

**操作系统进程管理项目**

题目 请求调页存储管理方式模拟

姓 名 刘淑仪

学 号 2251730

学 院 软件学院

专 业 软件工程

教 师 张惠娟

二〇二四 年 五 月 二十九 日

**请求调页存储管理方式模拟**

刘淑仪 2251730 软件学院 软件工程

**一、项目简介**

**（一）项目目的**

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

**（二）项目需求**

1. 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。
2. 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。
3. 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。
4. 置换算法可以选用FIFO或者LRU算法
5. 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。

**（三）项目开发环境**

系统：Windows 11 家庭中文版

IDE：WebStorm 2024.1

语言：Vue + Node.js

**二、请求调页系统的原理和实现过程**

请求调页系统（Demand Paging System）是一种内存管理技术，用于在虚拟内存系统中高效地管理物理内存和虚拟内存之间的映射。其核心思想是只在需要时（即请求时）将页面从磁盘加载到内存，而不是一次性将整个程序加载到内存。

·请求调页系统的原理如下：

1. 虚拟内存和物理内存：虚拟内存是由操作系统提供的一个抽象概念，给每个进程提供一个连续的内存空间，但实际使用的物理内存往往是不连续且有限的。
2. 页面（Page）和页框（Frame）：虚拟内存和物理内存都被划分成固定大小的块，称为页面和页框。页面是虚拟内存中的块，页框是物理内存中的块。
3. 页表（Page Table）：每个进程有一个页表，记录虚拟页面和物理页框的映射关系。当进程需要访问一个虚拟地址时，通过页表查找相应的物理地址。
4. 缺页中断（Page Fault）：如果进程访问的虚拟页面不在物理内存中，会触发缺页中断。操作系统会捕获这个中断并加载相应的页面到物理内存中。

·请求调页系统的实现过程如下：

* 1. 初始化：操作系统初始化虚拟内存和物理内存的管理结构（例如页表、空闲页框列表等）。
  2. 页面请求：当进程访问一个虚拟地址时，处理器会将该地址分解成页号和页内偏移量，然后通过页表查找对应的物理页框。
  3. 缺页处理：
  4. 如果页表中没有找到对应的物理页框，触发缺页中断。
  5. 操作系统的缺页处理程序会进行以下步骤：

1. 查找空闲页框：从空闲页框列表中查找一个空闲页框。如果没有空闲页框，需要使用页面置换算法（例如LRU、FIFO等）腾出一个页框。
2. 从磁盘加载页面：将缺页的虚拟页面从磁盘加载到选定的物理页框中。
3. 更新页表：更新页表，记录新的虚拟页面和物理页框的映射关系。
4. 恢复进程：将进程的状态恢复到缺页中断发生时的状态，并重新执行导致缺页的指令。
   1. 页面置换：
   2. 当物理内存满了时，需要选择一个已经在内存中的页面进行置换，以腾出空间给新页面。本程序所采用的页面置换算法包括：
5. LRU（Least Recently Used）：置换最近最少使用的页面。
6. FIFO（First In First Out）：置换最早进入内存的页面。
   1. 继续执行：页面成功加载后，进程可以继续执行，访问刚刚加载到内存中的页面。

**三、项目架构**

本项目是基于请求调页（Demand Paging）算法的内存管理模拟系统，使用Vue.js和Element UI进行前端开发。

**（一） 前端架构**

1. 组件结构：

1. 模板部分（<template>）：包含了整个页面的布局和显示内容，包括标题、图像链接、作者信息、分割线、基本介绍、内存中的页面图示、执行按钮和已执行指令列表。
2. 脚本部分（<script>）：主要负责数据处理、页面交互和逻辑控制。定义了组件的名称、属性、数据、观察者和方法。
3. 样式部分（<style>）：用于定义页面各部分的样式，包括按钮样式、文本样式和其他UI元素的样式。

