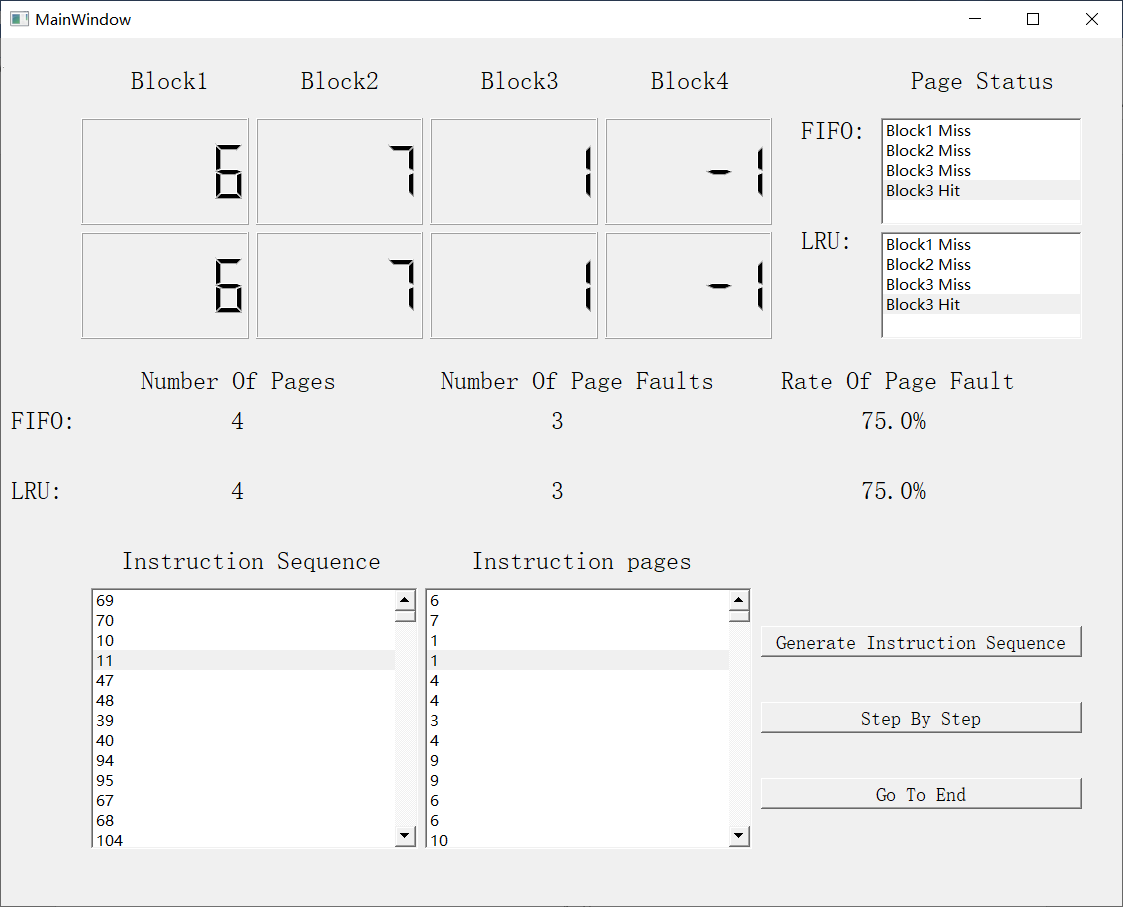
self.blockFIFOQueue[self.queueCurrent]=replaceBlock#调整队列

self.queueCurrent+=1

self.pageFaultNumber[0]+=1

self.pageNumber[0]+=1

self.pageStaus1.append('Block'+str(replaceBlock+1)+' Miss')



