**附录 实验报告**

**课程编号：A0801040060**

**操作系统**

**实验报告**



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **姓名** | **柳成林** | **班级学号** | **软件2001** | **成绩** |  |
| **报告名称** | **操作系统实验报告** | | | | |
| **开设学期** | **2021-2022春季学期** | | | | |
| **开设时间** | **第16周——第17周** | | | | |
| **报告完成日期** | **2022年6月22日** | | | | |
| **批改教师** |  | | | | |
| **批改日期** | **2022年6月20** | | | | |

**东北大学软件学院**

**2022年6月**

实验一 进程/线程的同步与互斥

**一、实验目的**

（1）加深对进程概念/线程概念的理解；

（2）进一步认识并发的实质；

（3）分析进程/线程竞争资源的现象，学习解决进程互斥的方法。

（4）了解Windows对进程/线程管理的支持。

**二、问题分析与程序设计**

**问题描述：**

编写一个简单的生产者/消费者示例程序。

生产者消费者问题是一个著名的线程同步问题，该问题描述如下：有一个生产者在生产产品，这些产品将提供给若干个消费者去消费，为了使生产者和消费者能并发执行，在两者之间设置一个具有多个缓冲区的缓冲池，生产者将它生产的产品放入一个缓冲区中，消费者可以从缓冲区中取走产品进行消费，显然生产者和消费者之间必须保持同步，**即不允许消费者到一个空的缓冲区中取产品，也不允许生产者向一个已经放入产品的缓冲区中再次投放产品。**

**问题分析与解决方案：**

生产者消费者问题中很重要的一点在于协调问题：只有生产者生产出了产品，消费者才能进行消费，缓冲区中有产品，消费者才能进行消费，缓冲区中有产品，生产者就必须把产品放在空的缓冲区中。

我们可以在程序开始时需要初始化信号量，为了保证进程的互斥，需要在程序中设置互斥锁，判断可用资源数目时通过信号量来表示（缓冲池空/缓冲池满）。

如果生产者已经生产出产品，消费者可以直接取；如果没有产品，则需要等待生产者进行生产过后才能从缓冲区中取到，取到后该缓冲区置为空。

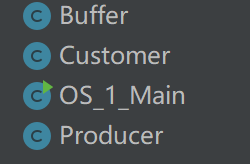
采用Java Thread，使用 Semaphore 类定义原子信号量，Semaphore（信号量）是用来控制同时访问特定资源的线程数量，它通过协调各个线程，以保证合理的使用公共资源。

Semaphore提供了acquire方法来获取一个许可，Semaphore提供release来释放许可；我们可以把acquire方法理解为生产产品，把release方法理解为消费商品。同时可以定义Semaphore变量的大小，来控制缓冲区的大小，比如说，当信号量大小为0时，则不能调acquire。

