操作系统项目说明文档

——内存管理项目之请求调页存储管理方式模拟

作 者 姓 名： 马威

学 号： 2151294

指 导 教 师： 王冬青

学院、 专业： 软件学院 软件工程

****

**目 录**

[1.项目分析 1](#_Toc13101)

[1.1 项目背景 1](#_Toc21362)

[1.2 项目要求 1](#_Toc21739)

[1.3 项目需求分析 1](#_Toc14167)

[2.项目设计 2](#_Toc29303)

[2.1 开发环境选择 2](#_Toc5400)

[2.2 界面设计 2](#_Toc32675)

[2.3 置换算法设计 3](#_Toc7531)

[2.3.1 FIFO 3](#_Toc28106)

[2.3.1.1 算法思路 3](#_Toc31155)

[2.3.1.2 算法代码 3](#_Toc31155)

[2.3.2 LRU 4](#_Toc18684)

[2.3.2.1 算法思路 4](#_Toc31155)

[2.3.2.2 算法代码 4](#_Toc31155)

[2.3.3 LFU 5](#_Toc6362)

[2.3.3.1 算法思路 5](#_Toc31155)

[2.3.3.2 算法代码 5](#_Toc31155)

[2.3.4 OPT 6](#_Toc16606)

[2.3.4.1 算法思路 6](#_Toc31155)

[2.3.4.2 算法代码 6](#_Toc31155)

[2.3.5 生成指令序列 7](#_Toc3889)

[2.3.5.1 算法思路 7](#_Toc31155)

[2.3.5.2 算法代码 7](#_Toc31155)

[2.3.6 线程同步 8](#_Toc6765)

[2.3.6.1 算法思路 8](#_Toc31155)

[2.3.6.2 算法代码 9](#_Toc31155)

[2.3.6.3 事件的处理 1](#_Toc31155)4

[3.项目测试 1](#_Toc29303)5

[4.项目评价 1](#_Toc29303)6

[4.1 可改进的地方 16](#_Toc5591)

[4.2 解决了的特殊问题 16](#_Toc26412)

**1.项目分析**

1.1 项目背景

计算机中每个作业都伴随着指令序列的处理，而指令都会储存在若干页中，在运行时调入内存，以便更快取得指令。最好的情况显然是所有的页都在内存中，这样便不会发生缺页中断。然而，为节省内存空间，有可能出现内存分配给这个作业的块无法装下所有页的情况。因此，如果访问的指令不在内存中，则会发生缺页中断并发生调块，这时如何选择被替代的块至关重要，若选择算法能使得缺页率低，则可以提高计算机的运行效率。

1.2 项目要求

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。

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。

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。

置换算法可以选用FIFO或者LRU算法。

1.3 项目需求分析

对于请求调页存储管理方式模拟系统，需考虑以下需求：

**·正确性**

程序应当能够正确模拟出现实的指令执行顺序，同时在发生缺页时执行正确的算法替代块，尽可能正确的模拟出内存管理的情况。

**·健壮性**

程序应当对各种指令序列和置换算法进行合适的处理。

**·交互性**

程序要以系统的形式出现，涉及到按按键等功能，需要给用户提供足够的提示与说明，使用方式要友好。同时由于有实时数据，需要在完成线程同步基础上显示实时数据，并在执行完成后显示缺页率等信息。

