|  |  |  |
| --- | --- | --- |
| **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**

**操作系统设计与实现**

专 业 名 称 ：计算机科学与技术

课 程 名 称 ：操作系统设计

指 导 教 师 ：李文海

学 生 学 号 ：2022302111038

学 生 姓 名 ：张智清

二○二四年七月

**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： 张智清 日期： 2024/7/20

摘要

本实验旨在让学生深入探究操作系统的核心原理，包括进程调度、虚拟内存管理以及文件系统的设计与实现。通过编写代码实践不同的调度策略、页面替换算法以及模拟文件系统的基本功能，学生不仅能够理解和掌握必要的理论知识，还能培养出设计和实现操作系统基本功能的能力。实验按照由浅入深、理论与实践相结合的方式进行设计，覆盖了进程调度、内存管理、文件系统等多个关键环节。实验结果指出，不同的调度和页面替换算法在性能和公平性上有着明显的区别，且它们各自适用于不同的场景。同时，文件系统的实验加深了学生对于文件存储和管理的认识。

**关键词：**操作系统；JAVA；类设计；进程设计；内存设计；文件系统；进程调度

目录

[郑 重 声 明 2](#_Toc501189979)

[摘要 3](#_Toc1069113519)

[1. 进程调度 6](#_Toc595239384)

[1.1. 实验目的 6](#_Toc1209499162)

[1.2. 实验原理 6](#_Toc2119696879)

[1.3. 实验设计与实现 7](#_Toc1139225270)

[1.3.1. 设计思路 7](#_Toc2142399885)

[1.3.2. 关键代码 7](#_Toc456557946)

[1.3.2.1. FCFS调度算法 7](#_Toc410327691)

[1.3.2.2. RR调度算法 7](#_Toc807512120)

[1.3.2.3. MLFQ调度算法 7](#_Toc1907035447)

[1.3.3. 数据结构 7](#_Toc351326254)

[1.4. 实验结果与分析 7](#_Toc1307805375)

[1.4.1. 测试数据集 7](#_Toc789810580)

[1.4.2. 结果展示 7](#_Toc749995953)

[1.4.2.1. FCFS调度结果 7](#_Toc1600457828)

[1.4.2.2. SJF调度结果 8](#_Toc1662036521)

[1.4.2.3. RR调度结果 8](#_Toc1528011918)

[1.4.2.4. MLFQ调度结果 8](#_Toc1686855000)

[1.4.3. 性能分析 8](#_Toc2040360953)

[1.5. 实验结论 8](#_Toc1327661775)

[2. 虚拟页式存储管理系统 10](#_Toc1656360095)

[2.1. 实验目的 10](#_Toc613600604)

[2.2. 实验原理 10](#_Toc568878534)

[2.3. 实验设计与实现 11](#_Toc544324494)

[2.3.1. 设计思路 11](#_Toc181434438)

[2.3.2. 关键代码 11](#_Toc2089304373)

[2.3.2.1. FIFO页面置换算法 11](#_Toc1433484914)

[2.3.2.2. LRU页面置换算法 11](#_Toc2109397552)

[2.3.2.3. CLOCK页面置换算法 11](#_Toc1984611788)

[2.3.2.4. 虚拟内存管理系统 11](#_Toc654315712)

[2.3.2.5. 控制台输入 11](#_Toc1967898944)

[2.4. 实验结果与分析 11](#_Toc1081904361)

[2.4.1. 测试数据集 11](#_Toc822556178)

[2.4.2. 结果展示 12](#_Toc1349447907)

[2.4.2.1. FIFO结果 12](#_Toc596176982)

[2.4.2.2. LRU结果 12](#_Toc1935323219)

[2.4.2.3. CLOCK结果 12](#_Toc1190024271)

[2.4.3. 性能分析 12](#_Toc1222718186)

[2.5. 实验结论 12](#_Toc953533959)

[3. 文件系统 14](#_Toc1522274999)

[3.1. 实验目的 14](#_Toc1903221482)

[3.2. 实验原理 14](#_Toc674525909)

[3.3. 实验设计与实现 15](#_Toc190780050)

[3.3.1. 设计思路 15](#_Toc247215379)

[3.3.2. 核心代码 15](#_Toc1715501555)

[3.3.2.1. 获取目录对象方法 15](#_Toc319190263)

[3.3.2.2. 分割路径方法 16](#_Toc216600035)

[3.3.2.3. 解析权限方法 17](#_Toc412006580)

[3.3.2.4. 查找文件方法 18](#_Toc1107312132)

[3.3.2.5. 查找子目录方法 19](#_Toc501717622)

[3.3.2.6. 创建文件方法 20](#_Toc1347274832)

[3.3.2.7. 打开文件方法 21](#_Toc580527456)

[3.3.2.8. 关闭文件方法 22](#_Toc906744671)

[3.3.2.9. 读取文件方法 23](#_Toc1113726385)

[3.3.2.10. 写文件方法 24](#_Toc931885443)

[3.3.2.11. 删除文件方法 25](#_Toc600402930)

[3.3.2.12. 创建目录方法 27](#_Toc2093870904)

[3.3.2.13. 删除目录方法 28](#_Toc873760139)

[3.3.2.14. 遍历方法 29](#_Toc793477987)

[3.3.2.15. 控制台输入 30](#_Toc111079639)

[3.4. 实验结果与分析 33](#_Toc752203430)

[3.4.1. 测试数据集 33](#_Toc46818121)

[3.4.2. 实验结果 34](#_Toc893144845)

[3.5. 实验结论 36](#_Toc174717385)

[教师评语评分 37](#_Toc864944246)

1. 进程调度
   1. 实验目的

* 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。
* 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。
* 通过在编写代码中对传统算法进行一些小的设计和修改体会算法设计过程
* 对比不同算法在跑不同测试用例时的效果区别
  1. 实验原理

进程调度是操作系统中关于处理系统进程运行顺序和方式的机制。在设计进程调度程序时，需要了解以下基本原理：

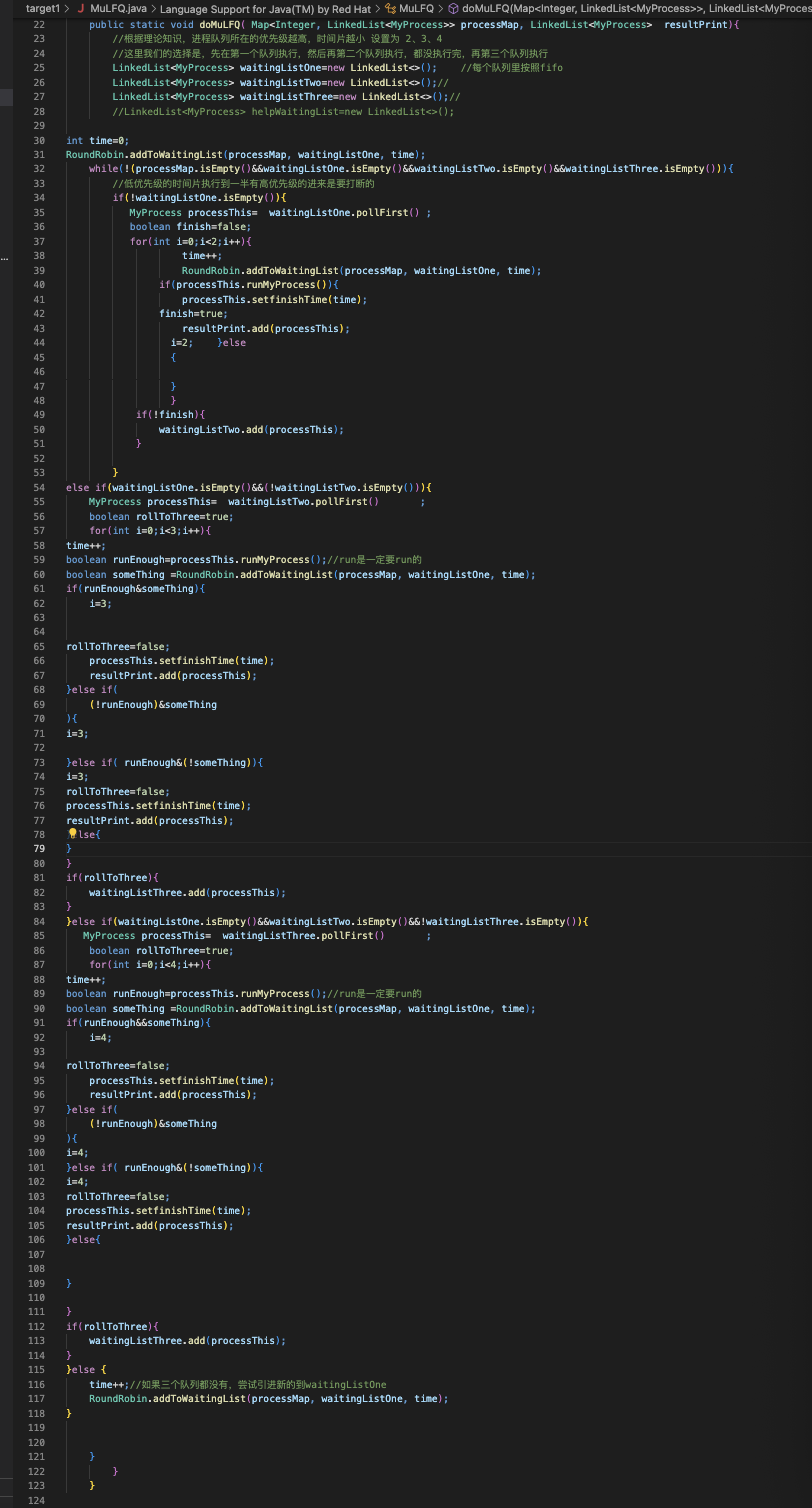
* 进程的概念：进程是程序的一次执行过程，是系统进行资源分配和调度的基本单位。它由程序代码和相关数据、堆栈、PCB (Process Control Block, 进程控制块) 等部分组成。在这次实验中，在这次实验的进程类中我们主要设计
* 调度器：负责根据特定策略，决定哪个进程占有CPU。调度器分为长程调度器（作业调度器）、短程调度器（CPU调度器）、中程调度器。本模拟实验仅涉及短程调度。
* 进程调度策略：进程调度策略决定了当一个进程执行完成式，或者执行遭到中断无法继续在CPU执行，调度策略主要有先来先服务（FCFS）、最短作业优先（SJF）、优先级调度、轮转法（RR）、多级反馈队列等。这些调度策略均有其特定的场合和使用环境。而这几个调度策略也是我们
* 多级反馈队列调度算法：这种算法兼顾了高响应比优先、短作业优先和时间片轮转等算法优点。因为它设置多个调度队列，而优先级较高的队列时间片设置的较短，这使得执行时间较短的进程能够先完成，同时选用时间片算法，能够保证不同的进程能在一定时间内得到相应，以达到最大程度地提高系统的吞吐量、响应时间以及带宽等性能指标。
* 上下文切换：当CPU从一个进程切换到另一个进程时，需要保存当前进程的状态，并加载新的进程状态，这个过程被称为上下文切换，它会产生一定的开销。
* 并发性：操作系统可以让多个进程“同时”运行，实际上是通过迅速地在不同进程间进行切换，为每个进程分配一段时间来执行，从而实现宏观上的并发。
  1. 实验设计与实现
     1. 设计思路

本实验通过模拟实现四种常见的进程调度算法：先来先服务（FCFS）、短进程优先（SJF）、时间片轮转（RR）和多级反馈队列（MLFQ）。每种算法的实现细节如下。

* + 1. 关键代码
       1. FCFS调度算法
       2. public static void DFCFS( Map<Integer, LinkedList<MyProcess>> FCFS, LinkedList<MyProcess> FCFSresult){
       3. // LinkedList<MyProcess> waitingList=new LinkedList<>();
       4. int time =0;
       5. for (Map.Entry<Integer, LinkedList<MyProcess>> entry : FCFS.entrySet()) {
       6. Integer key = entry.getKey();
       7. LinkedList<MyProcess> processList = entry.getValue();
       8. // 处理每个时间点的进程列表
       9. for (MyProcess process : processList) {
       10. // 例如打印进程信息
       11. time= time>process.getArriveTime() ? time:process.getArriveTime();//time先到进程到达的时间
       12. time+= process.getTotalNeedRuntime();//然后再执行
       13. process.setfinishTime(time);//应该是执行完只用设置一个finishtime
       14. FCFSresult.add(process);
       15. }
       16. }
* 代码讲解：
* int time = 0;：初始化一个变量time来跟踪当前时间点。
* for (Map.Entry<Integer, LinkedList<MyProcess>> entry : FCFS.entrySet())：遍历进程映射，映射的键是优先级，值是进程队列。
* LinkedList<MyProcess> processList = entry.getValue();：获取当前优先级的进程列表。
* for (MyProcess process : processList)：遍历当前优先级队列中的所有进程。
* time = time > process.getArriveTime() ? time : process.getArriveTime();：如果当前时间小于进程的到达时间，则将当前时间更新为进程的到达时间。这是为了确保进程在其到达后才开始执行。
* time += process.getTotalNeedRuntime();：将当前时间增加进程所需的执行时间，得到进程的完成时间。
* process.setfinishTime(time);：更新进程的完成时间。
* FCFSresult.add(process);：将处理过的进程添加到结果列表中。
* 这段代码的核心在于确保每个进程按照到达的顺序执行，并且每个进程的执行时间是连续的，不重叠。通过这种方式，我们可以模拟一个简单的先来先服务调度算法的行为。