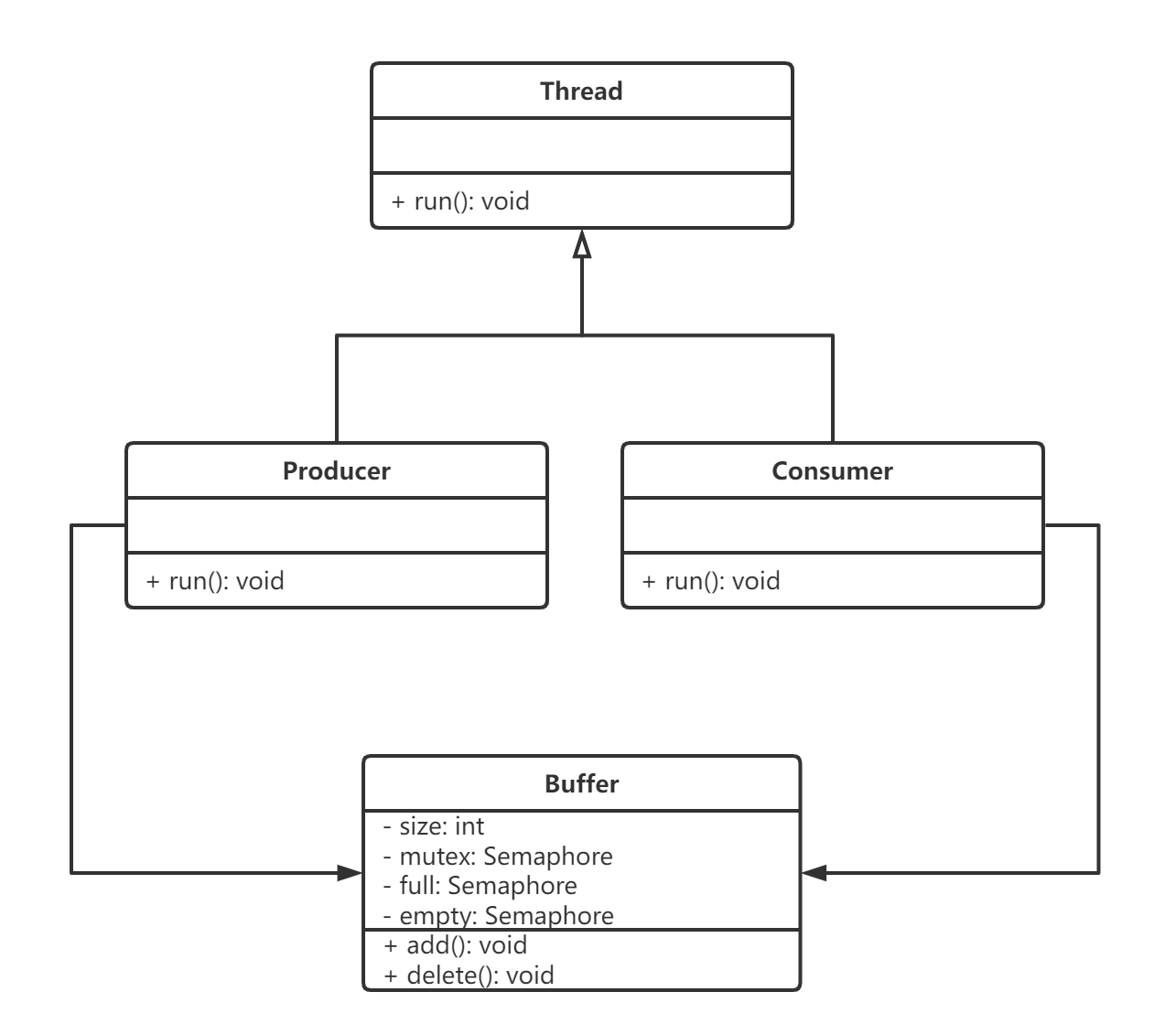
三个信号量：mutex、empty、full：mutex初始化为1，用来实现进程互斥；empty初始化为5，代表缓冲区大小为5；full初始化为0，代表生产者已生产数量为0。

**三、实现过程与测试结果分析**

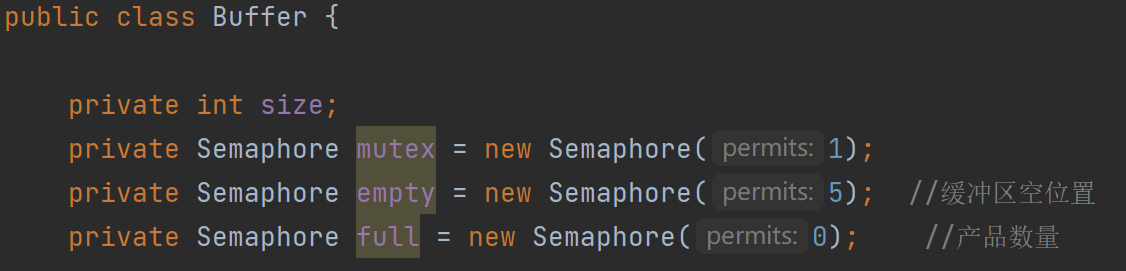
结构：



类图：

****

Buffer：缓冲区，size：int类型，表示当前缓冲区产品的数量，当生产出一个产品时，size加1，当消费一个产品时，size减1；mutex、full、empty为Semaphore类型。