**2.项目设计**

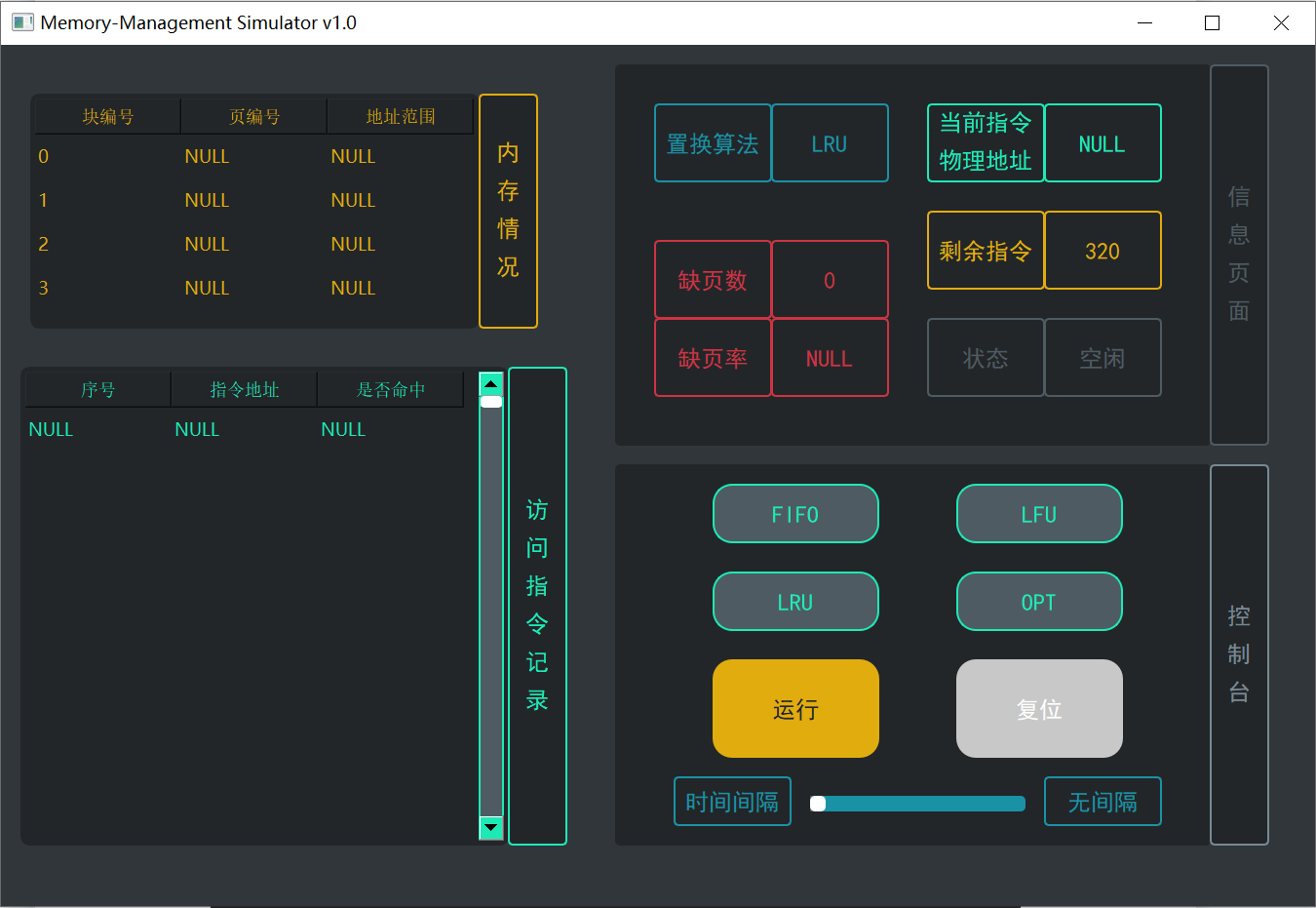
2.1 开发环境选择

本项目的开发环境：

操作系统：Windows10 编程语言：python 3.9 界面设计：PyQt5

2.2 界面设计

本项目分为四个部分。第一个是内存情况表格，它可以实时显示当前内存中各块的页号和存储指令的地址范围（逻辑）；第二个是访问指令记录表格，它可以实时显示每条指令的序号、逻辑地址和是否命中；第三个是信息页面，它可以实时显示替换算法、缺页信息、当前指令物理地址、剩余指令数和程序状态；第四个是控制台，可以在此选择置换算法、运行或复位、调整时间间隔。



2.3 置换算法设计

**2.3.1 FIFO**

**2.3.1.1 算法思路**

处理指令序列的过程中，维护一个队列queue。设新页为new\_page，内存数组为memory。若内存中有空闲位置则new\_page直接入队，若内存中无空闲位置则队头元素出队，对应的memory中的位置被new\_page取代，new\_page入队。

**2.3.1.2 算法代码**

# 置换算法：FIFO  
**def** FIFO(*page\_num*):  
 *index* = *queue*.qsize() # 要替换的块的下标，初始为队列的长度，在内存仍有空闲块时可以取到第一个空闲位置  
  
 **if** *queue*.qsize() == 4: # 若无空闲块，则队头元素出队，并找到其下标  
 *index* = *memory*.index(*queue*.get())  
  
 *queue*.put(*page\_num*) # 新块入队  
 *memory*[*index*] = *page\_num* # 新块替换内存中原来的块

**2.3.2 LRU**

**2.3.2.1 算法思路**

处理指令序列的过程中，维护一个特殊栈stack。设新页为new\_page，内存数组为memory。若内存中有空闲位置则new\_page直接放在栈顶，若内存中无空闲位置则new\_page取代栈底元素，对应的memory中的位置被new\_page取代。每访问一条指令，都将其对应的页抽出放到栈顶。

**2.3.2.2 算法代码**

# 置换算法：LRU  
**def** LRU(*page\_num*):  
 **if len**(*stack*) < 4: # 若仍有空闲块，直接向栈顶加新块，新块调入第一个空闲位置  
 *stack*.append(*page\_num*)  
 *memory*[**len**(*stack*) - 1] = *page\_num* **else**: # 若无空闲块，则找到栈底元素对应的块，既修改栈底元素，也修改内存中对应的块  
 *index* = *memory*.index(*stack*[0])  
 *stack*[0] = *page\_num  
 memory*[*index*] = *page\_num*

