内存管理 － 请求调页存储管理方式模拟

2152402 段婷婷

目录

1. 项目目的

- 设计页面、页表，加深对地址转换的理解
- 实现 FIFO 与 LRU 算法来置换页面
- 加深对请求调页系统的原理和实现过程的理解

2. 项目简述与功能

- 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。
- 首先**生成作业中的指令访问次序**，按照以下原则生成：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。
- **置换算法**采用 FIFO 与 LRU 算法。
- 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。

3. 项目设计

3.1 项目开发环境

- 系统：Windows 11 家庭中文版
- IDE：Visual Studio Code 1.78.2
- 语言：HTML、CSS、JavaScript

3.2 项目运行方式

- 双击 index.html 进入网页即可运行。

3.3 项目整体设计

- index.html：通过标记和元素来描述页面的结构，包括标题、段落、列表等。
- main.js：用于实现内存管理模拟的页面交互逻辑，包括生成指令访问次序、模拟请求调页管理方式、更新网页显示的内容等。
- main.css：用于控制网页样式和外观的样式。通过选择器和属性来选择页面元素，并定义其样式和布局。

4. 页面置换算法

4.1 FIFO（先进先出）算法

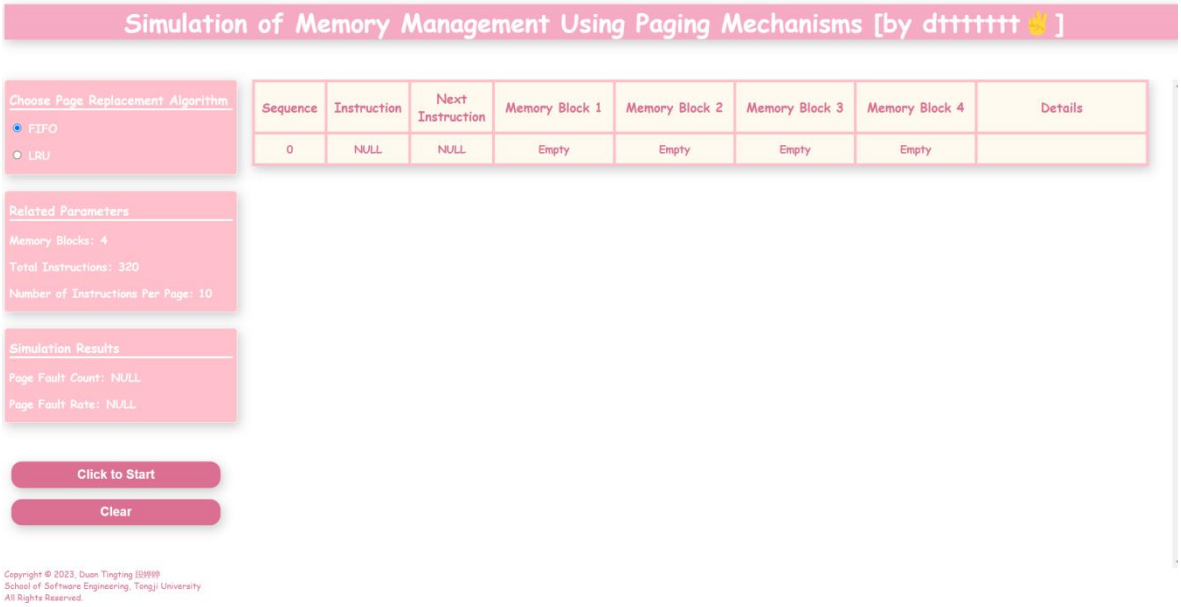
- FIFO 算法是一种简单的页面置换算法，它将最早进入内存的页面视为最先被置换出去的页面。类似于队列的先进先出原则。
- FIFO 算法的实现相对简单，适用于内存中页面访问没有明显的规律或者没有关联性的情况。然而，FIFO 算法有一个明显的缺点，就是无法区分页面的重要性和访问频率，可能会导致性能下降，尤其在长时间运行的情况下。

4.2 LRU（最近最久未使用）算法

- LRU 算法基于页面的使用历史，将最近最久未使用的页面视为最需要被置换出去的页面。它假设在最近一段时间内没有被访问的页面，在未来也不太可能被访问到。
- LRU 算法能够比较好地适应访问模式的变化，对于经常被访问的页面会保持在内存中，提高了缓存命中率和性能。然而，实现 LRU 算法需要维护访问顺序的数据结构，增加了额外的开销。