2. 主要功能：

1. 基本介绍：显示项目的基本信息，例如作业指令总数、每页存放指令数、作业占用内存页数、页面置换算法、下一条指令地址、缺页数、缺页率和执行速度。
2. 内存中的页面图示：动态显示当前内存中的页面内容和状态，使用Element UI的el-container和el-card组件。
3. 执行按钮：包括单步执行、连续执行和复位功能。按钮点击后触发相应的方法进行操作。
4. 已执行指令：使用el-table组件显示已经执行的指令信息，包括序号、地址、是否缺页、换出页和换入页。

**（二） 数据结构和方法**

1. 数据结构：

1. miss\_page\_num：记录缺页次数。
2. miss\_page\_rate：记录缺页率。
3. page\_algorithm：选择的页面置换算法（FIFO或LRU）。
4. frame：模拟的内存帧，包含页面编号和指令列表。
5. table\_data：记录已执行指令的信息。
6. is\_disabled：控制按钮的禁用状态。
7. page\_data：预先生成的页面数据。
8. next\_address：下一条指令地址。
9. pre\_address：前一条指令地址。
10. fifo\_queue：FIFO算法的队列。
11. lru\_queue：LRU算法的时间戳。
12. interval：控制连续执行的定时器。
13. frame\_style：控制内存帧的样式。
14. order\_style：控制指令样式。
15. current\_row：记录当前执行的行。
16. timePerExecution：控制连续执行的速度。

2. 方法：

1. single\_step：单步执行一条指令。
2. executable：连续执行指令，使用定时器控制。
3. init：复位所有数据和状态。
4. FIFO：实现FIFO页面置换算法。
5. LRU：实现LRU页面置换算法。
6. exec：执行一次指令，处理页面置换并更新数据。
7. getNextAddress：生成下一条指令地址。
8. setCurrent：设置当前执行的行高亮。

**（三） 工作流程**

1. 初始化：created钩子函数在组件创建时调用，初始化页表和第一条指令地址。
2. 指令执行：单步执行时调用single\_step方法，连续执行时调用executable方法，这些方法会调用exec方法执行具体的指令操作。
3. exec方法检查当前指令是否在内存中，如果不在，触发页面置换算法（FIFO或LRU），更新内存帧和已执行指令列表。
4. 页面置换：根据选择的置换算法（FIFO或LRU），调用对应的方法（FIFO或LRU），替换内存中的页面并更新相关数据。
5. 显示更新：每次执行指令后，更新内存帧的显示样式和已执行指令列表。

**四、界面设计**

**（一） 标题界面**

1. 学校和学院标识：界面左上角显示了“同济大学软件学院”的标识，这是一个图片链接，点进去可以进入同济大学软件学院官网。
2. 项目标题：界面中间部分显示了项目的标题：“操作系统 - 内存管理项目 - 请求调页存储管理方式模拟”。标题简洁明了，明确了项目的主题和内容。
3. 创作者信息：界面右上角显示了创作者的信息，包括学号“2251730”和姓名“刘淑仪”。此外，还提供了一个链接到“GitHub主页”的按钮，方便用户查看或管理自己的代码仓库。

**（二） 基本介绍界面**

1. 标题：“基本介绍”以较大的字体位于界面顶部，明确了这个部分的内容。
2. 作业指令总数：显示为320，表示这个模拟项目将处理的总指令数量。
3. 每页存放指令数：显示为10，表示每一页内存能够存放的指令数量。
4. 作业占用内存页数：显示为4，表示这个作业在内存中占用的页数。
5. 页面置换算法：提供两种选择：FIFO算法和LRU算法。当前选择的是FIFO算法（以红色背景突出显示），用户可以点击切换到LRU算法。
6. 下一条指令地址：显示为71，表示即将执行的下一条指令所在的地址。
7. 缺页数：显示为0，表示当前模拟过程中发生的缺页次数。
8. 缺页率：显示为0%，表示当前缺页次数相对于总指令数的比例。
9. 执行速度：默认显示为100毫秒，表示指令执行的速度。用户可以调整这个值来改变模拟执行的速度。