**if** *algorithm* == "LRU": # 选择算法为LRU时，维护特殊栈（将访问到的页抽出，放至栈顶）  
 *stack*.remove(*page\_num*)  
 *stack*.append(*page\_num*)

**2.3.3 LFU**

**2.3.3.1 算法思路**

处理指令序列的过程中，维护一个数组page\_visit，记录所有页的访问次数。设新页为new\_page，内存数组为memory。若内存中有空闲位置则new\_page直接放入空闲位置，若内存中无空闲位置则new\_page取代memory中在page\_visit里对应值最小元素。每访问一条指令，都将其对应的页在page\_visit中的值+1。

**2.3.3.2 算法代码**

# 置换算法：LFU  
**def** LFU(*page\_num*):  
 **if** *memory*.**\_\_contains\_\_**(-1): # 若仍有空闲块，直接向栈顶加新块，新块调入第一个空闲位置  
 *memory*[*memory*.index(-1)] = *page\_num* **else**: # 若无空闲块  
 *visit\_count* = [] # 四个块的被访问次数  
  
 **for** *i* **in range**(4):  
 *visit\_count*.append(*page\_visit*[*memory*[*i*]]) # 从page\_visit总数组中找到四个块的被访问次数  
  
 *max\_index* = *visit\_count*.index(**max**(*visit\_count*)) # 选出次数最多的块进行取代  
 *memory*[*max\_index*] = *page\_num*

**elif** *algorithm* == "LFU": # 选择算法为LFU时，维护页被访问次数数组（对应页+1）  
 *page\_visit*[*page\_num*] += 1

**2.3.4 OPT**

**2.3.4.1 算法思路**

处理指令序列的过程中，先预生成两个数组ins\_order、page\_order，记录指令序列和页号序列。设新页为new\_page，内存数组为memory。若内存中有空闲位置则new\_page直接放入空闲位置，若内存中无空闲位置则根据page\_order统计四个块下一次出现的位置，不再出现设为无穷大，取下一次出现位置离当前位置最远的块被new\_page取代。

实际上，程序运行过程中无法预先得知确定的指令序列及访问的页号序列，因此OPT无法实际应用。本系统通过预先生成序列进行了模拟，旨在对比不同的置换算法的缺页情况。

**2.3.4.2 算法代码**

# 1.若选择算法为OPT，则先取得指令序列和页号序列  
**if** *algorithm* == "OPT":  
 **for** *i* **in range**(320):  
 *address* = rand\_ins(*address*) # 将前一地址输入，获得当前地址  
 *visited\_ins*.add(*address*) # 将当前地址加入集合，防止获取重复地址  
 *ins\_order*.append(*address*) # 记录当前地址  
 *page\_order*.append(**int**((*address* - *address* % 10) / 10)) # 记录当前页号  
  
 *visited\_ins*.clear() # 预获取地址结束，集合清空

# 置换算法：OPT  
**def** OPT(*current*, *page\_num*):  
 **if** *memory*.\_\_contains\_\_(-1): # 若仍有空闲块，直接向栈顶加新块，新块调入第一个空闲位置  
 *memory*[*memory*.index(-1)] = *page\_num* **else**: # 若无空闲块  
 *indexes* = [] # 四个块的下一次访问的指令序号  
  
 **for** *i* **in range**(4): # 从当前指令往后，统计四个块的第一次出现序号  
 **if** *page\_order*[*current*:320].\_\_contains\_\_(*memory*[*i*]): # 若能找到则记录序号  
 *index* = *page\_order*.index(*memory*[*i*], *current*, 320)  
 **else**: # 若不能找到则记为无限大  
 *index* = 2147483647  
  
 *indexes*.append(*index*)  
  
 *max\_index* = *indexes*.index(**max**(*indexes*)) # 取序号最大的块在memory中的下标，并进行取代  
 *memory*[*max\_index*] = *page\_num*

