

验收成绩	报告成绩	总评成绩

武汉大学计算机学院

本科生实验报告

操作系统模拟实验

专业名称：计算机科学与技术

课程名称：操作系统课程设计

指导教师：郑鹏

学生学号：2021300004042

学生姓名：顾田

二〇二四年五月

# 郑 重 声 明

本人呈交的实验报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本实验报告不包含他人享有著作权的内容。对本实验报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本实验报告的知识产权归属于培养单位。

本人签名： 顾田

日期： 2024.05.03

## 摘 要

操作系统模拟实验的实验目的是理解操作系统中的关键概念和原理，并通过实践加深对这些概念的理解。实验设计主要遵循操作系统的基本原理和实践方法，确保实验能够有效地达到预期目标。实验内容主要包括模拟进程调度、虚拟内存管理和文件系统等方面的功能，通过实现不同的调度算法、页面置换算法和文件操作功能，并通过模拟数据进行测试和分析。最终得出实验结论，对不同算法和系统设计进行评价和总结。

**关键词：**操作系统；调度算法；页面置换算法；文件系统；模拟数据；测试分析。

# 目 录

1. 实验概述 .....	6
1.1 实验目的 .....	6
1.2 实验工具 .....	6
1.2.1 集成开发环境 (IDE): Visual Studio Code .....	6
1.2.2 版本控制系统: Git .....	6
1.3 实验准备 .....	6
1.3.1 进程调度实验设计 .....	6
1.3.2 虚拟页式存储管理系统实验设计 .....	7
1.3.3 文件系统实验设计 .....	7
2. 进程调度模拟实验 .....	8
2.1 实验目的 .....	8
2.2 实验步骤 .....	8
2.3 实验设计 .....	8
2.3.1 进程结构体 .....	8
2.3.2 数据的输入 .....	9
2.3.3 短进程优先调度算法 .....	9
2.3.4 时间片轮转调度算法 .....	12
2.3.5 多级队列调度算法 .....	14
2.4 实验结果测试 .....	16
2.4.1 短进程优先 .....	16
2.4.2 时间片轮转调度算法 .....	16
2.4.3 多级队列 (有抢占) 调度算法 .....	16
2.5 总结 .....	18
3. 页面置换算法模拟实验 .....	19
3.1 实验目的 .....	19
3.2 实验步骤 .....	19
3.3 实验设计 .....	19
3.3.1 内存结构 .....	19

3.3.2 页表和进程结构 .....	21
3.3.3 数据的输入 .....	22
3.3.4 页面置换算法 .....	23
3.4 测试 .....	26
3.5 总结 .....	29
4. 文件系统模拟实验 .....	30
4.1 实验目的 .....	30
4.2 实验步骤 .....	30
4.3 实验设计 .....	30
4.3.1 基本结构 .....	31
4.3.2 文件操作的设计 .....	32
4.3.3 内存管理设计 .....	35
4.3.4 数据的输入 .....	37
4.3.5 界面设计 .....	38
4.4 测试 .....	39
4.5 总结 .....	43
5. 实验总结 .....	45
5.1 实验结果 .....	45
5.2 问题和解决方案 .....	45
5.2.1 问题 .....	45
5.2.2 解决方案: .....	45
5.3 实验收获 .....	46

# 1. 实验概述

## 1.1 实验目的

操作系统模拟实验的实验目的在于通过实践来加深对操作系统中关键概念和原理的理解。具体来说，主要在：1) 理解在单处理器环境下的进程调度及状态转换的基本原理和关键点。2) 理解虚拟内存和物理内存之间的关系，以及缺页中断的发生情况。3) 理解文件系统的主要组成和工作原理，包括文件的存储方式、权限管理和空间管理等。

## 1.2 实验工具

在本次实验中，我使用了以下工具和环境来进行编程和版本控制：

### 1.2.1 集成开发环境 (IDE): Visual Studio Code

Visual Studio Code (VSCode) 是一款免费、轻量级的集成开发环境，提供了丰富的功能和插件，支持多种编程语言，包括 C++。

我选择了 VSCode 作为主要的编程工具，因为它具有直观的界面、强大的代码编辑功能和丰富的插件生态系统，能够满足我在实验中的需求。

### 1.2.2 版本控制系统: Git

Git 是一个分布式版本控制系统，用于跟踪文件的变化和协作开发。我使用 Git 来管理实验的代码和文档，以便进行版本控制和团队协作。

在实验过程中，我通过 Git 进行代码的提交、分支管理和合并操作，确保代码的版本控制和追踪。

## 1.3 实验准备

在本次实验中，我设计了一系列的实验步骤，以模拟操作系统的关键功能，并使用 C++ 语言实现。下面是我实验的设计思路和步骤：

### 1.3.1 进程调度实验设计

**数据准备：**设计进程类，包含到达时间、运行时间、优先级等属性。随机生

成或人工指定一组进程数据作为测试集。

**实验要求：**

1. 实现时间片轮转调度算法的模拟函数。
2. 实现短进程优先调度算法的模拟函数。
3. 实现多级反馈队列调度算法的模拟函数，包括至少三个队列，每个队列的时间片大小不同。
4. 对给定的一组进程，分别采用不同的调度算法进行仿真模拟，并记录进程状态转换过程。
5. 计算并比较不同调度算法的性能指标，如平均周转时间、平均等待时间、平均响应时间等。

### **1.3.2 虚拟页式存储管理系统实验设计**

**数据准备：**设计进程类和内存类，模拟进程的页面分配和页面置换过程。生成合适的测试数据集。

**实验要求：**

1. 设计并实现一个简单的虚拟页式存储管理系统，包括页表创建、页面分配、页面置换等功能。
2. 实现常见页面置换算法的模拟函数，如 FIFO、LRU、CLOCK 和随机置换算法。
3. 使用模拟数据对不同页面置换算法进行仿真模拟，并比较性能指标。

### **1.3.3 文件系统实验设计**

**数据准备：**设计文件和目录的数据结构，模拟文件和目录的创建、读写、删除等操作。

**实验要求：**

1. 设计并实现文件系统的基本功能，包括文件和目录的创建、打开、关闭、读取、写入、删除等操作。
2. 设计用户交互接口，通过终端命令进行文件系统操作的模拟。
3. 使用模拟数据测试文件系统的功能和性能。

## 2. 进程调度模拟实验

### 2.1 实验目的

本次实验的主要目的如下：

1. 理解在单处理器环境下的进程调度及状态转换的基本原理和关键点。
2. 编程实现时间片轮转调度、短进程优先调度和多级反馈队列调度算法，加深对调度算法的理解和掌握。
3. 分析并比较不同调度算法的效率和公平性，探讨其适用场景和优缺点。
4. 掌握 C++ 语言的编程技能，在模拟实验中实现进程调度算法，并对仿真结果进行评估和分析。

### 2.2 实验步骤

为了达到实验目的，我们将按以下步骤进行实验：

1. 设计进程类：首先，我们需要设计一个进程类，包含进程的属性，如到达时间、运行时间、优先级等。这个类将作为进程模拟的基本单元。
2. 实现短进程优先调度算法：编写另一个函数来实现短进程优先调度算法。该函数将根据进程的运行时间，选择最短的进程优先执行。
3. 实现时间片轮转调度算法：将短进程优先调度稍作修改，便能得到时间片调度算法的实现逻辑。
4. 多级反馈队列调度算法：实现第三个函数来模拟多级反馈队列调度算法。将时间片算法的函数稍作修改，实现不同大小的时间片共同作用，即可实现多级队列的逻辑。
5. 实验结果分析：根据性能评估的结果，对不同调度算法进行分析和比较。探讨各种算法的优缺点，以及在不同场景下的适用性。

### 2.3 实验设计

#### 2.3.1 进程结构体



```

You, last month | 1 author (You)
struct Process
{
    /* 表示一个进程 */
    string name;           // 进程的名称
    float arrive_time = 0; // 进程的到达时间
    float running_time = 0; // 进程的运行时间
    float remaining_time = 0; // 进程的剩余时间
    unsigned priority = 0; // 进程优先级, 用无符号整数表示
    /* 用于计算平均周转时间/平均带权周转时间, 平均等待时间、平均响应时间、利用率 */
    float turnaround_time = 0; // 周转时间
    float waiting_time = 0; // 等待时间
    float response_time = -0.25; // 响应时间
    float utilization = 0; // 利用率
};

```

进程结构体定义了一个进程的基本属性，包括名称、到达时间、运行时间、剩余时间、优先级以及用于计算性能指标的各种时间属性。

### 2.3.2 数据的输入

实验中使用了一个函数实现了从文件中读取进程数据并将其记录到程序中的功能。它打开一个文件流，逐行读取文件内容，并使用字符串流从每行数据中解析出进程的名称、到达时间、运行时间和优先级。然后，创建一个新的进程对象，将解析得到的数据赋值给进程的相应属性，并将进程对象添加到进程列表中。如果读取的数据格式错误，则输出错误信息。