**（三） 内存中的页面显示界面**

1. 标题：界面顶部显示“内存中的页面图示”，这是对整个界面的说明。
2. 页框：界面中间有四个框，每个框显示一个页面的内容。这些页框由标题和内容组成：
   1. 标题，每个框的顶部显示页面号，如“第14页”、“第0页”、“第25页”和“第17页”。在初始默认的界面中，标题显示为“第None页”，表示没有页面内容。
   2. 内容，每个框内显示页面的具体内容，如第14页的内容从140到149，第0页的内容从0到9，第25页的内容从250到259，第17页的内容从170到179。当前页标红，当前地址标紫。
3. 按钮：界面底部有三个按钮：
   1. 单步执行：执行一步操作，可能用于内存页面的模拟或演示。
   2. 连续执行：连续执行操作，用于连续展示内存页面的变化。再次点击停止连续执行，在此之前单步执行和复位按钮被禁用。
   3. 复位：重置操作，可能用于恢复初始状态或清空内存页面的显示。

**（四） 已执行指令概览界面**

1. 标题：面板的顶部有“已执行指令”的标题，说明这个面板用于显示已经执行过的指令。
2. 表头：面板中部有一个表格的表头，包含以下几列：
   * + 1. 序号：指令的序号，用于标识每一条指令。
       2. 地址：指令执行的地址。
       3. 是否缺页：指示该指令是否触发了缺页中断。
       4. 换出页：如果指令触发了页面置换，显示被换出的页面。
       5. 换入页：如果指令触发了页面置换，显示被换入的页面。
3. 内容：表格内容区域目前显示“No Data”，表示当前没有已执行的指令记录。有记录时，使用斑马格增强美观性。

**（五）** **整体界面预览**



图1 整体界面预览

**五、核心算法**

这个项目的核心算法是内存管理中的页面置换算法，具体实现了两种常用的页面置换策略：FIFO（First In, First Out）和LRU（Least Recently Used）。

**（一） FIFO（First In, First Out）**

1. 工作原理：FIFO页面置换算法通过维护一个队列，按照页面进入内存的顺序进行置换。最先进入内存的页面最先被置换出去。

2. 实现分析：

1. 通过this.fifo\_queue维护一个FIFO队列。
2. 获取队列中最老的帧编号，即队列首元素。
3. 将这个帧编号从队列中移除，并重新加入队尾。
4. 更新LRU队列中对应帧的时间戳。
5. 返回被替换的帧编号和其存储的页面编号。

3. 源代码：

FIFO(){

// 获取队列中最老的帧编号

let oldestFrameNum = this.fifo\_queue[0];

// 移除队列首元素，并将其放回队尾

this.fifo\_queue.shift();

this.fifo\_queue.push(oldestFrameNum);

// 同时更新LRU队列中对应帧的时间戳

this.lru\_queue[oldestFrameNum] = new Date().getTime();

// 返回被替换帧的编号和其存储的页面编号

return [oldestFrameNum, this.frame[oldestFrameNum].num];

}

**（二） LRU（Least Recently Used）**

1. 工作原理：LRU页面置换算法通过记录每个页面最近一次使用的时间，当需要进行置换时，选择最近最久未使用的页面进行置换。

2. 实现分析：

1. 通过this.lru\_queue记录每个帧的最近使用时间戳。
2. 初始化最久未使用帧编号为第一个帧编号。
3. 遍历this.lru\_queue，找出时间戳最小的帧编号，即最久未使用的帧。
4. 更新这个帧的时间戳为当前时间。
5. 在FIFO队列中，将这个帧从当前位置移除，并加到队尾。
6. 返回被替换的帧编号和其存储的页面编号。

3. 源代码：

LRU(){

// 初始化为第一个帧编号

let leastRecentlyUsedFrameNum = 0;

// 找出最久未使用的帧

for (let i = 1; i < this.lru\_queue.length; i++) {

if (this.lru\_queue[i] < this.lru\_queue[leastRecentlyUsedFrameNum]) {

leastRecentlyUsedFrameNum = i;

}

}

// 更新这个帧的时间戳为当前时间

this.lru\_queue[leastRecentlyUsedFrameNum] = new Date().getTime();