**2.3.5 生成指令序列**

**2.3.5.1 算法思路**

对于上一条指令地址last\_address，当前指令地址有70%可能顺序执行，15%可能向前跳转，15%可能向后跳转。因此，执行下列操作：

①先生成[0,1)的随机小数x

②0≤x＜0.7，且last\_address不是最后一条指令，则返回last\_address + 1

③0.7≤x＜0.85，且last\_address不是前两条指令，则返回[0,last\_address)的随机一个地址

④0.85≤x＜0.1，且last\_address不是倒数两条指令，则返回[last\_address + 1,319)的随机一个地址

⑤若①-④得出的返回值对应的指令已被访问，则将其作为新的last\_address重复①-④的操作，直至返回值对应的指令未被访问。

**2.3.5.2 算法代码**

# 产生当前指令的地址  
**def** rand\_ins(*last\_address*):  
 *result* = *last\_address* # 最终结果，初始化为上一条指令的地址  
  
 **while True**:  
 *rand* = random.random() # 随机产生小数  
  
 **if** 0 <= *rand* < 0.7 **and** *result* < 319: # 在上一条指令不是最后一条指令的前提下，70%的概率顺序执行  
 *result* += 1  
 **elif** 0.7 <= *rand* < 0.85 **and** *result* > 1: # 在上一条指令不是前两条指令的前提下，15%的概率向前跳转  
 *result* = random.randint(0, *result* - 1)  
 **elif** 0.85 <= *rand* < 1 **and** *result* < 318: # 在上一条指令不是倒数两条指令的前提下，15%的概率向后跳转  
 *result* = random.randint(*result* + 1, 319)  
  
 **if** *result* **not in** *visited\_ins*: # 若得出的指令确实没有被访问过，则得出结果；否则重新生成  
 **break  
  
 return** *result*

**2.3.6 线程同步**

**2.3.6.1 算法思路**

为了配合PyQt界面实现多线程操作，本项目使用QThread类进行线程设计。主要有两类：一类专门处理数据，进行指令序列处理和数据结构维护；一类专门负责实时更新界面。

通过下列信号变量进行通信，变为True时相关负责函数开始运行，运行结束后由负责函数调回False：

miss = False # 触发缺页时内存情况表格更新的标志

update = False # 触发信息更新的标志

execute = False # 触发处理指令序列的标志

reset\_ui = False # 触发复位界面的标志

reset\_data = False # 触发复位数据的标志

通过下列互斥锁进行同步控制，保证在一段时间内只有一个线程对共享信息进行修改：

lmiss = QtCore.QMutex() # 控制缺页时内存情况表格的更新（将其从lupdate中剥离，是因为只有发生缺页时才需要更新表格）

lupdate = QtCore.QMutex() # 控制信息更新

lexecute = QtCore.QMutex() # 控制指令序列处理

lreset = QtCore.QMutex() # 控制复位

**2.3.6.2 算法代码**

# 专门处理界面更新的线程，在run里没有耗时操作  
**class** UiThread(QThread):  
 signal\_memory = pyqtSignal(**int**) # 更新内存情况表格的信号对象  
 signal\_record = pyqtSignal(**int**, **int**, **bool**) # 更新记录表格的信号对象  
 signal\_miss = pyqtSignal() # 更新缺页情况标签的信号对象  
 signal\_reset = pyqtSignal() # 使表格复位的信号对象  
 ins\_num = -1 # 更新记录表格用的临时变量：指令序号  
 ins\_address = -1 # 更新记录表格用的临时变量：指令地址  
 ins\_hit = **False** # 更新记录表格用的临时变量：是否命中  
 new\_page = -1 # 更新内存情况表格用的临时变量：新页编号  
 physical\_address = -1 # 更新当前指令标签用的临时变量：物理地址  
 current\_interval = 0.05 # 更新时间间隔用的临时变量：当前时间间隔  
 run\_button\_enabled = **True** # 切换按键状态用的临时变量：运行键是否有效  
 reset\_button\_enabled = **False** # 切换按键状态用的临时变量：复位键是否有效  
  
 **def \_\_init\_\_**(**self**):  
 **super**(UiThread, **self**).**\_\_init\_\_**()  
  
 # 各信号与对应的槽函数连接  
 **self**.signal\_memory.connect(update\_memory)  
 **self**.signal\_record.connect(update\_record)  
 **self**.signal\_miss.connect(update\_miss\_info)  
 **self**.signal\_reset.connect(reset\_memory)  
 **self**.signal\_reset.connect(reset\_record)  
  
 **def** run(**self**):  
 **global** *update*, *lupdate*, *reset\_ui*, *miss\_count*, *reset\_data*, *mainWindow*, *record*, *interval*, *memory*, *miss* **while** 1: # 线程运行过程处在永真循环中  
 *latest\_interval* = *mainWindow*.IntervalEditSlider.value()  
  
 # 一、若时间间隔发生改变  
 **if** *latest\_interval* != **self**.current\_interval:  
 **self**.current\_interval = *latest\_interval* # 更新临时变量  
 *interval* = *latest\_interval* # 更新时间间隔  
 *mainWindow*.IntervalDataLabel.setText(f"{*latest\_interval* / 100}s" **if** *interval* != 0 **else** "无间隔") # 改标签  
  
 # 二、若发生缺页  
 **if** *miss*:  
 *lmiss*.lock() # ------------------------------------- 互斥锁上锁 -------------------------------------  
 **self**.signal\_memory.emit(*memory*.index(**self**.new\_page)) # 更新内存情况表格  
 *miss* = **False** # 更新结束，关闭信号变量  
 *lmiss*.unlock() # ------------------------------------ 互斥锁解锁 ------------------------------------  
  