在实验中，将进程可以按如下格式保存到.txt 文件中，程序开始时加载进程。

```

You, last month | 1 author
1 process1 0 2 0
2 process2 1 7 0
3 process3 2 4 0
4 process4 3 1 0
5 process5 5 11 0
6 process6 6 3 0
7 process7 8 2 0
8 process8 50 1 0

```

### 2.3.3 短进程优先调度算法

下面的函数实现了短进程优先调度算法。它首先按照进程的到达时间对进程

进行排序，然后循环执行调度过程，直到所有进程都完成调度为止。在每个时间片内，程序会检查当前时间是否有进程到达，如果有，则将其加入就绪队列；然后从就绪队列中选择运行时间最短的进程执行，直至该进程完成运行。在执行过程中，记录每个进程的周转时间、响应时间、等待时间和利用率等信息。最后，输出算法执行结束时的时间和 CPU 利用率，并展示调度结果。

```
// 短进程优先调度算法
void shortestJobFirstScheduling(const vector<Process>&
originalProcesses) {
    // 创建一个副本，用于修改进程状态
    vector<Process> processes = originalProcesses;
    // 按照到达时间进行排序
    sortByArriveTime(processes);
    // Cpu 空闲的时间
    float spare = 0;

    // 当前时间
    float curTime = 0;
    // 进程索引
    int index = 0;
    // 就绪的进程队列
    queue<Process> readyProcesses = {};
    // 完成的进程
    vector<Process> finished = {};

    // 算法的过程
    // 仍有进程没有完成调度，循环
    while(finished.size() < processes.size()){
        // 当前时间已经到达的进程更新加入队列
        for(int i=index;
(i < processes.size()) && (processes[i].arrive_time <= curTime); i++) {
            readyProcesses.push(processes[i]);
            index++;
        }
        // 没有进程就绪时，等待下一个进程到达
        if(readyProcesses.empty()){
            readyProcesses.push(processes[index]);
            spare += (processes[index].arrive_time - curTime);
            curTime = processes[index].arrive_time;
        }
        // 有进程就绪时，执行当前队列的时间最短的进程
        else {
            // 先按运行时间排序，再选出运行时间最短的进程执行
```

```

        sortByRunningTime(readyProcesses);
        Process curProcess = readyProcesses.front();
        cout << "[SJF] Time: " << curTime << ", Process " <<
curProcess.name << " begins. " << endl;

        curTime += curProcess.running_time;

        curProcess.turnaround_time = curTime -
curProcess.arrive_time; // 到达完成
        curProcess.response_time = curTime - curProcess.running_time
- curProcess.arrive_time; // 到达响应
        curProcess.waiting_time = curTime - curProcess.running_time
- curProcess.arrive_time; // 等待时间的和
        curProcess.utilization = (curProcess.turnaround_time -
curProcess.waiting_time)/curProcess.turnaround_time;

        finished.push_back(curProcess);
        readyProcesses.pop();
    }
}
// 展示结果
cout << "[Ending] Time: " << curTime << endl;
cout << "CPU Utilization Rate: " << 1 - spare/curTime << endl;
showResults(finished);
}

```

简单来说，短进程优先调度算法的过程如下：

#### 初始化：

- 创建一个副本的进程列表，以便于修改进程状态。
- 按照进程的到达时间对副本进程列表进行排序。
- 初始化当前时间 `curTime` 为 0。
- 初始化进程索引 `index` 为 0。
- 初始化就绪的进程队列 `readyProcesses` 为空队列。
- 初始化完成的进程列表 `finished` 为空。

**循环调度进程：**在所有进程都完成调度之前，循环执行以下步骤：

- 更新就绪队列：将当前时间已经到达的进程加入就绪队列中。
  - 如果就绪队列为空：
    - ◆ 将下一个进程加入就绪队列。
    - ◆ 计算 CPU 的空闲时间，并更新当前时间为下一个进程的到达时

间。

- 如果就绪队列不为空：
  - ◆ 从就绪队列中选出运行时间最短的进程。
  - ◆ 执行该进程直到完成。
- 更新当前时间为进程完成后的时间。
- 记录该进程的周转时间、响应时间、等待时间和利用率等信息。
- 将该进程添加到完成的进程列表中。
- 从就绪队列中移除该进程。

#### 输出结果：

- 输出算法执行结束时的时间。
- 计算并输出 CPU 的利用率。
- 展示完成的进程列表，包括每个进程的信息如周转时间、响应时间、等待时间和利用率等。

### 2.3.4 时间片轮转调度算法

时间片轮转与短进程优先算法的框架是一样的，只有细节上有差别。具体来说：

**循环调度进程：**在所有进程都完成调度之前，循环执行以下步骤：

- 更新就绪队列：将当前时间已经到达的进程加入就绪队列中。
  - 如果就绪队列为空：
    - ◆ 将下一个进程加入就绪队列。
    - ◆ 计算 CPU 的空闲时间，并更新当前时间为下一个进程的到达时间。
  - 如果就绪队列不为空：
    - ◆ 从就绪队列中选出队首进程执行。
  - 如果当前进程的剩余时间小于等于时间片大小：
    - ◆ 执行当前进程直到完成。
    - ◆ 更新当前时间为进程完成后的时间。
    - ◆ 计算并记录该进程的周转时间、等待时间和利用率等信息。
    - ◆ 将该进程添加到完成的进程列表中。

- 如果当前进程的剩余时间大于时间片大小：
  - ◆ 执行当前进程一个时间片的时间。
  - ◆ 更新当前进程的剩余时间。
  - ◆ 将当前进程重新加入就绪队列中。

```
// 时间片轮转调度算法
void roundRobinScheduling(const std::vector<Process>&
originalProcesses, float timeQuantum) {
    // .....
    // 有进程就绪时，按照时间片轮转的方式执行进程
    else {
        Process curProcess = readyProcesses.front();
        curProcess.response_time = (curProcess.response_time==0.25)? curTime : curProcess.response_time;
        cout << "[RR] Time: " << curTime << ", Process " <<
curProcess.name << " begins. " << endl;
        readyProcesses.pop();

        // 判断时间片是否足够执行完当前进程
        if (curProcess.remaining_time <= timeQuantum) {
            // 当前时间加上当前进程的运行时间
            curTime += curProcess.remaining_time;
            // 更新进程的属性
            curProcess.turnaround_time = curTime -
curProcess.arrive_time; // 周转时间
            curProcess.waiting_time = curProcess.turnaround_time -
curProcess.running_time; // 等待时间
            curProcess.utilization = curProcess.running_time /
curProcess.turnaround_time; // 利用率
            // 添加进程到完成队列
            finished.push_back(curProcess);
        } else {
            // 当前时间加上一个时间片
            curTime += timeQuantum;
            // 更新当前进程的运行时间和剩余时间
            curProcess.remaining_time -= timeQuantum;
            // 当前时间已经到达的进程更新加入队列
            for (int i = index; (i < processes.size()) &&
(processes[i].arrive_time <= curTime); i++) {
                readyProcesses.push(processes[i]);
                index++;
            }
            // 将进程重新加入队列
        }
    }
}
```

```

        readyProcesses.push(curProcess);
    }
}
}

```

对于时间片轮转调度算法, 在执行当前进程后, 如果当前进程的剩余时间大于时间片大小, 则将当前进程重新加入就绪队列中, 等待下次调度。这意味着每个进程在就绪队列中都有机会被调度, 而不像短进程优先调度算法那样只有最短作业才会被优先执行。

这种更新就绪队列的方法确保了每个进程都有公平的机会被执行, 避免了某些长时间运行的进程占用 CPU 时间过多的情况。

### 2.3.5 多级队列调度算法

将时间片轮转调度算法稍作修改, 即可得到多级队列优先调度算法的实现逻辑。

```

// 算法的过程
while (finished.size() < processes.size()) {
    // 将到达时间小于等于当前时间的进程加入就绪队列
    while (index < processes.size() &&
processes[index].p.arrive_time <= curTime) {
        readyProcesses.push(processes[index]);
        index++;
    }
    // 如果就绪队列非空, 则执行就绪队列中的进程
    if (!readyProcesses.empty()) {
        sortMLFQ(readyProcesses);
        MLFQ curMLFQ = readyProcesses.front();
        cout << "[MLFQ] Time: " << curTime << ", Process " <<
curMLFQ.p.name << " begins. " << endl;
        curMLFQ.p.response_time = (curMLFQ.p.response_time==0.25)?
curTime : curMLFQ.p.response_time;
        readyProcesses.pop();

        // 获取当前进程所在队列的时间片大小
        float timeQuantum = timeQuantums[curMLFQ.q];

        // 判断当前队列的时间片是否足够执行完当前进程
        if (curMLFQ.p.remaining_time <= timeQuantum) {
            // 更新当前进程的属性
            curTime += curMLFQ.p.remaining_time;

