同济大学

Tongji University

《操作系统》

项目报告II

报告名称：内存管理项目之请求调页

班　　级：操作系统02班

学　　号：2253744

姓　　名：林觉凯

指导老师：张惠娟

完成日期：2024年5月16日

目录

1.项目介绍

1.1 项目目的

- 掌握页面、页表、地址转换；
- 详细理解并掌握页面置换的过程；
- 加深对请求调页系统的原理和实现过程的理解。

1.2 项目需求

- 基本任务

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。

- 模拟过程

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。

- 置换算法和指令访问次序；

置换算法可以选用 FIFO 或者 LRU 算法；

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。具体实施方法如下(可以参考，不一定必须如此):

在 0-319 条指令之间，随机选取一个起始执行指令，如序号为 m

顺序执行下一条指令，即序号为 m+1 的指令

通过随机数，跳转到前地址部分 0-m-1 中的某个指令处，其序号为 m1

顺序执行下一条指令，即序号为 m1+1 的指令

通过随机数，跳转到后地址部分 m1+2~319 中的某条指令处，其序号为 m2

顺序执行下一条指令，即 m2+1 处的指令。

重复跳转到前地址部分、顺序执行、跳转到后地址部分、顺序执行的过程，直到执行完 320 条指令。

1.3 开发环境介绍

- 开发环境：Windows 11
- 开发软件：Visual Studio Code
- 开发语言：HTML、CSS 和 JavaScript

1.4 项目运行

按照下图所示目录将文件放置好(即提交的 os_project2_memory 文件夹顺序)，然后点击 os_project2_memory.html 即可进入界面运行。(建议使用 Chrome 打开)

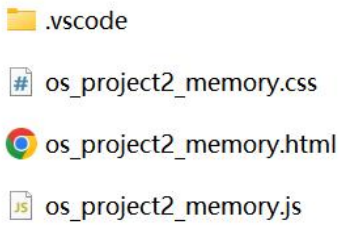

图 1.1 文件放置顺序

2.项目设计

2.1 前端页面设计

HTML、CSS 和 JavaScript 是在 Web 前端开发中最基础且最重要的三项技术。HTML 用于创建网页结构和内容的标记语言，它定义了网页的基本框架；CSS 用于控制网页的外观和布局，使网页更美观和易用；JavaScript 是一种脚本语言，用于为网页添加交互性和动态效果，它可以操作 HTML 和 CSS，响应用户事件实现复杂的功能。本次项目的前端界面就是使用前端三剑客来实现的。

前端界面的设计主要分为一下几个部分：

最顶上面的 navbar，它用于显示本次项目的主题和作者；

在左端的第一个方框内显示“Please choose one kind of Algorithm”，我们可以在第一个方框中选择我们所需要的请求调页算法(FIFO 和 LRU)；

在左端的第二个方框内显示“Relevant parameters are as follows”，它用来显示本次项目所用到的参数(即内存块个数、指令条数、每页的指令数量)；