 # 三、若需要更新信息  
 **if** *update*:  
 *lupdate*.lock() # ------------------------------------ 互斥锁上锁 ------------------------------------  
 **self**.signal\_record.emit(**self**.ins\_num, **self**.ins\_address, **self**.ins\_hit) # 更新表格  
 **self**.signal\_miss.emit() # 更新缺页信息标签  
 *mainWindow*.CurrentCmdDataLabel.setText(f"{**self**.physical\_address}") # 更新当前指令标签  
 *mainWindow*.LeftCmdDataLabel.setText(f"{319 - **self**.ins\_num}") # 更新剩余指令标签  
 QApplication.processEvents() # 处理所有emit()请求，以达到实时更新  
  
 time.sleep(*interval* / 100) # 延时使得画面更为流畅  
 *update* = **False** # 更新结束，关闭信号变量  
 *lupdate*.unlock() # ------------------------------------ 互斥锁解锁 ------------------------------------  
  
 # 四、若需要复位界面  
 **if** *reset\_ui*:  
 *lreset*.lock() # ------------------------------------- 互斥锁上锁 ------------------------------------  
 *miss\_count* = 0  
 **self**.signal\_miss.emit() # 复位缺页信息标签  
 **self**.signal\_reset.emit() # 复位两个表格  
 *mainWindow*.CurrentCmdDataLabel.setText("NULL") # 复位当前指令标签  
 *mainWindow*.LeftCmdDataLabel.setText("320") # 复位剩余指令标签  
 *mainWindow*.StateDataLabel.setText("空闲") # 复位状态标签  
 *mainWindow*.RunButton.setEnabled(**True**) # 复位运行键  
 *mainWindow*.ResetButton.setEnabled(**False**) # 复位复位键  
 **self**.run\_button\_enabled = **True** # 临时变量更新，运行键设为有效  
 **self**.reset\_button\_enabled = **False** # 临时变量更新，复位键设为无效  
 QApplication.processEvents() # 处理所有emit()请求，以达到实时更新  
 *reset\_ui* = **False** # 复位ui结束，关闭信号变量  
 *reset\_data* = **True** # 打开信号变量，通知复位数据开始  
 *lreset*.unlock() # ------------------------------------- 互斥锁解锁 ------------------------------------  
  
 # 五、若处在刚开始运行瞬间  
 **if** *execute* **and self**.run\_button\_enabled:  
 *mainWindow*.RunButton.setEnabled(**False**) # 运行键设为无效  
 *mainWindow*.FIFO.setEnabled(**False**) # 算法选择键均设为无效  
 *mainWindow*.LRU.setEnabled(**False**)  
 *mainWindow*.LFU.setEnabled(**False**)  
 *mainWindow*.OPT.setEnabled(**False**)  
 *mainWindow*.StateDataLabel.setText("运行") # 更新状态标签  
 **self**.run\_button\_enabled = **False** # 临时变量更新，运行键设为无效  
  
 # 六、若处在运行完毕的瞬间  
 **elif not** *execute* **and not self**.reset\_button\_enabled **and not** *reset\_ui* **and not** *reset\_data* **and len**(*record*) != 0:  
 *mainWindow*.ResetButton.setEnabled(**True**) # 复位键设为有效  
 *mainWindow*.FIFO.setEnabled(**True**) # 算法选择键均设为有效  
 *mainWindow*.LRU.setEnabled(**True**)  
 *mainWindow*.LFU.setEnabled(**True**)  
 *mainWindow*.OPT.setEnabled(**True**)  
 *mainWindow*.StateDataLabel.setText("停止") # 更新状态标签  
 **self**.reset\_button\_enabled = **True** # 临时变量更新，复位键设为有效  
  
 **def** set\_record\_info(**self**, *num*, *address*, *hit*): # 更新记录表格时，将临时变量也更新，方便传送到槽函数  
 **self**.ins\_num = *num* **self**.ins\_address = *address* **self**.ins\_hit = *hit* **def** set\_physical\_address(**self**, *physical\_address*):  
 **self**.physical\_address = *physical\_address* **def** set\_new\_page(**self**, *new\_page\_num*): # 更新内存情况表格时，将临时变量也更新，方便传送到槽函数  
 **self**.new\_page = *new\_page\_num*# 专门处理指令序列的线程，直接在run里执行耗时的对数据的操作  
**class** DataThread(QThread):  
 **def \_\_init\_\_**(**self**):  
 **super**(DataThread, **self**).**\_\_init\_\_**()  
  