```

```

        curMLFQ.p.turnaround_time = curTime -
curMLFQ.p.arrive_time;
        curMLFQ.p.waiting_time = curMLFQ.p.turnaround_time -
curMLFQ.p.running_time;
        curMLFQ.p.utilization = curMLFQ.p.running_time /
curMLFQ.p.turnaround_time;

        // 将当前进程加入完成队列
        finished.push_back(curMLFQ);
    } else {
        // 一个时间片的运行逻辑
        curTime += timeQuantum;
        curMLFQ.p.remaining_time -= timeQuantum;
        // 将当前进程加入下一优先级队列
        curMLFQ.q = min(curMLFQ.q + 1,
static_cast<unsigned>(timeQuantums.size() - 1));
        readyProcesses.push(curMLFQ);
    }
} else {
    // 如果就绪队列为空，将时间推进到下一个进程的到达时间
    spare += (processes[index].p.arrive_time - curTime);
    curTime = processes[index].p.arrive_time;
}
}

```

在多级队列调度算法中，与时间片轮转调度算法相比，主要的区别在于就绪队列的组织和调度逻辑：

- 对于多级队列调度算法：
  - 就绪队列中有多个优先级别的队列，每个队列都有自己的时间片大小。
  - 在执行当前进程后，如果当前进程剩余的运行时间超过了其所在队列的时间片大小，那么该进程将被移到下一个优先级更低的队列中等待后续调度。
  - 如果当前进程在其所在队列的时间片内完成了执行，则该进程会被移到已完成队列中。
  - 如果就绪队列为空，系统会等待下一个进程到达时间，然后再进行调度。

这种多级队列的组织方式允许不同优先级别的进程以不同的调度策略执行，从而更好地满足不同进程的执行需求。

## 2.4 实验结果测试

实验测试主要通过编写展示进程、调度过程等的相关函数和测试用例进行。下面分别介绍各个调度算法的测试：

### 2.4.1 短进程优先

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS
Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。
尝试新的跨平台 PowerShell https://aka.ms/pscore6

PS C:\zPersonal\Univ\CODE\operating_system_simulate_lab> & 'c:\Users\yigu\vscode\extensions\ms-vscode.cpptools-1.19.9-win32-x64\debugAdapters\bin\win
dowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-15ulj9b5.kx1' '--stdout=Microsoft-MIEngine-Out-i1r3lchu.z15' '--stderr=Microsoft-MIEngine-Error-e
bjs0lmy.sdc' '--pid=Microsoft-MIEngine-Pid-kaxql4g.gfk' '--dbgExe=C:\mingw64\bin\gdb.exe' '--interpreter=mi'
ID      Name      Arrive Time  Running Time  Priority
1        1         10          2             0
2        2        10.2         1             0
3        3        10.4         0.5           0
4        4        10.5         0.3           0
[SFJ] Time: 10, Process 1 begins.
[SFJ] Time: 12, Process 4 begins.
[SFJ] Time: 12.3, Process 3 begins.
[SFJ] Time: 12.8, Process 2 begins.
[Ending] Time: 13.8
CPU Utilization Rate: 0.275362
Process Name      Turnaround time  Response time  Waiting time  Utilization
1                  2                  0              0              0
4                  1.8              1.5            1.5            1.5
3                  2.4              1.9            1.9            1.9
2                  3.6              2.6            2.6            2.6
Average Turnaround Time: 2.45
Average Response Time: 1.5
Average Waiting Time: 1.5
PS C:\zPersonal\Univ\CODE\operating_system_simulate_lab>
```

测试使用课件上的一道例题完成，可以看到，程序输出了正确的调度过程。

### 2.4.2 时间片轮转调度算法

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS
Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。
尝试新的跨平台 PowerShell https://aka.ms/pscore6

PS C:\zPersonal\Univ\CODE\operating_system_simulate_lab> & 'c:\Users\yigu\vscode\extensions\ms-vscode.cpptools-1.19.9-win32-x64\debugAdapters\bin\win
dowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-24eefouk.rhz' '--stdout=Microsoft-MIEngine-Out-t31lrm3k.pto' '--stderr=Microsoft-MIEngine-Error-o
qvgbhyk.den' '--pid=Microsoft-MIEngine-Pid-fcmb53wx.w4j' '--dbgExe=C:\mingw64\bin\gdb.exe' '--interpreter=mi'
ID      Name      Arrive Time  Running Time  Priority
1        1         0           3             0
2        2         1           6             0
3        3         2           4             0
4        4         3           5             0
5        5         4           2             0
[RR] Time: 0, Process 1 begins.
[RR] Time: 3, Process 2 begins.
[RR] Time: 7, Process 3 begins.
[RR] Time: 11, Process 4 begins.
[RR] Time: 15, Process 5 begins.
[RR] Time: 17, Process 2 begins.
[RR] Time: 19, Process 4 begins.
[Ending] Time: 20
CPU Utilization Rate: 1
Process Name      Turnaround time  Response time  Waiting time  Utilization
1                  3                  0              0              0
3                  9                  7              5              5
5                  13                 15             11             11
2                  18                  3              12             12
4                  17                 11             12             12
Average Turnaround Time: 12
Average Response Time: 7.2
Average Waiting Time: 8
PS C:\zPersonal\Univ\CODE\operating_system_simulate_lab>
```

测试同样使用了一个课本例题，可以看到，程序输出了正确的结果。

### 2.4.3 多级队列（有抢占）调度算法



PROBLEMS	OUTPUT	DEBUG CONSOLE	TERMINAL	PORTS	GITLENS
CPU Utilization Rate: 1					
	Process Name	Turnaround time	Response time	Waiting time	Utilization
	scheduler.system	1	0	0	
	kernel.system	9	0	7	
	database.system	24	0	20	
	database.backup	1	0	0	
	printer.system	17	0	15	
	server.network	31	0	28	
	inventory.manager	1	0	0	
	controller.device	39	0	36	
	report.generator	27	0	25	
	ticketing.system	1	0	0	
	security.manager	53	0	50	
	search.engine	34	0	32	
	access.control	1	0	0	
	monitor.system	78	0	74	
	firewall.system	40	0	38	
	manager.app	95	0	89	
	virtualization.manager	42	0	39	
	dashboard.app	68	0	65	
	analytics.system	89	0	84	
	authentication.app	74	0	70	
	API.gateway	46	0	44	
	inventory.system	47	0	44	
	backup.system	99	0	95	
	authentication.system	51	0	48	
	billing.app	87	0	84	
	reporting.app	51	0	49	
	handler.file	139	0	134	
	mail.system	104	0	99	
	load.balancer	94	0	90	
	encryption.module	95	0	90	
	scanner.device	135	0	129	

测试使用了一个 ChatGPT 生成的包含 50 个进程的测试集，上图是测试的部分输出。

```
[PMLFQ] Time: 0, Process database.system begins. Quene: 0
[PMLFQ] Time: 1, Process manager.app begins. Quene: 0
[PMLFQ] Time: 2, Process database.system begins. Quene: 1
[PMLFQ] Time: 3, Process kernel.system begins. Quene: 0
[PMLFQ] Time: 4, Process handler.file begins. Quene: 0
[PMLFQ] Time: 5, Process manager.app begins. Quene: 1
[PMLFQ] Time: 6, Process server.network begins. Quene: 0
[PMLFQ] Time: 7, Process interface.app begins. Quene: 0
[PMLFQ] Time: 8, Process scheduler.system begins. Quene: 0
[PMLFQ] Time: 9, Process database.system begins. Quene: 1
[PMLFQ] Time: 10, Process input.user begins. Quene: 0
[PMLFQ] Time: 11, Process kernel.system begins. Quene: 1
[PMLFQ] Time: 12, Process monitor.system begins. Quene: 0
[PMLFQ] Time: 13, Process handler.file begins. Quene: 1
[PMLFQ] Time: 15, Process controller.device begins. Quene: 0
[PMLFQ] Time: 16, Process manager.app begins. Quene: 1
[PMLFQ] Time: 17, Process printer.system begins. Quene: 0
[PMLFQ] Time: 18, Process scanner.device begins. Quene: 0
[PMLFQ] Time: 19, Process server.network begins. Quene: 1
[PMLFQ] Time: 20, Process analytics.system begins. Quene: 0
[PMLFQ] Time: 21, Process interface.app begins. Quene: 1
[PMLFQ] Time: 22, Process security.manager begins. Quene: 0
[PMLFQ] Time: 23, Process database.system begins. Quene: 1
[PMLFQ] Time: 24, Process backup.system begins. Quene: 0
[PMLFQ] Time: 25, Process input.user begins. Quene: 1
[PMLFQ] Time: 26, Process notification.app begins. Quene: 0
[PMLFQ] Time: 27, Process monitor.system begins. Quene: 1
[PMLFQ] Time: 28, Process database.backup begins. Quene: 0
[PMLFQ] Time: 29, Process controller.device begins. Quene: 1
[PMLFQ] Time: 30, Process scheduler.task begins. Quene: 0
[PMLFQ] Time: 31, Process manager.app begins. Quene: 1
[PMLFQ] Time: 32, Process report.generator begins. Quene: 0
[PMLFQ] Time: 33, Process printer.system begins. Quene: 1
[PMLFQ] Time: 34, Process scanner.device begins. Quene: 1
[PMLFQ] Time: 35, Process dashboard.app begins. Quene: 0
```

