## 请求调页存储管理方式模拟REPORT

Created time: May 30, 2023 3:53 PM  
Email: [momoyama.sawa@gmail.com](mailto:momoyama.sawa@gmail.com)  
Last edited time: May 30, 2023 5:04 PM  
Owner: 寧々寝る

[请求调页存储管理方式模拟REPORT](#请求调页存储管理方式模拟report)  
[项目简介](#项目简介)  
[项目目的](#项目目的)  
[项目环境](#项目环境)  
[项目功能](#项目功能)  
[系统架构](#系统架构)  
[模块设计](#模块设计)  
 [核心算法模块](#核心算法模块)  
 [指令部分](#指令部分)  
 [内存部分](#内存部分)  
 [演示模块](#演示模块)  
[算法设计](#算法设计)  
 [置换算法](#置换算法)  
 [指令序列的模拟](#指令序列的模拟)  
[其他](#其他)  
[项目总结](#项目总结)  
 [项目亮点](#项目亮点)  
 [项目改进方向](#项目改进方向)

# 项目简介

本项目为同济大学软件学院2023年春季学期操作系统课程项目的请求调页存储管理方式模拟项目。通过模拟内存中指令的执行以及缺页置换的过程，本项目通过可视化很好的展现了内存管理的核心思路。

# 项目目的

* 体会页面、页表、地址转换
* 体会页面置换过程
* 加深对请求调页系统的原理和实现过程的理解。

# 项目环境

* 开发环境：
  + python pyqt6
  + win11
* 运行方法
  + cd dist\start
  + .\demo.exe

# 项目功能



* 实现了请求调页存储管理方式模拟的基本功能
* 指令的单步执行和连续执行，以及模拟运行的重置
* 指令总数、每页存放指令数、作业占内存页数可在可定范围内更改
* 可以更换页面置换算法，有FIFO和LRU
* 可以选择指令执行顺序，有随机执行和预设执行，其中预设执行推荐以默认参数执行，会演示17条预设指令，包括了LRU算法中的各种情况的展示
* 生动形象的内存中的界面展示，还附带动画
* 已执行指令的相关信息以列表展示，展示模拟运行的日志

# 系统架构

分为演示模块和核心算法模块，其中演示模块聚合核心算法模块，核心算法模块通过信号槽机制来和演示模块交互

* 核心算法模块：控制模拟运行底层指令的执行，发出信号给出日志等信息
* 演示模块：接收信号根据核心算法系统的执行信息进行展示，同时控制核心算模块的执行参数

# 模块设计

## 核心算法模块

核心算法模块分为指令部分和内存部分

### 指令部分

指令部分控制随机指令的生成或者读取预设指令，同时提供统一接口供演示模块模拟执行，控制指令读取的各种信息

### 内存部分

控制内存的各种信息，实现模拟内存的行为和算法，提供了统一接口进内存供指令部分调用，同时发出包含内存变动日志的信号，供演示模块演示

class MemAlgorithm(ABC):  
 @abstractmethod  
 def replace(self, pageID: int, list: list) -> list[int, list, int]:  
 pass  
  
 @abstractmethod  
 def check(self, pageID: int, list: list, new: bool) -> list:  
 pass

定义了一个内存算法抽象类，其中replace是在需要置换页面的时候调用，check是在不需要置换页面的时候调用

def \_\_check(self, pageID: int) -> bool:  
 # 检查页是否在内存中  
 # 在内存中  
 if pageID in self.spaceList:  
 self.spaceList = self.al.check(pageID, self.spaceList, False)  
 return True  
 # 内存中有空位  
 elif None in self.spaceList:  
 index = len(self.spaceList) - self.spaceList[::-1].index(None) - 1  
 self.spaceList[index] = pageID  
 self.spaceList = self.al.check(pageID, self.spaceList, True)  
 self.check = len(self.spaceList) - index - 1  
 return True  
 # 不在内存中，且内存已满  
 else:  
 return False  
  