 **def** run(**self**):  
 **global** *execute*, *lexecute*, *update*, *reset\_data*, *miss\_count*, *miss*, *miss\_rate* **while** 1:  
 # 一、若需要开始处理指令序列  
 **if** *execute*:  
 *lexecute*.lock() # ------------------------------------ 互斥锁上锁 -----------------------------------  
 *address* = random.randint(0, 319) # 初始随机地址  
 **print**("#--------------------------------------------------------------------------------------------#\n"  
 "start processing...") # 打印开始处理的标志  
  
 # 1.若选择算法为OPT，则先取得指令序列和页号序列  
 **if** *algorithm* == "OPT":  
 **for** *i* **in range**(320):  
 *address* = rand\_ins(*address*) # 将前一地址输入，获得当前地址  
 *visited\_ins*.add(*address*) # 将当前地址加入集合，防止获取重复地址  
 *ins\_order*.append(*address*) # 记录当前地址  
 *page\_order*.append(**int**((*address* - *address* % 10) / 10)) # 记录当前页号  
  
 *visited\_ins*.clear() # 预获取地址结束，集合清空  
  
 # 2.处理指令  
 **for** *i* **in range**(320):  
 # 若选择算法为OPT，直接从记录中取地址和页号；否则随机生成地址并计算对应页号  
 *address* = rand\_ins(*address*) **if** *algorithm* != "OPT" **else** *ins\_order*[*i*]  
 *page\_num* = **int**((*address* - *address* % 10) / 10) **if** *algorithm* != "OPT" **else** *page\_order*[*i*]  
 *hit* = **False** # 是否命中的标志  
  
 **for** *page* **in** *memory*: # 遍历内存中的4个块，看是否命中  
 **if** *page* == *page\_num*:  
 *hit* = **True  
  
 if not** *hit*: # 若不命中，则进行调块  
 **if** *algorithm* == "FIFO":  
 FIFO(*page\_num*)  
 **elif** *algorithm* == "LRU":  
 LRU(*page\_num*)  
 **elif** *algorithm* == "OPT":  
 OPT(*i*, *page\_num*)  
 **elif** *algorithm* == "LFU":  
 LFU(*page\_num*)  
  
 *miss\_count* += 1 # 缺页数+1  
  
 *physical\_address* = *memory*.index(*page\_num*) \* 10 + *address* % 10 # 计算物理地址  
 visit(*i*, *address*, *physical\_address*, *hit*, *page\_num*) # 访问该指令  
  
 **if not** *hit*:  
 *lmiss*.lock() # ---------------------- 互斥锁上锁 ----------------------  
 *ui\_thread*.set\_new\_page(*page\_num*) # 改变UiThread的临时变量  
 *miss* = **True** # 打开信号变量，通知更新内存情况表格开始  
 *lmiss*.unlock() # --------------------- 互斥锁解锁 ---------------------  
  
 *lupdate*.lock() # ----------------------- 互斥锁上锁 -----------------------  
 *ui\_thread*.set\_record\_info(*i*, *address*, *hit*) # 改变UiThread的临时变量  
 *ui\_thread*.set\_physical\_address(*physical\_address*) # 改变UiThread的临时变量  
 *update* = **True** # 打开信号变量，通知更新界面开始  
 *lupdate*.unlock() # ----------------------- 互斥锁解锁 -----------------------  
  
 **while** *miss* **or** *update*: # 若界面正在更新，则阻塞指令处理的进行，以免处理太快导致界面或数据不同步  
 **pass** # 3.打印本次运行信息：置换算法、缺页数、缺页率  
 **print**("#--------------------------------------------------------------------------------------------#\n"  
 "process success\n"  
 f"algorithm:{*algorithm*}\n"  
 f"miss\_count:{*miss\_count*}\n"  
 f"miss\_rate:{**round**(*miss\_count* / 3.2, 2)}%\n"  
 "press 复位 to continue next round\n"  
 "#--------------------------------------------------------------------------------------------#\n"  
 )  
  
 *execute* = **False** # 指令处理结束，关闭信号变量  
 *lexecute*.unlock() # ------------------------------------ 互斥锁解锁 -----------------------------------  
  
 # 二、若需要复位数据  
 **if** *reset\_data*:  
 *lreset*.lock() # ------------------------------------- 互斥锁上锁 ------------------------------------  
 *record*.clear() # 指令访问记录清空  
  
 **for** *i* **in range**(4): # 内存中块的情况重置  
 *memory*[*i*] = -1  
  
 *visited\_ins*.clear() # 访问过的指令集合清空  
  
 **for** *i* **in range**(*queue*.qsize()): # 清空FIFO使用的队列  
 *queue*.get()  
  
 **for** *i* **in range**(32): # 清空LFU使用的页面访问次数记录数组  
 *page\_visit*[*i*] = 0  
  
 *stack*.clear() # 清空LRU使用的特殊栈  
  
 *ins\_order*.clear() # 清空OPT使用的地址序列  
 *page\_order*.clear() # 清空OPT使用的页号序列  
  
 *reset\_data* = **False** # 数据复位结束，关闭信号变量  
 *lreset*.unlock() # ------------------------------------ 互斥锁解锁 -----------------------------------