// 同时更新FIFO队列，将这个帧从当前位置移除，并加到队尾

this.fifo\_queue.splice(this.fifo\_queue.indexOf(leastRecentlyUsedFrameNum), 1);

this.fifo\_queue.push(leastRecentlyUsedFrameNum);

// 返回被替换帧的编号和其存储的页面编号

return [leastRecentlyUsedFrameNum, this.frame[leastRecentlyUsedFrameNum].num];

}

**（三）** **页面置换的执行过程**

1. 实现分析：

1. 计算当前指令的页面编号currentPage。
2. 在内存帧中搜索当前页面，更新LRU队列中对应帧的时间戳。
3. 如果页面未找到，增加缺页次数，计算缺页率，调用页面置换算法（FIFO或LRU），更新内存帧和页面置换信息。
4. 如果页面找到，记录访问信息。
5. 更新当前指令和下一条指令的地址。

2. 源代码：

exec() {

if (this.table\_data.length === 320) {

// 当指令数量达到320时，停止执行并清理定时器

this.s\_exec\_name = '连续执行';

this.is\_disabled = false;

clearInterval(this.interval);

} else {

// 恢复上一条指令的样式

this.frame\_style = ['', '', '', ''];

this.order\_style[this.pre\_address] = '';

// 计算当前指令的页面编号

let currentPage = Math.floor(this.next\_address / 10);

let pageFound = false; // 标记页面是否在内存中找到

// 在内存帧中搜索当前页面

for (let i = 0; i < this.frame.length; i++) {

if (currentPage === this.frame[i].num) {

this.lru\_queue[i] = new Date().getTime(); // 更新页面的最近使用时间

pageFound = true;

break;

}

}

// 高亮显示当前指令

this.order\_style[this.next\_address] = { background: 'Orchid' };

if (!pageFound) {

// 页面未找到，处理页面置换

this.miss\_page\_num++;

this.miss\_page\_rate = Math.floor(this.miss\_page\_num \* 100 / (this.table\_data.length + 1));

let replacementResult = this.page\_algorithm === 'FIFO算法' ? this.FIFO() : this.LRU();

this.frame\_style[replacementResult[0]] = { background: '#FF8DACFF' };

this.frame[replacementResult[0]].num = currentPage;

this.frame[replacementResult[0]].list = this.page\_data[currentPage];

// 记录页面置换信息

this.table\_data.unshift({

order: this.table\_data.length,

address: this.next\_address,

loss\_page: 'Yes',

out\_page: replacementResult[1] ? replacementResult[1] : '',

in\_page: currentPage

});

} else {

// 页面找到，记录访问信息

this.table\_data.unshift({

order: this.table\_data.length,

address: this.next\_address,

loss\_page: 'No',

out\_page: '',

in\_page: ''

});

}

// 更新当前和下一条指令地址

this.setCurrent();

this.getNextAddress();