## 4.3 LRU算法

本项目中实现的LRU算法接受一个新页的变量进入函数。首先遍历四个内存块，如果他们的内部存放着的逻辑页号为-1，则将其替换为新的页号，除去这块以外的其他内存块的信号数加一。随后缺页增加，块数增加，缺页情况加入列表中，随后函数直接返回。如果遍历过程中发现物理块内有相同的页号，则总数增加，命中情况加入列表中，随后函数直接返回。遍历结束函数没有返回则发生替换情况，在信号列表中找出信号数最大的，表明需替换该块，将其替换为新的页号，除去这块以外的其他内存块的信号数加一。随后缺页增加，块数增加，缺页情况加入列表中。

def algorithmLRU(self,newPage):#LRU算法

for i in range(4):

if self.blockPage[1][i]==-1:#首先判断是否为放入任何块

self.blockPage[1][i]=newPage#放入新页

for j in range(4):#对于没有访问的块+1，表示时间+1

self.blockLRUSignal[j]+=1

self.blockLRUSignal[i]=0

self.pageNumber[1]+=1

self.pageFaultNumber[1]+=1

self.pageStaus2.append('Block'+str(i+1)+' Miss')

return

elif self.blockPage[1][i]==newPage:#命中情况

for j in range(4):

self.blockLRUSignal[j]+=1

self.blockLRUSignal[i]=0

self.pageNumber[1]+=1

self.pageStaus2.append('Block'+str(i+1)+' Hit')

