课程设计名称：操作系统A

题目 页式存储器管理置换算法模拟

|  |  |
| --- | --- |
| **年 级：** | 2022级 |
| **开发小组名称：** | 重生之我有一个操作系统 |
| **开发日期：** | 2025年5月19日-2025年5月30日 |

摘 要

本课程设计旨在开发一个模拟操作系统页面置换算法的实验平台，通过对比分析FIFO、LRU、LFU和OPT四种典型页面置换算法，帮助理解内存管理机制及其性能差异。实验平台支持用户自定义页面访问序列，或随机生成逻辑页号，同时可设定驻留内存页数、内存访问时间、缺页中断时间和快表访问时间等参数。系统通过多线程并发执行四种算法，并使用图形界面实时展示内存状态、缺页次数与访问耗时等结果。

平台采用 Flask 框架搭建后端，配合前端可视化交互设计，实现了多线程页面置换过程的动态演示和实验数据的实时收集。实验过程中，用户可暂停、继续模拟并保存数据，便于后续分析与比较。通过多组实验，本平台有效展现了各算法在不同访问模式下的性能表现，并揭示了快表、内存容量等关键因素对算法效率的影响。

关键词：页面置换算法；图形界面；多线程；快表

目 录

[摘 要 I](#_Toc31572)

[目 录 I](#_Toc15517)

[第1章 前 言 1](#_Toc23668)

[1.1 项目背景 1](#_Toc32712)

[1.2 目的和意义 1](#_Toc17267)

[第2章 主要任务 2](#_Toc3306)

[2.1 项目概述 2](#_Toc23251)

[2.2 需求分析 2](#_Toc31520)

[第3章 总体设计 3](#_Toc8530)

[3.1基本概念和原理 3](#_Toc27102)

[3.1.1基本概念 3](#_Toc11647)

[3.1.2项目原理 3](#_Toc24255)

[3.2 功能设计 4](#_Toc6854)

[3.2.1 参数设置 4](#_Toc27342)

[3.2.2 开始、暂停、继续、停止 4](#_Toc21517)

[3.2.3 全部清除 4](#_Toc6972)

[3.2.4 保存过程、保存结果 4](#_Toc20510)

[3.2.5 页面置换算法 4](#_Toc6298)

[3.2.6 结果对比 5](#_Toc13934)

[3.3 算法思想 5](#_Toc12560)

[3.3.1 FIFO 5](#_Toc22268)

[3.3.2 LFU 5](#_Toc11567)

[3.3.3 LRU 6](#_Toc30309)

[3.3.4 OPT 6](#_Toc3137)

[3.4 流程图 7](#_Toc14179)

[3.4.1总体流程图 7](#_Toc27728)

[3.4.2 FIFO 8](#_Toc30014)

[3.4.3 LFU 9](#_Toc17214)

[3.4.4 LRU 10](#_Toc16144)

[3.4.5 OPT 11](#_Toc31127)

[第4章 详细设计 12](#_Toc25022)

[4.1 整体思路 12](#_Toc15626)

[4.1.1 后端服务 12](#_Toc7950)

[4.1.2 算法管理 12](#_Toc20406)

[4.1.3 数据交互 12](#_Toc25174)

[4.2 算法实现 12](#_Toc669)

[4.2.1 FIFO（先进先出算法） 12](#_Toc1925)

[4.2.2 LFU（最少使用算法） 15](#_Toc15619)

[4.2.3 LRU（最近最少使用算法） 18](#_Toc12187)

[4.2.4 OPT（最佳置换算法） 21](#_Toc16264)

[4.3 基本功能实现 24](#_Toc13779)

[4.3.1 参数设定 24](#_Toc2356)

[4.3.2 手动输入序列 26](#_Toc13051)

[4.3.3 随机生成序列 26](#_Toc13783)

[4.3.4 线程并行模拟 27](#_Toc10461)

[4.3.5 暂停/继续/停止 27](#_Toc10342)

[4.3.6 快表模式切换 29](#_Toc32168)

[4.3.7 运行结果展示 30](#_Toc10555)

[4.3.8 过程结果保存 34](#_Toc28583)

[第5章 运行结果 37](#_Toc8037)

[总 结 39](#_Toc32039)

[参考文献 40](#_Toc2098)

[使用手册 41](#_Toc17341)

第1章 前 言

## 1.1 项目背景

本课程设计是为了深入理解和研究操作系统中页面置换算法的效率和性能。通过设计和实现一个模拟操作系统页面置换算法的实验平台，可以直观地观察和比较不同算法的运行结果，进一步分析各种算法的优缺点。同时，该实验平台也提供了一个可交互的环境，让用户能够自定义实验参数，深入探索页面置换算法的原理和效果。

## 1.2 目的和意义

通过设计和实现一个模拟操作系统页面置换算法的实验平台，帮助我们更好地理解和掌握页面置换算法的原理和实现。通过实际操作和观察，可以更深入地了解不同页面置换算法的性能和效率，并分析各种算法的优缺点。同时，该实验平台也提供了一个可交互的环境，让用户能够自定义实验参数，进一步探索页面置换算法的效果和影响。此外，通过完成多次不同设置的实验，我们可以总结实验数据，得出一些有意义的结论，从而提高他们对操作系统内存管理机制的理解和掌握。总之，本课程设计报告旨在通过实践操作和实验分析，加深了我们对操作系统页面置换算法的理解，提高了我们的实际操作能力和小组合作能力。

第2章 主要任务

2.1 项目概述

该项目旨在开发一个动态页面置换算法模拟器，通过模拟不同的页面置换算法，如FIFO、LRU、LFU和OPT，来深入理解操作系统中的内存管理机制。用户通过输入或随机生成页面访问序列，模拟不同算法的行为，实时展示内存状态、缺页次数和页面访问时间等信息。该系统不仅可以展示四种经典的页面置换算法，还支持调节内存大小、快表的使用以及其他相关参数，帮助用户对页面置换算法的性能进行可视化比较。

本项目通过使用Flask框架构建Web应用，前端负责与用户的交互，后端则使用Python实现各种置换算法的逻辑。该系统提供了直观的图形界面，能够实时显示算法的执行结果和内存状态，并且可以在用户交互过程中提供算法性能的实时反馈。

2.2 需求分析

(1) 输入一个逻辑页面访问序列和随机产生逻辑页面访问序列，由四个线程同时完成每个算法；

(2) 能够设定驻留内存页面的个数、内存的存取时间、缺页中断的时间、快表的时间，并提供合理省缺值，可以暂停和继续系统的执行；

(3) 能够随机输入存取的逻辑页面的页号序列；

(4) 能够随机产生存取的逻辑页面的页号序列；

(5) 能够设定页号序列中逻辑页面个数和范围；

(6) 能够设定有快表和没有快表的运行模式；

(7) 提供良好图形界面，同时能够展示四个算法运行的结果；

(8) 给出每种页面置换算法每个页面的存取时间；

(9) 能够将每次的实验输入和实验结果存储起来，下次运行时或以后可查询；

(10) 完成多次不同设置的实验，总结实验数据，看看能得出什么结论。

第3章 总体设计

## 3.1基本概念和原理

### 3.1.1基本概念

（1）多道程序：允许多个程序（作业）同时进入一个计算机系统的内存并启动进行交替计算的方法。

（2）进程：进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。

（3）线程指在程序执行过程中，能够执行程序代码的一个执行单位，每个程序至少都有一个线程，也就是程序本身。

（4）同步与互斥：同步是多个进程或线程之间的协同工作，确保按照预定顺序访问共享资源；互斥是多个进程或线程之间的互相排斥，确保在访问共享资源时不发生冲突。

### 3.1.2项目原理

本项目采用前后端分离的架构，后端使用Python结合Flask框架实现页面置换算法的模拟逻辑，前端通过异步请求与后端进行交互，实现数据的动态展示。

在后端部分，系统将每种置换算法（FIFO、LRU、LFU、OPT）封装为独立的类，每个类中实现对应的页面置换策略。Flask接收到前端请求后，根据用户选择的算法及输入的页面访问序列，调用对应算法类的处理函数，生成内存页帧变换过程和统计结果。

为提高响应效率和模拟并发处理效果，部分逻辑采用多线程处理。主线程负责接收和解析前端请求，同时通过设置信号触发各个算法类的运行线程。在算法运行过程中，系统使用Python中的threading模块配合Lock互斥锁机制，确保多个算法类在访问共享资源时的线程安全。