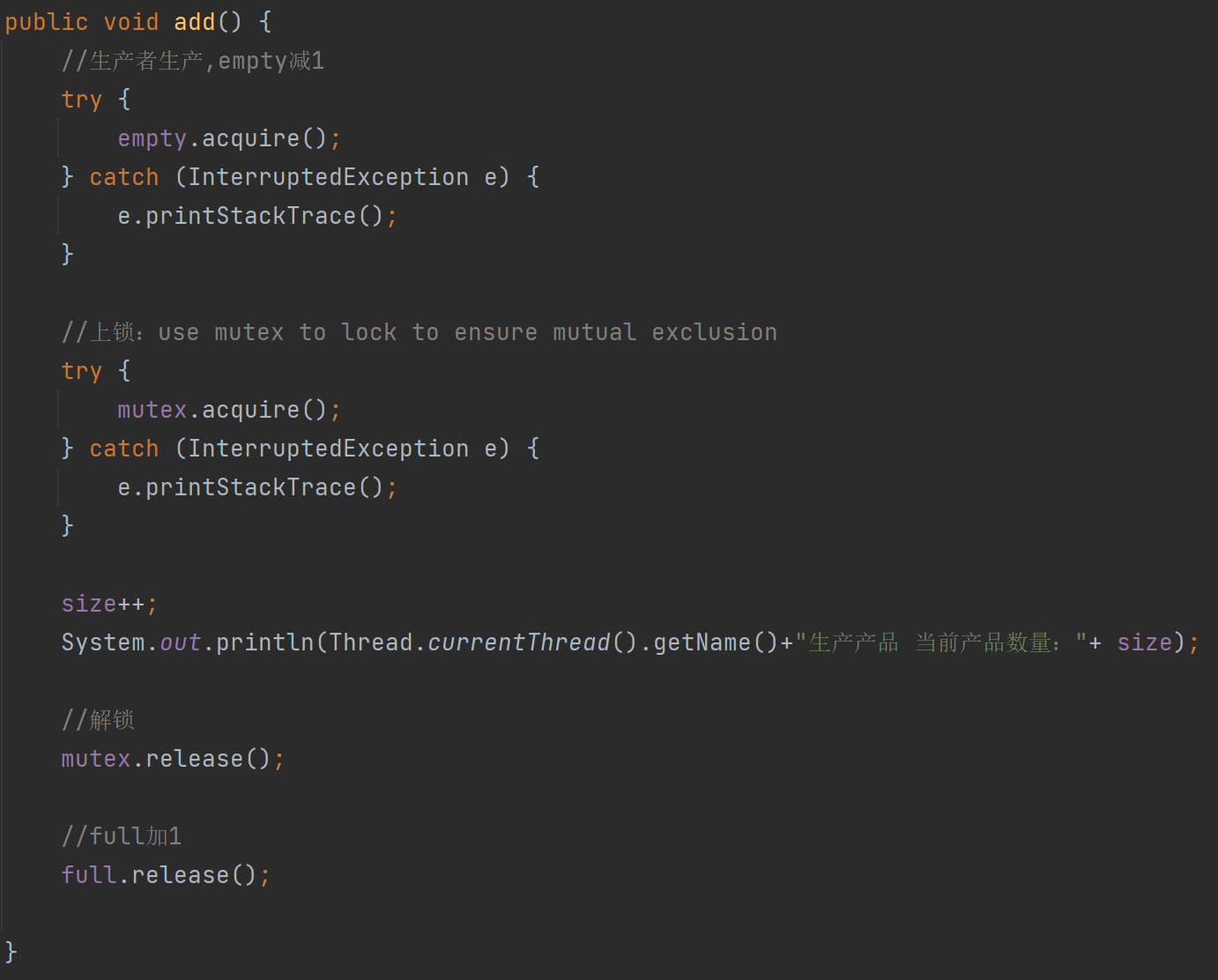
初始化时定义好每一个信号量的大小，这样，当empty等于0时，便不能允许生产者再次生产产品，当full等于0时，也不能让消费者消费产品。