可以看到，这个代码还是比较简单的

* + - 1. RR调度算法

1. public static void doDD(
2. Map<Integer, LinkedList<MyProcess>> RR,
3. LinkedList<MyProcess> RRresult,int timeSliceValue
4. ){
6. int time=0;
7. LinkedList<MyProcess> waitingList=new LinkedList<>();
8. MyProcess nowdo = null;
9. while (!(RR.isEmpty() && waitingList.isEmpty())) {
10. //刚开始 time是0，只用增加waiting list就好了·
11. addToWaitingList(RR, waitingList, time);
12. if (!waitingList.isEmpty()) {
13. if (nowdo != null) {
14. System.out.println("有问题结束");
15. throw new RuntimeException("处理中断，进程未完成"); // 使用异常处理错误
16. }
17. nowdo = waitingList.removeFirst();
18. boolean helpJudgeFinish=false;
19. for (int i = 0; i < timeSliceValue; i++) {
20. time++;addToWaitingList(RR, waitingList, time);
21. if (nowdo.runMyProcess()) {
22. nowdo.setfinishTime(time);
23. RRresult.add(nowdo);
24. nowdo = null; // 如果进程完成，确保清除 nowdo
25. i=timeSliceValue;//这里要中断
26. helpJudgeFinish=true;
27. }
28. }
29. if(!helpJudgeFinish){
30. if(nowdo==null) { throw new RuntimeException("..");};
31. waitingList.add(nowdo);
32. nowdo=null;
33. }
34. } else {
35. time++; // 如果 waitingList 为空，增加时间
36. }
37. }
    * + 1. 上面这段代码在随着time变量增大的同时遍历进程输入的hashmap（进程输入后储存的hashmap，这个进程输入稍后再讲），然后让他们轮流执行，
        2. 比较细节的地方是.runMyProcess()会返回一个boolean类型，如果返回的是true，那说明这个进程运行完毕了，时间片应该停止转动，开始下一个进程，下一个时间片，在time每增大1时
        3. MLFQ调度算法

整体思路：

初始化多个等待队列，分别对应不同的优先级。

使用一个循环来模拟进程调度，直到所有队列都为空。

根据进程的优先级分配不同的时间片，并在每个时间片内执行进程。

如果进程在时间片内完成，则将其从当前队列中移除；如果未完成，则根据规则移动到下一个队列。

RoundRobin.addToWaitingList(processMap, waitingListOne, time);：这个方法负责将新到达的进程添加到第一个等待队列中。

waitingListOne.pollFirst();：从第一个队列中取出第一个进程。

processThis.runMyProcess();：执行当前进程。如果返回true，表示进程执行完毕。

processThis.setfinishTime(time);：设置当前进程的完成时间。

resultPrint.add(processThis);：将完成的进程添加到结果列表中。

if (!finish) { ... }：如果当前进程未在时间片内完成，根据其优先级将其移动到下一个队列。

* + - 1. SJF调度算法

1. public static void DoShortFS(Map<Integer, LinkedList<MyProcess>> SJF , LinkedList<MyProcess> SJFresult){
2. int time=0;
3. MyProcess nowdo=null;
4. PriorityQueue<MyProcess> sjfWaiting = new PriorityQueue<>(Comparator.comparingInt(MyProcess::getleftTime));
5. addToWaitingList(SJF,sjfWaiting, time);
6. while(!(SJF.isEmpty()&&sjfWaiting.isEmpty())){
7. time++;
8. //System.out.println("time+1是"+time);
9. nowdo = sjfWaiting.poll();//刚开始有可能是null，因为还没到对应的key
10. if(nowdo!=null){
11. if(nowdo.runMyProcess()){//如果不是
12. nowdo.setfinishTime(time);
13. // System.out.println("time是"+time);
14. SJFresult.add(nowdo);
15. nowdo=null;
16. }else{
17. //System.out.println("leftTime"+nowdo.getleftTime());
18. sjfWaiting.add(nowdo);
19. nowdo=null;
20. }
21. }
22. else{
23. // System.out.println(time+"时没有能执行的");
24. }
25. addToWaitingList(SJF, sjfWaiting, time);
26. }
27. }

定义了一个名为DoShortFS的静态方法，它接受一个包含进程的映射和一个结果列表作为参数。

第40行：初始化一个变量time来跟踪当前时间。

第43行：创建一个优先队列sjfWaiting，它根据进程的剩余执行时间（getleftTime方法返回的值）进行排序。

第44行：调用addToWaitingList方法将初始进程添加到优先队列中。

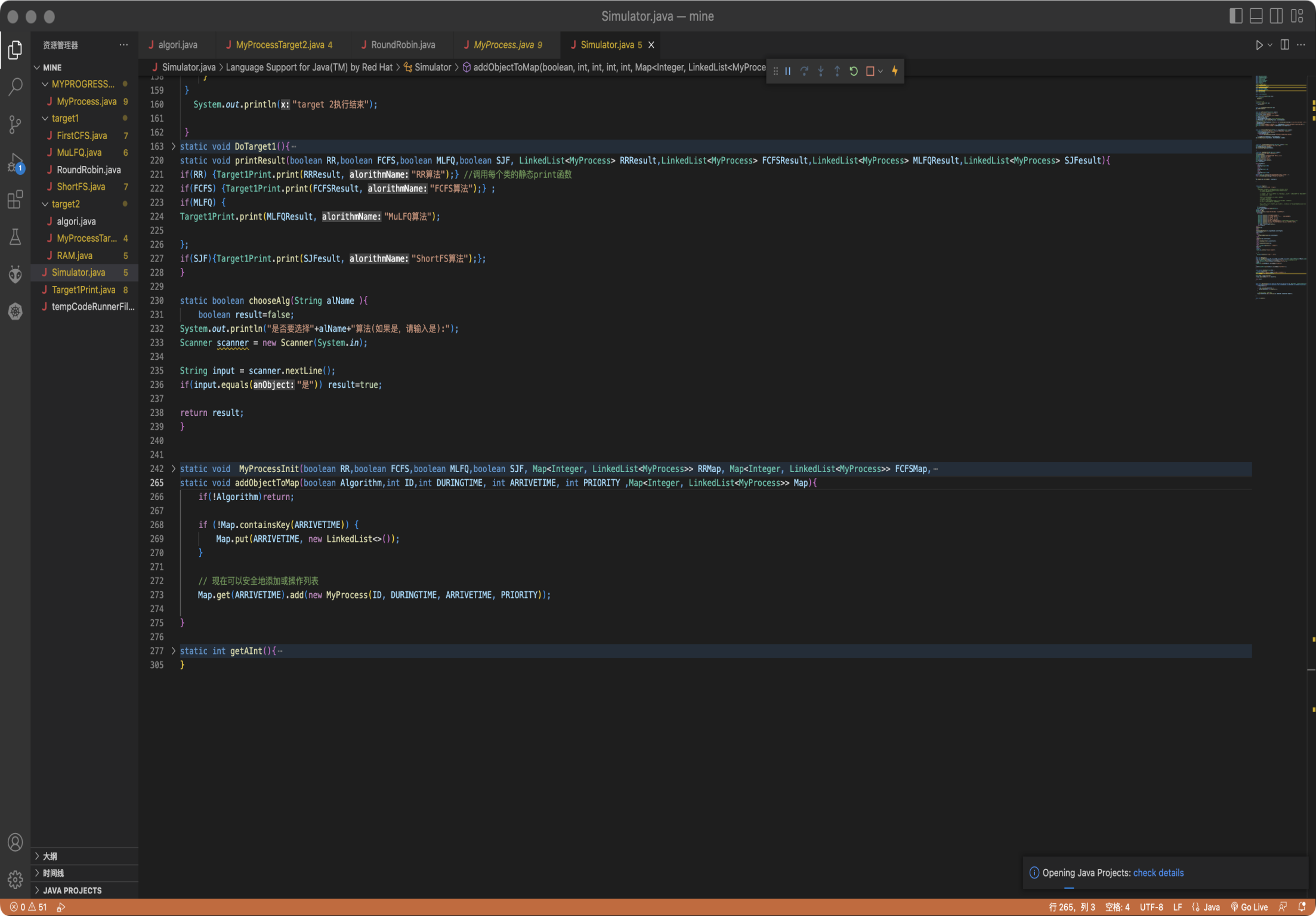
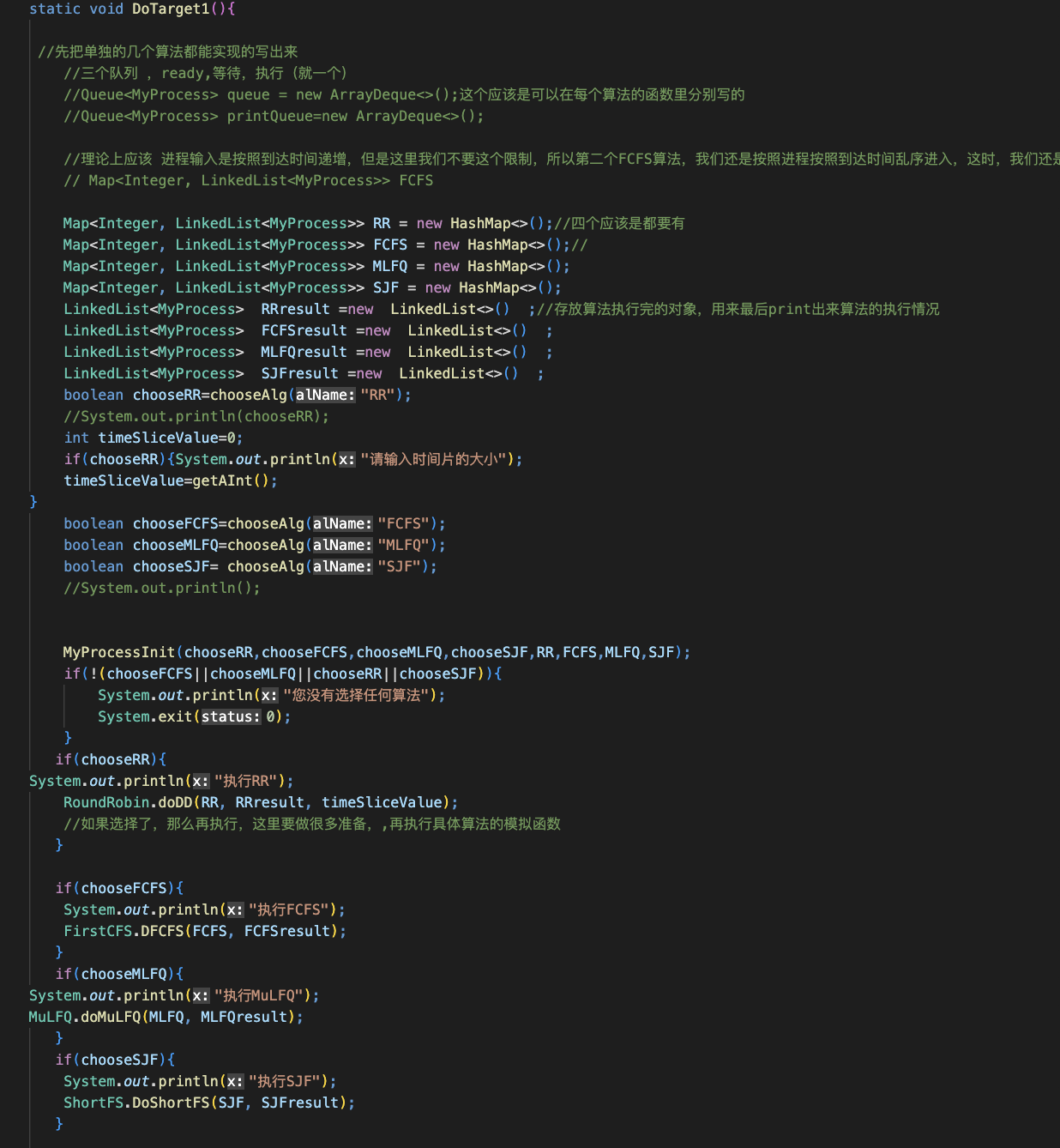
第45行：开始一个循环，直到所有进程都已经被处理（即进程映射和优先队列都为空）。