def inMem(self, ins: int):  
 pageID, offset = getPage(ins, self.pageLen)  
 # 算法实现写入或替换  
 # 首先要检查是不是已经在内存中吧，如果已经在内存中或者没装满都返回true  
 miss = None  
 repalce = None  
 self.check = -1  
 if not self.\_\_check(pageID):  
 miss = True  
 repalce, self.spaceList, self.check = self.al.replace(  
 pageID, self.spaceList  
 )  
 else:  
 miss = False  
 # 同时要信号槽更新界面  
 self.signal.emit(Item(ins, pageID, offset, miss, repalce))  
 self.memSignal.emit(self.check, self.spaceList, pageID, offset)

进内存的函数会首先检查所需页面是否已经在内存中，没在的话要考虑添加页面/考虑置换，在的话也要考虑使用的算法是否要调整页面顺序，同时最后还要发出信号传递信息给演示模块展示

## 演示模块

聚合核心模块，接收各种信息来进行演示，控制核心算模块的执行参数，提供可视化图形化演示界面，控制项目程序运行

# 算法设计

## 置换算法

本项目中实现了两个经典页面置换算法，即**先进先出(FIFO)**和**最近最少使用(LRU)**算法。

首先是***FIFO算法***：

先进先出算法是最简单的页面换算法，是指每次有新的分页需要调入时，会选择调入内存时间最久的分页换出。它简单，容易实现，但这种绝对的公平方式容易导致效率的降低。

本项目中，FIFO算法被设计为算法抽象类的子类，实现上类似于队列结构，先进先出

| 方法 | 描述 |
| --- | --- |
| replace | 用于需要置换的场合，置换存储位置最前的页，其他页前移，把新放入的页放到最后，实现类似队列的效果 |
| check | 用于不需要置换的场合，若没有新页加入则什么也不做（命中），若有新页加入（就是分配的内存没被填满的时候有新页加入），则存储位置放到已有页面之后 |

class FIFO(MemAlgorithm):  
 def replace(self, pageID: int, list: list) -> list[int, list, int]:  
 # 置换最前面的元素，其他前移，把pageID放到最后  
 replace = list[0]  
 list = list[1:] + [pageID]  
 check = len(list) - 1  
 return replace, list, check  
  
 def check(self, pageID: int, list: list, new: bool) -> list:  
 if new:  
 index = list.index(pageID)  
 list = list[:index] + list[index + 1 :] + [list[index]]  
 return list

而***LRU算法***的原理则如下所示：

最近最少使用算法，是一种常用的页面置换算法，选择最近最久未使用的页面予以淘汰。该算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 t，当须淘汰一个页面时，选择现有页面中其 t 值最大的，即最近最少使用的页面予以淘汰。

本项目中，LRU算法被设计为算法抽象类的子类，实现上类似于栈结构，最近被使用的压到栈底，而很久没被使用的会在栈顶，当要置换的时候被弹出

| 方法 | 描述 |
| --- | --- |
| replace | 用于需要置换的场合，置换存储位置最前的页，其他页前移，把新放入的页放到最后，实现 |
| check | 用于不需要置换的场合，若没有新页加入则将命中页放到最后（命中，并且实现放回栈底），若有新页加入（就是分配的内存没被填满的时候有新页加入），则存储位置依旧放到最后（放入栈底） |

class LRU(MemAlgorithm):  
 def replace(self, pageID: int, list: list) -> list[int, list, int]:  
 # 置换最前面的元素，其他前移，把pageID放到最后  
 replace = list[0]  
 list = list[1:] + [pageID]  
 check = len(list) - 1  
 return replace, list, check  
  
 def check(self, pageID: int, list: list, new: bool) -> list:  
 # 移动元素到最后  
 index = list.index(pageID)  
 list = list[:index] + list[index + 1 :] + [list[index]]  
 return list