上图是程序刚开始执行的部分输出结果。这些输出展示了程序的执行过程。

可以看到，所有进程到达后，优先执行第一个队列 Queue-0（大小为 1），当所有进程执行完第一级队列后，有第二级队列 Queue-1 的进程开始执行（大小为 2）。

```
[PMLFQ] Time: 137, Process dashboard.generator begins. Quene: 1
[PMLFQ] Time: 139, Process backup.manager begins. Quene: 1
[PMLFQ] Time: 141, Process handler.file begins. Quene: 2
[PMLFQ] Time: 143, Process mail.system begins. Quene: 2
[PMLFQ] Time: 145, Process load.balancer begins. Quene: 2
[PMLFQ] Time: 146, Process scheduler.task begins. Quene: 2
[PMLFQ] Time: 150, Process encryption.module begins. Quene: 2
[PMLFQ] Time: 152, Process scanner.device begins. Quene: 2
[PMLFQ] Time: 153, Process logger.system begins. Quene: 2
[PMLFQ] Time: 155, Process scheduler.process begins. Quene: 2
[PMLFQ] Time: 158, Process deployment.manager begins. Quene: 2
[PMLFQ] Time: 159, Process interface.app begins. Quene: 2
[PMLFQ] Time: 160, Process monitoring.app begins. Quene: 2
[PMLFQ] Time: 164, Process billing.system begins. Quene: 2
[PMLFQ] Time: 166, Process input.user begins. Quene: 2
[PMLFQ] Time: 168, Process scheduler.job begins. Quene: 2
[PMLFQ] Time: 172, Process gateway.system begins. Quene: 2
[PMLFQ] Time: 176, Process database.manager begins. Quene: 2
[PMLFQ] Time: 177, Process notification.app begins. Quene: 2
[PMLFQ] Time: 179, Process dashboard.generator begins. Quene: 2
[PMLFQ] Time: 182, Process backup.manager begins. Quene: 2
[PMLFQ] Time: 185, Process scheduler.task begins. Quene: 2
[PMLFQ] Time: 186, Process monitoring.app begins. Quene: 2
[Ending] Time: 187
```

而如图所示，在程序结束前，所有程序都在最后一级队列（大小为 4），直到运行结束。

## 2.5 总结

通过本次实验，我们深入学习了进程调度算法的原理与实现，并通过模拟实验对三种常见的调度算法进行了测试和比较。在短进程优先调度算法（SJF）中，我们发现该算法能够优先执行运行时间较短的进程，从而降低了平均周转时间和平均等待时间；而时间片轮转调度算法（Round Robin）则能够保证每个进程都能获得公平的 CPU 时间片，但可能存在较长的响应时间和较高的平均周转时间；多级队列调度算法（MLFQ）结合了前两种算法的优点，通过多个优先级队列实现了对不同类型进程的不同调度策略，能够在不同场景下灵活应对。

通过对比测试结果，我们可以根据不同的应用场景选择合适的调度算法，以最大程度地提高系统的性能和吞吐量。此外，本次实验还加深了我们对操作系统调度算法的理解，为进一步深入学习和应用提供了坚实的基础。

## 3. 页面置换算法模拟实验

### 3.1 实验目的

页面置换算法模拟实验的目的是通过实际模拟不同页面置换算法的运行过程，评估它们在不同工作负载下的性能表现。具体来说，我们的目标包括：

- 理解不同页面置换算法的工作原理：包括先进先出（FIFO）、最近最少使用（LRU）、时钟（Clock）等经典页面置换算法的基本原理和策略。
- 深入了解每种算法在处理页面置换时的策略和效率：通过实验模拟，探究各种算法在不同场景下的表现，了解它们如何选择页面进行置换以及对系统性能的影响。
- 比较不同页面置换算法的性能差异：通过对比不同算法在相同工作负载下的表现，如缺页率、命中率等指标，为选择适合特定场景的最佳算法提供参考。

### 3.2 实验步骤

本实验的步骤主要包括以下几个方面：

1. 内存设计：设计基本的内存、外存和页表结构。
2. 实现页面置换算法：根据选定的页面置换算法，编写相应的代码实现。  
常见的页面置换算法包括 FIFO、LRU、Clock 等，每种算法都有不同的逻辑和数据结构要求。
3. 编写模拟器：编写一个模拟器程序，模拟内存访问过程和页面置换过程。  
该模拟器需要能够接受指定的工作负载，调用相应的页面置换算法，并输出模拟结果，如缺页率、命中率等。
4. 运行实验：在模拟器中加载选择的工作负载，并运行页面置换算法模拟。  
记录模拟过程中的关键数据，如每个算法的缺页次数、命中次数等。

### 3.3 实验设计

#### 3.3.1 内存结构

本实验内存的设计主要包括内存页面、物理内存和外存三个部分，其中，页面需要考虑外存中的页面和内存中的页面。

```
// 内存页面
struct page
{
    int pageNumber = 0;           // 页号
    int disk_pn = -1;             // 在磁盘中的位置
    int dirty = 0;                // 修改位（在物理内存中）/ 使用标记
    //（在磁盘外存中）
    bool valid = false;           // 有效位（在物理内存中）
    int data[PAGE_SIZE];          // 页面存储的数据

    // 在物理内存中的页面需要一些其他数据
    int pid = -1;                 // 对应进程的id
    int vpn = -1;                 // 对应页表的物理页号
    int in_time = 0;              // 加入物理内存的时间
    int hit_time = 0;             // 上次命中的时间
    int ref_sign = 1;             // Clock 算法中的引用位

    // 初始化一个页面
    page(){
        for(int& d: data) {
            d = 0;
        }
    };
    page(int pageNumber): pageNumber(pageNumber){
        for(int& d: data) {
            d = 0;
        }
    };
};

// 物理内存
struct memory
{
    page data[P_MEM_SIZE];        // 物理内存中的数据

    // 初始化物理内存
    memory(){
        for(int i=0; i<P_MEM_SIZE; i++) {
            page newPage(i);
            data[i] = newPage;
        }
    }
};
```

```

    };
};
memory PhysicalMemory;

struct disk
{
    page data[DISK_SIZE];           // 磁盘中的数据

    // 初始化物理内存
    disk(){
        for(int i=0; i<P_MEM_SIZE; i++) {
            page newPage(i);
            data[i] = newPage;
            data[i].disk_pn = i;
        }
    };
};
disk Disk;                         // 磁盘/外存

```

page 结构体表示内存中的一页或磁盘中的一页。

memory 结构体表示物理内存，包含一个包含 page 结构体的数组 data，表示内存中的页面数据。

disk 结构体表示磁盘（或外存），包含一个包含 page 结构体的数组 data，表示磁盘中的页面数据。初始化时，每个页面的 disk\_pn 被设置为对应的索引值。

### 3.3.2 页表和进程结构

本实验中每个进程对应一个页表，结构体的设计如下：

```

// 页表项
struct pt_item
{
    bool valid = false;    // 页表项有效位

    int vpn = 0;           // 虚拟页号
    int ppn = 0;           // 物理页号
    int disk_pn = 0;       // 数据在磁盘中的位置

    int in_time = 0;       // 加入物理内存的时间
    int hit_time = 0;      // 上次命中的时间
    int ref_sign = 1;      // Clock 算法中的引用位
    pt_item(){};
};

```

```

};

// 页表
struct pt
{
    int size = 0;
    vector<pt_item> items; // 页表项

    // 初始化一个页表
    pt(){};
    pt(int size): size(size){
        for (int i=0; i<size; i++) {
            pt_item new_pti = {};
            new_pti.vpn = i; // 分配虚拟页号
            items.push_back(new_pti);
        }
    };
};

// 进程
struct process
{
    int pid = -1; // pid
    string name = "undefined"; // name of process

    int size = 0; // 进程页表的大小
    pt process_pt; // 进程的页表

    process(){};
    process(int size): size(size) {
        pt a(size);
        process_pt = a;
    };
};

```

pt\_item 结构体表示页表中的一个表项， pt 结构体表示页表， process 结构体表示一个进程。这些数据结构用于管理进程的页表信息，包括虚拟页号到物理页号的映射关系，以及页面在磁盘中的位置等信息。这种设计允许模拟操作系统中的进程管理功能，包括页面置换算法对页表的影响。

### 3.3.3 数据的输入



各个进程访问内存的顺序仍然记录在 txt 文件中。加载程序时，程序将文本加载到这样的特殊结构中：

```
// 表示各个进程的页表访问顺序
struct access
{
    int pid = -1;    // 进程
    int logicAddress = 0;    // 逻辑地址
    int op = 0;    // 操作 ( 0 读 | 1 写 )
    int wd = 0;    // 写操作写入的数据
};
```

程序开始时，我们让所有访问信息保存在一个 access 结构体的数组中，然后按顺序执行。

### 3.3.4 页面置换算法

与进程模拟实验的调度类似，不同调度算法间仅有极少细节是不同的，因此，这次我将所有算法编写在同一个函数中。

我们可以这样分解页面调度的过程：

- 检查页表是否命中。
  - 如果命中：
    - ◆ 更新页表项信息。
  - 如果没有命中，但地址是有效的，这时需要从外存调页：
    - ◆ 寻找一个新的空闲物理块。
    - ◆ 如果物理块都被占用，需要调用调度算法：
      - 根据算法的不同，确定需要被替换的物理页号。
      - 将需要替换的物理页放回外存，从外存中取新的物理页。
      - 更新页表信息