第46行：每次循环时增加时间。

第49行：从优先队列中取出预计执行时间最短的进程。

第50-62行：如果取出的进程不为空，则执行该进程。如果进程执行完毕，设置其完成时间并将其添加到结果列表中；如果进程未执行完毕，则将其重新加入优先队列。

第63-68行：如果当前没有可执行的进程，输出日志信息（已注释）。然后，将新到达的进程添加到优先队列中。

* + - 1. 进程信息输入
      2. 在生成随机数创造进程，文件输入、键盘输入创造进程这几种方法中，我选择了键盘输入进程
      3. 下图显示了与终端交互的代码、创造进程的代码以及把新建的进程放入hashmap的代码
      4. 
      5. print函数

显然我们要把算法执行的顺序和运行效果展示给用户，因此在算法执行过程中，已经执行完的进程（也就是leftTime==0）的进程，我们需要将他们统一管理，因此，我们还需要一个Resultlist<MyProgress>，并且在调度算法执行结束后，把这个List里面的进程信息print出来,因此我们有了如下代码：

1. public static void print(LinkedList<MyProcess> RRresult,String alorithmName){
2. double ATT=0;//平均周转时间
3. double AWTT=0; //平均带权周转时间
4. double AverageWaitingTime=0 ;//平均等待时间
5. double AverageRef=0;//平均相应时间
6. double count=0;
7. System.out.println("选用"+alorithmName+"算法的运行结果如下");
8. //写平均周转时间和等待时间
9. for (MyProcess process : RRresult) {
10. // 这里是对每个MyProcess对象进行的操作
11. // 例如打印process的信息
12. System.out.println("进程"+process.getId()+"的到达时间是"+process.getArriveTime()+" 离开时间是"+process.getFinishtime()+" 停留时间是"+process.getStayTime()+" 有效运行时间是"+process.getTotalNeedRuntime());
13. ATT+=process.getStayTime();
14. AWTT+=process.getStayTime()/process.getTotalNeedRuntime();
15. AverageWaitingTime+= process.getStayTime()-process.getTotalNeedRuntime() ;
16. count++;
17. }
18. ATT=ATT/count;
19. AWTT=AWTT/count;
20. AverageRef=AverageRef/count;
21. AverageWaitingTime=AverageWaitingTime/count;
22. System.out.println("平均周转时间是"+ATT);
23. System.out.println("平均带权周转时间"+AWTT);
24. System.out.println("平均等待时间是"+AverageWaitingTime);
25. System.out.println(alorithmName+"算法信息输出结束");
26. }
    * 1. 数据结构

在代码上，显然我们要设计一个进程类，而其中应该有显示调度算法效率的相关变量，例如arrivetime,finishTime等等，同时，要有辅助算法能够顺利实现的变量，例如leftTime，等等，当然，也要写外接访问这些变量的接口函数

下面截取关键代码



* 1. 实验结果与分析
     1. 测试数据集

实验中，我特意设计了能够验证算法正确性的一些数据集，例如在SJF算法中，应该设计一些晚到但是执行时间较短的进程，如果我们看到这样的进程被先执行完毕了，证明了我们的算法正确，再比如，在多级队列调度算法中，如果我们看到在二级优先级或者三级优先级的队列正在运行的过程中，有一个hashmap的进程进入了1级waitinglist，那么这个进程应该抢占正在执行的进程。下面是我的测试数据和终端运行，可以看到结果都证明了我的算法执行正确

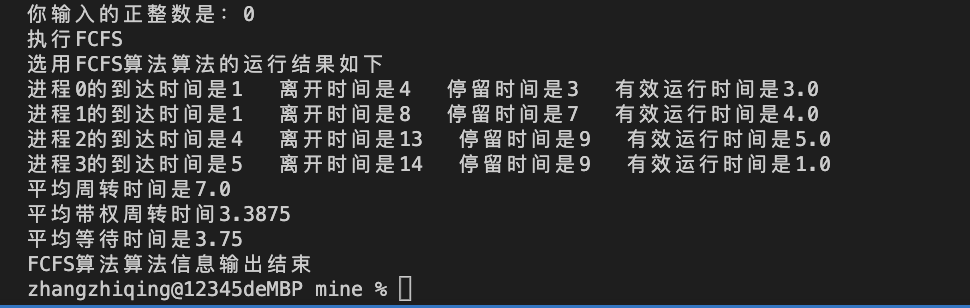
* + 1. 结果展示
       1. FCFS调度结果

先进先出

进程1 1到达 执行时间3

进程2 1到达 执行时间4

进程3 4到达 执行时间5

进程 4 5到达 执行时间1

* + - 1. SJF调度结果

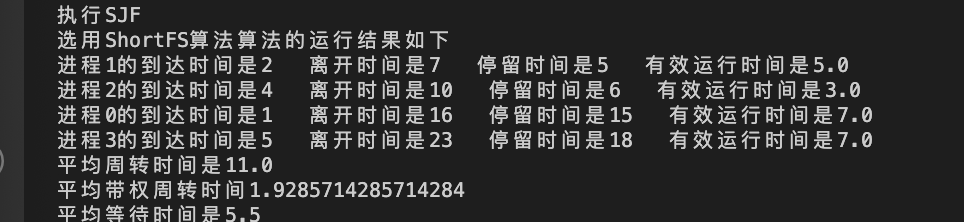
最短剩余

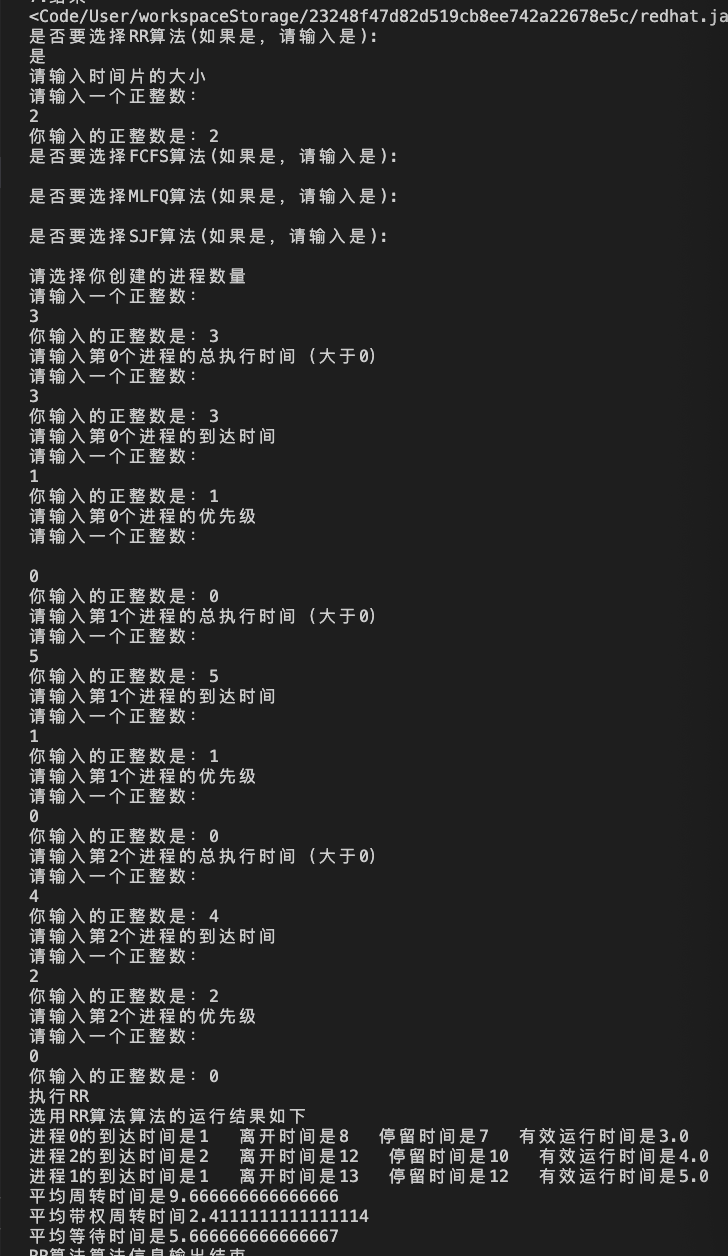
进程1 ：1到 执行时间7

进程2 ：2到执行时间5

进程3 ：4到，执行时间 3

进程4： 5到，执行时间7

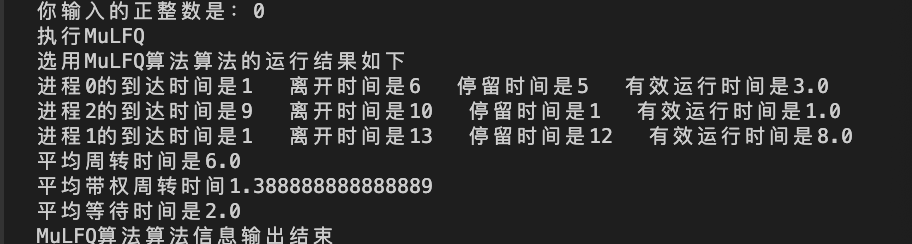


* + - 1. RR调度结果
      2. MLFQ调度结果

进程1：1时刻到 执行3，

进程2： 1时刻到，执行8

进程3 ：9时刻到执行1



* + 1. 性能分析
    2. 除了上述几个验证算法正确性
* **FCFS**: 简单易实现，但可能导致长时间等待，尤其是短作业到达后需等待长作业完成。
* **SJF**: 提供最短平均等待时间，但难以实现，因为需要预知运行时间，可能导致“饥饿”问题。
* **RR**: 适用于时间共享系统，通过时间片轮转分配CPU时间，响应时间较好，但可能导致频繁的上下文切换。
* **MLFQ**: 综合多种调度策略，兼顾高响应比优先、短作业优先和时间片轮转的优点，较为复杂，但能够适应不同类型的任务。
  1. 实验结论

不同调度算法在性能和公平性方面存在显著差异：

* FCFS算法简单，但可能导致进程长时间等待。
* SJF算法在平均等待时间方面表现最好，但难以实现，且可能导致“饥饿”问题。
* RR算法在响应时间方面表现较好，但可能导致频繁的上下文切换。
* MLFQ算法综合了多种调度策略的优点，适应性较强，但实现较为复杂。

通过本次实验，实现并比较了FCFS、SJF、RR和MLFQ四种调度算法。结果表明，各调度算法在不同应用场景下具有不同的适用性，需根据具体需求选择合适的调度算法。FCFS适用于简单任务，SJF适用于已知运行时间的任务，RR适用于时间共享系统，MLFQ适用于综合多种任务类型的复杂系统。

1. 虚拟页式存储管理系统
   1. 实验目的

* 理解虚拟内存和物理内存之间的关系及转化，在什么情况下会发生缺页中断。
* 理解和掌握各种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）、时钟置换（CLOCK）和随机置换算法，并通过模拟实验观察它们的执行过程。
* 通过比较不同页面置换算法下系统的性能，理解不同算法的优缺点以及在何种情况下更适用。
* 学习如何评估系统性能。通过实验结果，理解如何通过缺页中断次数、平均访问时间等指标去评估系统性能，并能合理地解释实验结果。
* 从实验中体验和理解内存访问的局部性原理，提高自己对操作系统虚拟内存管理的理解。
  1. 实验原理
* **虚拟内存技术**：虚拟内存技术使得计算机系统看上去具有比实际RAM更大的内存，通过使用硬盘空间作为对RAM的扩充，并将之作为应用程序的虚拟存储器。
* **页（Page）**：操作系统管理内存的一个单位，将逻辑地址空间分割为一系列连续的、固定大小的区域，即页。
* **帧（Frame）**：与页的大小一致，操作系统管理对应实际内存的单位。
* **页表**：存放虚拟页面和物理内存帧之间的映射。为了提高地址转换效率，操作系统通常采取一些优化策略，如多级页表、倒置页表、TLB等。
* **地址映射**：操作系统维护页表，记录页号和帧号的映射关系。当CPU发出虚拟地址时，通过页表将虚拟地址转换为实际的物理地址。
* **页面置换**：当分配给进程的所有页面都在使用中且需要加载新的页面时，操作系统必须选择一个页面移出内存以空出空间给新的页面。常见页面置换算法包括FIFO、LRU、CLOCK和随机置换。
* **缺页中断**：程序引用的页不在内存中时，会产生缺页中断，操作系统从磁盘中将页调入内存，并更新页表。
  1. 实验设计与实现
     1. 设计思路

