请求调页存储管理方式模拟设计方案

内存管理之项目二

|  |  |
| --- | --- |
| 姓名 | 学号 |
| 王蔚达 | 2151300 |

⼀ 、项目简介

1. 项目内容

本项目是⼀个模拟请求分页分配的系统 ， 目的是为了加深理解请求调页系统的原理和实现过程。在这个 模拟系统中 ，我们模拟⼀个含有320条指令的作业执行过程 ，通过特定的算法模拟指令的执行顺序 ，并 采用FIFO和LRU来处理处理可能出现的页面缺页和置换问题。

2. 开发环境

 系统 ：Windows 11 家庭中文版

 IDE ：Visual Studio Code

 HTML 版本 ：HTML5

 CSS 版本 ：CSS3

 JavaScript 版本 ：ES6

二、系统架构

请求分页分配模拟系统采用前端技术实现 ，包括HTML、CSS和JavaScript ，用于展示和模拟请求调页系统 的执行过程。系统的架构主要包括以下几个组件 ：

1. 前端界面 ：使用HTML和CSS构建用户界面 ，提供交互元素和可视化展示。用户可以通过点击按钮来执 行模拟过程 ，查看缺页次数和缺页率等结果。

2. **JavaScript**逻辑 ：通过JavaScript编写的逻辑代码 ，实现了模拟系统的核心功能。它包括生成符合要求 的指令序列、执行页面分配、地址转换和页面置换等操作 ，并更新界面上的数据和状态。

3. 数据交互 ：使用JavaScript与HTML页面进行数据交互 ，将模拟过程中的结果实时更新到界面上。通过 DOM操作 ，动态添加和修改表格中的数据 ，显示缺页次数、缺页率等信息。

4. **CSS**样式 ：采用CSS样式设计 ，为界面提供美观的外观和交互效果。通过定义样式类、选择器和样式 属性 ，调整页面元素的布局、颜色和样式。

设计的两个数据结构如下 ：

 Page ：表示进程的页面 ，其包括id (页号) ，instructions (包含的指令 ，10条)

 Memory ：表示内存 ，其包括method ( 置换方法) 、frames (页框数组 ，4个) ，lastUsedTime ( ⼀个数

组 ，记录各个页框最近被访问的时间)

三、核心算法

核心算法部分包括指令访问序列的生成以及置换算法 ( FIFO与LRU) 。

**3.1** 指令访问序列生成算法

其实现方法与作业中提供的相同 ，即①在0~319之间随机取⼀个起始指令m ；②顺序执行m+1 ；③通过随机 数跳转到m前地址部分*m*1 并执行该指令 ；④顺序执行指令*m*1+1 ；⑤通过随机数 ，跳转到m的后地址部分 *m*1 并执行该指令 ；⑥顺序执行指令*m*2+1 ；重复上面的步骤直到产生320条指令。

//生成符合题目要求的指令序列

}

function

function generateInstructionSequence() {

let instructions = Array .from({ length : 320 }, (\_, i) => i);

let executionOrder = []; // 用于存储执行顺序的数组

let count = 0; // 记录执行的次数

while (count < 320) {

let m = Math .floor(Math .random() \* 320);

executionOrder .push(m);

count++;

// 如果存在编号为m+1的指令 ，则执行这个指令

if (m + 1 < 320) {

executionOrder .push(m + 1);

count++;

}

// 如果m大于0，即存在可以跳转的前地址部分的指令

if (m > 0 && count < 320) {

// 在0到m-1之间随机选择⼀个指令

let m1 = Math .floor(Math .random() \* m);

executionOrder .push(m1);

count++;

// 如果存在编号为m1+1的指令 ，则执行这个指令

if (m1 + 1 < m) {

executionOrder .push(m1 + 1);

count++;

}

}

// 如果存在可以跳转的后地址部分的指令

if (m + 2 < 320 && count < 320) {

// 在m+2到319之间随机选择⼀个指令

let m2 = m + 2 + Math .floor(Math .random() \* (319 - m - 1));

executionOrder .push(m2);

count++;

// 如果存在编号为m2+1的指令 ，则执行这个指令

if (m2 + 1 < 320) {

executionOrder .push(m2 + 1);

count++;

}

}

}

//只需要320条指令

if (executionOrder .length > 320) { executionOrder .pop(); }

// 返回执行顺序数组

return executionOrder;

**3.2** 置换算法 **-** **FIFO**

FIFO算法 ( First-In-First-Out ，先进先出算法) 是⼀种简单的页面置换算法 ，它基于先进入内存的页面先被 替换的原则。该算法维护⼀个页面队列 ，当需要置换页面时 ，选择队列中最早进入的页面进行替换。在这 里使用index来指示该发生置换的页框。

fifoSimulation() {

clearMemoryTable(); // 清空memory\_table内容

let pages = Array .from({ length : 32 }, (\_, i) => new Page(i));//32页

let memory = new Memory(4);

memory .setMethod( 'FIFO ')

let instructions = generateInstructionSequence();

let missingPages = 0;

var count = 0;

var index = 0;

for (let i of instructions) {

count++;

let pageId = Math .floor(i / 10);

if ( !memory .hasInstruction(i)) {

//需要判断是否已经装满

let removedPage = memory .pages .length >= memory .size ? memory .pages[index] .id : null;

memory .pages[index] = pages[pageId];

missingPages++;

addInstructionToTable(count, i, memory, "是", index, removedPage);

index = (index + 1) % 4;