}

}

**六、项目整体运行测试**

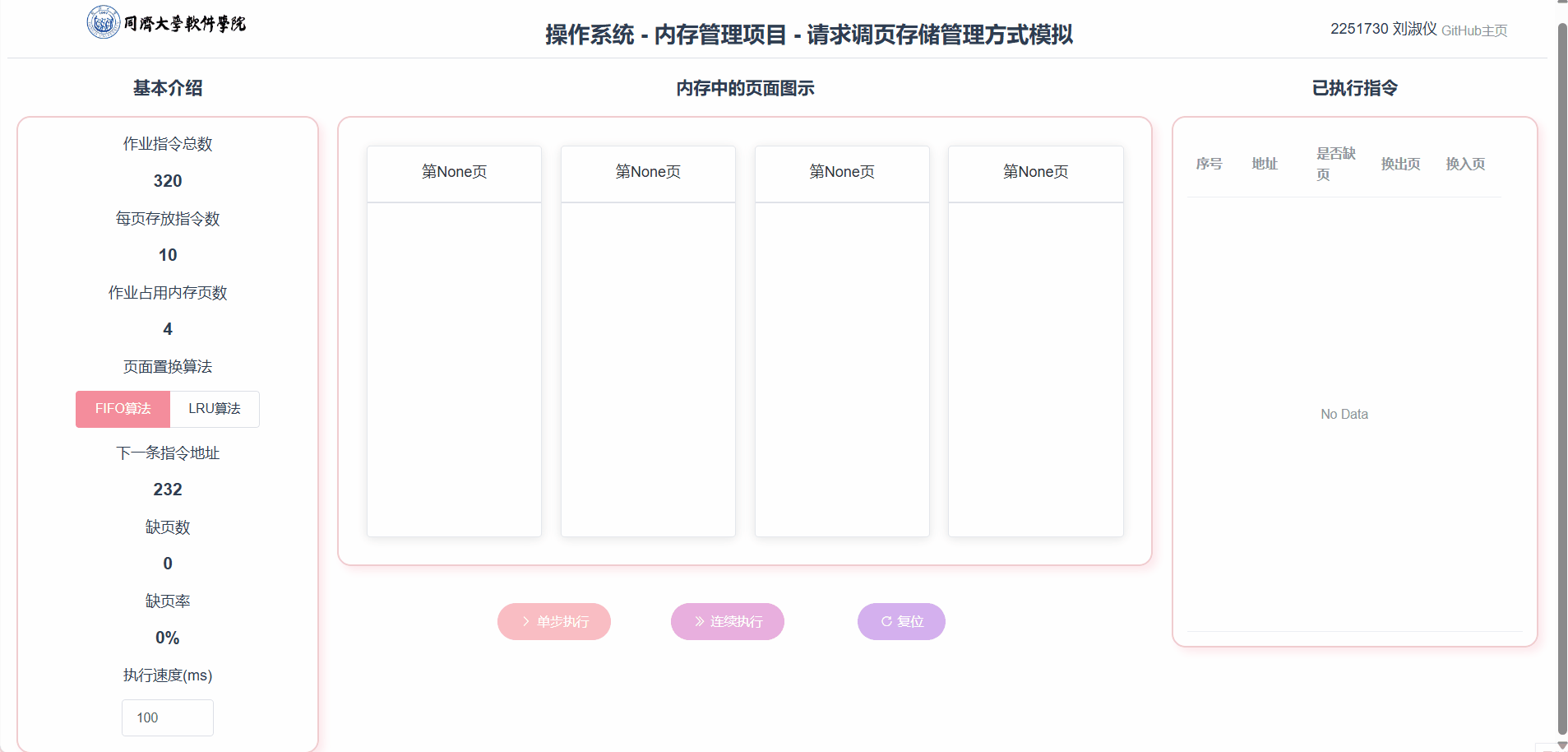


图2 整体运行测试图示例图

**七、项目总结**

**（一）项目改进建议**

1. 算法扩展：可以增加其他页面置换算法（如OPT (Optimal Page Replacement)、CLOCK等）以提供更全面的学习内容。这将帮助用户了解更多种类的页面置换策略及其优缺点，从而更深入地理解内存管理机制。
2. 用户交互：增加用户自定义参数的功能，例如允许用户设置内存大小、页面大小、指令数等。这将使项目更具灵活性和可操作性，用户可以根据不同的需求进行模拟和实验，提升学习效果。
3. 性能优化：针对大规模指令集和内存配置，进一步优化算法和数据结构，提高模拟效率。例如，通过改进数据结构以减少时间复杂度，优化内存访问以提升速度，确保项目在处理更复杂的情况时仍能保持良好的性能。

**（二）总结**

该项目通过实现和可视化操作系统中的内存管理和页面置换算法，为用户提供了一种高效且直观的学习工具。项目不仅动态展示了页面置换的过程，还通过详细的统计数据帮助用户深入理解算法的性能和效果。未来，通过扩展算法种类和增加用户交互功能，可以进一步提升项目的教育价值和用户体验，使其成为一个更全面、更灵活的学习平台。

**八、项目运行**

本地运行：

npm install

npm run serve

Local : http://localhost:8080/

Network : unavailable