设计一个模拟内存系统有很多种设计方法，例如我们可以设计多级页表，或者是设置可以由用户决定内存有多少个物理页号，由用户决定一个进程可以在物理页面中占用多少个物理框，在这里我选了不设置多级页表，把物理页框数设置为16，给每个进程最多分配4个物理页框。如果一个进程的虚拟页号没有超过4，那么它有几个虚拟页号，就给他分配几个物理页号（这时候也不需要替换算法了），另外，我在RAM类中设计了一个两行的TLB，所有进程公用，TLB采用LRU替换算法（实际上的不同操作系统中，可能会给每个进程都设计TLB，但这取决于怎么设计，而TLb命中率，缺页率等数据就是很好的反映了设计效果）

接下来简单列出一些我在设计过程中遇到的难点以及解决方法

首先是在实现LRU替换算法中，有可能在RAM类使用访问TLb就命中了，这样就不会进入progress类去触发维护LRU算法的函数，LRU算法的相关数据就会发生错误，所以我们在tlb命中的时候要多执行一个函数，去维护进程对象的LRU相关变量里面的数据仍然正确。

再比如，在统计访问页表的命中次数时，我设计了每一个更新替换算法，就把之前的访问记录数据清空，而一次统计总访问页表次数时，有可能之前有的访问过的进程已经被删除，因为我们要在删除进程的时候get这个进程的访问数据到RAM类的数据里

另外在代码中还要设计一些防止运行出错和保证用户有好的代码，例如，在访问或者删除一个进程时，首先要遍历一下RAM的ObjectList，看这个进程存不存在，如果不存在，给用户sout ：该进程不存在，然后直接return，避免在深入调用其他函数，可能引发一些错误。

* + 1. 关键代码

以下是主要代码实现，包括页面置换算法和虚拟内存管理系统：

* + - 1. 控制台输入

1. static void Dotaget2(){
2. System.out.println("下面开始target2");
3. /\*\*
4. \*物理内存有多少个帧理论上也可以键盘输入，但是我们这里固定为16个页
5. 给进程首次分配帧也可以采用首次适应算法、循环首次适应算法等等
6. 我们这里就选用首次适应算法，
8. 在进程不断的执行的过程中我们可以根据一个进程的缺页率，进行固定分配局部置换，可变分配全局置换，可变分配局部置换等等
9. 我们这里就选用固定局部置换
11. 另外，由于模拟的内存页数较小，我们就不用多级页表了
12. 多个进程共用一个2行的tlb
13. （是否要模拟访问时间待定）
14. 我们假设一个进程的虚拟内存有8个页面，我们假设字节寻址、一个页面64byte
15. （也就是6位为页内偏移地址，我们不需要考虑）
17. 这里我们不限制进程虚拟空间的下限，也就是说它可以小于 4，有可能把 16个桢的物理内存拆散，不是mod 4==0，
18. 上限限制为8个页面
19. \*/
20. int progressNumber=1;
21. Scanner scannerTarget2=new Scanner(System.in);
22. boolean loop=true;
23. RAM ram=new RAM();
24. System.out.println("系统初始化完毕，默认开始选择FIFO算法");
25. while(loop){
27. System.out.println("请输入你想选择的功能");
28. System.out.println("1.显示内存中现有的进程和他们所占据的物理页面号");
29. System.out.println("2.尝试新建一个进程");
30. System.out.println("3.尝试删除一个进程");
31. System.out.println("4.读写某进程的 内容");
32. System.out.println("5.更改进程的页表的替换算法");//更改算法时先显示算法的效果，再清空
33. System.out.println("6.查看目前算法内存访问的tlb命中情况和各进程的页表命中情况");
34. System.out.println("7.结束");
35. int number = scannerTarget2.nextInt() ;
36. switch(number){
37. case 1:
38. showInfo(ram);
39. break;
40. case 2:
41. tryToCreateANewProgress(ram,progressNumber,scannerTarget2);
42. progressNumber++;
43. break;
44. case 3:
45. tryToDeleteANewProgress(ram,scannerTarget2);
46. break;
47. case 4:
48. useProcess(ram,scannerTarget2);
49. break;
50. case 5:changeAlgorithm(ram,scannerTarget2);
51. break;
52. case 6:showNowAlPerformance(ram);
53. break;
54. case 7: System.out.println("退出内存模拟");
55. loop=false;
56. break;
57. default :
58. System.out.println("Invalid command");
60. }
61. }
62. System.out.println("target 2执行结束");
64. }
65. static void showNowAlPerformance(RAM ram){
66. ram.showPerformance();
67. }
68. static void useProcess(RAM ram,Scanner scanner){
69. System.out.println("请输入您想使用的进程号");
70. int ProgressId= scanner.nextInt();
71. System.out.println("请输入您想使用该进程的地址");
72. int address=scanner.nextInt();
73. int PageOffset=address%64;
74. int virtualPageNumber=address/64+1;//求出了 这个地方要看好0，1
75. int RAMPageNumber = ram.TryDoRead(ProgressId, virtualPageNumber);
76. //假设一个进程64byte
77. if(RAMPageNumber<=0&&RAMPageNumber!=-999){throw new RuntimeException("在main函数里，doread的执行返回值出了错误");}
78. if(RAMPageNumber>0){
79. System.out.println("该地址现在在物理内存中是在第"+RAMPageNumber+"（内存的物理框的范围是1到16，0不作数）");//
80. System.out.println("在物理内存中的地址是"+((RAMPageNumber-1)\*64+PageOffset));
81. }}
82. static void tryToCreateANewProgress(RAM ram,int progressNumber,Scanner scanner){
83. System.out.println("请输入该进程有多少个虚拟页面(请输入一个正整数)");
84. int VitualPageNumber=scanner.nextInt();
85. if(VitualPageNumber<=0){
86. throw new RuntimeException("您输入的虚拟页面的个数不是正整数");
87. }
88. int needRam=VitualPageNumber<4 ?VitualPageNumber:4 ;
89. ram.CreateNewProgress(progressNumber, VitualPageNumber, needRam);
90. }
91. static void tryToDeleteANewProgress(RAM ram,Scanner scanner){//完成了
92. System.out.println("请输入你想要删除的进程的进程号");
93. int wantToDelete=scanner.nextInt();
94. ram.tryToDeleteAProgress(wantToDelete);
95. }
96. static void changeAlgorithm(RAM ram,Scanner scanner){
97. System.out.print("目前的单进程的替换算法是");
98. ram.showNowALGORI();
99. System.out.println("输入您想要替换成什么算法(1,2,3)");
100. System.out.println("1.FIFO");
101. System.out.println("2.LRU");
102. System.out.println("3.CLOCK");
103. int newAlNumber= scanner.nextInt();
104. algori targetAlgori;
105. switch(newAlNumber){
106. case 1:
107. targetAlgori=algori.FIFO;
108. break;
109. case 2:
110. targetAlgori=algori.LRU;
111. break;
112. case 3:
113. targetAlgori=algori.CLOCK;
114. break;
115. default :System.out.println("您未输入想要替换成什么算法,程序即将退出");
116. throw new RuntimeException("您输入的算法替换不正确");
117. }
118. ram.changeAL(ram.returnAlGORI(), targetAlgori);
119. }

以下是代码的主要功能：

初始化:

设置进程编号为1。

创建Scanner对象用于接收用户输入。

创建RAM对象模拟内存管理。

默认选择FIFO算法。

主循环:

提示用户输入想要执行的操作编号。

输出可用的功能选项，包括显示内存信息、创建新进程、删除进程、读写进程内容、更改算法、查看算法性能和退出程序。

根据用户输入的编号执行相应的操作。

功能方法:

showNowAlPerformance(RAM ram): 显示当前算法的性能。

useProcess(RAM ram, Scanner scanner): 允许用户读写指定进程的内存内容。

tryToCreateANewProgress(RAM ram, int progressNumber, Scanner scanner): 允许用户创建新进程。

tryToDeleteANewProgress(RAM ram, Scanner scanner): 允许用户删除进程。

changeAlgorithm(RAM ram, Scanner scanner): 允许用户更改页面置换算法。

结束:

当用户选择退出时，循环结束，程序输出提示信息表示target 2执行结束。

这段代码提供了一个基本的内存管理模拟环境，用户可以通过文本界面与模拟的内存系统交互，执行各种内存管理操作。程序中使用了一个RAM类来模拟内存的分配和置换，以及一个Scanner类来接收用户输入。每个功能都封装在单独的方法中，以便于管理和维护