可以看出，只有“确定被替换的物理页号”这一过程与选择的调度算法有关，各算法其他部分是完全一样的。我们在调度函数中加入一个 int &al 的参数选择算法，再使用条件语句就可以把所有算法写在一起。

```
// 使用不同算法实现调度（通用函数）
void schedule(const access& a, const int &al, vector<process>& p) {
    pt& pageTable = p[a.pid].process_pt;    // 对应进程的页表
    int vpn = a.logicAddress / PAGE_SIZE;    // 访问虚拟页号
    int bias = a.logicAddress % PAGE_SIZE;    // 访问的页内偏移量
```

```

int ppn = -1;    // 物理页号

// 检查页表是否命中
if (vpn < pageTable.size && pageTable.items[vpn].valid){
    // 页表命中
    ppn = pageTable.items[vpn].ppn;
    // 更新命中时间
    PhysicalMemory.data[ppn].hit_time = tick;
    // 更新CLOCK 算法的参考位
    PhysicalMemory.data[ppn].ref_sign = 1;
    cout << tick << "\t" << a.pid << "\t" << a.logicAddress << "\t"
<< vpn << "\tHIT\t" << ppn << "\t" <<
PhysicalMemory.data[ppn].data[bias] << endl;
}
else if(vpn < pageTable.size) {
    // 页表未命中, vpn 有效
    cout << tick << "\t" << a.pid << "\t" << a.logicAddress << "\t"
<< vpn << "\tMISS\t" << endl;

    // 寻找一个新的空闲块
    ppn = getEmptyPage();
    if(ppn == -1) {
        // 不存在空闲页, 调用磁盘调度算法
        ppn = 0;
        // 找到最早加入的物理块
        switch (al)
        {
            case 0:
                // FIFO
                for(int i=0; i<P_MEM_SIZE; i++){
                    page pg = PhysicalMemory.data[i];
                    ppn = (pg.in_time <
PhysicalMemory.data[ppn].in_time) ? i : ppn;
                }
                break;
            case 1:
                // RS
                ppn = generateRandomNumber(P_MEM_SIZE);
                break;
            case 2:
                // LRU
                for(int i=0; i<P_MEM_SIZE; i++){
                    page pg = PhysicalMemory.data[i];

```



```

        ppn = (pg.hit_time <
PhysicalMemory.data[ppn].hit_time) ? i : ppn;
    }
    break;
case 3:
    // CLOCK
    ppn = -1;
    while (true)
    {
        page& c = PhysicalMemory.data[clk];
        if(!c.valid || c.ref_sign == 0){
            // 物理块无效或者ref 位为0, 替换这一个物理快, clk+1
            ppn = clk;
            clk = (clk + 1) % P_MEM_SIZE;
            break;
        }
        else {
            // 物理块在内存中, 且ref 位为1, ref 位置0, clk+1,
继续循环

            c.ref_sign = 0;
            clk = (clk + 1) % P_MEM_SIZE;
        }
    }
    break;
default:
    return;
}
}

// 从外存向ppn 调页
int old_addr = (PhysicalMemory.data[ppn].valid)?
PhysicalMemory.data[ppn].disk_pn : -1;
// show(PhysicalMemory);
int new_addr = pageTable.items[vpn].disk_pn;
replacePage(PhysicalMemory, Disk, ppn, old_addr, new_addr);
updatePageTable(p[a.pid], vpn, ppn);
cout << "[PageTable] pid: " << a.pid << ", Updated: Vpn-" << vpn
<< ", Ppn-" << ppn << ". " << endl;
} else {
    // 页表未命中, vpn 无效
    cout << tick << "\t" << a.pid << "\t" << a.logicAddress << "\t"
<< vpn << "\tMISS\t" << endl;
    cout << "[PageFault] Failed to handle this page missing, as the
virtual page number is too large." << endl;

```

```
}  
}
```

在确定被替换的物理页号时：

1. FIFO (First In, First Out):

- 需要替换的物理页号是最早加入物理内存的页号，即物理内存中入队时间最早的页。

2. RS (Random Selection):

- 随机选择一个物理页号进行替换，通过生成随机数在物理内存中选择一个页号。

3. LRU (Least Recently Used):

- 需要替换的物理页号是最近未被访问的页号，即物理内存中上次被访问时间最早的页。

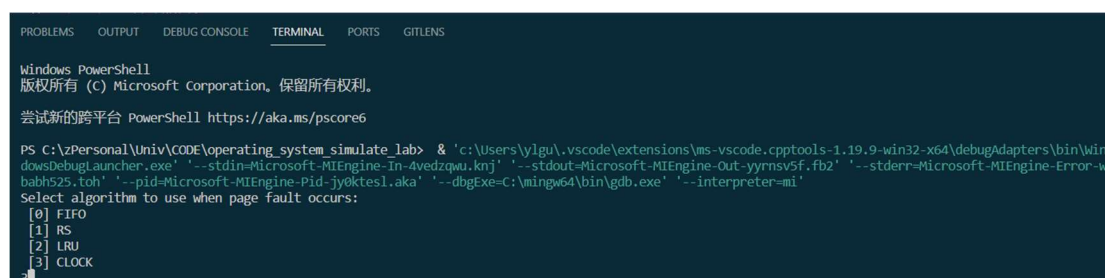
4. CLOCK:

- CLOCK 算法维护一个时钟指针 `clk`，指向物理内存中的页面。
- 遍历物理内存，如果遇到一个页面的 `valid` 为 `false`，或者 `ref_sign` 为 0，则选择该页进行替换，将 `ref_sign` 设置为 0 并将时钟指针移至下一个页面。
- 如果遍历一轮都没有找到合适的页面，则继续遍历，直到找到一个符合条件的页面为止。

程序根据选择的算法不同，会采用相应的方法确定被替换的物理页号，并进行页面的替换操作。

### 3.4 测试

这次仅需要一个测试函数即可完成测试，在测试前我们选择不同的调度算法：



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS  
Windows PowerShell  
版权所有 (C) Microsoft Corporation。保留所有权利。  
尝试新的跨平台 PowerShell https://aka.ms/pscore6  
PS C:\zPersonal\Univ\CODE\operating_system_simulate_lab> & 'c:\Users\y1gu\.vscode\extensions\ms-vscode.cpptools-1.19.9-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-4vedzqwu.knj' '--stdout=Microsoft-MIEngine-Out-yyrns5f.fb2' '--stderr=Microsoft-MIEngine-Error-wbabh525.toh' '--pid=Microsoft-MIEngine-Pid-jy0ktes1.aka' '--dbgExe=C:\mingw64\bin\gdb.exe' '--interpreter=mi'  
Select algorithm to use when page fault occurs:  
[0] FIFO  
[1] RS  
[2] LRU  
[3] CLOCK
```

以 CLOCK 算法为例，我编写了一个包含多个大小不同的进程的测试用例，页

表大小为 4，不同进程按照预定的顺序访问页表。

```
2.memory > test_access.txt
You, 3 weeks ago | 1 author (You)
1  *TESTSET ACCESS
2  *INPUT:
3  * 0 PROCESS PID
4  * 8 ADDRESS
5  * 0 OPERATION CODE
6  * 0 DATA TO WRITE
7  *MISS
8  0 8 0
9  *HIT
10 0 8 0
11 *MISS
12 0 16 0
13 0 24 0
14 0 0 0
15 *HIT
16 0 8 0
17 0 0 0
18 *MISS REPLACE PPN 0 VPN 1
19 0 33 0
20 *MISS REPLACE PPN 1 VPN 2
21 0 8 0
22 *HIT
23 0 0 0
24 *MISS REPLACE PPN 2 VPN 0
25 1 0 0
```

程序首先输出访问的结果，包括进程、时间、地址、页号、是否命中和访问数据等信息。

```

CLOCK:
Time    Pid    Address VPN    Result PPN    Data
0       0       8       1       MISS
[PageTable] pid: 0, Updated: Vpn-1, Ppn-0.
1       0       8       1       HIT    0       6
2       0       16      2       MISS
[PageTable] pid: 0, Updated: Vpn-2, Ppn-1.
3       0       24      3       MISS
[PageTable] pid: 0, Updated: Vpn-3, Ppn-2.
4       0       0       0       MISS
[PageTable] pid: 0, Updated: Vpn-0, Ppn-3.
5       0       8       1       HIT    0       6
6       0       0       0       HIT    3       0
7       0       33      4       MISS
[PageTable] pid: 0, Updated: Vpn-1 to Invalid.
[PageTable] pid: 0, Updated: Vpn-4, Ppn-0.
8       0       8       1       MISS
[PageTable] pid: 0, Updated: Vpn-2 to Invalid.
[PageTable] pid: 0, Updated: Vpn-1, Ppn-1.
9       0       0       0       HIT    3       0
10      1       0       0       MISS
[PageTable] pid: 0, Updated: Vpn-3 to Invalid.
[PageTable] pid: 1, Updated: Vpn-0, Ppn-2.
11      1       0       0       HIT    2       1
12      1       8       1       MISS

```

程序执行结束后，显示内存和页表的当前状态。