每个算法线程在执行完毕后，会将计算结果封装为结构化数据，并通过队列或信号机制将结果返回主线程。主线程统一收集各类算法的运行结果后，通过JSON格式发送回前端。

前端页面使用JavaScript结合Ajax技术实现异步数据交互，在接收到结果数据后，自动刷新图表、表格等界面组件，实现算法过程的可视化展示，包括内存变化过程、缺页次数和访问耗时等关键指标。

通过这种机制，本项目实现了模拟过程的并行执行、结果的统一回传和前端的动态刷新，提升了用户交互体验和系统性能。

3.2 功能设计

### 3.2.1 参数设置

基本设置：

可交互的图形化界面，用户可以在参数设置里规定页面个数、快表个数等从而生成一个随机的页面序列，点击保存参数，才能点击开始执行。

时间设置：

可以选择是否启动快表，设置快表大小、快表存取时间、内存存取时间、缺页中断时间。

### 3.2.2 开始、暂停、继续、停止

开始执行后，暂停按钮或者停止按钮才可操作，暂停后，继续按钮才可操作。

### 3.2.3 全部清除

清除后，页面置换算法模拟中生成序列的数据将被全部清除，参数设置不会被清除。

### 3.2.4 保存过程、保存结果

将过程和结果完整保存在txt文档并自动下载。

### 3.2.5 页面置换算法

页面分为四部分，记录四个算法的动态执行过程，是否使用快表在参数设置里决定。模拟表会记录中断次数、总时间、平均时间、缺页率。

### 3.2.6 结果对比

将算法执行结果对比分析，并选出性能最优的。

## 3.3 算法思想

### 3.3.1 FIFO

FIFO（First - In - First - Out）页面置换算法的基本原理是基于页面进入内存的先后顺序。它会记录每个页面进入内存的时间，当发生缺页中断且内存已满时，优先选择最早进入内存的页面进行置换。这种算法认为，最早进入内存的页面在未来一段时间内被访问的可能性相对较低。

FIFO算法的基本步骤：

（1）内存帧管理：要对内存中的帧进行管理，记录每个帧当前存储的页面信息。可以使用一个数组来表示内存帧，数组的每个元素对应一个内存帧，存储着该帧中当前页面的编号。

（2）页面进入顺序记录：为了明确哪个页面是最早进入内存的，需要记录每个页面进入内存的先后顺序。可以使用一个变量或者队列来辅助记录。

（3）缺页中断处理：在访问页面时，检查该页面是否在内存中。若不在，就发生了缺页中断。此时，若内存还有空闲帧，就将该页面调入空闲帧；若内存已满，就找出最早进入内存的页面，将其置换出去，再把新页面调入。

### 3.3.2 LFU

LFU（Least Frequently Used，最不经常使用）的核心思想是：当缓存空间不足时，淘汰使用频率最低的数据项。如果有多个数据项具有相同的使用频率，则淘汰其中最久未使用的数据项。

LFU（Least Frequently Used，最不经常使用）页面置换算法的模拟过程。LFU算法基于页面的使用频率来决定哪些页面应该被置换出去，使用频率最低的页面会被优先置换。首先初始化物理块数量和页面引用序列，然后通过遍历页面引用序列来模拟页面访问过程。对于每个页面，检查是否已经在内存中，如果不在则发生缺页，需要将该页面调入内存。当内存已满时，根据页面的使用频率选择置换目标，使用频率最低的页面会被置换出去，如果多个页面具有相同的最低使用频率，则选择最久未使用的页面进行置换。每次页面访问后，会记录当前内存状态并输出。最后统计缺页次数、缺页率和置换次数，并将结果保存到历史记录中。

### 3.3.3 LRU

**LRU（Least Recently Used，最近最少使用） 是一种基于页面访问历史的置换算法。其核心思想是：当内存中没有空闲物理块且发生缺页时，置换最近最久未被访问的页面。该算法假设“过去长时间未使用的页面，未来也很可能不再使用”，通过跟踪每个页面的最后访问时间来实现这一目标。它具有时间局部性：优先保留最近被频繁访问的页面。动态调整：每次页面访问都会更新其“最后使用时间”，确保置换策略始终反映最新的访问模式。**

**LRU算法的基本步骤：**

**（1）首先进行初始化：清空内存块（memory）。创建字典 last\_used，记录每个页面的最后访问时间（用页面引用序列的索引表示时间戳）。**

**（2）遍历页面引用序列：首先命中检查：检查当前页面是否已在内存中：若命中：更新该页面的 last\_used 时间戳。若缺页：进入缺页处理流程。**

**（3）缺页处理：检查空闲块：若存在空闲物理块，直接将页面加载到空闲块，并记录其访问时间。若无空闲块，遍历内存中所有页面，找到 last\_used时间戳最小的页面（即最久未使用的页面），将其替换为当前页面。**

**（4）更新与记录：更新当前页面的last\_used时间戳。记录每一步的内存状态（用于后续分析）。**

**（5）统计结果：计算缺页次数、缺页率及置换次数。**

### 3.3.4 OPT

OPT（最佳置换算法）的核心思想是：当缓存空间不足时，淘汰最长时间内不再被访问的页面，或是以后永不使用的页面，以便保证获得最低的缺页率。这种算法需要预先知道整个页面访问序列，因此在实际系统中无法实现，主要用作理论上的参考标准，去评价其他算法。