* + - 1. RAM方法设置

1. public class RAM {
2. private static algori ALGORI;
3. PageInfo[] TLBArray ;//TLB的替换也需要选用算法，
4. //这里选择，如果一个发生tlb替换，如果另一个的recent use 数值大于
5. //CLOCK，
6. //FIFO
7. //LRU需要在这里实现
8. private static LinkedList<MyProcessTarget2> ObjectList;//记录目前有的进程，
9. private static int[] progressId ;
10. //RAM固定有16个帧
11. private static double TLBhitTime;
12. private static double totalReadTime;
13. private static double deletedProcessReadTime;
14. private static double deletedProcessMissingTime;
15. public RAM(){
16. progressId=new int[16];
17. ObjectList=new LinkedList<>();
18. TLBArray = new PageInfo[2];
19. TLBArray[0]=new PageInfo(-1, -1, -1);
20. TLBArray[1]=new PageInfo(-1, -1, -1);
21. ALGORI=algori.FIFO;
22. for(int i=0;i<16;i++){
23. progressId[i]=-1;
24. }
25. TLBhitTime=0;
26. totalReadTime=0;
27. }
28. public void showNowALGORI(){
29. System.out.println(ALGORI);
31. }
32. public void showInfo(){
33. //这里不显示每个进程的信息了，显示内存的信息
34. showNowALGORI();
35. System.out.println("下面输出内存中页框的信息");
36. for(int i=0;i<16;i++){
37. if(progressId[i]!=-1){
38. System.out.println("内存中的第"+(i+1)+"个页框是"+progressId[i]+"进程的");//我们全部统一，在对外展示时都没有0
39. }
40. }
41. System.out.println("下面是tlb的信息");
42. boolean helpJudgeTLBContainsSomethind=false;
43. for(int i=0;i<2;i++){
44. if(TLBArray[i].getPageInFoprogressNumber()>0) {System.out.println(TLBArray[i]);
45. helpJudgeTLBContainsSomethind=true;
46. }
47. }
48. if(!helpJudgeTLBContainsSomethind){System.out.println("\*\*\*\*\*\*\*\*目前TLB中还没有有效的信息");}
49. System.out.println("内存信息输出结束，");
50. }
51. public void CreateNewProgress(int NewProgressId,int VirtualPageAmount,int needInRam){//这个need给main函数的地方处理，如果比4大，那就算成4
52. //这个地方为了后面加算法，肯定还得加参数
53. int [] recordIndexInRam=new int[needInRam];//虚拟页号就是顺着的
54. int count=0;
55. for(int i=0;i<16;i++ ){
56. if(progressId[i]==-1){
57. if(count<needInRam) recordIndexInRam[count]=i; //这个地方先记录页框号
58. count++;}
59. }
60. if(count<needInRam){
61. System.out.println("您内存目前剩余的空闲帧无法创造一个需要分配"+needInRam+"物理内存中的帧的进程");
62. return;
63. }
64. MyProcessTarget2 newProgress=new MyProcessTarget2(NewProgressId ,VirtualPageAmount,needInRam,ALGORI,recordIndexInRam);
65. //这里要初始化clock,fifolru,放在构造函数里
66. ObjectList.add(newProgress);
67. for(int i=0;i<needInRam;i++){
68. newProgress.writePageTable(i, recordIndexInRam[i]+1);//页表中的物理页框的范围也是 1到16
69. progressId[recordIndexInRam[i]]=NewProgressId;
70. }
71. }
72. private int ReadTlb(int progressNumber,int virtualPageNumber){
73. int help=-1;
74. boolean TLBhas=false;
75. for(int i=0;i<2;i++){
76. if(TLBArray[i].isThisRow(progressNumber, virtualPageNumber)>-1){
77. TLBhas=true;
78. help=i;
79. }
80. }
81. if (TLBhas){ System.out.println("查找tlb成功，无需访问页表");
82. TLBhitTime++;
83. TLBArray[help].recentUsePlus();
84. int result=TLBArray[help].isThisRow(progressNumber, virtualPageNumber);
86. return result;
87. }
88. else{
90. return -1;
91. }
92. }
93. private void changeTlb(int progressNumber,int virtualPageNumber,int RamNumber ){//change
94. //在这里要完成对tlb优先级的
95. int index= TLBArray[0].getRecentUseCount()<=TLBArray[1].getRecentUseCount()? 0:1 ;
96. //这里recentUseCount 给一个1
97. TLBArray[1-index].recentUseCountSubIfOtherReplace();
98. PageInfo newPageinfo=new PageInfo(progressNumber, virtualPageNumber, RamNumber,1);
99. System.out.println("TLB中的条目:"+TLBArray[index]+"即将替换为"+newPageinfo);
100. TLBArray[index]=newPageinfo;
101. }
102. private int doRead(int progressNumber,int virtualPageNumber,MyProcessTarget2 inList){
103. totalReadTime++;
104. int RAMNumber=ReadTlb(progressNumber, virtualPageNumber);
105. if(RAMNumber>0){//这个地方后面
106. if(ALGORI==algori.LRU){ inList.increaseWhenTlbAndLru(RAMNumber);
107. }
109. return RAMNumber;
110. }
111. int readResult=inList.ReadPageForm(virtualPageNumber);//返回了物理页号
112. changeTlb(progressNumber, virtualPageNumber, readResult);
113. return readResult;
114. }
115. private int ReadPageForm(MyProcessTarget2 inList ,int virtualPageNumber){
116. return inList.ReadPageForm(virtualPageNumber);
117. }
118. /\* \*/
119. public int TryDoRead(int progressNumber,int virtualPageNumber){
120. boolean helpJudge=false;
121. MyProcessTarget2 helpProcess=null;
122. for (MyProcessTarget2 process : ObjectList) {
123. if(process.getprocessId()==progressNumber)
124. {helpJudge=true;
125. helpProcess=process;
126. break;}
127. }
129. if(!helpJudge){System.out.println("您想访问的进程不存在");
130. return -999;
131. } if(helpProcess==null){
132. throw new RuntimeException("处理中断，进程未完成");
133. //这个是应该不会被触发才对
134. }
135. if( virtualPageNumber<=0||(virtualPageNumber>helpProcess.getVirtualPageCount()) )//虚拟页号是1到virtualPageCount
136. {System.out.println("您想访问的"+progressNumber+"号进程没有"+"该地址");
137. return -999;
138. }
139. //然后给do Read 不要都写在一个函数里
140. return doRead(progressNumber, virtualPageNumber,helpProcess);
141. }
142. public boolean tryToDeleteAProgress(int deleteProgressId){
143. boolean hasThisProgress=false;
144. for (MyProcessTarget2 process : ObjectList) {
145. if(process.getprocessId()==deleteProgressId){
146. hasThisProgress=true;
147. break;
148. }
149. }
150. if(!hasThisProgress){
151. System.out.println("您尝试删除的进程本身就不存在");
152. return false;
153. }
154. //这里怎么delete一会儿要看一下语法
155. //int
156. Iterator<MyProcessTarget2> iterator = ObjectList.iterator();
157. boolean helpAgain=false;
158. while (iterator.hasNext()) {
159. MyProcessTarget2 processTarget = iterator.next();
160. if (processTarget.getprocessId() == deleteProgressId) {
161. deletedProcessMissingTime+=processTarget.getPageTableMissTime();
162. deletedProcessReadTime+=processTarget.getReadPagetime();
163. iterator.remove(); // 删除当前对象
164. helpAgain=true;
165. }
166. }
167. if(!helpAgain){
168. throw new RuntimeException("应该是能删除到"+deleteProgressId+"的，但是出现问题了");
169. }
170. for(int i=0;i<16;i++ ){
171. if(progressId[i]==deleteProgressId){
172. progressId[i]=-1;
173. }
174. }
175. //在这里把tlb也删了
176. for(int i=0;i<2;i++){
177. if(TLBArray[i].getPageInFoprogressNumber()==deleteProgressId){
179. System.out.println(TLBArray[i]+"即将被删除");
180. TLBArray[i]=new PageInfo(-1, -1, -1);
181. }
182. }
183. System.out.println("删除"+deleteProgressId+"成功");
184. return true;
185. }
186. public void changeAL(algori NowAL ,algori Targetal){
187. if(NowAL==Targetal){
188. System.out.println("源算法和目的算法相同");
189. return;}
190. System.out.println("将替换算法从"+NowAL+"变为"+Targetal);
191. //替换算法前
192. showPerformance();
193. ALGORI=Targetal;
194. TLBhitTime=0;
195. totalReadTime=0;
196. deletedProcessMissingTime=0;
197. deletedProcessReadTime=0;
198. if(!ObjectList.isEmpty()){
199. for (MyProcessTarget2 process : ObjectList) {
200. process.swtichAL(NowAL, Targetal);
201. }
202. }else{
203. System.out.println("目前没有存在的进程，算法替换完成");
204. }
205. }
206. public algori returnAlGORI(){
207. return ALGORI;
208. }
209. public void showPerformance(){
210. showNowALGORI();
211. double ExisteachTotalread=0;
212. double ExistMissTime=0;
213. for (MyProcessTarget2 process : ObjectList) {
214. ExisteachTotalread+=process.getReadPagetime();
215. ExistMissTime+=process.getPageTableMissTime();
216. }
217. if(ExisteachTotalread+TLBhitTime+deletedProcessReadTime!=totalReadTime){
218. throw new RuntimeException("数量对不上");
219. }else{
220. System.out.println("从开始执行目前的算法到目前为止");
221. System.out.println("一共进行了"+totalReadTime+"次访问");
222. System.out.println("TLB命中"+TLBhitTime+"次");
223. System.out.println("访问已经被删除的进程页表一共"+deletedProcessReadTime+"次，其中有"+deletedProcessMissingTime+"次缺页");
224. System.out.println("访问目前内存内的页表"+ExisteachTotalread+"次其中有"+ExistMissTime+"缺页，缺页率为");
225. if(totalReadTime!=0){
226. System.out.println("总TLB命中率为"+TLBhitTime/totalReadTime);
227. System.out.println("总缺页率为"+(ExistMissTime+deletedProcessMissingTime) /totalReadTime);
228. System.out.println();
229. }
230. }
232. }
233. }
     * + 1. 进程类方法设置
234. /\*
235. \* 由于一个进程创建完了之后在内存中占据的位置已经固定了，所以在页表中只要改，哪行是哪行就行了，值都是index+1的值
236. \* （只要初始化的时候是+1就可以）
237. \*
238. \*
239. \*/
240. //后面这里要有一个缺页中断利用率的统计，这个后面有时间再弄
241. //private int
242. private double readPagetime;
243. private void readPagetimereset(){
244. readPagetime=0;
245. }
246. private void readPagetimePlusOne(){
247. readPagetime++;
248. }
249. public double getReadPagetime(){
250. return readPagetime;
251. }
252. private double PageTableMissTime;
254. private void PageTableMissTimeReset(){
255. PageTableMissTime=0;
256. }
257. private void PageTableMissTimePlusOne(){
258. PageTableMissTime++;
259. }
260. public double getPageTableMissTime(){
261. return PageTableMissTime;
262. }
263. private int CLOCKHELPINT;
264. private int processId;
265. private int virtualPageCount;//这个进程的虚拟页的总数
266. private int PhysicalPageNumber;//这个进程的物理页框的总数
267. private algori ALGORI ;
268. private int[] PageTable;//页表里存着的是+1的
269. private int [][] CLOCK;//CLCOK 每个的int [][0]装虚拟页号 int[][1]装物理页号 存着的都是+1的
270. //物理页号是正的代表标志位是1，负的代表是0，再轮到要被替换掉
271. //HashMap<Integer, Integer> LRUhashMap ;//LRU 存着的都是+1的
272. HashMap<Integer, int[]> LRUhashMap = new HashMap<>();//key是物理页号，后面是 int[1]是次数
273. //最后实现的实际上不是low recent 是low total
274. LinkedList<int[]> FIFO ; //FIFO 存着的都是+1的 同样的 int[0]是物理页号，int[1]是虚拟页号，
276. public MyProcessTarget2(int ID ,int VPC,int PhysiPN,algori ALGORIFromRam,int []Init ){
277. this.ALGORI=ALGORIFromRam;
278. PhysicalPageNumber=PhysiPN;
279. processId=ID;
280. virtualPageCount=VPC;
281. readPagetime=0;
282. PageTableMissTime=0;
283. PageTable=new int[VPC];
284. LRUhashMap=new HashMap<>();
285. CLOCK=new int[PhysicalPageNumber][2];
286. FIFO=new LinkedList<>();
287. for(int i=0;i<VPC;i++){
288. PageTable[i]=-1;
289. }
290. switch(ALGORIFromRam){
291. case FIFO:
292. FIFOInit(Init);
293. break;
294. case LRU:
295. LRUInit(Init);
296. break;
297. case CLOCK:
298. CLOCKInit(Init);
299. break;
300. default:
301. break;
303. }
304. CLOCKHELPINT=0;
305. }// 1,2,3,4 -> 5,2\*(1),3(1),4
306. public void FIFOInit(int[]Init){/\* 这个init是没+1的 \*/
308. for(int i=0; i<PhysicalPageNumber;i++){
309. int []toADD=new int[2];
310. toADD[0]=i+1;
311. toADD[1]=Init[i]+1;
312. FIFO.add( toADD);
313. }
314. }
315. public void LRUInit(int[]Init){
316. //key是物理页号
317. for(int i=0;i<PhysicalPageNumber;i++){
318. int []toADD=new int[2];
319. toADD[0]=i+1;//虚拟页号
320. toADD[1]=0;
321. LRUhashMap.put(Init[i]+1, toADD);
322. }
323. }
324. public void CLOCKInit(int[]Init){
325. for(int i=0;i<PhysicalPageNumber;i++){
326. CLOCK[i][0]=i+1;//虚拟页号,装的
327. CLOCK[i][1]=Init[i]+1;//实际的物理页号
328. }
329. }
330. public void increaseWhenTlbAndLru(int RamNumber){
331. int [] VALUE= LRUhashMap.get(RamNumber);
332. VALUE[1]++;
333. LRUhashMap.put(RamNumber,VALUE);
334. }
335. public int getprocessId(){
336. return processId;
337. }
338. public int getVirtualPageCount(){
339. return virtualPageCount;
340. }
341. public void writePageTable(int row,int realPageFrameNumber){
342. PageTable[row]=realPageFrameNumber;
343. }
344. //pub
345. public void swtichAL(algori sourceAlgori,algori targetAlgori){
346. readPagetimereset();
347. PageTableMissTimeReset();
348. if(sourceAlgori==algori.FIFO&&targetAlgori==algori.LRU)FIFOtoLRU();
349. if(sourceAlgori==algori.FIFO&&targetAlgori==algori.CLOCK)FIFOtoCLOCK();
350. if(sourceAlgori==algori.LRU&&targetAlgori==algori.FIFO)LRUtoFIFO();
351. if(sourceAlgori==algori.LRU&&targetAlgori==algori.CLOCK)LRUtoCLOCK();
352. if(sourceAlgori==algori.CLOCK&&targetAlgori==algori.FIFO)CLOCKtoFIFO();
353. if(sourceAlgori==algori.CLOCK&&targetAlgori==algori.LRU) CLOCKtoLRU();
354. }
355. private void FIFOtoLRU()
356. {
357. //很简单，从队首先出来的，被访问次数是0，后面的依次递增即可
358. System.out.println("进程"+processId+"切换完毕");
359. }
360. private void FIFOtoCLOCK(){
361. // //两个多的标志位给1，两个少的标志位给0
362. //指针放在第一个零上
363. System.out.println("进程"+processId+"切换完毕");
364. }
365. private void LRUtoFIFO(){
366. //和FIFOtoLRU返回来
367. System.out.println("进程"+processId+"切换完毕");
368. }
369. private void LRUtoCLOCK(){
370. //两个多的标志位给1，两个少的标志位给0
371. System.out.println("进程"+processId+"切换完毕");
372. }
373. private void CLOCKtoFIFO(){
374. //不好做模拟，就随缘写了
375. System.out.println("进程"+processId+"切换完毕");
376. }
377. private void CLOCKtoLRU(){
378. //也简单写了，标志位是1的给访问次数是2，标志位是0的，给访问次数是1
379. System.out.println("进程"+processId+"切换完毕");
380. }
381. public int ReadPageForm(int virtualPageNumber){//返回的是物理页号
382. readPagetimePlusOne();//每次访问
383. //这里可能会引到缺页中断
384. if(PageTable[ virtualPageNumber-1 ]>0){
386. changeDSWhenHit(PageTable[ virtualPageNumber-1 ]);
387. return PageTable[ virtualPageNumber-1 ];///\*\*\*\*\*\*\*\*\*\*\*后面如果有溢出或者是报错，注意这个地方 \*/
388. }
389. else{
390. System.out.println("进程"+processId+"的"+virtualPageNumber+"页不在内存中,发生缺页中断");
391. return PageMissing(virtualPageNumber);
392. }
393. }
394. private void changeDSWhenHit(int RamPageNumber){
395. switch(ALGORI){
396. case FIFO://啥都不用做
397. break ;
398. case LRU:
399. LRUchangeWhenHit(RamPageNumber);
400. break;
401. case CLOCK ://这里我们不扫表盘，只循环查找一下把对应的标志位置为1
402. CLOCKchangeWhenHit(RamPageNumber);
403. break;
404. }
405. }
406. private void LRUchangeWhenHit(int RamPageNumber){//给对应的物理页号的访问次数加一
407. int[] array = LRUhashMap.get(RamPageNumber);
408. if (array == null) {
409. throw new RuntimeException("LRU没有正确维护");
410. }
411. array[1] += 1; // 对数组第一个元素加1
412. LRUhashMap.put(RamPageNumber, array); // 更新HashMap中的值
413. }
414. private void CLOCKchangeWhenHit(int RamPageNumber){
415. boolean find=false;
416. for(int i=0;i<PhysicalPageNumber;i++){
417. if(RamPageNumber== Math.abs(CLOCK[i][1])){
418. CLOCK[i][1]=Math.abs(CLOCK[i][1]);
419. find=true;
420. i=PhysicalPageNumber;
421. }
422. }
424. if(!find){
425. throw new RuntimeException("CLOCK算法没有正确维护");
426. }
427. }
428. private int PageMissing(int virtualPageNumber ){//
429. PageTableMissTimePlusOne();//
430. int []changeInfo;//第一个是选择了哪个物理页框，第二个是原来被替换出去的虚拟页框号是多少
432. switch (this.ALGORI) {
433. case LRU:
434. changeInfo = LRUMissing(virtualPageNumber);
435. break;
436. case FIFO:
437. changeInfo = FIFOMissing(virtualPageNumber);
438. break;
439. case CLOCK:
440. changeInfo= CLOCKMissing(virtualPageNumber);
441. break;
442. default:
443. throw new RuntimeException("在切换算法时枚举类型出错了");
444. }
445. System.out.println(ALGORI+"进程"+processId+"把进程"+changeInfo[1]+"号虚拟页框从"+changeInfo[0]+"号物理页框替换了出去，"+"把"+virtualPageNumber+"号虚拟页面替换了进来");
446. writePageTable(virtualPageNumber-1, changeInfo[0]);//写页表
447. return changeInfo[1];
448. }
450. private int[] LRUMissing(int virtualPageNumber){//缺少异常处理
451. int[]result=new int[2]; //虚拟页号和物理页号都是加一的
452. //在替换时要把其他没有被换掉的相关数据减少一点
453. Integer minKey = null;
454. int minValue = Integer.MAX\_VALUE;
455. for (HashMap.Entry<Integer, int[]> entry : LRUhashMap.entrySet()) {
456. int[] valueArray = entry.getValue();
457. if (valueArray.length > 1 && valueArray[1] < minValue) {
458. minValue = valueArray[1];
459. minKey = entry.getKey();
460. }
461. //valueArray[1] =valueArray[1]>=3? valueArray[1]-1: valueArray[1];
462. // LRUhashMap.put(entry.getKey(), valueArray);
463. //可以进行各种设计，这里就用最基础的，不减少，并且给1
464. }
465. result[0]=minKey;
466. result[1]=LRUhashMap.get(minKey)[0];
467. int [] WantToInsert=new int[2] ;
468. WantToInsert[0]=virtualPageNumber;
469. WantToInsert[1]=1;//这里不要给1
470. LRUhashMap.put(minKey,WantToInsert);
472. return result;
473. }
474. private int[] FIFOMissing(int virtualPageNumber){//虚拟页号没有减1，改页表要改减过的index
475. int[]result=new int[2];
476. int[] firstElementRemoved = FIFO.poll();
477. result[1]= firstElementRemoved[1];
478. result[0]=firstElementRemoved[0];
479. firstElementRemoved[1]=virtualPageNumber;
480. FIFO.add(firstElementRemoved);
481. return result;
482. }
483. private int[] CLOCKMissing(int virtualPageNumber){
484. int[]result=new int[2]; //
485. boolean BREAK=false;
486. CLOCKHELPINT=CLOCKHELPINT%PhysicalPageNumber;
487. while(!BREAK){
488. if(CLOCK[CLOCKHELPINT][1]==0){
489. throw new RuntimeException("clock算法中出现了有的 物理页号是0，不应该是这样");
490. }
491. if(CLOCK[CLOCKHELPINT][1]>0){
492. CLOCK[CLOCKHELPINT][1]\*=-1;
493. CLOCKHELPINT=(CLOCKHELPINT+1)%PhysicalPageNumber;
494. }else{//找到是0的了，可以不用接着循环了
495. BREAK=true;
497. CLOCK[CLOCKHELPINT][1]\*=-1;
498. result[1]=CLOCK[CLOCKHELPINT][0];
499. result[0]=CLOCK[CLOCKHELPINT][1];
500. CLOCK[CLOCKHELPINT][0]=virtualPageNumber;
502. CLOCKHELPINT=(CLOCKHELPINT+1)%PhysicalPageNumber;
504. }
506. }
508. return result;
509. }
511. }