在左端的第三个方框内显示“Simulation results are as follows”，它用来显示在运行后的结果：缺页数和缺页率。

在左端的下方则有两个按钮：Start 和 Clear，前者用来开始模拟请求调页的

过程，后者用来清空每一次请求调页的结果。

在右侧便是每一次请求调页面的结果，它包含了以下显示数据：

指令序列(Sequence)：1-320，是第几条执行指令的序号；

指令编号(Instruction)：0-319，是每一条指令的编号(这里注意区别指令的编号和序号，序号仅代表指令执行的顺序，编号则决定了指令所在页号)；

内存块(Memory Block)：1-4，即代表四个内存块；

缺页情况(Page Fault)：来判断当前指令是否缺页；

载入块(Load Block)：如果缺页，则应该将当前也放入哪一个内存块中；

移除页面(Remove Page)：如果缺页，则替换的页面需要被移除相应的内存块。

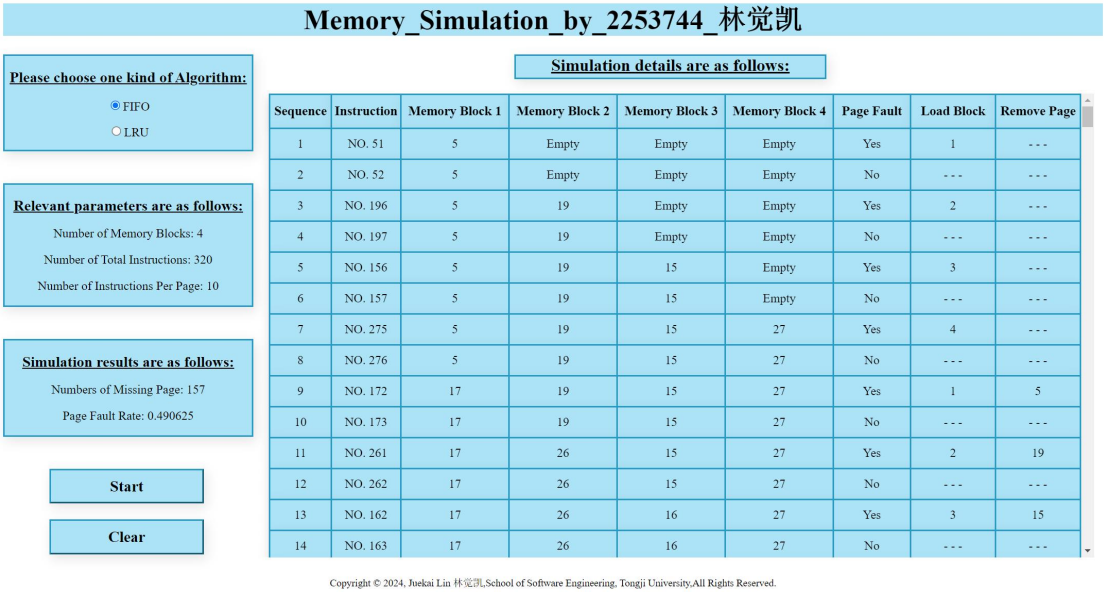

图 2.1 整体前端页面设计

2.2 后端算法设计

请求调页(Page Replacement)算法在计算机操作系统中用于管理虚拟内存。当一个进程访问的页面不在物理内存中时，需要将一个页面从内存中移出，以腾出空间将所需页面加载进来。本次项目采用的页面置换算法有 FIFO(First In, First Out)和 LRU(Least Recently Used)。

FIFO(First In, First Out)：FIFO 算法根据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移出。维护一个队列，队列按页面进入内存的顺序排列。当需要置换页面时，移出队列最前面的页面，并将新页面加入队列的末尾。

LRU(Least Recently Used)：LRU 算法根据页面最近使用的时间进行置换。最

久未使用的页面最先被移出。维护一个数据结构(如列表或字典)，记录每个页面最近被访问的时间。当需要置换页面时，移出最近最少使用的页面，并更新新页面的最近使用时间。

两者之间的比较。FIFO：简单实现，但是可能不太高效，因为它不考虑页面的使用频率或时间。在某些情况下增加帧数反而可能导致更多的缺页。LRU：较为复杂，但一般更高效。考虑页面的使用时间，可以更好地预测哪些页面将不再需要。但是实现时需要维护额外的数据结构来记录页面使用时间，可能增加开销。