OPT算法的基本步骤：

（1）查找所请求的页面是否在内存中，如果在，则直接访问。

（2）如果不在内存中且内存有空闲页帧，则将页面装入空闲页帧。

（3）如果内存已满，则找出未来最长时间内不会被再次访问的页面将其置换出去。

（4）将所请求的页面装入内存。

## 3.4 流程图

### 3.4.1总体流程图

总体流程图如图3-1所示。

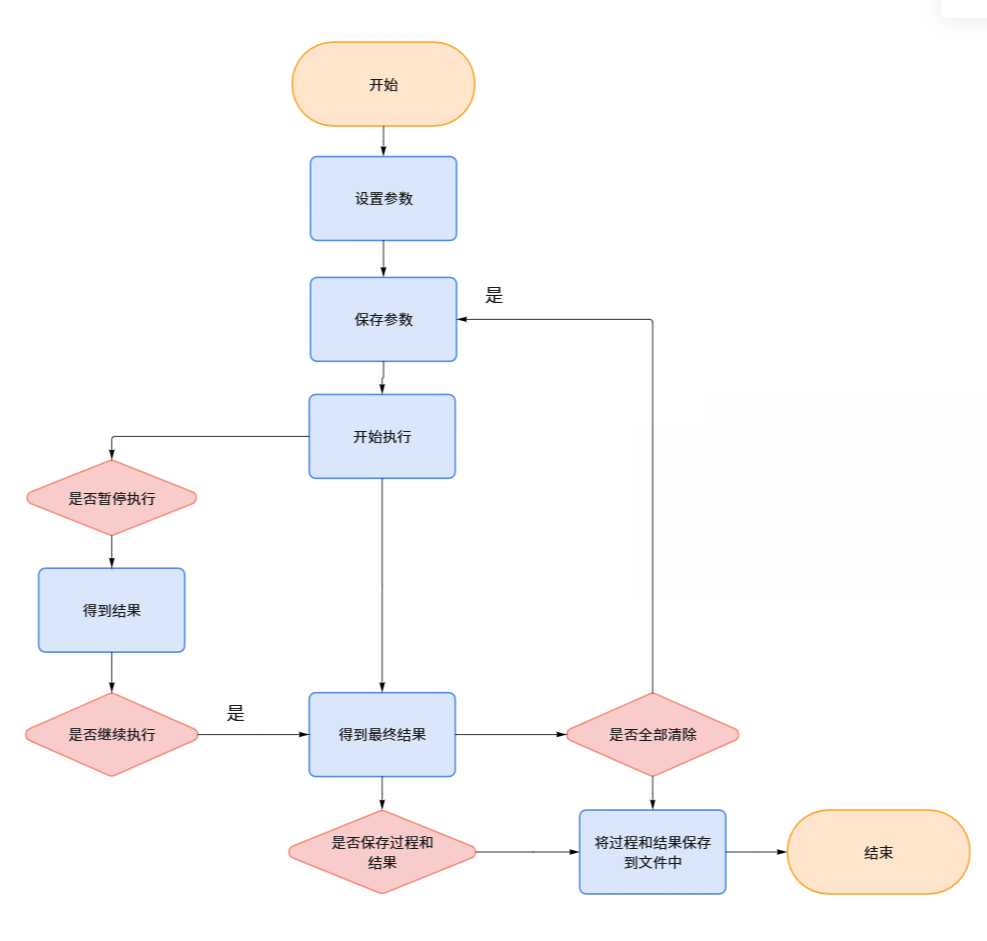


图3-1总体流程图

### 3.4.2 FIFO

FIFO置换算法流程图如图3-2所示。

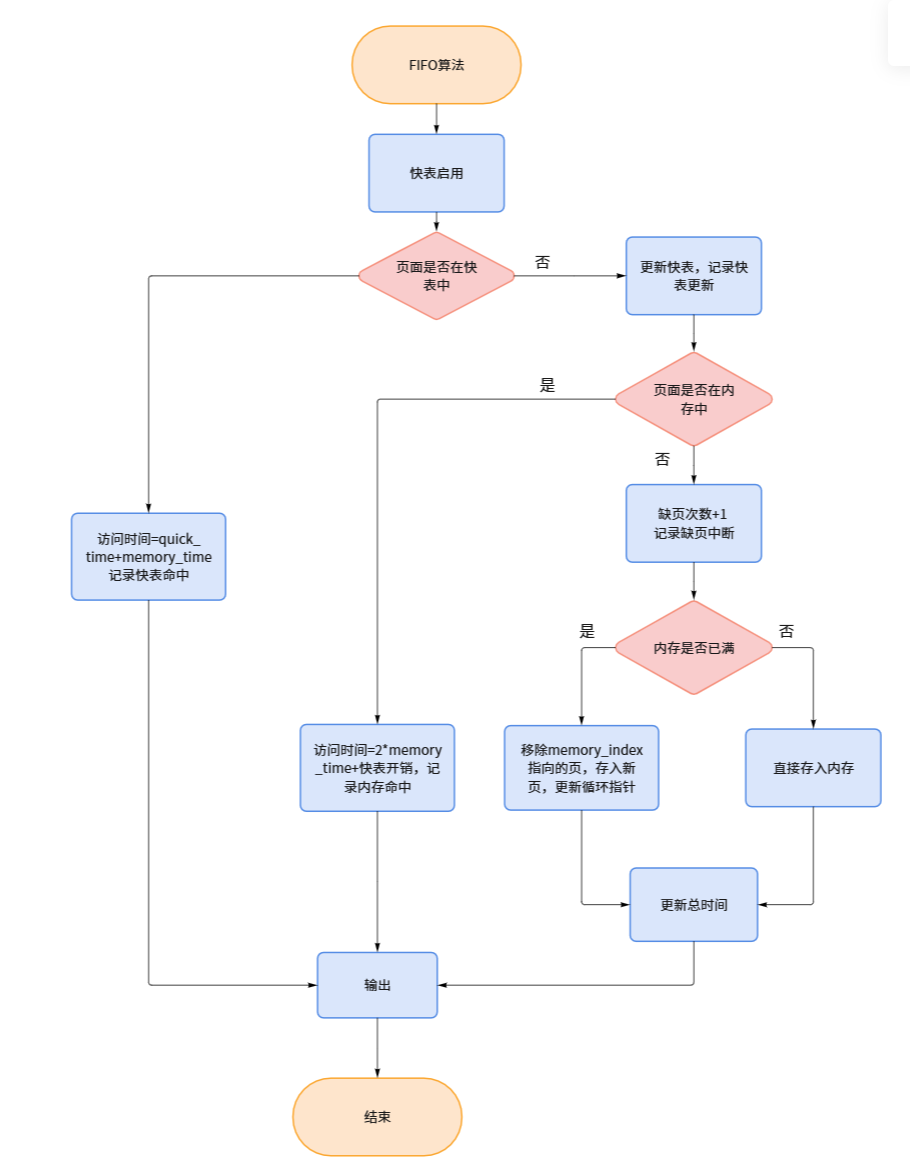


图3-2 FIFO置换算法流程图

### 3.4.3 LFU

LFU置换算法流程图如图3-3所示。

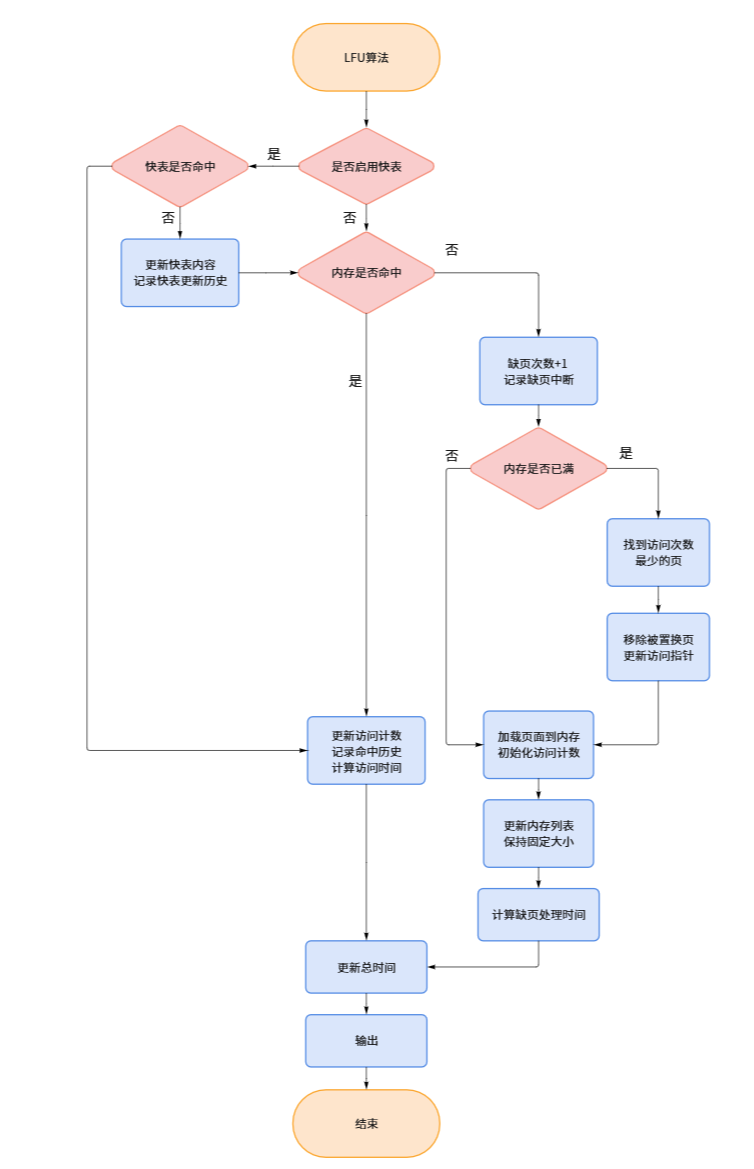


图3-3 LFU置换算法流程图

### 3.4.4 LRU

LRU置换算法流程图如图3-4所示。

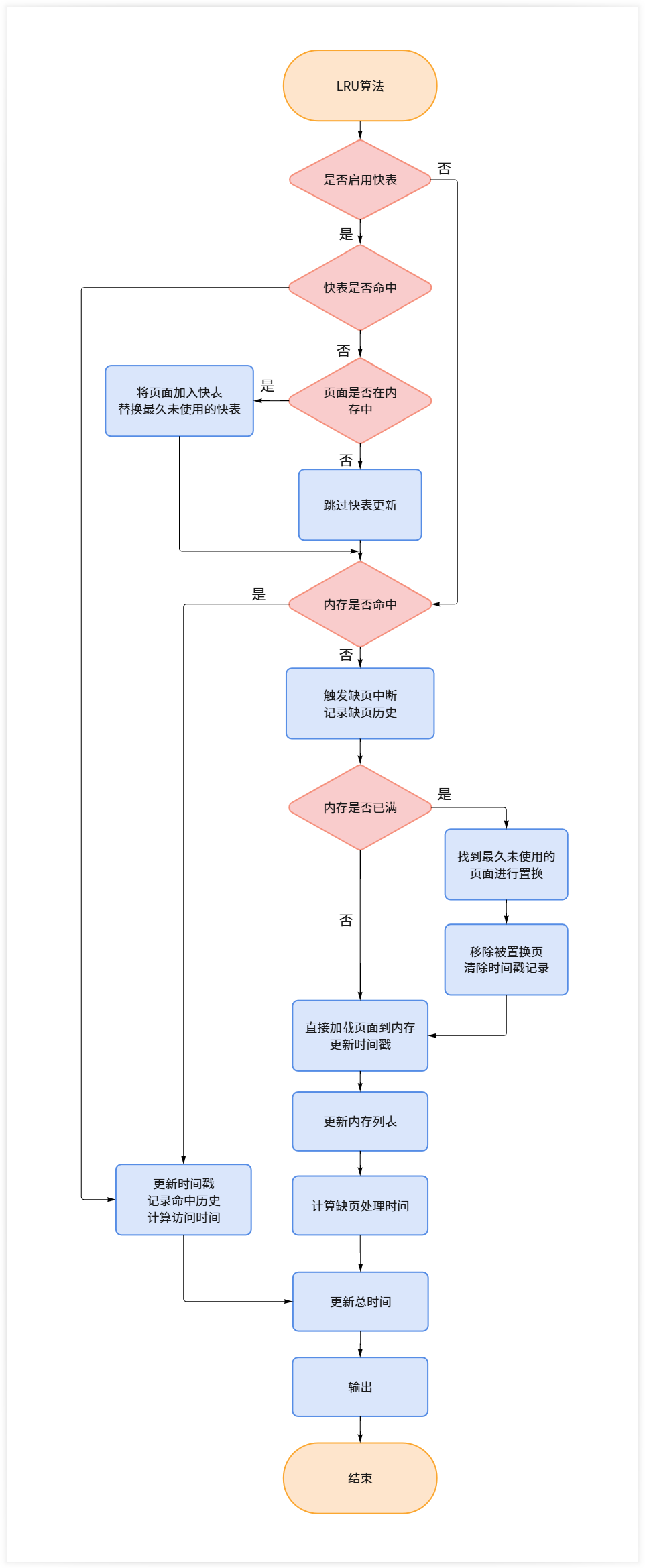


图3-4 LRU置换算法流程图

### 3.4.5 OPT

OPT置换算法流程图如图3-5所示。

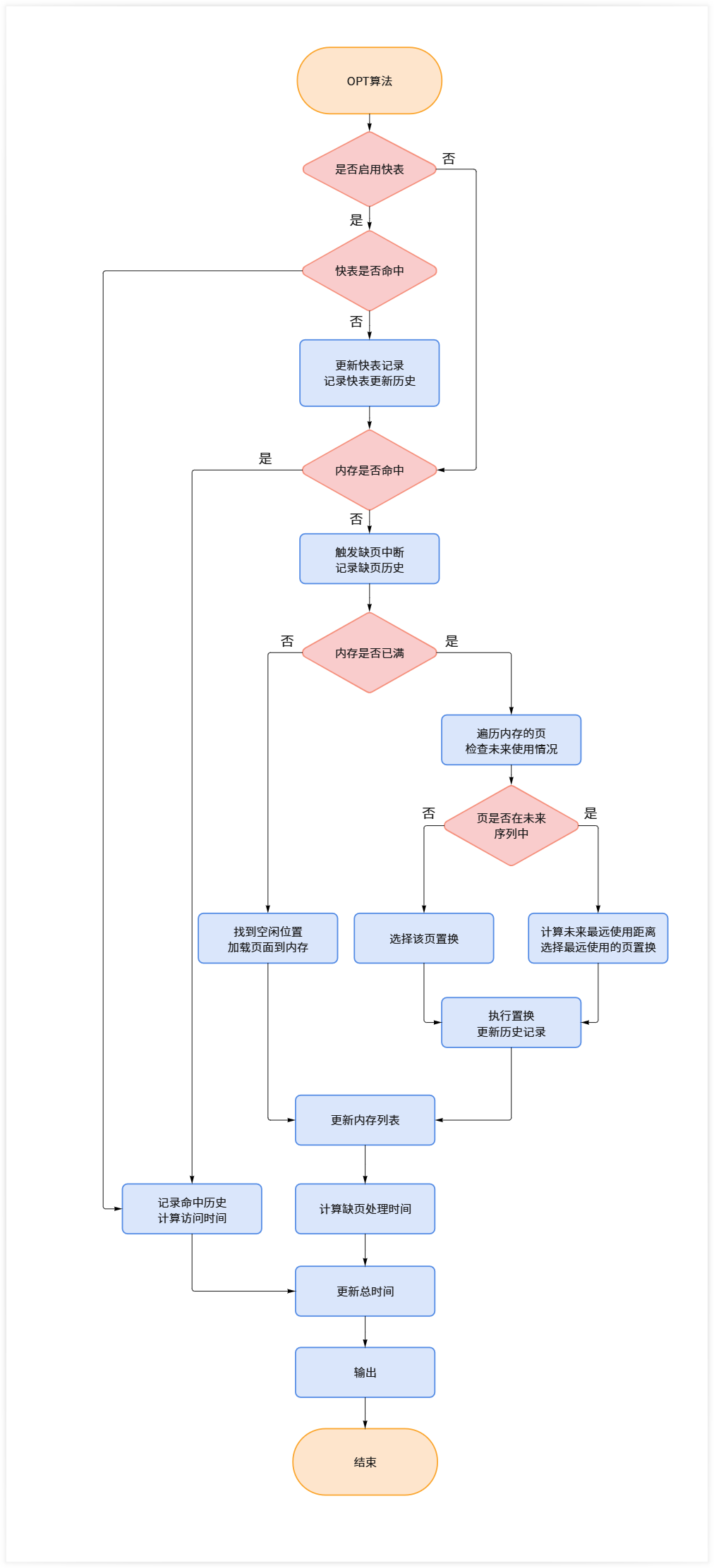


图3-5 OPT置换算法流程图

第4章 详细设计

## 4.1 整体思路

本项目基于Flask框架构建Web应用，实现对操作系统页面置换算法的模拟。前端负责用户交互和界面展示，后端通过Python实现算法逻辑。系统支持FIFO、LRU、OPT、LFU四种页面置换算法，可动态处理页面访问序列，并实时返回内存状态、缺页信息及性能指标。

4.1.1 后端服务

使用Flask搭建RESTful API，接收前端发送的页面访问请求。

维护algorithms字典存储各算法实例，避免重复创建对象。

通过日志系统（logging）记录请求处理过程，便于调试和问题定位。

4.1.2 算法管理

每个算法封装为独立类，包含初始化、页面处理、状态获取等方法。

支持动态创建算法实例：首次调用时根据算法名称（fifo/lru/opt/lfu）初始化对象，后续请求复用实例。

4.1.3 数据交互

前端通过JSON格式传递页面访问数据（如页号、内存大小、是否启用快表等）。

后端处理后返回内存状态、快表内容、缺页次数、总时间等信息，供前端实时更新界面。

## 4.2 算法实现

### 4.2.1 FIFO（先进先出算法）

**（1）核心逻辑**

**内存管理**：使用队列存储内存中的页号，遵循 “先入先出” 原则，置换时移除队首元素。

**快表处理**：若启用快表，维护固定大小的快表数组，采用循环覆盖策略更新快表内容。

**时间计算**：根据是否命中快表和内存，计算访问时间（快表命中时间 + 内存访问时间）。

**（2）代码实现**

class FIFO:

def \_\_init\_\_(self, memory\_size, use\_quick\_table=False, quick\_table\_size=4):

self.memory\_size = memory\_size

self.use\_quick\_table = use\_quick\_table

self.quick\_table\_size = quick\_table\_size

self.memory = [] # 内存队列

self.quick\_table = [] # 快表

self.faults = 0 # 缺页次数

self.total\_time = 0 # 总访问时间

self.history = [] # 访问历史记录

self.quick\_index = 0 # 快表更新索引

self.memory\_index = 0 # 内存队列索引（循环使用）

self.reset()

def reset(self):

"""初始化内存和快表"""

self.memory = [-1] \* self.memory\_size

if self.use\_quick\_table:

self.quick\_table = [-1] \* self.quick\_table\_size

self.faults = 0

self.total\_time = 0

self.history = []

self.quick\_index = 0

self.memory\_index = 0

def process\_page(self, page, time\_settings, future\_sequence=None):

"""处理页面访问"""

quick\_time = time\_settings['quick\_time']

memory\_time = time\_settings['memory\_time']

break\_time = time\_settings['break\_time']

is\_page\_fault = False

replaced\_page = -1

access\_time = 0

# 检查快表

if self.use\_quick\_table and page in self.quick\_table:

access\_time = quick\_time + memory\_time # 快表命中

self.history.append(f"快表命中: 页{page}")

else:

# 快表未命中，更新快表

if self.use\_quick\_table:

self.quick\_table[self.quick\_index] = page

self.quick\_index = (self.quick\_index + 1) % self.quick\_table\_size

self.history.append(f"快表更新: 页{page}")

# 检查内存

if page in self.memory:

access\_time = (2 \* quick\_time if self.use\_quick\_table else 0) + 2 \* memory\_time # 内存命中，访问页表+内存

self.history.append(f"内存命中: 页{page}")

else:

# 缺页处理

is\_page\_fault = True

self.faults += 1

access\_time = (2 \* quick\_time if self.use\_quick\_table else 0) + 2 \* memory\_time + break\_time # 缺页中断

self.history.append(f"缺页中断: 页{page}")

# 置换内存中的页（FIFO）

if -1 not in self.memory: # 内存已满

replaced\_page = self.memory[self.memory\_index]

self.memory[self.memory\_index] = page

self.memory\_index = (self.memory\_index + 1) % self.memory\_size

self.history.append(f"置换: 移除页{replaced\_page}, 加入页{page}")

else: # 内存未满，直接加入

self.memory[self.memory.index(-1)] = page

self.total\_time += access\_time

return is\_page\_fault, replaced\_page, access\_time

def get\_state(self):

"""获取当前算法状态"""

return {

'memory': self.memory.copy(),

'quick\_table': self.quick\_table.copy(),

'faults': self.faults,

'total\_time': self.total\_time,

'step': len(self.history)

}

### 4.2.2 LFU（最少使用算法）

**（1）核心逻辑**

**内存管理：**使用字典记录页号的访问次数，置换时移除访问次数最少的页（若次数相同，按 FIFO 处理）。

**时间计算：**同 FIFO，根据访问命中情况计算时间。

**（2）代码实现**

class LFU:

def \_\_init\_\_(self, memory\_size, use\_quick\_table=False, quick\_table\_size=4):

self.memory\_size = memory\_size

self.use\_quick\_table = use\_quick\_table

self.quick\_table\_size = quick\_table\_size

self.memory = [] # 内存列表

self.quick\_table = [] # 快表

self.access\_count = {} # 记录页访问次数

self.faults = 0

self.total\_time = 0

self.history = []

self.quick\_index = 0

self.reset()

def reset(self):

self.memory = [-1] \* self.memory\_size

if self.use\_quick\_table:

self.quick\_table = [-1] \* self.quick\_table\_size

self.access\_count.clear()

self.faults = 0

self.total\_time = 0

self.history = []

self.quick\_index = 0

def process\_page(self, page, time\_settings, future\_sequence=None):

quick\_time = time\_settings['quick\_time']

memory\_time = time\_settings['memory\_time']

break\_time = time\_settings['break\_time']

is\_page\_fault = False

replaced\_page = -1

access\_time = 0

# 检查快表

if self.use\_quick\_table and page in self.quick\_table:

access\_time = quick\_time + memory\_time

self.access\_count[page] += 1

self.history.append(f"快表命中: 页{page}")

else:

# 快表未命中，更新快表

if self.use\_quick\_table:

self.quick\_table[self.quick\_index] = page

self.quick\_index = (self.quick\_index + 1) % self.quick\_table\_size

self.history.append(f"快表更新: 页{page}")

# 检查内存

if page in self.memory:

access\_time = (2 \* quick\_time if self.use\_quick\_table else 0) + 2 \* memory\_time

self.access\_count[page] += 1

self.history.append(f"内存命中: 页{page}")

else:

# 缺页处理

is\_page\_fault = True

self.faults += 1

access\_time = (2 \* quick\_time if self.use\_quick\_table else 0) + 2 \* memory\_time + break\_time

self.history.append(f"缺页中断: 页{page}")

# 置换内存中的页（LFU）

if -1 not in self.memory:

# 找到访问次数最少的页（若次数相同，按FIFO处理）

min\_count = min(self.access\_count.values())

candidates = [p for p in self.memory if self.access\_count[p] == min\_count]

# 按FIFO选择最早进入的页

replaced\_page = candidates[0]

self.memory.remove(replaced\_page)

self.access\_count.pop(replaced\_page)

self.history.append(f"置换: 移除页{replaced\_page}, 加入页{page}")

self.memory.append(page)

self.access\_count[page] = 1

# 保持内存大小固定

if len(self.memory) > self.memory\_size:

self.memory = self.memory[-self.memory\_size:]

self.total\_time += access\_time

return is\_page\_fault, replaced\_page, access\_time

### 4.2.3 LRU（最近最少使用算法）

**（1）核心逻辑**

**内存管理**：使用字典记录页号的最后访问时间，每次访问时更新时间，置换时移除时间戳最小的页。

**快表处理**：与 FIFO 类似，但快表更新时优先保留最近使用的页。

**时间计算**：同 FIFO，根据快表和内存命中情况计算时间。

**（2）代码实现**

class LRU:

def \_\_init\_\_(self, memory\_size, use\_quick\_table=False, quick\_table\_size=4):

self.memory\_size = memory\_size

self.use\_quick\_table = use\_quick\_table

self.quick\_table\_size = quick\_table\_size

self.memory = [] # 内存列表

self.quick\_table = [] # 快表

self.last\_used = {} # 记录页最后访问时间（时间戳）

self.faults = 0

self.total\_time = 0

self.history = []

self.quick\_index = 0

self.timestamp = 0 # 时间戳计数器

self.reset()

def reset(self):

self.memory = [-1] \* self.memory\_size

if self.use\_quick\_table:

self.quick\_table = [-1] \* self.quick\_table\_size

self.last\_used.clear()

self.faults = 0

self.total\_time = 0

self.history = []

self.quick\_index = 0

self.timestamp = 0

def process\_page(self, page, time\_settings, future\_sequence=None):

quick\_time = time\_settings['quick\_time']

memory\_time = time\_settings['memory\_time']

break\_time = time\_settings['break\_time']

is\_page\_fault = False

replaced\_page = -1

access\_time = 0

# 检查快表

if self.use\_quick\_table and page in self.quick\_table:

access\_time = quick\_time + memory\_time

self.last\_used[page] = self.timestamp # 更新时间戳

self.timestamp += 1

self.history.append(f"快表命中: 页{page}")

else:

# 快表未命中，更新快表

if self.use\_quick\_table:

if page in self.memory:

# 若内存中有该页，将其加入快表（替换最久未使用的快表项）

self.quick\_table[self.quick\_index] = page

self.quick\_index = (self.quick\_index + 1) % self.quick\_table\_size

self.history.append(f"快表更新: 页{page}")

# 检查内存

if page in self.memory:

access\_time = (2 \* quick\_time if self.use\_quick\_table else 0) + 2 \* memory\_time

self.last\_used[page] = self.timestamp # 更新时间戳

self.timestamp += 1

self.history.append(f"内存命中: 页{page}")

else:

# 缺页处理

is\_page\_fault = True

self.faults += 1

access\_time = (2 \* quick\_time if self.use\_quick\_table else 0) + 2 \* memory\_time + break\_time

self.history.append(f"缺页中断: 页{page}")

# 置换内存中的页（LRU）

if len(self.memory) == self.memory\_size:

# 找到最久未使用的页

oldest\_page = min(self.last\_used, key=self.last\_used.get)

replaced\_page = oldest\_page

self.memory.remove(oldest\_page)

self.last\_used.pop(oldest\_page)

self.memory.append(page)

self.last\_used[page] = self.timestamp

self.timestamp += 1

# 保持内存大小固定

if len(self.memory) > self.memory\_size:

self.memory = self.memory[-self.memory\_size:]

self.total\_time += access\_time

return is\_page\_fault, replaced\_page, access\_time

### 4.2.4 OPT（最佳置换算法）

**（1）核心逻辑**

**内存管理：**根据未来页面访问序列，置换最远将使用的页。需预先知道未来访问序列（仅用于模拟）。

**时间计算：**与FIFO类似，但置换策略基于未来访问预测。

**（2）代码实现**

class OPT:

def \_\_init\_\_(self, memory\_size, use\_quick\_table=False, quick\_table\_size=4):

self.memory\_size = memory\_size

self.use\_quick\_table = use\_quick\_table

self.quick\_table\_size = quick\_table\_size

self.memory = [] # 内存列表

self.quick\_table = [] # 快表

self.faults = 0

self.total\_time = 0

self.history = []

self.quick\_index = 0

self.reset()

def reset(self):

self.memory = [-1] \* self.memory\_size

if self.use\_quick\_table:

self.quick\_table = [-1] \* self.quick\_table\_size

self.faults = 0

self.total\_time = 0

self.history = []

self.quick\_index = 0

def process\_page(self, page, time\_settings, future\_sequence):

quick\_time = time\_settings['quick\_time']

memory\_time = time\_settings['memory\_time']

break\_time = time\_settings['break\_time']

is\_page\_fault = False

replaced\_page = -1

access\_time = 0

# 检查快表

if self.use\_quick\_table and page in self.quick\_table:

access\_time = quick\_time + memory\_time

self.history.append(f"快表命中: 页{page}")

else:

# 快表未命中，更新快表

if self.use\_quick\_table:

self.quick\_table[self.quick\_index] = page

self.quick\_index = (self.quick\_index + 1) % self.quick\_table\_size

self.history.append(f"快表更新: 页{page}")

# 检查内存

if page in self.memory:

access\_time = (2 \* quick\_time if self.use\_quick\_table else 0) + 2 \* memory\_time

self.history.append(f"内存命中: 页{page}")

else:

# 缺页处理

is\_page\_fault = True

self.faults += 1

access\_time = (2 \* quick\_time if self.use\_quick\_table else 0) + 2 \* memory\_time + break\_time

self.history.append(f"缺页中断: 页{page}")

# 置换内存中的页（OPT）

if -1 not in self.memory:

# 找到未来最远使用的页

max\_distance = -1

replaced\_page = self.memory[0]

for mem\_page in self.memory:

if mem\_page not in future\_sequence:

replaced\_page = mem\_page

break

distance = future\_sequence.index(mem\_page) - future\_sequence.index(page)

if distance > max\_distance:

max\_distance = distance

replaced\_page = mem\_page

self.memory[self.memory.index(replaced\_page)] = page

self.history.append(f"置换: 移除页{replaced\_page}, 加入页{page}")

else:

self.memory[self.memory.index(-1)] = page

self.total\_time += access\_time

return is\_page\_fault, replaced\_page, access\_time

## 4.3 基本功能实现

4.3.1 参数设定

（1）滑块控件（Slider）

用户可以通过拖动滑块在指定范围内选择值，用于控制速度：

<div class="speed-control">

<label>速度:</label>

<input type="range" class="speed-slider" id="speedSlider" min="1" max="10" value="5">

<span id="speedValue">5x</span></div>

JavaScript 部分监听滑块变化事件，并更新模拟速度：

const speedSlider = document.getElementById('speedSlider');

speedSlider.addEventListener('input', function() {

const speed = parseInt(this.value);

document.getElementById('speedValue').textContent = `${speed}x`;

// 根据速度调整模拟延迟时间

simulationDelay = 1000 / speed; // 例如，速度为5时，延迟200ms});

（2）输入框（Input）

输入框允许用户直接输入精确的参数值。例如，用户可以指定内存块数量：

<label for="frameCount">内存块数量:</label>

<input type="number" id="frameCount" min="1" max="10" value="3">

JavaScript 部分获取输入值并应用到模拟中：

const frameCount = parseInt(document.getElementById('frameCount').value);

（3）下拉菜单（Select）

下拉菜单提供预定义的参数选项。例如，选择不同的页面置换算法：

<label for="algorithmSelect">选择算法:</label><select id="algorithmSelect">

<option value="fifo">FIFO</option>

<option value="lru">LRU</option>

<option value="opt">OPT</option>

<option value="lfu">LFU</option></select>

（4）高级参数面板

对于复杂系统，参数被分组到不同面板中：

<div class="parameters-panel">

<div class="panel-group">

<h3>模拟参数</h3>

<div class="parameter-row">

<label>速度:</label>

<input type="range" ...>

</div>

<div class="parameter-row">

<label>内存块数量:</label>

<input type="number" ...>

</div>

</div>

<div class="panel-group">

<h3>序列参数</h3>

<div class="parameter-row">

<label>序列长度:</label>

<input type="number" ...>

</div>

<div class="parameter-row">

<label>最大页面号:</label>

<input type="number" ...>

</div>

</div></div>

4.3.2 手动输入序列

手动输入序列允许用户自行指定页面访问序列。在HTML中，通过一个输入框来实现：

<input type="text" class="sequence-input" id="pageSequence"

placeholder="输入页面访问序列，用逗号或空格分隔 (如: 4,1,5,6,6,3,5,6,4,5,2,4,2,3,2,4,2,1,4,3)"

value="4,1,5,6,6,3,5,6,4,5,2,4,2,3,2,4,2,1,4,3">

在 JavaScript 中，通过获取输入框的值，并将其转换为数组来使用：

const pageSequenceInput = document.getElementById('pageSequence');

const pageSequence = pageSequenceInput.value.split(/[,\s]+/).map(Number);

4.3.3 随机生成序列

随机生成序列功能可以通过点击按钮触发，调用JavaScript函数来实现。

<button class="btn btn-secondary" onclick="generateRandom()">随机序列</button>

function generateRandom() {

const sequenceLength = 20; // 序列长度

const maxPageNumber = 10; // 最大页面编号

const sequence = [];

for (let i = 0; i < sequenceLength; i++) {

sequence.push(Math.floor(Math.random() \* maxPageNumber) + 1);

}

const pageSequenceInput = document.getElementById('pageSequence');

pageSequenceInput.value = sequence.join(',');}

4.3.4 线程并行模拟

在JavaScript中，由于其单线程的特性，通常使用异步操作（如 setTimeout 或 Promise）来模拟并行执行。在本项目中，通过 setTimeout 来控制模拟的执行：

function runAlgorithm(algo) {

const speed = parseInt(document.getElementById('speedSlider').value);

const delay = 1000 / speed; // 根据速度计算延迟时间

const timeout = setTimeout(() => {

// 执行算法逻辑

if (algorithms[algo].step < pageSequence.length) {

// 继续执行下一步

runAlgorithm(algo);

}

}, delay);

timeouts.push(timeout);}

4.3.5 暂停/继续/停止

暂停/继续/停止功能通过点击相应的按钮触发，调用JavaScript函数来实现。

<button class="btn" id="startBtn" onclick="startSimulation()">开始模拟</button>

<button class="btn btn-secondary" id="pauseBtn" onclick="pauseSimulation()" disabled>暂停</button>

<button class="btn btn-danger" id="stopBtn" onclick="stopSimulation()" disabled>停止</button>

let isPaused = false;let isRunning = false;let timeouts = [];

function pauseSimulation() {

isPaused = !isPaused;

const pauseBtn = document.getElementById('pauseBtn');

if (isPaused) {

pauseBtn.textContent = '继续';

// 清除所有定时器

timeouts.forEach(timeout => clearTimeout(timeout));

timeouts = [];

['fifo', 'lru', 'opt', 'lfu'].forEach(algo => {

updateStatus(algo, 'paused');

});

} else {

pauseBtn.textContent = '暂停';

// 重新启动所有算法

['fifo', 'lru', 'opt', 'lfu'].forEach(algo => {

if (algorithms[algo].step < pageSequence.length) {

updateStatus(algo, 'running');

runAlgorithm(algo);

}

});

}}

async function stopSimulation() {

isRunning = false;

isPaused = false;

// 清除所有定时器

timeouts.forEach(timeout => clearTimeout(timeout));

timeouts = [];

// 更新UI状态

updateControlButtons();

hideProgress();

// 重置所有算法

await resetAllAlgorithms();

['fifo', 'lru', 'opt', 'lfu'].forEach(algo => {

updateStatus(algo, 'waiting');

});}

4.3.6 快表模式切换

用户通过前端复选框启用/禁用快表；切换时重置所有算法状态：

// 设置快表事件监听

document.getElementById('useQuickTable').addEventListener('change', function() {

useQuickTable = this.checked;

// 显示/隐藏所有快表区域

document.querySelectorAll('.quick-table-section').forEach(section => {

section.classList.toggle('show', useQuickTable);

});

// 重置所有算法

if (!isRunning) {

resetAllAlgorithms();

} else {

// 如果正在运行，重置并继续

resetAllAlgorithms().then(() => {

if (isRunning && !isPaused) {

currentStep = 0;

startSimulation();

}

});