return

replaceBlock= self.blockLRUSignal.index(max(self.blockLRUSignal))#没有命中，取出时间状态最大的

self.blockPage[1][replaceBlock]=newPage#放入新页

for j in range(4):#对于没有访问的块+1，表示时间+1

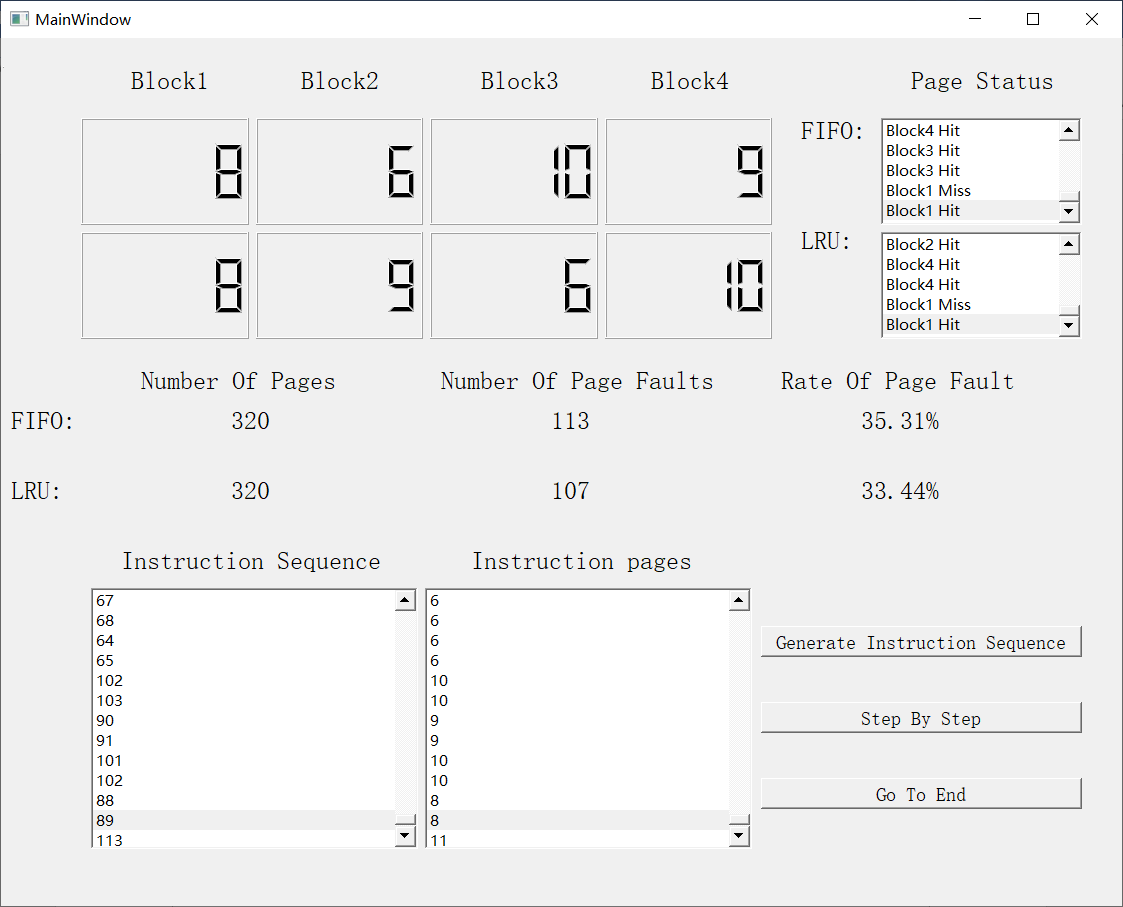
self.blockLRUSignal[j]+=1

self.blockLRUSignal[replaceBlock]=0

self.pageFaultNumber[1]+=1

self.pageNumber[1]+=1

self.pageStaus2.append('Block'+str(replaceBlock+1)+' Miss')



## 4.4 产生指令序列

首先在1-318之间产生一个随机数代表是第几条指令。将其加入指令列表，随后对其进行除以10并向下取整代表是该指令在第几块中，将其加入页面列表。随后指令数加一加入指令列表，除以10并向下取整代表是该指令在第几块中，将其加入页面列表。随后根据局部性原理，其他的指令只会产生在该条指令的附近。在前址部分中取离开该指令距离为六十以内的指令产生随机数并按照上述方式将该指令和随后的指令以及他们所在的页面加入列表中。随后在后址部分寻找同样是距离六十以内的指令，重复上述操作，直到列表中含有320条指令。