3.项目实现

3.1 生成指令序列实现

本次项目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。所以我们使用以序列的 index 为依据，如果序列的 index 除以 2 等于 0，则下一条指令顺序执行；如果序列的 index 除以 4 余 1，则下一条指令均匀分布在后地址部分；如果序列的 index 除以 4 余 3，则下一条指令均匀分布在前地址部分。这种数学方式的计算，可以保证有百分之 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。

```
    //生成指令序列的函数
        function generateInstructions() {
            var Index = 0;
            var preInstruction = -1;
            //在 0-319 中随机生成一条指令
            var curInstruction = Math.floor(Math.random() * numTotalInstruction);
            //将这一条指令放在指令序列的第一个
            Instruction[0] = curInstruction;
            //循环生成剩下的指令序列
            while (Index < numTotalInstruction - 1) {
                preInstruction = curInstruction;
                //50%的指令是顺序执行的
                if (Index % 2 === 0 && curInstruction < numTotalInstruction - 1)
                    curInstruction++;
                //25%的指令是均匀分部在后地址的
                else if (Index % 4 === 1 && curInstruction < numTotalInstruction - 2)
                    curInstruction = Math.floor(Math.random() * (numTotalInstruction -
curInstruction - 2)) + curInstruction + 2;
                //25%的指令是均匀分布在前地址的
                else if (Index % 4 === 3 && curInstruction > 0)
```

```
                    curInstruction = Math.floor(Math.random() * curInstruction);
                //如果两次生成的指令同一条，则随机再生成一条指令
                else {
                    while (preInstruction === curInstruction)
                        curInstruction = Math.floor(Math.random() *
numTotalInstruction);
                }
                //将生成的指令加到指令序列中
                Instruction[++Index] = curInstruction;
            }
        }
```

3.2 算法 FIFO 实现

FIFO 算法根据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移出。我们可以发现，FIFO 算法每次移动出的页面都是非常顺序的，(这里我们也可以理解为按照内存块 1-2-3-4 的顺序依次移除相应内存块中的页面)所以我们这里使用应该 fifoBlock 变量来记录最先到达的指令，并且每次移除页面时都给它++，再给它%4 来判断该移除的内存块时那一块就可以了。

```
    //FIFO 算法的实现函数
        function fifoSimulation() {
            //fifoBlock:用来记录最先到达的指令
            var fifoBlock = 0;
            for (var Index = 0; Index < Instruction.length; Index++) {
                //获得当前所需执行的指令
                curInstruction = Instruction[Index];
                var removePage = -1;
                //获得当前所需执行的指令的页面并且判断是否缺页
                var currentPage = Math.floor(curInstruction / numInstructionPerPage);
                var instructionAvailable = judgeAvailable(curInstruction);
                if (!instructionAvailable) {
                    //设置缺页相关的参数
                    numMissingPage++;
                    pageMissingCount.textContent = numMissingPage;
                    pageFaultRate.textContent = numMissingPage / numTotalInstruction;
                    //记录被移除页面页号和更新当前页面
                    removePage = Memory[(fifoBlock) % 4];
                    Memory[(fifoBlock++) % 4] = currentPage;
                };
                //将新的一行添加入结果显示表格中
                addInstructionsToTable(Index, instructionAvailable, (fifoBlock - 1) % 4 +
1, removePage);
            };
    }
```

3.3 算法 LRU 实现

LRU 算法根据页面最近使用的时间进行置换。最久未使用的页面最先被移出。这个时候我们就需要一个类似于队列的数据结构，这里我们第一一个 orderSequence，来记录最近使用的页面所在的内存块的 index，我们这里每次使用一个内存块我们都将原先内存块所在的 orderSequence 消除，再将该内存块重新压入 orderSequence 中，这样就可以保证 orderSequence[0]时最久未被使用的内存块，这个时候我们便可以通过这些函数得到需要移出的页面的页号，并且可以将新的页面加入到内存块中。

```
    //LRU 算法的实现函数
        function lruSimulation() {
            //访问顺序数组，越靠后越近访问
            var orderSequence = [0, 1, 2, 3];

            for (var Index = 0; Index < Instruction.length; Index++) {
                //获得当前所需执行的指令
                curInstruction = Instruction[Index];
                var removePage = -1;
                //获得当前所需执行的指令的页面并且判断是否缺页
                var currentPage = Math.floor(curInstruction / numInstructionPerPage);
                var instructionAvailable = judgeAvailable(curInstruction);
                //如果缺页
                if (!instructionAvailable) {
                    //设置缺页相关的参数
                    numMissingPage++;
                    pageMissingCount.textContent = numMissingPage;
                    pageFaultRate.textContent = numMissingPage / numTotalInstruction;
                    //记录被移除页面页号和更新当前页面
                    removePage = Memory[orderSequence[0]];
                    Memory[orderSequence[0]] = currentPage;
                };
                //记录当前使用页面的所在内存块
                var lruBlock = Memory.indexOf(currentPage);
                //从指令序列中删去当前块号
                orderSequence.splice(orderSequence.indexOf(lruBlock), 1);
                //将当前块号重新加入队尾(即最大的下标，最近使用)
                orderSequence.push(lruBlock);
                //将新的一行添加入结果显示表格中
                addInstructionsToTable(Index, instructionAvailable, lruBlock + 1,
removePage);
            };
        }
```