5. 项目界面

5.1 起始界面



总体来说分为三个部分：

（1）顶部：页面标题——Simulation of Management Using Paging Mechanisms，即内存的请求调页管理方式的模拟。

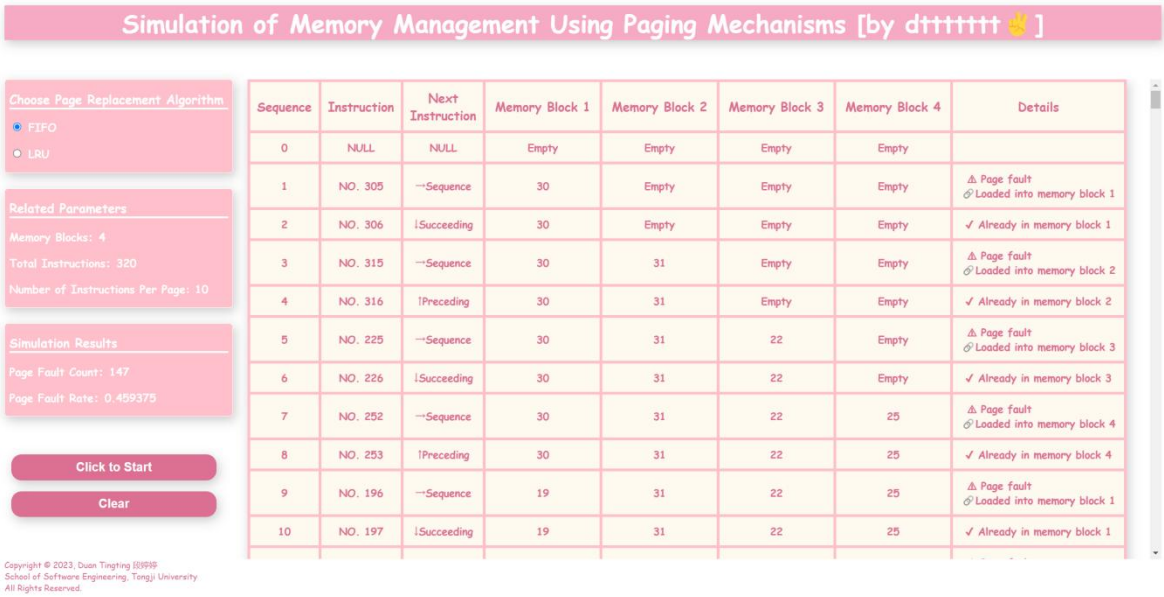
（2）左侧：

- 选择调页算法：可选 FIFO 或 LRU
- 显示模拟相关参数：内存块数目、作业指令总数目、单页指令数目
- 显示模拟结果：缺页次数、缺页率

- 开始与清空按钮：点击"Click to Start"开始模拟，点击"Clear"清空页面

（3）右侧：显示内存块中存储的页号与指令运行情况等。

5.2 点击"Click to Start"开始模拟



下面详细解释右侧表格的内容：

- **Sequence**：当前执行过的指令数目，按照设定，一共会执行 320 条指令。
- **Instruction**：当前执行的指令序号，这个序号是根据指令所在的地址标定的。
- **Next Instruction**：下一条执行指令的类型，有 Sequence（顺序执行）、Preceding（跳转至后地址）、Succeeding（跳转至前地址）三种类型。
- **Memory Block i**：描述第 i 个内存块中存储的页号，一共有 4 个内存块。
- **Details**：当前指令的调入情况——若已在内存中，则显示所在的内存块号；若不在内存中，则调入内存，显示调入块号。注意调入时可能需要置换页面。

左侧模拟结果一栏会显示模拟结果的缺页次数与缺页率。

5.3 多次模拟

- 方式一：点击"Clear"清空页面，然后重新选择换页算法并点击"Click to Start"运行。
- 方式二：直接重新选择换页算法并点击"Click to Start"运行。