def generateInstructions():#用于产生队列，50%顺序，25%前址，25%后址

m = random.randint(1, 318)

i=1

instructions = []

pages = []

instructions.append(m)#找到第一个随机数

pages.append(m//10)

instructions.append(m + 1)

pages.append((m + 1)//10)

while len(instructions) <= 318:#产生320条

if m-60>=0:#产生的范围在前后60条指令内以此模拟程序局部性原理

m1 = random.randint(m-60, m - 1)

else:

m1 = random.randint(0, m - 1)

instructions.append(m1)

instructions.append(m1 + 1)

pages.append(m1//10)

pages.append((m1 + 1) // 10)

if m1+60<=318:#产生的范围在前后60条指令内以此模拟程序局部性原理

m2 = random.randint(m1 + 2, m1+60)

else:

m2 = random.randint(m1 + 2, 318)

instructions.append(m2)

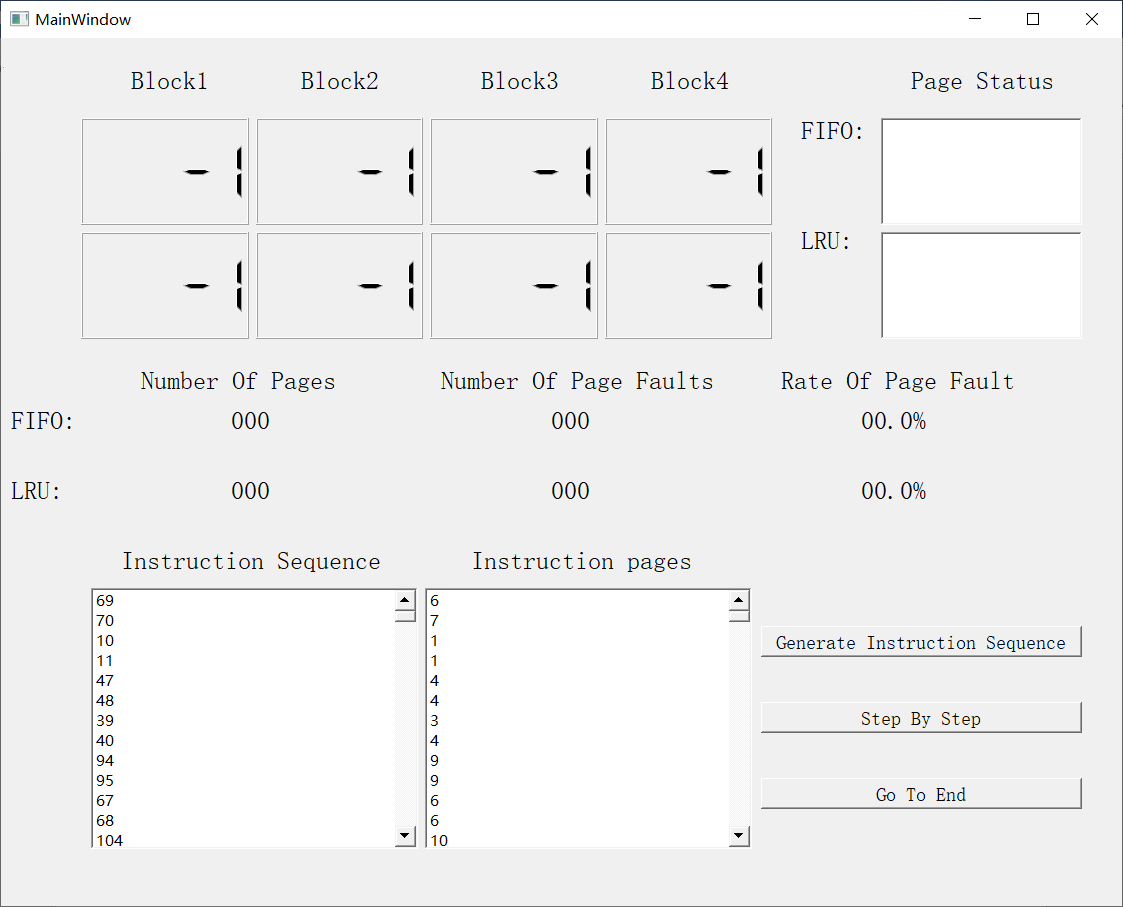
instructions.append(m2 + 1)