3.4 更新前端页面实现

更新前端页面主要由 addInstructionsToTable 函数来实现，对每一行的每一个单元格逐渐赋值。最主要是 Load in 和 Remove Page 这两个单元格，其中 Load in 单元格就由两种算法得出结果参数传入，Remove Page 填入的时候要注意，缺页的时候由两种情况，第一是一开始缺页，移入的内存块中并没有其它页面，这时候就不需要写出 Remove Page，填“- - -”；之后内存块中页面移出时，便填入要移出的页面的页号。

```
    //更新并显示表格内容的函数
        function addInstructionsToTable(index, instructionAvailable, block,
removePage) {
            var curInstruction = Instruction[index];
            //每一条指令都插入新建的一行中
            var newRow = document.getElementById("showTable").insertRow()
            //每一行的第 1 个单元格：序号
            newRow.insertCell(0).innerHTML = index + 1;
            //每一行的第 2 个单元格：指令编号
            newRow.insertCell(1).innerHTML = "NO. " + curInstruction;
            //每一行的第 3、4、5、6 个单元格：存放页面的页号或者是空的
            for (var i = 0; i < 4; i++)
                newRow.insertCell(i + 2).innerHTML = Memory[i] == undefined ? "Empty" :
Memory[i];
            //每一行的第 7、8 个单元格，判断是否缺页和载入、移出情况
            if (!instructionAvailable) {
                //如果缺页，显示缺页情况和载入块号、移出页号
                newRow.insertCell(6).innerHTML = "Yes";
                newRow.insertCell(7).innerHTML = block;
                if (removePage == -1 || removePage == undefined)
                    newRow.insertCell(8).innerHTML = "- - -";
                else
                    newRow.insertCell(8).innerHTML = removePage;
            }
            else {
                //如果不缺页，显示 No
                newRow.insertCell(6).innerHTML = "No";
                newRow.insertCell(7).innerHTML = "- - -";
                newRow.insertCell(8).innerHTML = "- - -";
            }
        }
```

3.5 清空按键实现

清空按键需要重置三个全局变量，同时改变 HTML 页面上的缺页率和缺页次数，最后删除结果表格的非表头行。

```
//清除函数，重置信息
    function Clear() {
        //重置三个全局变量
        Memory = new Array(numMemoryBlock);
        Instruction = new Array(numTotalInstruction);
        numMissingPage = 0;

        //删除目前存在的表格信息
        var showTable = document.getElementById("showTable");
        while (showTable.rows.length > 1)
            showTable.deleteRow(1);
        //将调页结果设为 NULL
        pageMissingCount.textContent = "NULL";
        pageFaultRate.textContent = "NULL";
    }
```

3.6 开始按键实现

开始按键绑定的是整个请求调页过程的执行，首先要清除原有记录，接下来生成指定的指令序列，最后执行指令产生结果并且显示。

```
//开始函数，开始执行
    function Start() {
        //先 disable 两个按键
        startButton.disabled = true;
        clearButton.disabled = true;
        //再清除已有信息、生成指令序列、执行指令
        Clear();
        generateInstructions();
        executeInstructions();
        //最后将两个按键再次置为可以使用状态
        startButton.disabled = false;
        clearButton.disabled = false;
    }
```

3.7 算法选择的实现

算法选择通过获取 HTML 上的相应标签的 value，通过 if-else 判断，来选择各自的算法进行执行。

```
//执行指令函数
    function executeInstructions() {
```

```
        //获取页面上所选择的算法
        var chooseAlgorithm = document.querySelector("input:checked").value;
        if (chooseAlgorithm == "FIFO")
            fifoSimulation();
        else
            lruSimulation();
    }
```