6. 模拟结果

算法	缺页次数	缺页率
FIFO	141	0.44
LRU	143	0.44

7. 项目实现

5.1 全局变量

```
// 获取"开始模拟"按钮
var Start_Simulation = document.getElementById("Start_Simulation");
var Clear_Button = document.getElementById("Clear")

//获取参数信息
var Memory_Blocks_Count = parseInt(document.getElementById("Memory_Blocks_Count").textContent); // 4
var Instructions_Count = parseInt(document.getElementById("Instructions_Count").textContent); // 320
var Instructions_Count_Per_Page = parseInt(document.getElementById("Instructions_Count_Per_Page").textContent); // 10

//获取需要改变的标签元素
var Page_Fault_CountSpan = document.getElementById("Page_Fault_Count");
var Page_Fault_RateSpan = document.getElementById("Page_Fault_Rate");

//定义变量
var memory = [];// 内存块
var instructions = [];// 记录指令访问次序
var page_fault_count = 0;// 缺页个数
```

5.2 函数

5.2.1 init 函数

init 函数用于模拟的初始化，包括清空页面中的模拟信息和初始化全局变量。

```
function init() {

    //清空表格
    var table = document.getElementById("simulation-information");
    while (table.rows.length > 2) {
        table.deleteRow(table.rows.length - 1);
    }

    //初始化变量
    memory = new Array(Memory_Blocks_Count);
    instructions = new Array(Instructions_Count);

    page_fault_count = 0;

    Page_Fault_CountSpan.textContent = page_fault_count;
    Page_Fault_RateSpan.textContent = page_fault_count / Instructions_Count;

};
```

5.4.2 generate_instructions 函数

generate_instructions 函数用于随机生成指令访问次序，

生成原则：

① 50%的指令是顺序执行的

② 25%是均匀分布在前地址部分

③ 25％是均匀分布在后地址部分。

具体实施方法：

① 随机选取一个起始执行指令

② 依次按照顺序执行、跳转至前地址、顺序执行、跳转至前地址的顺序确定下一条指令。其中跳转的指令都是通过随机数生成跳转指令。

③ 直至生成 320 条指令。

生成的指令访问序列存入 instructions[]中。

```
function generate_instructions() {
    var cur_ins = Math.floor(Math.random() * Instructions_Count); //随机生成起始指令 current_instruction
    var pre_ins = -1;

    //按照顺序执行、跳转到后地址、顺序执行、跳转到前地址的顺序生成指令
    //所以可以根据 idx 来确定当前指令应当如何生成

    var idx = 0;
    instructions[0] = cur_ins;

    while (idx < Instructions_Count - 1) {
        pre_ins = cur_ins;

        if (idx % 2 === 0 && cur_ins < Instructions_Count - 1) //顺序执行
            ++cur_ins;

        else if (idx % 4 === 1 && cur_ins < Instructions_Count - 2) //跳转到后地址
            cur_ins = Math.floor(Math.random() * (Instructions_Count - (cur_ins + 2))) + cur_ins + 2;

        else if (idx % 4 === 3 && cur_ins > 0) //跳转到前地址
            cur_ins = Math.floor(Math.random() * cur_ins);

        else {
            while (cur_ins === pre_ins) //如果指令没变，说明前述的跳转规则不适用，直接随机一条新指令
                cur_ins = Math.floor(Math.random() * Instructions_Count);
        }
        instructions[++idx] = cur_ins;

    }
}
```

5.4.3 execute_simulation 函数

execute_simulation 函数用于执行请求调页内存管理方式的模拟。

首先获取对换页算法的选择，然后按照此前生成的指令访问序列依次访问指令，对于每一个，执行：检查指令是否在内存中，若存在则更新页面中的内容，显示所在内存快好；若不存在则需要调页。