这段Java代码定义了一个MyProcessTarget2类，用于模拟操作系统的内存管理，特别是页表和页面置换算法。该类包含多个成员变量，用于跟踪进程的物理页框数量、虚拟页数量、当前使用的页面置换算法、页表、FIFO队列、LRU哈希表和CLOCK算法所需的数据结构。MyProcessTarget2类提供了多种方法来初始化进程的页表和置换算法，以及处理页面缺失时的替换策略。代码中实现了FIFO、LRU和CLOCK三种页面置换算法，并提供了方法来在它们之间切换。此外，代码还包含了统计页面访问时间和页表缺失时间的功能，用于评估算法的性能。

构造函数

public MyProcessTarget2(int ID, int VPC, int PhysiPN, algori ALGORIFromRam, int[] Init) {

// 初始化进程的ID、虚拟页数量、物理页框数量和页面置换算法

// 根据算法类型初始化页表、LRU哈希表、CLOCK数组或FIFO队列

// 初始化页面访问时间和页表缺失时间的统计变量

}

FIFO初始化方法

public void FIFOInit(int[] Init) {

// 根据初始化数组，将虚拟页号和物理页框号存储在FIFO队列中

// 每个队列元素包含物理页框号和对应的虚拟页号

}

LRU初始化方法

public void LRUInit(int[] Init) {

// 根据初始化数组，将虚拟页号和访问次数存储在LRU哈希表中

// 每个哈希表项包含物理页框号和对应的虚拟页号以及访问次数

}

CLOCK初始化方法

public void CLOCKInit(int[] Init) {

// 根据初始化数组，将虚拟页号和物理页框号存储在CLOCK数组中

// 每个数组元素包含虚拟页号和对应的物理页框号

}

增加LRU访问次数方法

public void increaseWhenTlbAndLru(int RamNumber) {

// 根据物理页框号，增加LRU哈希表中对应页面的访问次数

}

获取进程ID方法

public int getprocessId() {

// 返回进程的ID

}

获取虚拟页数量方法

public int getVirtualPageCount() {

// 返回进程的虚拟页数量

}

写入页表方法

public void writePageTable(int row, int realPageFrameNumber) {

// 根据虚拟页号，将物理页框号写入页表的指定行

}

切换算法方法

public void swtichAL(algori sourceAlgori, algori targetAlgori) {

// 重置页面访问时间和页表缺失时间的统计变量

// 根据源算法和目标算法，调用相应的切换方法

}

FIFO到LRU切换方法

private void FIFOtoLRU() {

// 将FIFO队列中的页面信息转换为LRU哈希表中的访问次数信息

}

FIFO到CLOCK切换方法

private void FIFOtoCLOCK() {

// 将FIFO队列中的页面信息转换为CLOCK数组中的标志位信息

}

LRU到FIFO切换方法

private void LRUtoFIFO() {

// 将LRU哈希表中的页面信息转换为FIFO队列中的信息

}

LRU到CLOCK切换方法

private void LRUtoCLOCK() {

// 将LRU哈希表中的页面信息转换为CLOCK数组中的标志位信息

}

CLOCK到FIFO切换方法

private void CLOCKtoFIFO() {

// 将CLOCK数组中的页面信息转换为FIFO队列中的信息

}

CLOCK到LRU切换方法

private void CLOCKtoLRU() {

// 将CLOCK数组中的页面信息转换为LRU哈希表中的访问次数信息

}

从页表中读取页面方法

public int ReadPageForm(int virtualPageNumber) {

// 增加页面访问时间的统计

// 检查虚拟页号是否在页表中，如果不在，则触发页面缺失

// 如果在，则更新数据结构并返回物理页框号

}

更新数据结构方法

private void changeDSWhenHit(int RamPageNumber) {

// 根据当前使用的页面置换算法，更新数据结构以反映页面命中

}

LRU页面命中更新方法

private void LRUchangeWhenHit(int RamPageNumber) {

// 增加LRU哈希表中对应页面的访问次数

}

CLOCK页面命中更新方法

private void CLOCKchangeWhenHit(int RamPageNumber) {

// 将CLOCK数组中对应页面的标志位置为1

}

处理页面缺失方法

private int PageMissing(int virtualPageNumber) {

* + - 1. 页表类

1. class PageInfo {
2. int progressNumber;
3. int virtualPageNumber;
4. int RamNumber;
5. int recentUseCount;
6. public int getPageInFoprogressNumber(){
7. return progressNumber;
8. }
9. public PageInfo(int progressNumber, int virtualPageNumber, int number) {
10. this.progressNumber = progressNumber;
11. this.virtualPageNumber = virtualPageNumber;
12. this.RamNumber = number;
13. recentUseCount=0;
14. }
15. public PageInfo(int progressNumber, int virtualPageNumber, int number,int RecentUseCount) {
16. this.progressNumber = progressNumber;
17. this.virtualPageNumber = virtualPageNumber;
18. this.RamNumber = number;
19. recentUseCount=RecentUseCount;
20. }
21. public void recentUseCountSubIfOtherReplace(){
22. if(recentUseCount>=2) recentUseCount--;
23. }
24. public int getRecentUseCount(){
25. return recentUseCount;
26. }
27. public int isThisRow(int ProgressNumber,int VirtualPageNumber){//tlb命中时要给那一项的recent值加1
28. if(ProgressNumber==this.progressNumber&&VirtualPageNumber==this.virtualPageNumber)
29. {
30. // recentUseCount++;
31. return RamNumber;
32. }
33. else
34. return -1;
35. }
36. public void recentUsePlus(){
37. recentUseCount++;
38. }
39. public String toString(){
40. return "在TLB中访问到"+progressNumber+"号进程中"+virtualPageNumber+"号虚拟页号在"+RamNumber+"号物理页框中";
41. }
43. }

这段Java代码定义了一个名为PageInfo的类，用于存储与页面置换相关的信息。PageInfo类包含五个成员变量和一个构造函数，以及几个方法来获取和更新这些信息。以下是每个成员变量和方法的详细解释：

成员变量

progressNumber: 表示进程编号。

virtualPageNumber: 表示虚拟页号。

RamNumber: 表示物理页框号。

recentUseCount: 表示最近使用次数，用于LRU算法。

构造函数

PageInfo(int progressNumber, int virtualPageNumber, int number): 初始化进程编号、虚拟页号和物理页框号，并设置最近使用次数为0。

PageInfo(int progressNumber, int virtualPageNumber, int number, int RecentUseCount): 初始化进程编号、虚拟页号、物理页框号和最近使用次数。

方法

getPageInFoprogressNumber(): 返回进程编号。

recentUseCountSubIfOtherReplace(): 如果其他页面被替换，减少最近使用次数（如果大于等于2）。

getRecentUseCount(): 返回最近使用次数。

isThisRow(int ProgressNumber, int VirtualPageNumber): 检查给定的进程编号和虚拟页号是否与当前页面信息匹配。如果匹配，返回物理页框号；如果不匹配，返回-1。

recentUsePlus(): 增加最近使用次数。

toString(): 返回页面信息的字符串表示形式，用于打印输出。

类的用途

PageInfo类用于跟踪每个页面的信息，包括它属于哪个进程、它在虚拟内存中的位置以及它在物理内存中的位置。这个类在页面置换算法中特别有用，因为它可以帮助算法决定哪些页面应该被替换，哪些页面应该保留。例如，LRU算法可以使用最近使用次数来确定哪个页面是最久未使用的，因此应该被替换。