```

[PageTable] pid: 2, Updated: Vpn-1, Ppn-0.
[MEMORY SHOW]
Valid  Page  Process Vpn    Disk  Data
1      0      2      1      8      8      8      8      8      8      8      8
1      1      3      1      4      4      4      4      4      4      4      4
1      2      2      0      2      2      2      2      2      2      2      2
1      3      3      3      62     62     62     62     62     62     62     62
Process ID: 0, Name: system

```

```

0      14     0     0
0      15     0     0

Process ID: 2, Name: desktop.layout
Page Table:
Valid  VPN    PPN    Hits time
1      0      2      21
1      1      0      23
0      2      0      0
0      3      0      0
0      4      0      0
0      5      0      0
0      6      0      0
0      7      0      0

Process ID: 3, Name: printer
Page Table:
Valid  VPN    PPN    Hits time
0      0      0      13
1      1      1      14
0      2      2      15
1      3      3      16

```

可以看到，现在内存中的 4 个页面分别对应进程 2 的两个页面和进程 3 的两个页面，而其他进程的页面和进程 2、进程 3 的其他页面都不在内存中。

### 3.5 总结

在本实验中，我们介绍了页面置换算法的模拟实验，并实现了常见的四种页面置换算法：FIFO、Random Selection (RS)、Least Recently Used (LRU) 和 CLOCK。通过模拟实验，我们可以深入理解这些页面置换算法的工作原理和性能特点。

在实验过程中，我们首先设计了表示页面、物理内存、磁盘、页表项、页表和进程的数据结构。然后，我们实现了一个通用的调度函数，该函数可以根据选择的页面置换算法进行页面的调度和替换操作。在确定被替换的物理页号时，每种算法都有不同的策略，如 FIFO 选择最早加入物理内存的页面，而 LRU 选择最近未被访问的页面。

通过实验结果的测试和分析，我们可以评估每种页面置换算法的性能，并比较它们在不同场景下的表现。例如，我们可以比较它们的命中率、缺页率以及对系统整体性能的影响。这些评估结果可以帮助我们选择合适的页面置换算法，以优化系统的内存管理和性能。

总的来说，页面置换算法是操作系统中重要的一部分，对系统的性能和资源利用率有着重要的影响。通过本次实验，我们加深了对页面置换算法的理解，并学习了如何使用模拟实验来评估和比较不同算法的性能，从而为操作系统的设计和优化提供了有益的参考。

## 4. 文件系统模拟实验

### 4.1 实验目的

文件系统模拟实验的目的是通过模拟文件系统的基本操作，加深对文件系统原理的理解，并探索不同文件系统设计对性能和功能的影响。具体目的包括：

- 理解文件系统的基本概念和组成部分，如文件、目录、索引节点等。
- 掌握文件系统的基本操作，包括文件的创建、读取、写入、删除，目录的创建、删除、重命名等。
- 研究不同的文件系统设计，比如 FAT、NTFS、EXT 等，了解它们的特点、优缺点以及适用场景。
- 分析文件系统的性能指标，如读写速度、存储利用率、数据安全性等，以及不同设计对这些指标的影响。

通过实验，可以加深对文件系统的工作原理和设计思想的理解，为系统管理员和开发人员提供设计、配置和优化文件系统的参考依据。

### 4.2 实验步骤

文件系统模拟实验的步骤如下：

1. 设计文件系统结构：设计文件系统的基本结构，包括索引节点、目录项、文件等。
2. 实现文件系统操作函数：编写代码实现文件系统的基本操作函数，包括文件的创建、读取、写入、删除，目录的创建、删除、重命名等。这些函数的实现需要根据文件系统的结构和算法来进行。
3. 构建文件系统模拟环境：创建一个模拟的文件系统环境，包括虚拟磁盘、文件、目录等，并初始化文件系统的相关参数和数据结构。
4. 测试文件系统功能：编写测试用例，验证文件系统的各项功能是否正常工作，包括文件的读写操作、目录的管理操作、文件系统的容量管理等。测试用例需要覆盖文件系统的各个方面，确保其功能的完整性和稳定性。

### 4.3 实验设计

### 4.3.1 基本结构

文件系统主要包括文件夹和文件索引两种结构。文件夹用于存放文件和其他文件夹的 ID，而文件索引存放文件名、文件在内存中的位置等信息。

```
// 文件索引
struct file
{
    // 编号
    int id = -1;
    // 文件名
    std::string fileName = "undefined";
    // 文件大小
    int size = 0;
    // 文件的起始地址
    int head = 0;
    // 例如, head = 0, size = 16, 则文件占用内存的[0]到[15]的空间

    // 文件索引项的初始化
    file(int size, int head, string name = "newFile", int id =
globalFileId) : size(size), head(head), id(id), fileName(name)
    {
        globalFileId = max(globalFileId+1, id+1);
        cout << "[SYSTEM] A new file [" << id << "]" << name << " is
created: from address (" << head << ") to (" << head + size - 1 << ").
" << endl;
    }
};

// 树状文件目录的文件夹
struct folder
{
    // 编号
    int id = -1;
    // 文件夹名
    string folderName = "undefined";

    // 存放下级文件夹的id
    vector<int> subFolders = {};
    // 存放文件夹内文件的id
    vector<file> files = {};

    // 文件夹的初始化
```

```

    folder(string name = "newFolder", int id = globalFolderId) : id(id),
folderName(name)
    {
        globalFolderId = max(globalFolderId+1, id+1);
        cout << "[SYSTEM] A new folder [" << id << "]"_ << name << " is
created. " << endl;
    }
};

```

文件索引（file 结构体）包括文件的编号、文件名、文件大小和在内存中的起始地址。文件夹（folder 结构体）包括文件夹的编号、文件夹名、下级文件夹的编号列表和文件夹内文件的编号列表。这样的设计可以方便地组织文件和文件夹，并通过 ID 进行索引和访问。

此外，程序引入了一些全局变量，便于后续算法的实现。

```

// 当前文件夹的id
int currentFolderId = 0;
// 全局文件的id
int globalFileId = 0;
// 全局文件夹的id
int globalFolderId = 0;
// 表示用到内存尾端的变量
int memoryBack = 0;

```

通过为文件和文件夹定义统一的结构，我们可以轻松地扩展文件系统，同时全局变量可以追踪文件和文件夹的创建情况，并确保每个元素都有唯一的标识符。

#### 4.3.2 文件操作的设计

文件操作主要包括文件的创建、读写、删除、重命名等，文件夹操作主要包括文件夹的创建、删除、重命名等。此外，文件系统还需要其他操作，如切换当前路径、获取指令帮助等。

文件/文件夹操作的过程比较相似，下面讲解比较有代表性的文件创建和删除操作。

需要说明的是，所有文件夹保存在这样一个向量中，文件夹中包括文件索引。

```

// 所有文件夹
vector<folder> folders = {};

```

下面是创建一个文件的过程：

```

// 新建文件

```



```

int newFile(const string &name, const int &size, const int &dirId, int
head, int id, vector<folder> &fs)
{
    folder &directory = fs[getFolder(dirId)];
    // 如果 head 没有输入, 计算新文件的起始地址
    if (head < 0)
    {
        head = memoryBack;
    }
    if (head + size < MEMORY_SIZE)
    {
        // 文件没有超出内存大小限制
        // 创建文件
        file newFile(size, head, name, id);
        // 添加文件到现有路径
        directory.files.push_back(newFile);
        // 更新内存尾端
        memoryBack = head + size;
    }
    else
    {
        // 合并文件碎片后继续尝试放入
        mergeMemory();
        head = memoryBack;
        if (head + size < MEMORY_SIZE)
        {
            // 碎片清除后可以创建文件
            // 创建文件
            file newFile(size, head, name, id);
            // 添加文件到现有路径
            directory.files.push_back(newFile);
            // 更新内存尾端
            memoryBack = head + size;
        }
        else
        {
            cout << "[Error] Failed to create file, as the memory is not
big enough. " << endl;
        }
    }
    return head;
}

```

newFile 函数接受文件名、文件大小、所在文件夹的 ID、起始地址、文件的

编号以及文件系统的向量作为参数。

- 首先，通过 `getFolder(dirId)` 函数获取对应文件夹的引用，存储在 `directory` 中。如果没有指定起始地址 (`head`)，则将起始地址设置为当前内存尾端 `memoryBack`。接着，检查文件大小加上起始地址是否超出了内存大小限制。如果没有超出，就可以创建新文件。
- 如果超出了内存大小限制，则尝试合并内存碎片后再次尝试创建文件。
- 如果仍然无法创建文件（即内存不足），则输出错误信息。合并碎片的算法将在稍后讲解。
- 最后，程序返回文件的起始地址。

而文件的删除过程也与之相似：主要可以描述为 1) 找到文件所在的文件夹，2) 找到文件，3) 删除文件索引。