3.8 判断缺页的实现

判断缺页通过遍历四个内存块，看看是否存在内存块中有当前指令所在的页面，如果有则不缺页，反之则缺页。

```
//判断是否缺页函数
    function judgeAvailable(instruction) {
        //遍历四个内存块，检查是否有内存块包含该指令所在的页面
        for (var i = 0; i < Memory.length; i++)
            if (Math.floor(instruction / numInstructionPerPage) === Memory[i])
                return true;
        return false;
    }
```

4.项目示例与分析

点开 os_project2_memory.html，我们分别采用不同的两种算法进行请求调页的模拟，相应的结果与分析如下：

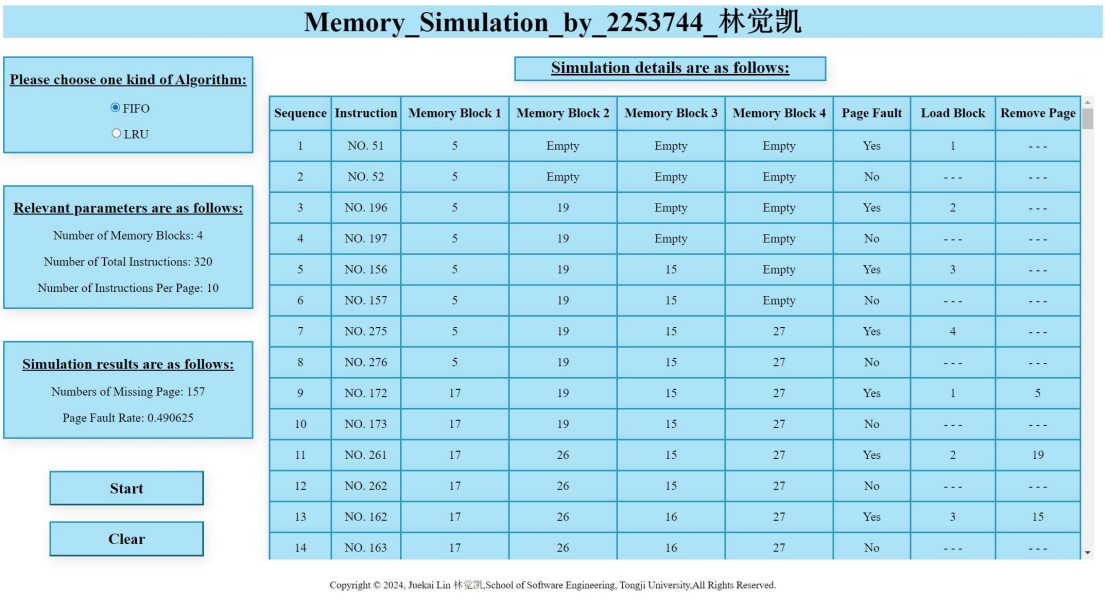

图 4.1 使用 FIFO 算法请求调页示例

以上是使用 FIFO 算法请求调页的示例，我们从前面几条指令开始看。第一条指令所在页号为 5，放入内存块 1，之后的几条指令分别将四个内存块填满。来到第 9 条指令，此时发生缺页，需要从内存块中移出一张页面，