通过提供这些信息和方法，PageInfo类为内存管理程序提供了一个灵活和强大的工具，可以用来实现和管理不同类型的页面置换算法。

* 1. 实验结果与分析
     1. 测试数据集

由于这个target2实现的功能比target1多很多，不方便把每一种测试用例的执行过程都截图，下面简述代码都测试通过了哪些类型的测试用例，并且会节选一些截图

1:创建进程（包括虚拟页面不到4的）、创建进程到满、展示页面、删除进程、展示页面

2 创建进程1、创建进程2、读进程2页面2两次、读进程2页面1一次，这时候再读进程2的虚拟页面3，应该根据LRU算法把进程2的虚拟页面1从tlb中替换掉

3.创造一个有5个虚拟页面的进程、这时他会把1，2，3，4页放到内存里

这时候访问第五个页面，观察到发生了缺页中断，然后再接着访问一次，这时候应该是访问tlb命中，然后创造一个新进程

有1，2，3，4，5，6，7，8八个虚拟页面，这个时候访问一下7，8会造成两次缺页中断，这时候会把进程1在tlb中的顶出去

这个时候再访问进程1的第五个页面，应该是tlb没命中，但也不发生缺页中断，

4.从FIFO切换到LRU算法，新建进程1，有六个页，然后对四个页面一个访问一次，两个访问3次

此时是3310

然后访问第五个页面，这时候应该把0的那个替换出去，变成3311

然后重复访问这个新页面一次

变成3312

然后再访问一个新页面

变成3312

后面在这里衔接一个缺页率计算，然后再把这个进程删除，再来个缺页率计算，把缺页率计算的功能验证

5

clock

新建一个进程有6个虚拟进程，

0，1，2，3，4，5

这时默认clock是在0，然后0，1，2，3在内存里 这时候我们访问4，应该是0被替换出去

4，1，2，3

这时候指针是在1上，我们访问一下1，它应该标志位变成1，

然后我们访问5，这时候如果把2替换出去，就对了

（在这之中要注意不能让tlb起作用）

* + 1. 性能分析

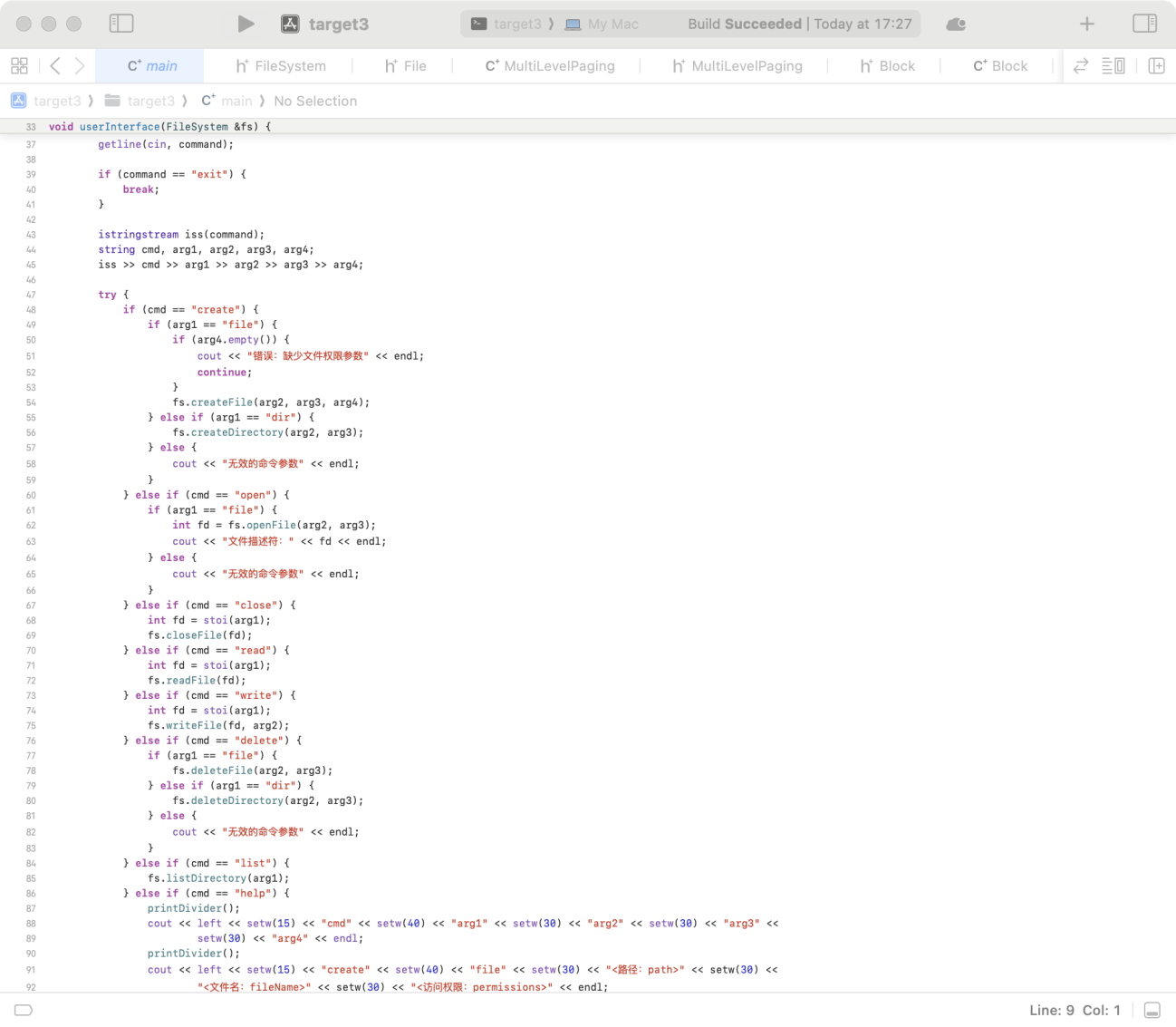
从在代码调试的过程中，给我最明显的感受就是，TLB的命中率相当之高！几个替换算法之间，由于数据集大小有限，没有感受到太大的差异

* 1. 实验总结

这个target2的任务相比起target1难度大了不少，主要在于类的设计以及函数件相互调用的关系以及函数的设计，从这个过程中我体会到，函数更多是为了类的成员变量服务。

1. 文件系统
   1. 实验目的

* 理解文件系统的主要组成和工作原理。
* 学习并掌握文件的存储方式，文件的创建、读写、删除等操作的实现机制。
* 提高对文件系统内部组织和管理的认识及熟悉实际系统使用上的细节。
  1. 实验原理
* 文件系统的基本构造和功能：文件系统由文件和目录组成，带有访问权限和管理信息，实现了对底层存储设备的抽象。文件系统是操作系统用于存储、组织、共享和访问数据的一种机制。
* 文件的存储方式：包括连续存储方式、链接存储方式和索引存储方式。
* 文件和目录：文件系统将所有的信息存储在文件中。每个文件都有一个唯一的名字，并放置在目录中。目录本身也是文件，它包含一组链接，每个链接将一个文件名与一个文件相关联。
* 权限和特权：文件系统为每个文件和目录都定义了一组权限，如读（r）、写（w）、执行（x）等。还可以定义谁可以使用这些权限，如文件所有者，所在组的成员，或者其他人。
* 空间管理：文件系统管理存储设备上的可用空间。当文件被创建或扩展时，文件系统需要找到可用的空间。当文件被删除或缩小时，其空间需要被回收。
* 文件系统的层次结构：文件系统通常使用一个树形结构来组织文件和目录。树的根是一个特殊的目录，称为根目录。每个目录可以包含文件和其他目录。
* 硬链接和软链接：硬链接是另一个指向同一个文件的引用，而软链接（或符号链接）则是指向另一个链接的引用。使用链接可以方便地共享和组织文件。
* 一致性和完整性：为了保证在系统崩溃或断电情况下数据的一致性和完整性，文件系统需要在修改数据时使用某种形式的原子操作或者日志。当操作失败时，可以回到操作前的状态。
  1. 实验设计与实现
     1. 设计思路
* 确定基本结构和组织方式：设计一个可以支持树状目录的文件系统，包括文件和目录的数据结构，选择合适的文件存储方式和空闲空间管理策略。文件系统是操作系统管理存储设备上数据的一种机制，它通过抽象底层存储设备，实现了数据的存储、组织、共享和访问。文件系统主要由文件和目录组成，每个文件和目录都带有访问权限和管理信息。文件的存储方式包括连续存储、链接存储和索引存储。文件系统中的每个文件都有一个唯一的名字，并放置在目录中，目录本身也是文件，包含一组链接，每个链接将一个文件名与一个文件相关联。文件系统为每个文件和目录定义了权限，如读、写、执行等，并规定了谁可以使用这些权限。文件系统还负责管理存储设备上的可用空间，确保在文件创建或扩展时能够找到空间，并在文件删除或缩小时回收空间。文件系统通常使用树形结构来组织文件和目录，根目录是树的根。硬链接是另一个指向同一个文件的引用，而软链接（或符号链接）是指向另一个链接的引用。为了保证数据的一致性和完整性，文件系统需要在修改数据时使用原子操作或日志，以便在操作失败时恢复到操作前的状态。
  + 1. 核心代码
       1. 获取目录对象方法