```
// 删除文件
bool deleteFile(int fileId, vector<folder> &fs = folders){
    bool found = false; // Flag to indicate if the file is found and
    deleted

    // Iterate through all folders
    for (folder &f : fs)
    {
        auto it = find_if(f.files.begin(), f.files.end(), [fileId](const
        file &fi)
            { return fi.id == fileId; });
        if (it != f.files.end())
        {
            // Remove the file from the folder's file list
            f.files.erase(it);
            found = true; // Set found flag to true
            cout << "[SYSTEM] File with ID [" << fileId << "] deleted
            successfully." << endl;
        }
    }

    // If the file is not found in any folder
    if (!found)
    {
        cout << "[Error] File with ID [" << fileId << "] not found." <<
        endl;
        return false; // Return false to indicate failure
    }
}
```

```
    return true; // Return true to indicate success
}
```

在这个系统中，文件删除后对应的内存会被释放，但后续的文件不会被主动提前。也就是说，文件并没有在内存中被覆盖。只有在创建文件时，内存不够用的情况下才会尝试覆盖之前的文件。

#### 4.3.3 内存管理设计

创建文件时，程序如果发现当前内存尾端 `memoryBack` 到内存末尾 `MEM_SIZE` 的空间不够时，会调用 `Merge` 函数，尝试合并碎片。碎片是文件系统中之前删除的文件留下的内存空间间隙。由于删除文件时不改变内存本身，该文件系统中允许碎片的存在。

下面是算法的执行过程：

```
// 合并文件碎片的尝试
bool mergeMemory(vector<folder> &fs = folders, memory &mem = Disk)
{
    // 遍历文件夹，获取所有文件的信息
    vector<Merge> fileInfo = {};
    for (const folder f : fs)
    {
        for (const file fi : f.files)
        {
            Merge n(fi.id, f.id, fi.head, fi.size);
            fileInfo.push_back(n);
        }
    }
    // 按开始地址排序获取到的文件
    sort(fileInfo.begin(), fileInfo.end(), compareMerge);
    // 没有文件，不需要合并碎片
    if (fileInfo.empty())
    {
        return true;
    }

    // 检查每两个内存上连续的文件之间是否存在碎片，如果存在，将第二个文件提前
    if (fileInfo[0].start != 0)
    {
        // 如果第一个文件前有碎片
        int forward = fileInfo[0].start;
        // 将第一个文件提前
    }
}
```

```

        forwardMemory(forward, fileInfo[0].start, fileInfo[0].size,
mem);
        fileInfo[0].start -= forward;
        fileInfo[0].dirty = true;
    }
    for (int i = 1; i < fileInfo.size(); ++i)
    {
        // 如果其他两个文件间有碎片空间
        if (fileInfo[i].start > fileInfo[i - 1].start + fileInfo[i -
1].size)
        {
            // 计算空间的大小
            int gapSize = fileInfo[i].start - (fileInfo[i - 1].start +
fileInfo[i - 1].size);
            // 将第i 个文件提前
            forwardMemory(gapSize, fileInfo[i].start, fileInfo[i].size,
mem);

            fileInfo[i].start -= gapSize;
            fileInfo[i].dirty = true;
        }
    }

    // 根据merge 结构体, 修改文件夹向量内的内存索引
    for (const Merge &m : fileInfo)
    {
        // 如果merge 结构体被修改过
        if (m.dirty)
        {
            // 找到id 对应的文件夹
            vector<file> &files = fs[getFolder(m.folderId)].files;
            for (file &f : files)
            {
                // 找到id 对应的文件
                if (f.id == m.fileId)
                {
                    // 更新文件索引
                    f.head = m.start;
                }
            }
        }
    }

    memoryBack = fileInfo.back().start + fileInfo.back().size;
    cout << "[Merge] Fragments merged, now the memory is used: " <<
memoryBack << " / " << MEMORY_SIZE << ". " << endl;

```

```
    return true;
}
```

- 首先，函数遍历文件夹中的所有文件，将文件的信息存储在 fileInfo 向量中，每个文件信息包括文件编号、所在文件夹编号、起始地址和文件大小。
- 接着，按照文件的起始地址对 fileInfo 向量进行排序，以便后续合并操作。
- 如果 fileInfo 为空，即没有文件，那么就不需要合并碎片，直接返回 true。
- 否则，遍历 fileInfo 向量，检查每两个连续文件之间是否存在碎片空间。
- 如果存在碎片空间，则将后一个文件提前填充到碎片空间，即向前移动文件。
- 最后，根据合并后的结果更新文件夹向量中文件的内存索引，并更新内存尾端 memoryBack，输出合并碎片后的内存使用情况。

合并碎片后，memoryBack 到内存末尾的空间便是当前内存中所能存放的最大文件的大小。如果再次发现空间不足，将会提示创建失败。

#### 4.3.4 数据的输入

程序支持将文件系统数据保存到系统外的.txt 文件中，保存的格式如下：

```
3.file > data > ≡ tree.txt
    You, 3 weeks ago | 1 author (You)
  1  in 0 file 0 systemconfig 529 17
  2  in 0 folder 1 user
  3  in 1 file 1 userconfig 546 14
  4  in 1 file 3 userinfo_gutian 560 34
  5  in 1 folder 2 user_gutian
  6  in 2 file 2 gutian 0 512
  7  in 2 file 4 mergetest 528 1
  8  in 2 file 9 gutianInfo 594 21
  9  in 2 file 10 gpa 615 8      You, 3 we
 10
```

ID 为 0 的是根目录，在系统开启时自动创建，因此无需再创建。该系统可以

读入一系列的文件夹和文件索引信息。例如：in 0 file 0 的意思是：在 0 号文件夹中有 0 号文件，后续的内容是文件名和文件在内存中的位置、大小等信息。

在用户界面中，使用/load 命令可以加载文件，使用/save 命令可以将当前文件索引和内存保存到外部.txt 文件中。下一节将介绍用户界面。

### 4.3.5 界面设计

用户界面用于用户和文件系统交互，打开文件系统时，展示如下界面：

```
[SYSTEM] A new folder [0]_root is created.  
  
[fileSystem] welcome to file system!  
/help: get help.  
:) [0]_root >> █
```

此时，用户处于根目录下。

用户界面可以解析用户输入的指令，并根据指令给出相应的操作。实现这一功能的是一个名为 parsing 的函数：

```

// 命令的解析, 如果退出界面则返回true, 否则返回false.
bool parsing(string &line)
{
    // 使用 stringstream 分割命令
    stringstream ss(line);
    string action;           // 命令操作
    vector<string> options = {}; // 命令的其他参数

    // 提取命令和参数
    ss >> action;
    string option;
    while (ss >> option)
    {
        options.push_back(option);
    }

    // 解析和执行命令
    if (action == "/help") ...
    else if (action == "/dbg") ...
    else if (action == "/quit") ...
    else if (action == "/load") ...
    else if (action == "/save") ...
    else if (action == "/list") ...
    else if (action == "/new") ...
    else if (action == "/cd") ...
    else if (action == "/mov") ...
    else if (action == "/dlt") ...
    else if (action == "/dltf") ...
}

```

这个函数使用条件语句分析用户的输入，并调用相应的函数，循环获取、执行用户的指令。

## 4.4 测试

测试过程模拟一个用户使用文件系统的真实过程，同时，我也编写了一个简单的文件目录，可以加载到文件系统中进行测试。

我们先打开用户界面，尝试通过/help 指令获取用户帮助。

```

[fileSystem] Welcome to file system!
/help: get help.
:) [0]_root >> /help

[HELP]
[***DEBUG COMMANDS***]:
/dbg m: Show current memory.
/dbg f: Show files.
/dbg g: Test fragment merging algorithm.
[***SYSTEM COMMANDS***]:
/help: Get help.
/quit: Quit the file system.
/load: Get files, memory data and folders from /data.
/save: Store data and directories to /data folder.
[***LIST COMMANDS***]:
/list: List current directory.
/list [+id]: List a specific directory. (eg. /list 5)
[***DIRECTORY COMMANDS***]:

```

接下来，我们使用/load 指令读入已经提前编写好的文件系统。

```

:) [0]_root >> /load
[SYSTEM] File system reset successfully.
[SYSTEM] A new folder [0]_root is created.
[SYSTEM] A new file [0]_systemconfig is created: from address (529) to (545).
[SYSTEM] A new folder [1]_user is created.
[SYSTEM] A new file [1]_userconfig is created: from address (546) to (559).
[SYSTEM] A new file [3]_userinfo_gutian is created: from address (560) to (593).
[SYSTEM] A new folder [2]_user_gutian is created.
[SYSTEM] A new file [2]_gutian is created: from address (0) to (511).
[SYSTEM] A new file [4]_mergetest is created: from address (528) to (528).
[SYSTEM] A new file [9]_gutianInfo is created: from address (594) to (614).
[SYSTEM] A new file [10]_gpa is created: from address (615) to (622).
[SYSTEM] File index loaded successfully from ./data/tree.txt
:) [0]_root >> █

```

我们看到，有一串文件和文件夹已经被创建。我们现在可以访问它们。

在根目录中，我们输入/list，可以看到当前目录有一个文件和一个文件夹。