}});

后端在算法实例创建时接收快表状态；前端根据状态显示/隐藏快表区域：

@app.route('/process\_page', methods=['POST'])

def process\_page():

try:

data = request.json

use\_quick\_table = data['use\_quick\_table'] # 获取快表状态

# 检查快表状态是否变化

current\_algorithm = algorithms[algorithm\_name]

if current\_algorithm is not None and current\_algorithm.use\_quick\_table != use\_quick\_table:

logger.debug(f"Resetting {algorithm\_name} due to TLB change")

algorithms[algorithm\_name] = None

# 创建新实例时传入快表状态

if algorithms[algorithm\_name] is None:

if algorithm\_name == 'fifo':

algorithms[algorithm\_name] = FIFO(memory\_size, use\_quick\_table)

# ...其他算法

4.3.7 运行结果展示

实时显示内存状态、快表状态和性能指标；详细记录每一步操作历史；结束时显示算法比较结果：

// 更新内存可视化

function updateMemoryVisualization(algoName, replacedPage = null) {

const container = document.getElementById(algoName + 'Memory');

container.innerHTML = '';

// 创建内存块

for (let i = 0; i < memorySize; i++) {

const block = document.createElement('div');

block.className = 'memory-block';

if (i < algorithms[algoName].memory.length) {

block.textContent = algorithms[algoName].memory[i];

block.classList.add('filled');

// 添加动画效果

if (replacedPage && algorithms[algoName].memory[i] === replacedPage) {

block.classList.add('replaced');

} else if (i === algorithms[algoName].memory.length - 1) {

block.classList.add('new-page');

}

}

container.appendChild(block);

}

}

// 更新快表可视化

function updateQuickTableVisualization(algoName) {

if (!useQuickTable) return;

const container = document.getElementById(algoName + 'QuickTable');

container.innerHTML = '';

// 创建快表块

for (let i = 0; i < quickTableSize; i++) {

const block = document.createElement('div');

block.className = 'quick-block';

if (i < algorithms[algoName].quickTable.length) {

block.textContent = algorithms[algoName].quickTable[i];

block.classList.add('filled');

}

container.appendChild(block);

}

}

// 更新历史记录

function updateHistoryTable(algoName, history) {

const tbody = document.getElementById(algoName + 'HistoryTable');

tbody.innerHTML = '';

history.forEach(entry => {

const row = document.createElement('tr');

let memoryContent = entry.memory?.join(', ') || '-';

let quickTableContent = useQuickTable && entry.quick\_table?.length > 0 ?

entry.quick\_table.join(', ') : '-';

row.innerHTML = `

<td>${entry.page}</td>

<td>${memoryContent}</td>

<td>${quickTableContent}</td>

<td>${getReplacementText(entry)}</td>

<td>${entry.time}ns</td>

`;

tbody.appendChild(row);

});

}

// 获取替换描述文本

function getReplacementText(entry) {

switch(entry.type) {

case 'quick\_table\_hit': return '快表命中';

case 'memory\_hit': return '内存命中';

case 'page\_fault\_new': return `载入页面 ${entry.page}`;

case 'page\_fault\_replace': return `${entry.page} 替换 ${entry.replaced\_page}`;

default: return '-';

}

}

// 显示算法比较结果

function showComparison() {

const tbody = document.getElementById('comparisonTableBody');

tbody.innerHTML = '';

['fifo', 'lru', 'opt', 'lfu'].forEach(algo => {

const alg = algorithms[algo];

const row = document.createElement('tr');

row.innerHTML = `

<td>${algo.toUpperCase()}</td>

<td>${alg.faults}</td>

<td>${(alg.faults / alg.step \* 100).toFixed(2)}%</td>

<td>${(alg.time / 1000).toFixed(1)}μs</td>

<td>${(alg.time / alg.step).toFixed(1)}ns</td>

`;

tbody.appendChild(row);

});

document.getElementById('comparisonSection').style.display = 'block';

}

4.3.8 过程结果保存

收集所有算法状态和操作历史；格式化为易读文本；含时间设置、内存状态、快表状态等完整信息使用Blob API创建下载文件：

function saveResults() {

// 1. 准备数据

const now = new Date();

const timestamp = `${now.getFullYear()}${(now.getMonth()+1).toString().padStart(2,'0')}${now.getDate().toString().padStart(2,'0')}\_${now.getHours().toString().padStart(2,'0')}${now.getMinutes().toString().padStart(2,'0')}${now.getSeconds().toString().padStart(2,'0')}`;

const filename = `page\_replacement\_results\_${timestamp}.txt`;

// 2. 构建报告内容

let content = '页面置换算法模拟报告\n';

content += '='.repeat(50) + '\n\n';

// 基础信息

content += `模拟时间: ${now.toLocaleString()}\n`;

content += `内存大小: ${memorySize}\n`;

content += `快表状态: ${useQuickTable ? '启用' : '禁用'}\n`;

content += `快表大小: ${quickTableSize}\n`;

content += `页面序列: ${pageSequence.join(', ')}\n\n`;

// 时间设置

content += '时间设置:\n';

content += `- 内存访问时间: ${timeSettings.memoryAccessTime}ns\n`;

content += `- 缺页中断时间: ${timeSettings.pageFaultTime}ns\n`;

content += `- 快表访问时间: ${timeSettings.quickTableTime}ns\n\n`;

// 各算法结果

['fifo', 'lru', 'opt', 'lfu'].forEach(algo => {

const alg = algorithms[algo];

content += `${algo.toUpperCase()} 算法结果:\n`;

content += '-'.repeat(40) + '\n';

content += `缺页次数: ${alg.faults}\n`;

content += `缺页率: ${(alg.faults / alg.step \* 100).toFixed(2)}%\n`;

content += `总时间: ${(alg.time / 1000).toFixed(1)}μs\n`;

content += `平均时间: ${(alg.time / alg.step).toFixed(1)}ns\n`;

// 内存和快表最终状态

content += `\n最终内存状态: ${alg.memory.join(', ') || '空'}\n`;

if (useQuickTable) {

content += `最终快表状态: ${alg.quickTable.join(', ') || '空'}\n`;

}

// 操作历史

content += '\n操作历史:\n';

const historyTable = document.getElementById(`${algo}HistoryTable`);

const rows = historyTable.querySelectorAll('tr');

rows.forEach(row => {

const cols = row.querySelectorAll('td');

if (cols.length >= 5) {

content += `步骤 ${parseInt(document.getElementById(`${algo}StepNum`).textContent} | `;

content += `访问: ${cols[0].textContent} | `;

content += `内存: ${cols[1].textContent} | `;

content += `快表: ${cols[2].textContent} | `;

content += `操作: ${cols[3].textContent} | `;

content += `时间: ${cols[4].textContent}\n`;

}

});

content += '\n\n';

});

// 3. 创建并下载文件

const blob = new Blob([content], { type: 'text/plain;charset=utf-8' });

const link = document.createElement('a');

link.href = URL.createObjectURL(blob);

link.download = filename;

document.body.appendChild(link);

link.click();

document.body.removeChild(link);