1. // 获取目录对象
2. // 参数:
3. //   path - 目录的路径
4. // 返回:
5. //   指向目录的指针，如果目录不存在则返回 nullptr
6. Directory \*FileSystem::getDirectory(const string &path) {
7. // 检查路径是否为空
8. if (path.empty()) {
9. return nullptr;
10. }
12. // 获取根目录的指针
13. Directory \*currentDir = root.get();
15. // 如果路径是根目录，直接返回根目录指针
16. if (path == "/") {
17. return currentDir;
18. }
20. // 将路径分割成各级目录名
21. vector<string> dirs = splitPath(path);
23. // 遍历各级目录名，逐级查找目录
24. for (const string &dirName: dirs) {
25. // 在当前目录的子目录中查找匹配的目录名
26. auto it = find\_if(currentDir->directories.begin(), currentDir->directories.end(), [&](const auto &subDir) {
27. return subDir->name == dirName;
28. });
30. // 如果找到匹配的子目录，更新当前目录指针
31. if (it != currentDir->directories.end()) {
32. currentDir = it->get();
33. } else {
34. // 如果找不到匹配的子目录，返回 nullptr
35. return nullptr;
36. }
37. }
39. // 返回找到的目录指针
40. return currentDir;
41. }
    * + 1. 分割路径方法
42. // 将路径字符串分割成各级目录名
43. // 参数:
44. //   path - 目录路径字符串
45. // 返回:
46. //   包含各级目录名的向量
47. vector<string> FileSystem::splitPath(const string &path) {
48. vector<string> dirs; // 用于存储各级目录名
49. size\_t pos = 0; // 当前处理的位置
50. size\_t nextPos; // 下一个分隔符的位置
52. // 循环查找路径中的每一个 '/' 分隔符
53. while ((nextPos = path.find("/", pos)) != string::npos) {
54. // 如果分隔符不在起始位置，则提取子串作为一个目录名
55. if (nextPos != pos) {
56. dirs.push\_back(path.substr(pos, nextPos - pos));
57. }
58. // 更新当前位置，跳过分隔符
59. pos = nextPos + 1;
60. }
62. // 处理最后一个目录名
63. if (pos < path.length()) {
64. dirs.push\_back(path.substr(pos));
65. }
67. return dirs; // 返回包含各级目录名的向量
68. }
    * + 1. 解析权限方法
69. // 解析权限字符串，将其转换为权限标志的整数表示
70. // 参数:
71. //   permissions - 权限字符串，如 "r", "w", "rw", "rwx"
72. // 返回:
73. //   权限标志的整数表示，如果权限字符串无效则返回 -1
74. int FileSystem::parsePermissions(const string &permissions) {
75. // 如果权限字符串是 "r"，返回读权限标志
76. if (permissions == "r") {
77. return Permissions::READ;
78. }
79. // 如果权限字符串是 "w"，返回写权限标志
80. else if (permissions == "w") {
81. return Permissions::WRITE;
82. }
83. // 如果权限字符串是 "rw"，返回读和写权限标志
84. else if (permissions == "rw") {
85. return Permissions::READ | Permissions::WRITE;
86. }
87. // 如果权限字符串是 "rwx"，返回读、写和执行权限标志
88. else if (permissions == "rwx") {
89. return Permissions::READ | Permissions::WRITE | Permissions::EXECUTE;
90. }
91. // 如果权限字符串无效，返回 -1
92. else {
93. return -1;
94. }
95. }
    * + 1. 查找文件方法
96. // 在指定目录中查找文件
97. // 参数:
98. //   dir - 指向目录对象的指针
99. //   fileName - 要查找的文件名
100. // 返回:
101. //   指向文件对象的指针，如果文件不存在则返回 nullptr
102. File \*FileSystem::findFile(Directory \*dir, const string &fileName) {
103. // 使用 find\_if 函数在目录的文件列表中查找匹配的文件名
104. auto it = find\_if(dir->files.begin(), dir->files.end(), [&](const auto &file) {
105. return file->name == fileName;
106. });
108. // 如果找到匹配的文件，返回指向文件对象的指针；否则返回 nullptr
109. return it != dir->files.end() ? it->get() : nullptr;
110. }
     * + 1. 查找子目录方法
111. // 在指定目录中查找子目录
112. // 参数:
113. //   dir - 指向目录对象的指针
114. //   dirName - 要查找的子目录名
115. // 返回:
116. //   指向子目录对象的指针，如果子目录不存在则返回 nullptr
117. Directory \*FileSystem::findDirectory(Directory \*dir, const string &dirName) {
118. // 使用 find\_if 函数在目录的子目录列表中查找匹配的子目录名
119. auto it = find\_if(dir->directories.begin(), dir->directories.end(), [&](const auto &subDir) {
120. return subDir->name == dirName;
121. });
123. // 如果找到匹配的子目录，返回指向子目录对象的指针；否则返回 nullptr
124. return it != dir->directories.end() ? it->get() : nullptr;
125. }
     * + 1. 创建文件方法
126. // 在指定目录中创建文件
127. // 参数:
128. //   path - 目录路径
129. //   fileName - 文件名
130. //   permissions - 文件权限字符串
131. // 抛出:
132. //   runtime\_error 如果目录不存在或文件已存在或权限非法
133. void FileSystem::createFile(const string &path, const string &fileName, const string &permissions) {
134. // 获取指定路径的目录指针
135. Directory \*dir = getDirectory(path);
136. if (!dir) {
137. throw runtime\_error("目录不存在");
138. }
140. // 检查文件是否已存在
141. if (findFile(dir, fileName)) {
142. throw runtime\_error("文件已存在");
143. }
145. // 创建新的文件对象
146. auto newFile = make\_unique(fileName, permissions);
148. // 解析权限字符串
149. int perm = parsePermissions(permissions);
150. if (perm == -1) {
151. throw runtime\_error("非法权限");
152. }
154. // 初始化文件大小为 0
155. newFile->size = 0;
157. // 将文件添加到目录的文件列表中
158. dir->files.push\_back(move(newFile));
159. cout << "文件创建成功" << endl;
160. }
     * + 1. 打开文件方法
161. // 在指定目录中打开文件
162. // 参数:
163. //   path - 目录路径
164. //   fileName - 文件名
165. // 抛出:
166. //   runtime\_error 如果目录不存在或文件不存在或文件不可读
167. int FileSystem::openFile(const string &path, const string &fileName) {
168. // 获取指定路径的目录指针
169. Directory \*dir = getDirectory(path);
171. // 如果目录不存在，抛出异常
172. if (!dir) {
173. throw runtime\_error("目录不存在");
174. }
176. // 在目录中查找指定文件
177. File \*file = findFile(dir, fileName);
179. // 如果文件不存在，抛出异常
180. if (!file) {
181. throw runtime\_error("文件不存在");
182. }
184. // 解析文件权限
185. int perm = parsePermissions(file->permissions);
187. // 检查文件是否可读
188. if (!(perm & Permissions::READ)) {
189. throw runtime\_error("文件不可读");
190. }
192. // 生成新的文件描述符并将文件与之关联
193. int fd = nextFileDescriptor++;
194. fileDescriptorMap[fd] = file;
196. // 输出文件打开成功信息
197. cout << "打开文件：" << file->name << endl;
199. // 返回文件描述符
200. return fd;
201. }
     * + 1. 关闭文件方法
202. // 关闭指定文件描述符对应的文件
203. // 参数:
204. //   fileDescriptor - 要关闭的文件描述符
205. // 抛出:
206. //   runtime\_error 如果文件描述符无效
207. void FileSystem::closeFile(int fileDescriptor) {
208. // 查找文件描述符对应的文件
209. auto it = fileDescriptorMap.find(fileDescriptor);
211. // 如果找到了文件描述符对应的文件
212. if (it != fileDescriptorMap.end()) {
213. // 从文件描述符映射中移除该文件
214. fileDescriptorMap.erase(it);
216. // 输出关闭文件描述符的信息
217. cout << "关闭文件描述符：" << fileDescriptor << endl;
218. } else {
219. // 如果文件描述符无效，则抛出异常
220. throw runtime\_error("无效的文件描述符");
221. }
222. }
     * + 1. 读取文件方法
223. // 读取指定文件描述符对应的文件内容
224. // 参数:
225. //   fileDescriptor - 要读取的文件描述符
226. // 抛出:
227. //   runtime\_error 如果文件描述符无效或文件没有读权限
228. void FileSystem::readFile(int fileDescriptor) {
229. // 查找文件描述符对应的文件
230. auto it = fileDescriptorMap.find(fileDescriptor);
232. // 如果找不到对应的文件描述符，则抛出异常
233. if (it == fileDescriptorMap.end()) {
234. throw runtime\_error("无效的文件描述符");
235. }
237. // 获取文件对象指针
238. File \*file = it->second;
240. // 检查文件的读权限
241. int perm = parsePermissions(file->permissions);
242. if (!(perm & Permissions::READ)) {
243. throw runtime\_error("文件没有读权限");
244. }
246. // 读取文件内容
247. string content = file->mlp->read();
249. // 输出读取的文件内容
250. cout << "读取文件内容：" << content << endl;
251. }
     * + 1. 写文件方法
252. // 写入数据到指定文件描述符对应的文件
253. // 参数:
254. //   fileDescriptor - 要写入的文件描述符
255. //   data - 要写入的数据
256. // 抛出:
257. //   runtime\_error 如果文件描述符无效或文件没有写权限
258. void FileSystem::writeFile(int fileDescriptor, const string &data) {
259. // 查找文件描述符对应的文件
260. auto it = fileDescriptorMap.find(fileDescriptor);
261. if (it == fileDescriptorMap.end()) {
262. throw runtime\_error("无效的文件描述符");
263. }
265. // 获取文件对象指针
266. File \*file = it->second;
268. // 检查权限
269. int perm = parsePermissions(file->permissions);
270. if (!(perm & Permissions::WRITE)) {
271. throw runtime\_error("文件没有写权限");
272. }
274. // 写入文件
275. file->mlp->write(data);
276. file->size += data.size();
278. // 输出写入文件成功信息
279. cout << "写入文件成功" << endl;
280. }
     * + 1. 删除文件方法
281. // 删除指定目录中的文件
282. // 参数:
283. //   path - 目录路径
284. //   fileName - 要删除的文件名
285. // 抛出:
286. //   runtime\_error 如果目录不存在或文件不存在
287. void FileSystem::deleteFile(const string &path, const string &fileName) {
288. // 获取指定路径的目录指针
289. Directory \*dir = getDirectory(path);
290. if (!dir) {
291. throw runtime\_error("目录不存在");
292. }
294. // 在目录的文件列表中查找匹配的文件
295. auto it = find\_if(dir->files.begin(), dir->files.end(), [&](const auto &file) {
296. return file->name == fileName;
297. });
299. // 如果文件不存在，抛出异常
300. if (it == dir->files.end()) {
301. throw runtime\_error("文件不存在");
302. }
304. // 在文件描述符映射中查找并删除指向该文件的条目
305. for (auto fdIt = fileDescriptorMap.begin(); fdIt != fileDescriptorMap.end();) {
306. if (fdIt->second == it->get()) {
307. fdIt = fileDescriptorMap.erase(fdIt);
308. } else {
309. ++fdIt;
310. }
311. }
313. // 从目录的文件列表中移除文件，智能指针会自动释放内存
314. dir->files.erase(it);
316. // 输出删除文件成功信息
317. cout << "删除文件成功" << endl;
318. }
     * + 1. 创建目录方法
319. // 在指定目录中创建新目录
320. // 参数:
321. //   path - 目标目录路径
322. //   dirName - 新目录的名称
323. // 抛出:
324. //   runtime\_error 如果目标目录不存在或同名目录已存在
325. void FileSystem::createDirectory(const string &path, const string &dirName) {
326. // 获取指定路径的目录对象
327. Directory \*dir = getDirectory(path);
329. // 如果目录不存在，则抛出异常
330. if (!dir) {
331. throw runtime\_error("目录不存在");
332. }
334. // 检查指定目录下是否已存在同名目录，如果存在，则抛出异常
335. if (findDirectory(dir, dirName)) {
336. throw runtime\_error("目录已存在");
337. }
339. // 创建新目录并添加到指定目录下
340. auto newDir = make\_unique(dirName);
341. dir->directories.push\_back(move(newDir));
343. // 输出创建成功信息
344. cout << "目录创建成功" << endl;
345. }
     * + 1. 删除目录方法
346. // 删除指定目录中的子目录
347. // 参数:
348. //   path - 目标目录路径
349. //   dirName - 要删除的子目录名称
350. // 抛出:
351. //   runtime\_error 如果目标目录不存在或子目录不存在
352. void FileSystem::deleteDirectory(const string &path, const string &dirName) {
353. // 获取指定路径的目录对象
354. Directory \*dir = getDirectory(path);
356. // 如果目录不存在，则抛出异常
357. if (!dir) {
358. throw runtime\_error("目录不存在");
359. }
361. // 查找指定目录下是否存在目标目录
362. auto it = find\_if(dir->directories.begin(), dir->directories.end(), [&](const auto &subDir) {
363. return subDir->name == dirName;
364. });
366. // 如果目标目录不存在，则抛出异常
367. if (it == dir->directories.end()) {
368. throw runtime\_error("目录不存在");
369. }
371. // 删除目标目录
372. dir->directories.erase(it);
374. // 输出删除成功信息
375. cout << "删除目录成功" << endl;
376. }
     * + 1. 遍历方法
377. // 列出指定目录的内容
378. // 参数:
379. //   path - 目标目录路径
380. // 抛出:
381. //   runtime\_error 如果目录不存在
382. void FileSystem::listDirectory(const string &path) {
383. // 获取指定路径的目录对象
384. Directory \*dir = getDirectory(path);
386. // 如果目录不存在，则抛出异常
387. if (!dir) {
388. throw runtime\_error("目录不存在");
389. }
391. // 输出目录内容
392. cout << "目录内容：" << endl;
394. // 遍历目录中的文件，并输出文件信息
395. for (const auto &file: dir->files) {
396. cout << "文件： " << file->name << " （权限： " << file->permissions << "），大小： " << file->size << " 字节" << endl;
397. }
399. // 遍历目录中的子目录，并输出子目录名
400. for (const auto &subDir: dir->directories) {
401. cout << "目录： " << subDir->name << endl;
402. }
403. }
     * + 1. 控制台输入
     1. 实验结果与分析
        1. 测试数据集

使用以下命令集进行测试：

* 创建目录：  
  create dir / dir1
* 创建文件并设置权限：

create file /dir1 file1 rw

* 打开文件：

open file /dir1 file1

* 写入文件：

write 0 "Hello,world!"

* 读取文件：

read 0

* 关闭文件：

close 0

* 创建具有不同权限的文件：

create file /dir1 file2 r

create file /dir1 file3 w

create file /dir1 file4 rw

* 尝试写入只读文件：

open file /dir1 file2

write 1 "Text1"

* 尝试读取只写文件：

open file /dir1 file3

* 删除文件并验证删除：

delete file /dir1 file1

list /dir1

* + 1. 实验结果

实验结果表明，文件系统正确实现了目录和文件的创建、打开、写入、读取、关闭等基本操作，并且文件权限管理也按预期工作，能够阻止非法的读写操作。

* 1. 实验结论

在本次实验中，我们成功实现了一个基础文件系统，该系统支持树状目录结构，并实现了文件的创建、打开、关闭、读取、写入和删除等基本操作。通过这个实验，我们深入理解了文件系统的基本原理，包括文件和目录的管理方法，以及文件系统的层次结构和权限管理机制。实验过程不仅加深了我们对文件系统实现细节和操作机制的认识，而且达到了预期的学习目标。

**教师评语评分**

评语：

评分：

评阅人： 年 月 日