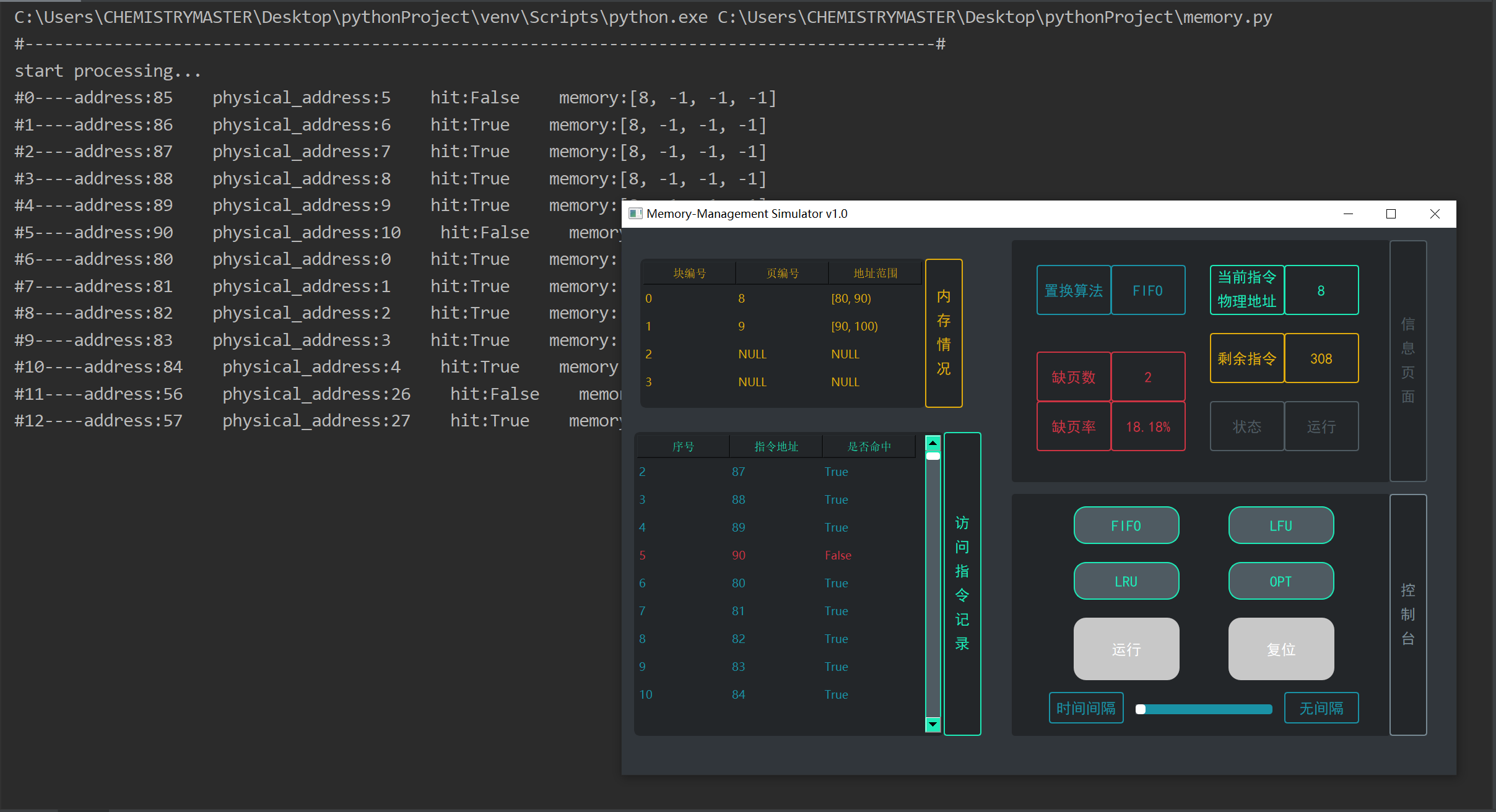
**2.3.6.3 事件的处理**

运行键的点击挂接了trigger\_execute函数

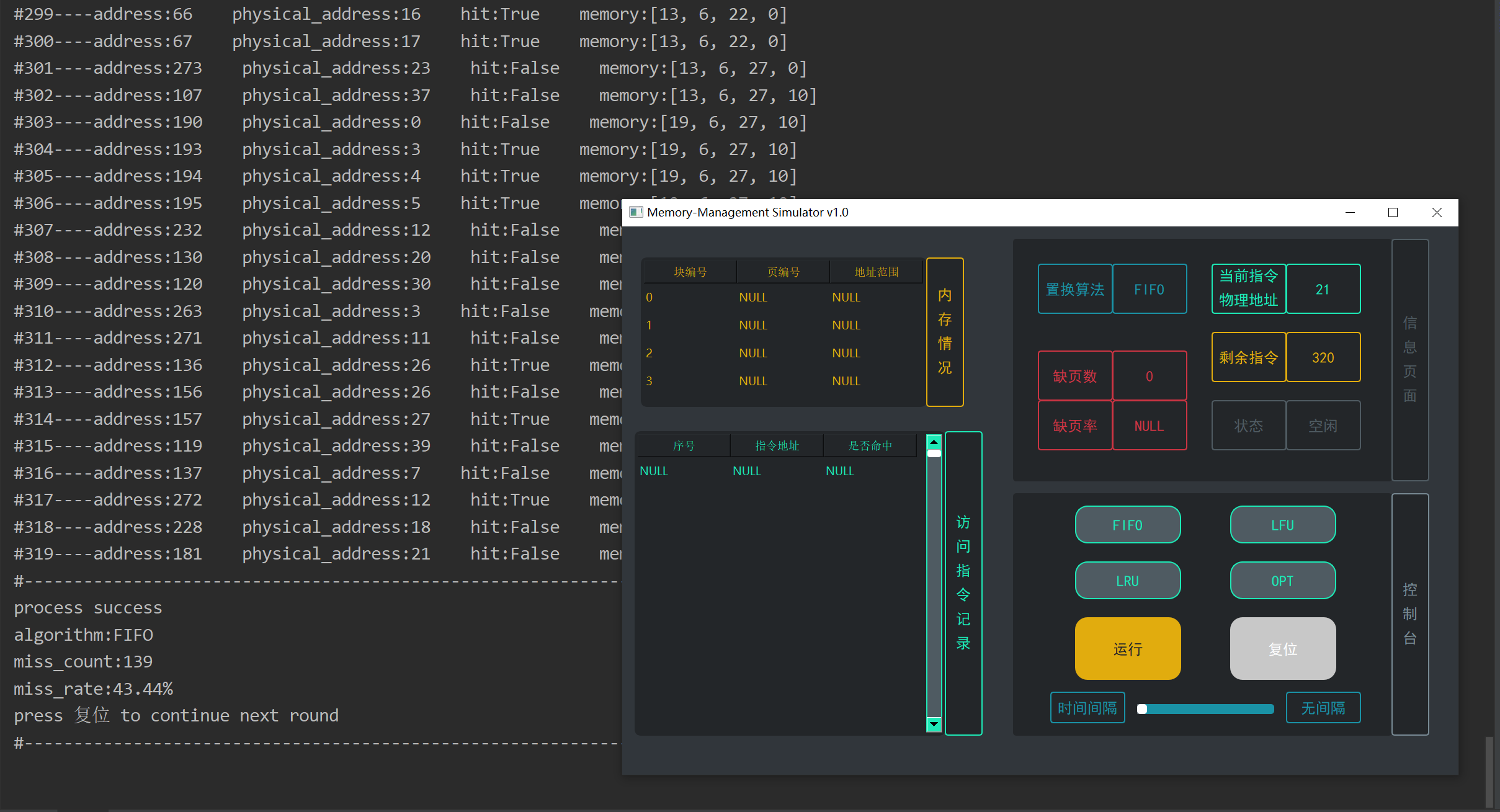
复位键的点击挂接了trigger\_reset函数

算法选择键的点击挂接了select\_algorithm函数

**3.项目测试**

****

****

****

**4.项目评价**

4.1 可改进的地方

①由于实时展示了较多的信息，使用了较多的同步控制，导致指令处理速度实际上慢了一些，即使是时间间隔没有设限制也比较慢。在单步处理上可以寻找更为高效的处理办法。

②展示信息的布局上可以改进，如深入思考哪些信息应该显示实时值、哪些信息应该显示一条条记录；哪些信息应该显示在系统界面上、哪些信息可以显示在控制台（计算机的）上。

③一旦运行，控制台上的按键全部失效，无法暂停或进行单步调试，对于观察序列处理过程并不友好。

④表格字号可能较小，界面设计上可以再改进为更友好。

4.2 解决了的特殊问题

①实现了多种信息，包括当前算法、缺页信息、物理地址、剩余指令数、系统状态在内的实时显示

②实现了复位功能

③由于PyQt中表格的特殊使用方法，刚进入程序不能直接复位；且运行完成后若不复位则结果没有意义且有可能出错。本项目实现了空闲状态下只能运行、运行状态下所有按键无法按下、停止状态下只能复位的按键状态切换

④实现了用滑动条控制时间间隔，从而控制处理速度

⑤实现了置换算法LFU、OPT