Producer：生产者，继承Thread类，重写run方法，调用buffer的add方法。

Consumer：消费者，继承Thread类，重写run方法，调用buffer的delete方法。

Buffer与Producer、Consumer为1对1关系。

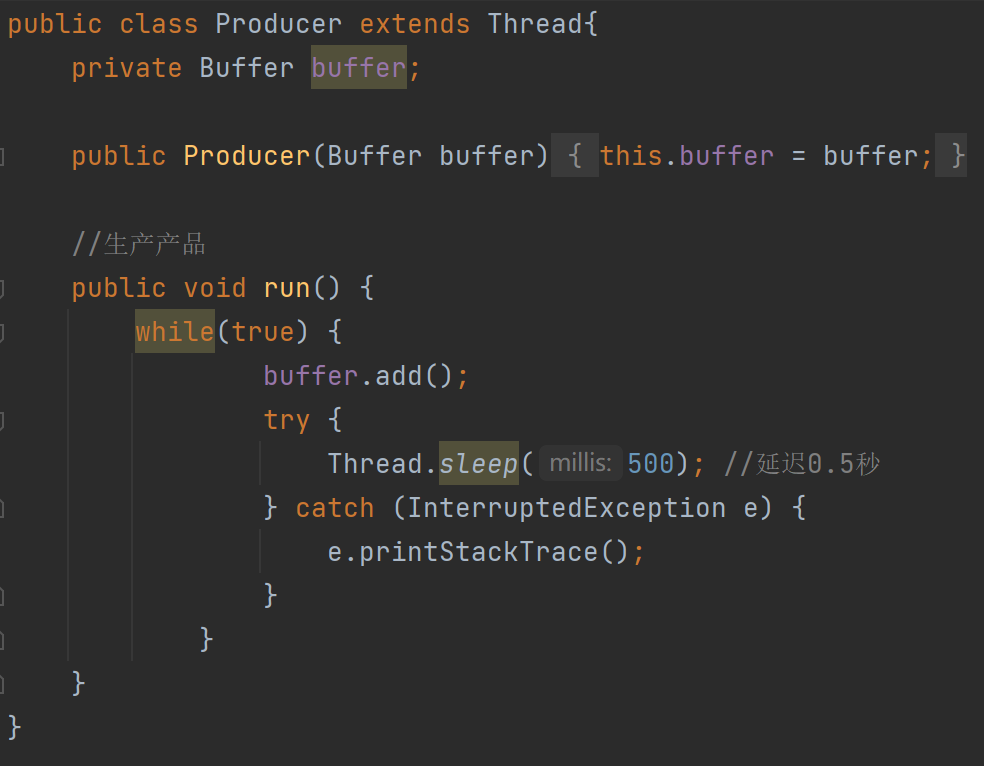
Buffer核心代码：





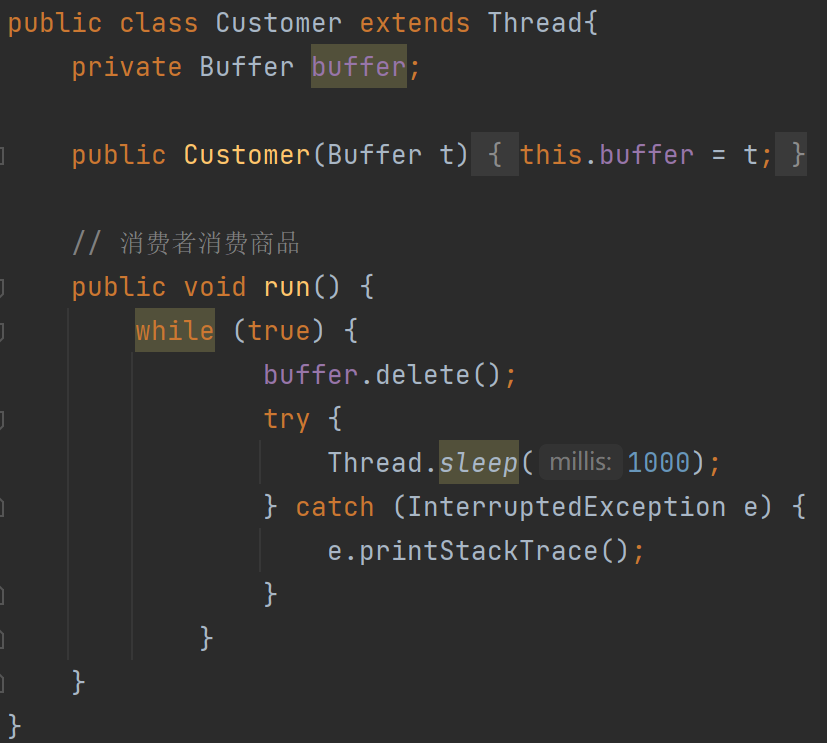
Producer核心代码：

生产者每隔0.5秒生产一个产品，调用buffer的add方法。



Consumer核心代码：

消费者每隔1秒生产一个产品，调用buffer的delete方法。



测试过程：

设置缓冲区大小为5，两个生产者，两个消费者。调试过程中发现生产者生产速度总是与消费者相当，即生产不满5个，总是生产到2被消费者消费掉，再生产到1或2，循环如此。

解决方法：让生产者和消费者的run方法执行时间不同，比如让生产者和消费者的Sleep时间不同。

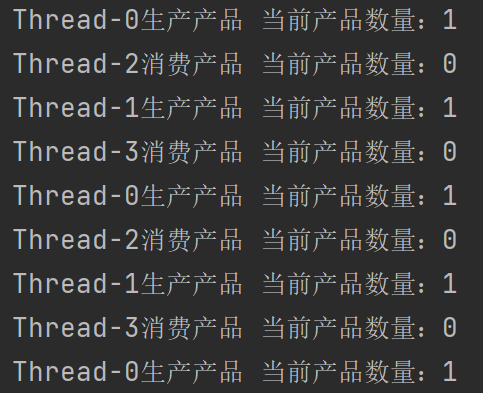
**四、实验结果总结**

运行结果：

缓冲区大小为5：



缓冲区大小为1：



可以看出没有发生冲突。

**五、创新的部分**

实现多生产者多消费者线程同步问题。

**六、对实验的意见与建议**

通过本次实验，我对进程的同步与互斥有了更深的理解，对生产者消费者模型有了更深入的认识。

**七、思考题**

1. 如何控制进程间的互相通信

1、信号量：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

2、管道通信：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