pages.append(m2 // 10)

pages.append((m2 + 1) // 10)

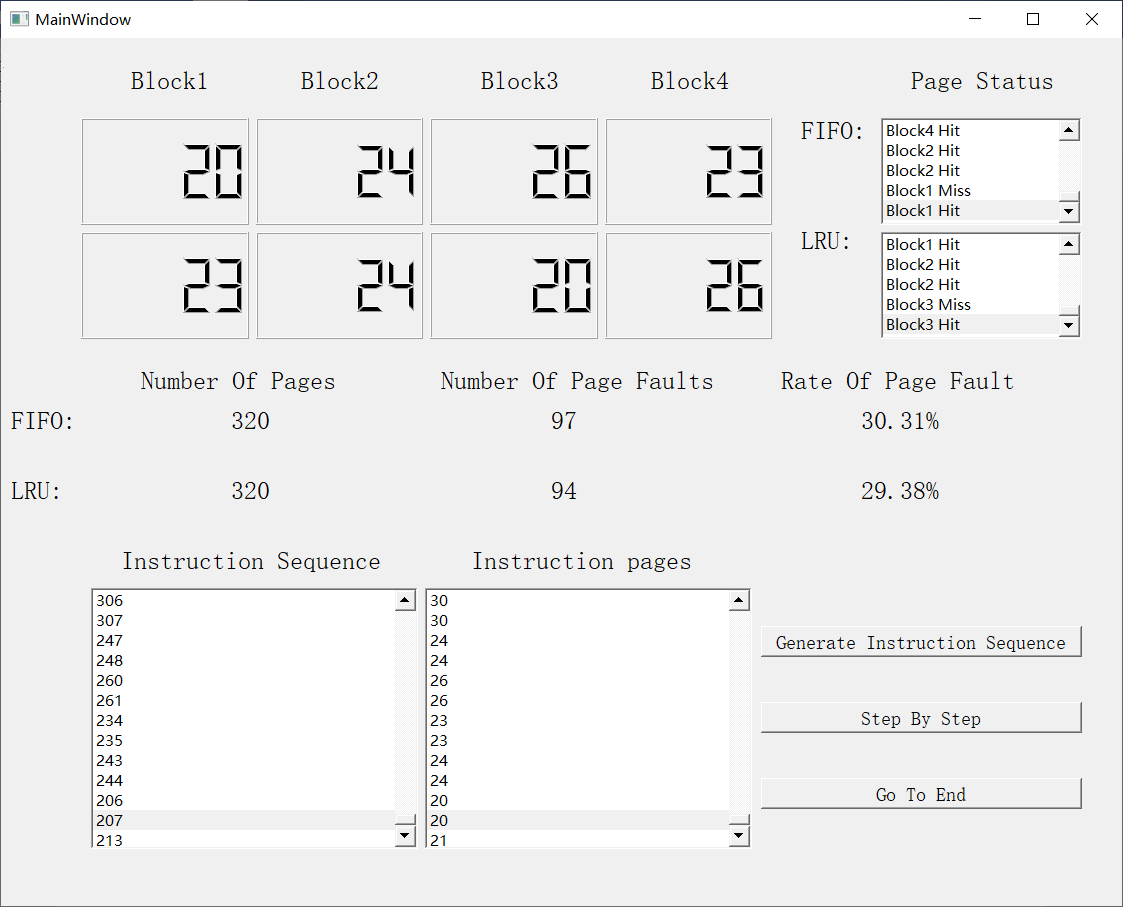
m = m2

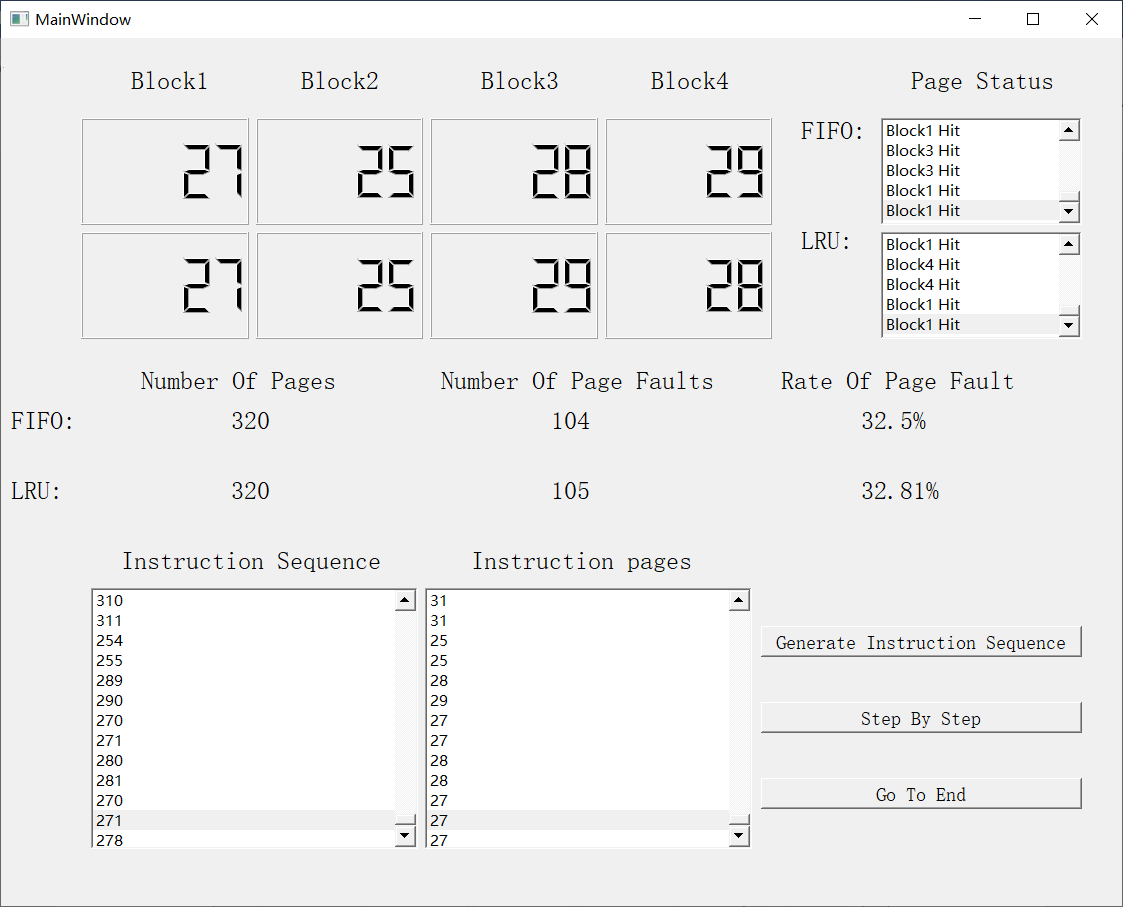
return instructions,pages

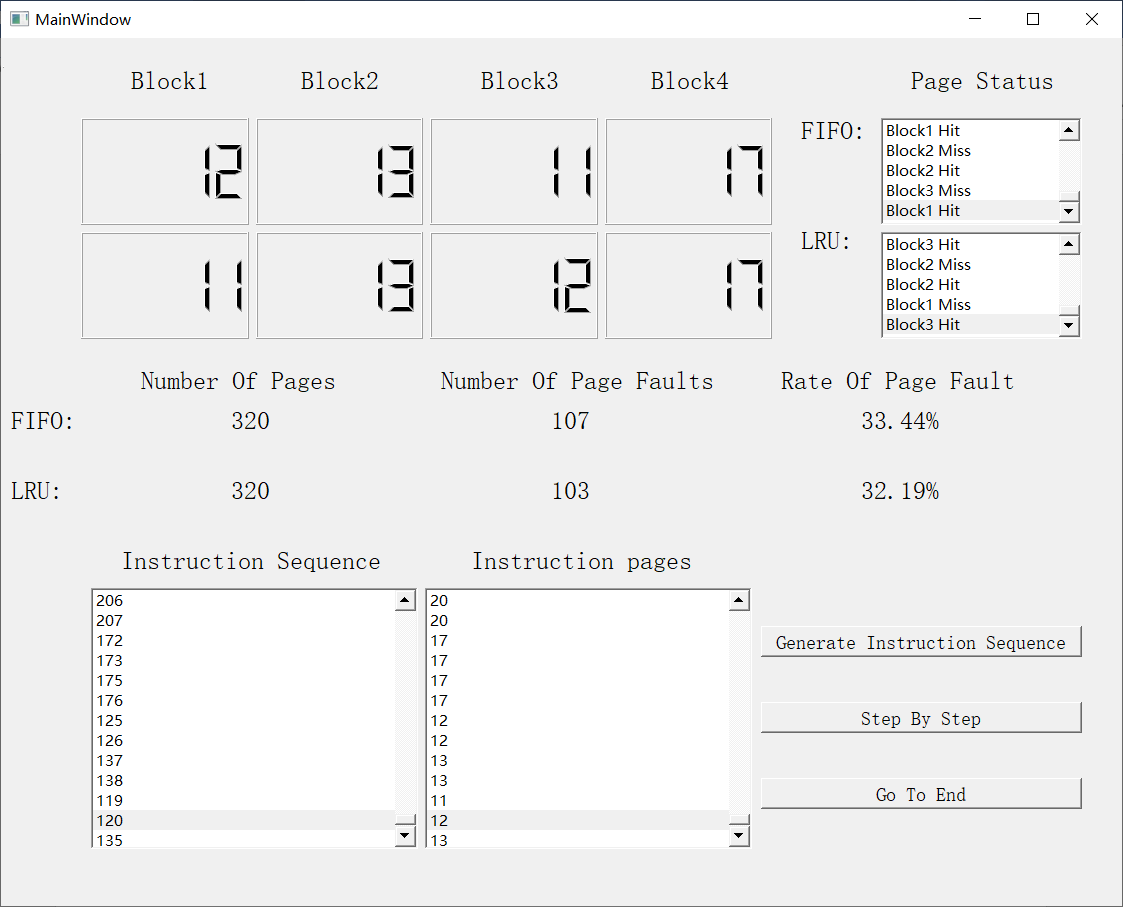


# 5 算法性能比较

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 缺页率1 | 缺页率2 | 缺页率3 | 平均缺页率 |
| FIFO | 30.31% | 32.50% | 33.44% | 32.08% |
| LRU | 29.38% | 32.81% | 32.19% | 31.46% |







# 6 附录

## 6.1 文件说明

MemoryManagement.py为调度算法的核心代码

MemoryManagementUI.py为调度算法的界面代码

MemoryManagementUI.exe为生成的可执行文件

platforms文件夹内部存放了运行时所需要的依赖

由于python环境问题，exe文件运行并不稳定（使用pyinstaller生成），所以使用时建议执行MemoryManagementUI.py，所需环境为：PyQt5

## 6.2 开发环境

操作系统：windows10 1909 64bit

语言：python 3.7

相关python包：PyQt5, PyQt-tools, PyUIC