FIFO 算法的实现：一直按照内存块 1~内存块 4 的顺序装入即可。

LRU 算法的实现：记录一个 vis_seq 数组，为访问顺序，靠近末尾的为最近访问的。每次调入 vis_seq[0]的内存块中，即为最近最久未使用的内存块。每次访问指令后，需要将指令所在的内存块设置为最新访问的块，即在 vis_seq 数组中移至末尾。

```
function execute_simulation() {
    var page_replacement_algorithm = document.querySelector("input:checked").value; //获取对换页算法的选择

    var vis_seq = [0, 1, 2, 3];// visit_sequence 访问顺序，靠近末尾的为最近访问的
    var FIFO_block = 0;

    for (var idx = 0; idx < instructions.length; ++idx) {

        cur_ins = instructions[idx];
        var cur_page = Math.floor(cur_ins / Instructions_Count_Per_Page); //current_page

        // 判断选中指令是否在内存中
        var instruction_available = is_Available(cur_ins);
        if (!instruction_available) { // 不在内存中，缺页

            page_fault_count++; //跟新缺页数目

            // 更新相应html标签
            Page_Fault_CountSpan.textContent = page_fault_count;
            Page_Fault_RateSpan.textContent = page_fault_count / Instructions_Count;

            // 替换
            if (page_replacement_algorithm === "FIFO")
                memory[(FIFO_block++) % 4] = cur_page;

            else if (page_replacement_algorithm === "LRU")
                memory[vis_seq[0]] = cur_page;

        };

        if (page_replacement_algorithm === "FIFO")
            update_table(idx, instruction_available, (FIFO_block - 1) % 4 + 1);

        else if (page_replacement_algorithm === "LRU") {
            // 更新访问顺序
            var LRU_block = memory.indexOf(cur_page);

            // 将当前块在访问顺序数组中挪到最后一位
            vis_seq.splice(vis_seq.indexOf(LRU_block), 1);
            vis_seq.push(LRU_block);

            update_table(idx, instruction_available, LRU_block + 1);
        }

    };
}
```

5.4.4 start 函数

当“Click to Start”按钮被点击后，会执行 start()函数：

① 首先禁用 Start 与 Clear 按钮，避免按钮点击对模拟过程造成干扰。

② 调用 init()函数来初始化页面与变量。

③ 调用 generate_instructions()函数来按照指定原则随机生成指令访问序列。

④ 调用 execute_simulation()函数来模拟请求调页内存管理方式，并显示结果。

⑤ 启用 Start 与 Clear 按钮，以便进行下一次模拟。

```
function start() {
    // 禁用"Start"和"Clear"按钮
    Start_Simulation.disabled = true;
    Clear_Button.disabled = true;

    init(); // 初始化表格和变量
    generate_instructions(); //生成指令序列
    execute_simulation(); //开始模拟

    // 启用"Start"和"Clear"按钮
    Start_Simulation.disabled = false;
    Clear_Button.disabled = false;
}
```

5.4.5 is_Available 函数

Is_Available(number)函数用于判断指令 number 是否在内存中。

```
function is_Available(number) {
    for (var i = 0; i < memory.length; i++)
        if (Math.floor(number / Instructions_Count_Per_Page) === memory[i])
            return true; // 已经存在，没有发生缺页
    return false;// 缺页
};
```

5.4.6 update_table 函数

该函数用于跟新页面中用于显示模拟信息的表格。

```
function update_table(idx, instruction_available, block) {

    var cur_ins = instructions[idx]; //current_instruction
    var next_ins = instructions[idx + 1]; //next_instruction

    var new_row = document.getElementById("simulation-information").insertRow()
    new_row.insertCell(0).innerHTML = idx + 1;
    new_row.insertCell(1).innerHTML = "NO. " + cur_ins;

    if (next_ins == cur_ins + 1)
        new_row.insertCell(2).innerHTML = "→Sequence"
    else if (next_ins < cur_ins)
        new_row.insertCell(2).innerHTML = "↑Preceding"
    else if (next_ins > cur_ins)
        new_row.insertCell(2).innerHTML = "↓Succeeding"
    else
        new_row.insertCell(2).innerHTML = "NULL"

    for (var i = 0; i < 4; ++i)
        new_row.insertCell(i + 3).innerHTML =
            memory[i] == undefined ? "Empty" : memory[i];

    if (!instruction_available)
        new_row.insertCell(7).innerHTML = "⚠ Page fault             &nb
    else
        new_row.insertCell(7).innerHTML = "✓ Already in memory block " + block;
}
```

8. 项目总结与心得

通过这次项目实现，我对请求调页的内存管理方式的认识更加深刻，对 FIFO 和 LRU 两种页面置换算法的理解也更加清晰。

此外，通过这个项目，我第一次接触、学习并尝试使用 HTML、CSS 和 JavaScript 来构建网页的前端和后端。通过编写 HTML 页面结构、应用 CSS 样式以及使用 JavaScript 实现交互和逻辑，我成功地将内存管理模拟器呈现为一个功能完善的网页应用。这个经历让我熟悉了前端和后端开发的基本概念，并培养了我的网页开发技能。

总的来说，这个项目带给我很多收获。我不仅加深了对内存管理方式的理解，还学到了如何运用前端和后端技术构建一个实用的网页应用。我期待将这些学到的知识和经验应用到未来的项目中，并不断提升自己在软件开发领域的能力。