根据 FIFO 的原理，内存块 1 中的 5 号页面是最早到达内存块中的，所以我们移出内存块 1 中的 5 号页面。之后按照整个算法不断调页。

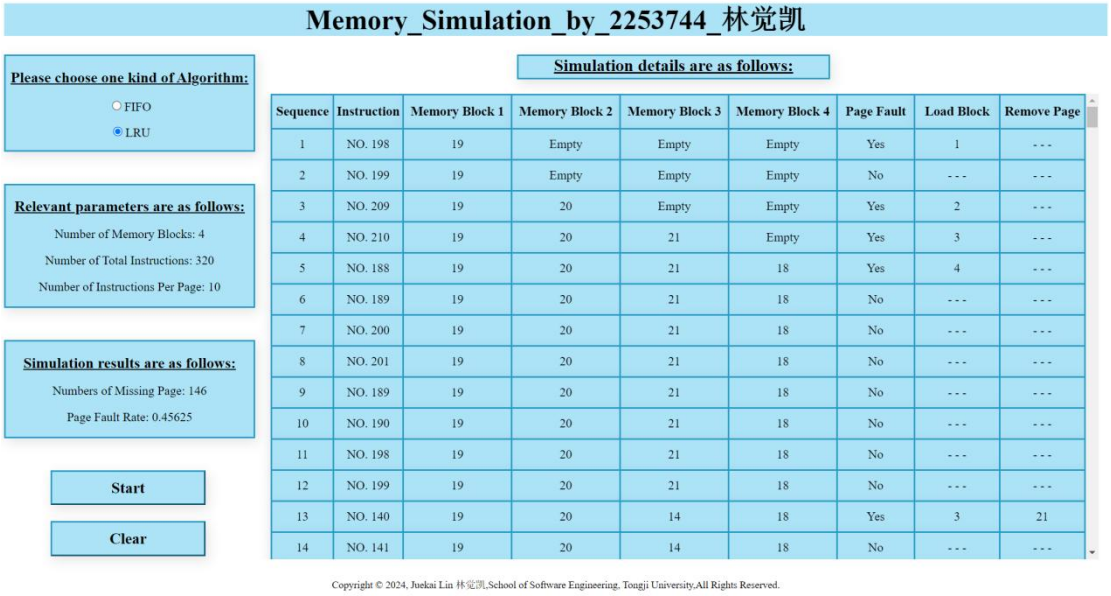

图 4.2 使用 LRU 算法请求调页示例

以上是使用 LRU 算法请求调页的示例，我们从前面几条指令开始看。第一条指令 198 所在页面页号为 19，将它放入内存块 1。之后逐渐执行指令将四个内存块填满。观察到第 13 条指令 140 时，发生缺页，这时，不是按照 FIFO 算法将最先到达的内存块 1 中的 19 号页面置换出。这时我们分析，1 号内存块中的 19 号页面在第 12 条指令时使用，第 2 号内存块中的 20 号页面在第 8 条指令时使用，第 3 号内存块中的 21 号页面在第 4 条指令时使用，第 4 号内存块中的第 18 号指令在第 9 条指令时使用。我们发现，第 3 号内存中的第 21 号页面时最久未被使用的，使用置换它。

通过两个示例我们可以看到，使用 FIFO 算法最后的的缺页数为 157，缺页率为 0.490625；使用 LRU 算法最后的缺页数为 146，缺页率为 0.45625.这也验证了 LRU 置换算法比 FIFO 置换算法性能更加优异些。(当然这是大多数情况，不排除有些情况 FIFO 置换算法性能更好)

5.项目总结

5.1 项目遇到的问题

本次项目较为简单，加之我本身有 HTML、CSS 和 JavaScript 前端开发的基础，这次作业比较顺利地完成了。本次作业重点的部分便是两种算法的实现与编

码，两种算法是请求调页模拟的关键。在编程的过程中，重点是要注意到 FIFO 算法我们需要一个辅助的变量来表示每一次需要被移出的页面的页号的基本信息，LRU 算法则是需要维护一个类似队列的数据结构，其中在 JavaScript 中相关的函数需要我们查找资料进行学习。

5.2 项目心得与收获

本次项目，我利用编程的方法重现了在内存管理中的请求调页的实现过程。通过本次作业，我对请求调页的实现过程有了更加深入的理解，最主要的是对请求调页中的两种算法 FIFO 和 LRU 有了更加清晰的认识，这对于我理论课收获的知识有着巩固的作用，也相当于仔细地复习了一边理论课这部分的内容。更重要的是，在这次项目的编程过程中，我对 HTML、CSS 和 JavaScript 这些前端开发的技能有了更加熟练的掌握，前端开发技能是作为一位软件工程专业学生必不可少的技能，本学期许多课程也将使用相应的技能来完成项目。这次项目的顺利完成将为我之后的学习与工作打下坚实的实践基础。