```

[SYSTEM] File index loaded successfully
:) [0]_root >> /list
List: [0] root
Folder ID: 0, Name: root
Files:
ID      Size   Head   Name
0       17     529    systemconfig

Folders In root:
[1]      user

```



我们使用/cd 1，可以切换到 1 号文件夹。我们再次/list 这个文件夹。

```
Folders In root:
[1]      user
:) [0]_root >> /cd 1
:) [1]_user >> /list
List: [1] user
Folder ID: 1, Name: user
Files:
ID      Size    Head    Name
1       14      546     userconfig
3       34      560     userinfo_gutian

Folders In user:
[2]      user_gutian
:) [1]_user >> █
```

可以看到，这个文件中有一个大小为 14 的 userconfig 文件，其 id 为 1，我们用/rd 1 读取这个文件。

```
[2]      user_gutian
:) [1]_user >> /rd 1
user_01_gutian
:) [1]_user >> █
```

文件系统显示了这个文件的内容。

接下来，我们测试文件的创建，删除功能。我们删除 1 号文件，发现它已经被移出文件系统。

```
:~
:) [1]_user >> /dlt 1
[SYSTEM] File with ID [1] deleted successfully.
:) [1]_user >> /list
List: [1] user
Folder ID: 1, Name: user
Files:
ID      Size    Head    Name
3       34      560     userinfo_gutian

Folders In user:
[2]      user_gutian
:) [1]_user >> █
```

接下来，我们可以创建一个内容为“test”的新文件 new。

```
[2]      user_gutian
:) [1]_user >> /new -s new test
[SYSTEM] A new file [11]_new is created: from address (623) to (626).
:) [1]_user >> █
```

系统为它分配了 623 到 626 的内存空间，现在我们可以读取这个新建的文件。

```
:~  
:) [1]_user >> /rd 11  
test  
:) [1]_user >> █
```

最后，我们还可以测试碎片合并。由于刚才我们删除了 1 号文件，因此合并后这个 11 号 new 文件会被提前。我们来试试是不是这样。

我们使用 debug 命令/dbg m， 现在系统显示内存的内容。

```
:~  
:) [1]_user >> /dbg m  
// // // // // // // // // // // // // // // //  
// // // // // // // // // // // // // // // //  
// // // // // // // // // // // // // // // //  
// // // // // // // // // // // // // // // //  
// // // // // // // // // // // // // // // //  
// // // // // // // // // // // // // // // //  
// // // // // // // // // // // // // // // //  
// // // // // // // // // // // // // // // //  
// // // // // // // // // // // // // // // //  
// // // // // // // // // // // // // // // //
```

我们再删除大小为 512 的 2 号文件，然后使用/dbg g 指令进行碎片合并，可以看到，系统提示碎片合并完成，内存开头的内容已经发生了改变。

```
:~  
:) [1]_user >> /dlt 2  
[SYSTEM] File with ID [2] deleted successfully.  
:) [1]_user >> /dbg g  
[Merge] Fragments merged, now the memory is used: 85 / 1024.  
:) [1]_user >> /dbg m  
/systemconfig.fi  
league:21 | sex:m  
ale | birthday:1  
total:1024  
free:939
```

合并完成后，再使用/list 查看 11 号文件的位置，发现它已经被提前到 81。

```
:~  
:) [1]_user >> /list  
List: [1] user  
Folder ID: 1, Name: user  
Files:  
ID      Size  Head  Name  
3       34    18    userinfo_gutian  
11      4      81    new  
  
Folders In user:  
[2] user_gutian  
:) [1]_user >> █
```

最后，我们使用/save 指令将这个系统的数据重新保存到外部，系统现在已经自动切换到根目录。

```
[2] user_gutian
:) [1]_user >> /save
[SYSTEM] File index saved successfully to ./data/tree.txt
:) [0]_root >>
```

查看外部文件，可以确认修改完成。刚才创建的 11 号文件位于第 4 行。

```
3.file > data > ≡ tree.txt
You, 1 minute ago | 1 author (You)
1  in 0 file 0 systemconfig 1 17
2  in 0 folder 1 user
3  in 1 file 3 userinfo_gutian 18 34
4  in 1 file 11 new 81 4
5  in 1 folder 2 user_gutian
6  in 2 file 4 mergetest 0 1
7  in 2 file 9 gutianInfo 52 21
8  in 2 file 10 gpa 73 8
9
```

## 4.5 总结

通过文件系统模拟实验，我们深入理解了文件系统的基本概念、组成部分以及操作。在设计和实现文件系统的过程中，我们掌握了文件的创建、读写、删除，文件夹的创建、删除、重命名等基本操作函数。通过对不同文件系统设计的研究，我们了解了各种文件系统的特点、优缺点以及适用场景。同时，我们分析了文件系统的性能指标，如读写速度、存储利用率和数据安全性，以及不同设计对这些指标的影响。

在实验中，我们设计了文件系统的基本结构，包括文件和文件夹的索引节点，以及文件夹中包含文件和下级文件夹的信息。通过全局变量追踪文件和文件夹的创建情况，确保每个元素都有唯一的标识符。在文件操作的设计中，我们实现了文件的创建、删除等功能，并且支持文件的读写操作。内存管理方面，我们实现了合并碎片的算法，以最大程度地利用内存空间。

在数据的输入方面，我们支持将文件系统数据保存到外部文件中，可以通过

特定格式的.txt 文件进行保存和读取，从而实现数据的持久化。用户界面设计简洁明了，能够清晰地展示文件系统的当前状态和支持的操作，用户可以通过界面进行文件系统的交互操作。

通过测试过程，我们验证了文件系统的各项功能是否正常工作，包括文件的创建、读写、删除，目录的管理，以及内存的管理等。测试用例覆盖了文件系统的各个方面，确保了其功能的完整性和稳定性。

综上所述，文件系统模拟实验加深了我们对文件系统原理和设计的理解，为系统管理员和开发人员提供了设计、配置和优化文件系统的参考依据。

## 5. 实验总结

### 5.1 实验结果

在进程调度模拟实验中，我们成功地模拟了多种常见的进程调度算法，包括最短作业优先（SJF）、优先级调度、轮转调度（RR）等。通过模拟实验，我们深入理解了各种调度算法的工作原理和特点，并对它们的性能进行了比较分析，为选择合适的调度算法提供了参考依据。

在内存置换模拟实验中，我们实现了常见的页面置换算法，如先进先出算法（FIFO）、最近最久未使用算法（LRU）等。通过模拟实验，我们熟悉了各种页面置换算法的思想和实现方法，并对它们的性能进行了评估和比较，深入了解了内存管理的重要性和挑战。

在文件系统模拟实验中，我们设计并实现了一个简单的文件系统模拟器，包括文件和文件夹的基本操作，如创建、读写、删除、重命名等。通过模拟实验，我们加深了对文件系统的理解，掌握了文件系统的基本概念和操作方法，为文件系统的设计和优化提供了实践基础。

### 5.2 问题和解决方案

在完成这三个实验的过程中，我们也遇到了一些问题，但通过不断探索和解决，取得了如上所述的实验结果和收获。

#### 5.2.1 问题

**算法实现复杂度：** 在进程调度和内存置换模拟实验中，某些算法的实现比较复杂，尤其是 LRU 算法和 OPT 算法，需要维护大量的数据结构和状态信息。

**资源管理困难：** 在文件系统模拟实验中，需要合理管理内存资源，包括文件的分配和释放，以及内存碎片的处理，这给实现带来了一定的困难。

**调试和测试难度：** 由于涉及到多个算法和复杂的系统设计，调试和测试过程相对较为繁琐，需要仔细排查代码逻辑和数据结构是否正确。

#### 5.2.2 解决方案：

**分步实现和调试：** 针对复杂算法，采用分步实现和调试的策略，先实现基本

功能，逐步完善和优化，确保每个功能模块的正确性。

**模块化设计：** 采用模块化设计思想，将整个系统划分为多个模块，每个模块负责一个特定的功能，降低系统的复杂度，便于管理和维护。

**数据结构优化：** 针对内存管理和文件系统设计中的数据结构，选择合适的结构和算法，优化资源管理和性能，提高系统的效率和稳定性。

**测试用例设计：** 设计全面的测试用例，覆盖各种边界情况和异常情况，验证系统的稳定性和鲁棒性，及时发现和解决潜在的问题和缺陷。

### 5.3 实验收获

通过三个实验的完成，我们获得了以下收获：

- 深入理解了进程调度、内存管理和文件系统的工作原理和实现方法，提高了对计算机系统的整体认识和理解水平。
- 掌握了多种常见的进程调度算法、页面置换算法和文件系统设计方法，增强了问题分析和解决问题的能力。
- 提高了编程实践能力和算法设计能力，加深了对数据结构和算法的理解。
- 通过模拟实验，加深了对操作系统和计算机系统的实践经验，为今后深入学习和研究相关领域打下了坚实的基础。