## 指令序列的模拟

为了模拟出：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。

本项目通过先在0~n 中产生一个随机数i，作为第一条指令的序号

然后采用0-100的随机数来表示概率，概率执行以下下三条

顺序执行下一条指令i+1；

之后在0~i-1中产生一个随机数作为指令序号j，即在跳转到前地址部分；

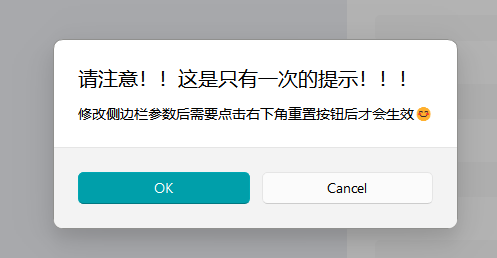
之后在i+2~n中产生一个随机数执行，代表跳转到后地址，再顺序执行下一条指令。

# 其他

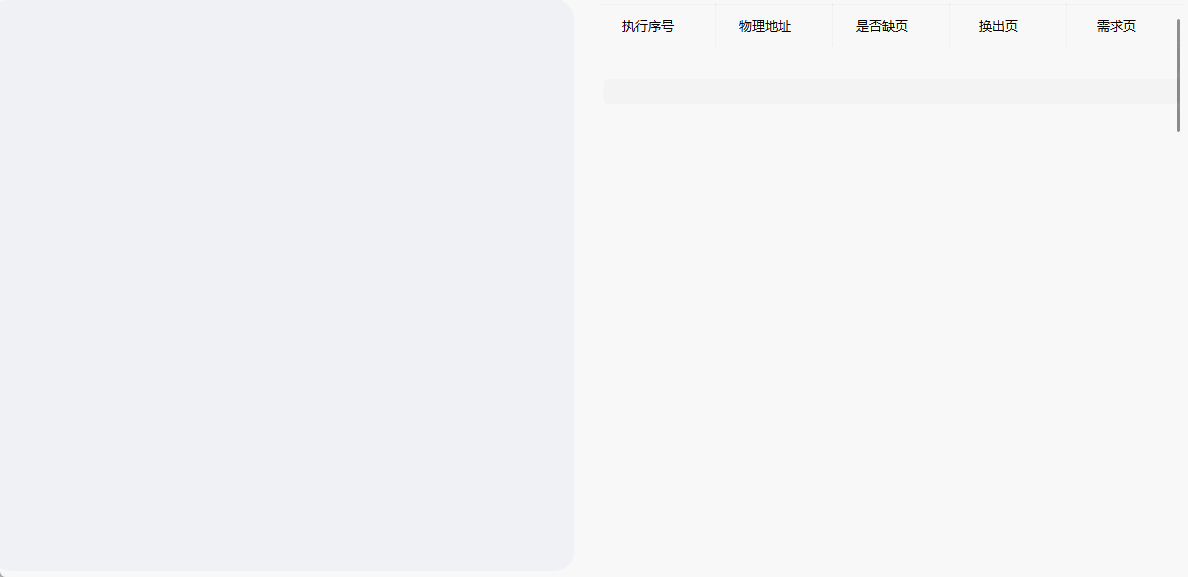
* 提示清晰，引导好







* 动画效果好，不管是交换栈里页面的顺序还是新加页面都很生动，展示清晰



* 交互性好



# 项目总结

## 项目亮点

* 模拟运行参数可自定义，实现了两种算法
* 页面美观，动画流畅
* 演示生动形象，加上有日志显示，便于观察
* 提示信息清晰
* 代码架构较好

## 项目改进方向

* 可以考虑更多的置换算法
* 暂无撤回上一步操作的功能，可以做一个状态机
* 可以实现模拟演示更多地自定义功能
* 可以导入指定指令文件执行
* 代码结构可以进一步优化（写到后面就再也没在意代码结构了×）