}

第5章 运行结果

## 5.1 并行运行算法



图5-1 并行运行算法（无快表）

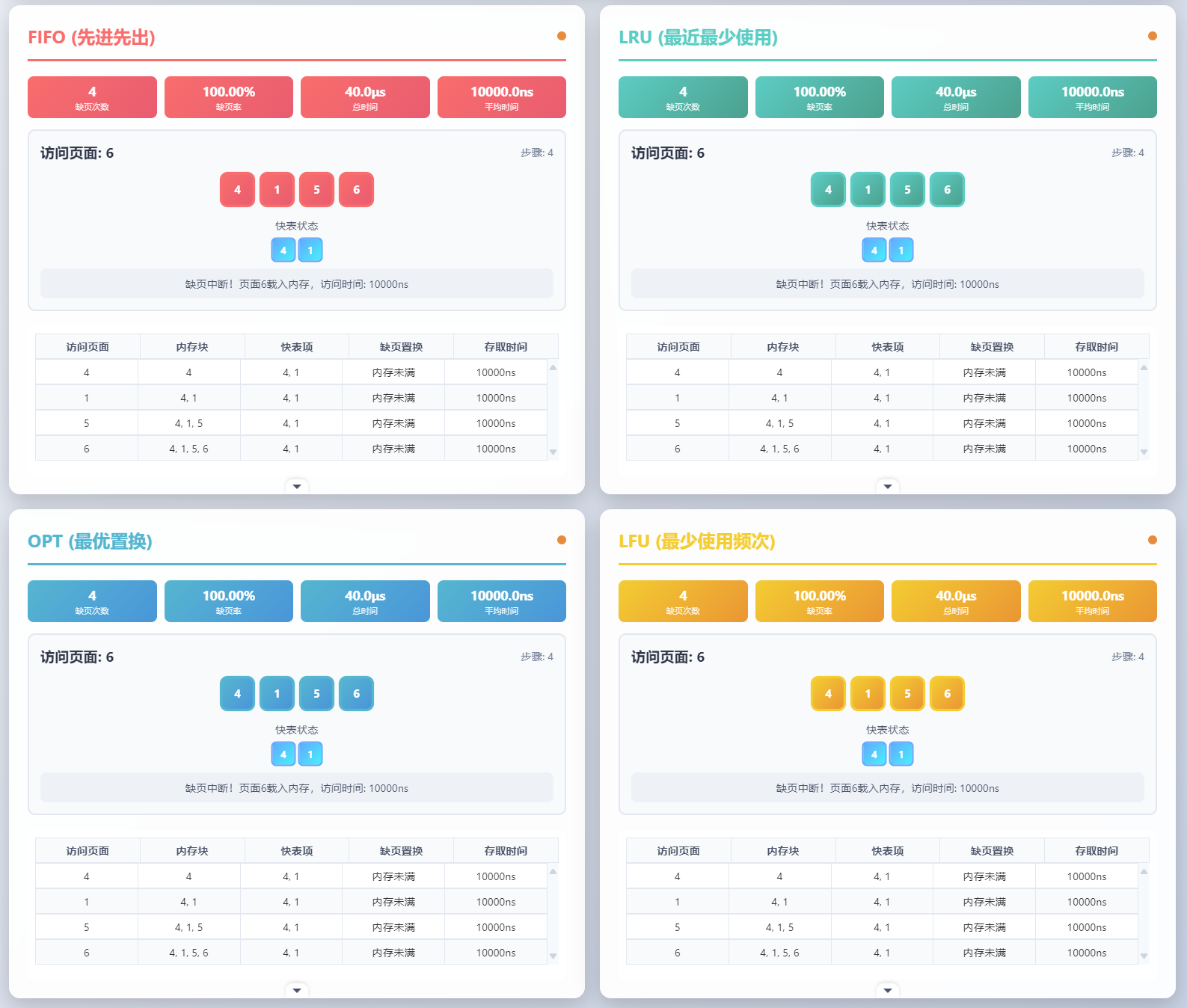


图5-2 并行运行算法（有快表）

四个算法并行运行，同时显示实时的缺页次数、缺页率、总时间、平均时间，记录访问过程，并将内存块和快表变化可视化。

总 结

本课程设计报告通过设计和实现一个模拟操作系统页面置换算法的实验平台，对各种页面置换算法进行了深入的研究和比较。实验平台支持输入和随机生成逻辑页面访问序列，并由四个线程同时完成每种算法的执行。用户可以自定义驻留内存页面的个数、内存的存取时间、缺页中断的时间和快表的时间，并提供合理的省缺值。同时，实验平台允许用户随机输入和生成存取的逻辑页面页号序列，并可设定其个数和范围。用户还可以根据需求选择是否启用快表功能。

实验平台提供了良好的图形界面，能够同时展示四种算法的运行结果，并详细列出每个页面访问对应的时间开销。用户可在系统运行过程中随时暂停或继续执行，实验结果支持存储与回溯查询，方便进行多组参数配置下的算法比较。

通过完成多次不同设置的实验，本报告总结实验数据并得出以下结论：

不同页面置换算法的性能存在显著差异。FIFO在随机访问场景下易导致高缺页率，而LRU能在具有局部性的访问模式下有效降低缺页次数。LFU在统计稳定性较强的序列中表现良好，OPT虽然理论最优，但需预知未来访问，难以实际应用。

快表的使用显著提升了访问效率，通过缓存近期访问页号减少了内存读取次数，尤其在页表较大时效果更明显。其性能依赖于命中率与缓存容量，需结合具体应用环境优化配置。

驻留内存页数、访问延迟参数等设置对算法性能具有关键影响，系统在实际部署中应依据硬件条件与应用需求进行合理调优。

综上所述，本实验平台不仅加深了对页面置换机制的理解，也为后续操作系统设计与内存优化提供了可量化、可视化的支持手段。

参考文献

[1]《计算机操作系统（第四版）》西安电子科技大学出版社

使用手册

## 1.操作步骤

①点击[高级设置]按钮和参数输入框设置参数。

②点击[随机序列]按钮或手动输入生成序列。

③点击[开始模拟]按钮各线程开始运行。

④点击[暂停]按钮各线程暂停运行。

⑤点击[继续]按钮暂停的线程可以继续执行。

⑥点击[停止]按钮线程全部结束。

⑦点击[保存结果]按钮可以将运行完整的过程和结果保存到文件中。

## 2.基本功能

通过演示，可以更直观，深入的明确操作系统中进程、线程的概念，可以将课本上P、V操作的思想运用到实践当中，提高动手能力。

本程序有五个基本的功能：参数设置、开始模拟、暂停\继续、停止、保存过程和结果、结果对比总结。

## 2.1 输入设置



图1 输入设置

可以自由更改内存块数，快表块数，是否使用快表。可以选择随机序列或者手动输入序列。

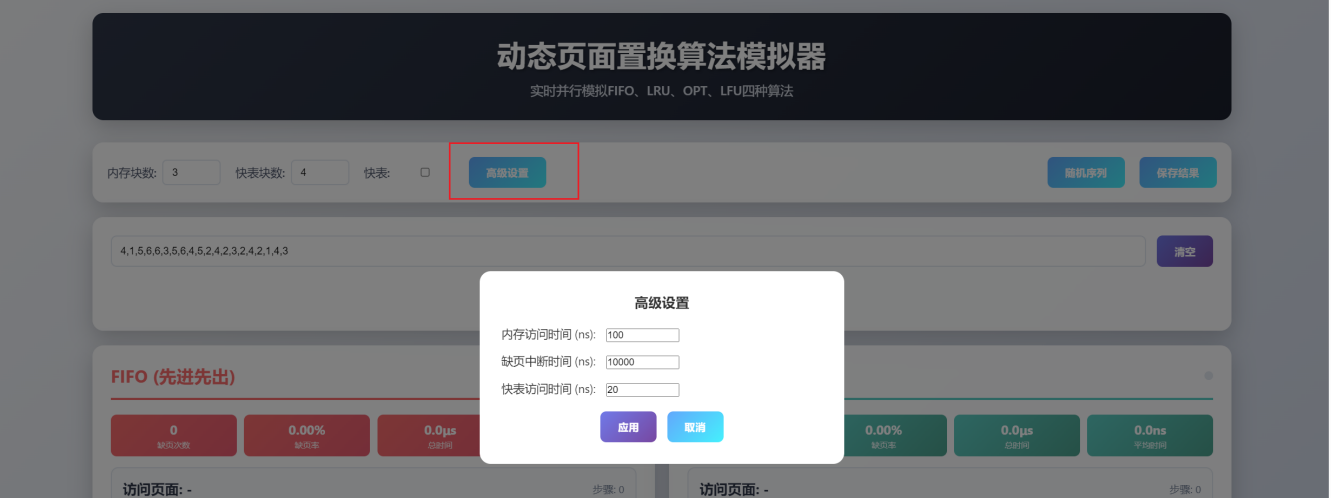


图2 高级设置

可以点击高级设置，设置内存访问时间，缺页中断时间，快表访问时间。

## 2.2 开始/暂停/继续/停止



图3 开始模拟

点击开始模拟按钮，运行算法。

图4 算法运行



图5 暂停

点击暂停按钮，随时暂停进程，且同时暂停按钮变成继续按钮。



图6 继续

点击继续按钮，继续进程，且同时继续按钮变成暂停按钮。



图7 停止

点击停止按钮，结束所有进程。

## 2.3 结果对比总结

图8 结果对比总结

在算法运行结束后，将在页面下方展示算法性能比较表格，并高亮性能最佳的算法。

## 2.4 结果保存

图9 结果保存

点击保存结果按钮，将会将本次算法输入、过程、结果均记录在txt文档中并自动下载。

## 3.运行环境

本程序运行需要Flask应用程序框架。