} else {

addInstructionToTable(count, i, memory, "否", null, null);

}

}

return missingPages / 320; // 返回缺页率

}

**3.3** 置换算法 **-** **LRU**

LRU算法 ( Least-Recently-Used ，最近最久未使用算法) 是⼀种基于页面使用频率的页面置换算法 ，它认 为最近最少使用的页面是最有可能被再次使用的 ，因此优先选择最久未使用的页面进行替换。在实现上 ， 使用Memory.lastUsedTime数组来标记每个页框的最近访问时间 ，当需要置换时 ，优先选择数组中最小的 位置。

function lruSimulation() {

clearMemoryTable(); // 清空memory\_table内容

let pages = Array .from({ length : 32 }, (\_, i) => new Page(i));

let memory = new Memory(4);

memory .setMethod( 'LRU ')

let instructions = generateInstructionSequence();

let missingPages = 0;

var count = 0;

var index = 0;

for (let i of instructions) {

count++;

let pageId = Math .floor(i / 10);

if ( !memory .hasInstruction(i, count)) {

//找到最早被访问的

index = memory .lastUsedTime .indexOf(Math .min( . . .memory .lastUsedTime));

let removedPage = memory .frames[index] === undefined ? null : memory .frames[index] .id;

memory .frames[index] = pages[pageId];

missingPages++;

addInstructionToTable(count, i, memory, "是", index, removedPage);

} else {

addInstructionToTable(count, i, memory, "否", null, null);

}

}

return missingPages / 320;

}

四、请求调页系统的原理和实现过程

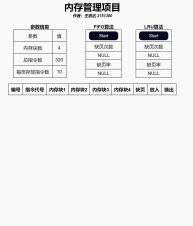
请求调页系统的原理和实现过程是指操作系统中⼀种内存管理技术。在请求调页系统中 ，内存被分为固定 大小的页框 (page frame) ，进程的地址空间被分为相同大小的页面 (page) 。当进程需要访问某个页面 时 ，如果该页面已经在内存中 ，直接访问即可 ；否则 ，需要将该页面从磁盘中读入内存。 当内存中没有足 够的空闲页框时 ，需要使用页面置换算法将某些页面置换出去 ，以便为新的页面腾出空间。请求调页系统 的实现需要解决许多问题 ，如页面分配、地址转换、页面缺页和置换等。

在本项目中 ，采用了FIFO和LRU两种页面置换算法 ，模拟了请求调页系统的执行过程。在模拟过程中 ，我 们生成了320条指令 ，通过特定的算法模拟指令的执行顺序 ，并使用FIFO和LRU算法处理可能出现的页面 缺页和置换问题。在模拟过程中 ，记录了每⼀条指令的执行情况以及内存中页框的状态 ，以便分析缺页率

和置换次数等数据。通过模拟实验 ，可以更清楚地了解请求调页系统的原理和实现过程 ，加深对操作系统 内存管理的理解。

五、用户界面设计

**5.1** 初始界面



**5.2** 实验结果展示

点击FIFO算法和LRU算法的“Start”按钮便可以开始进行实验。

实验结果的展示 ：编号 (指令序号) 、指令代号 (指令的内容) 、内存块1~4 (展示每个页框中存放的页面 页号 ，蓝色表明发生了置换或装入) 、缺页 (“是”则表示装入该指令时发生缺页 ，为“否”时整行会变成灰 色) 、放入 (该页面放入的存储块的编号) 、换出 (展示换入内存块中原有的页号)

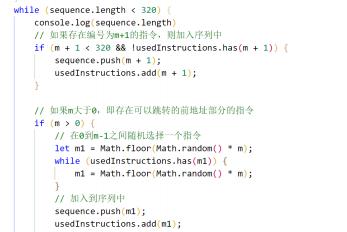


六、心得体会

**6.1** 设计过程中遇到的问题及解决办法

 在生成指令执行序列的时候出现循环卡死的情况 ，导致界面⼀直没有反应。

解决方案 ：使用控制台进行调试 ，观察过程中sequence序列的变化 ，发现问题所在while循环 ，不再使 用while判断是否找对 ，采用规定上下界的方式来处理特殊情况。改正后的代码见5.1。



**6.2** 经验

 本次实验让我更深入地了解了操作系统中请求调页存储管理方式的原理和实现过程。通过编写

JavaScript代码模拟实验过程 ，我更加清楚地了解了请求调页系统中的页面分配、页面置换、地址转换 等过程 ，同时也更加熟悉了FIFO和LRU两种页面置换算法的原理和实现方式。

学会使用chorme的“检查”中的控制台进行调试 ，使用console.log可以输出⼀些程序在执行过程中的信 息 ，进而对代码进行修正 ；同时也可以使用“调试”模式 ，看到每⼀步值的变化。