3、消息队列：消息队列是消息的链接表。对消息队列有读权限的进程则可以从消息队列中读取信息。其基本思想是:根据”生产者-消费者”原理,利用内存中公用消息缓冲区实现进程之间的信息交换.内存中开辟了若干消息缓冲区,用以存放消息.每当一个进程向另一个进程发送消息时,便申请一个消息缓冲区,并把已准备好的消息送到缓冲区,然后把该消息缓冲区插入到接收进程的消息队列中,最后通知接收进程.接收进程收到发送里程发来的通知后,从本进程的消息队列中摘下一消息缓冲区,取出所需的信息,然后把消息缓冲区不定期给系统.系统负责管理公用消息缓冲区以及消息的传递.一个进程可以给若干个进程发送消息,反之,一个进程可以接收不同进程发来的消息.显然,进程中关于消息队列的操作是临界区.当发送进程正往接收进程的消息队列中添加一条消息时,接收进程不能同时从该消息队列中到出消息。

4、共享内存：依靠某种同步操作，如互斥锁和信号量等。

1. 什么是进程同步？什么是进程互斥？分别有哪些实现方式？

进程同步：同时访问共享数据可能会导致数据不一致，保持数据间的一致性要求有一定的机制去约束进程。

进程互斥：是指某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。

实现方式：p、v操作。p、v操作是定义在信号量上的两个原语操作。信号量是表示资源的物理量，也是一个与队列有关的整形变量，用s表示。其值仅能由p，v操作原语来改变。当s<0时，其绝对值代表进入信号量执政队列的处于等待状态的进程数：当s>0时，其值代表系统中可用的资源数：当s=0是，表示所有资源都分配给了所有进程。系统利用信号量对进程控制和管理，及控制进程对临界资源或者公共变量的访问，以实现进程的同步与互斥。

实验二 处理机调度（优先权调度算法）

**一、实验目的**

（1）加深对处理机调度的作用和工作原理的理解。

（2）进一步认识并发执行的实质。

**二、问题分析与程序设计**

**问题描述：**

在采用多道程序中，往往有若干个进程同时处于就绪状态。当就绪进程个数大于处理机个数时，就必须依照某种策略来决定哪些进程优先占用处理机。本实验模拟在单处理机情况下的处理机调度，加深我们对处理机调度的作用和工作原理的理解，并帮助我们进一步认是并发执行的实质。

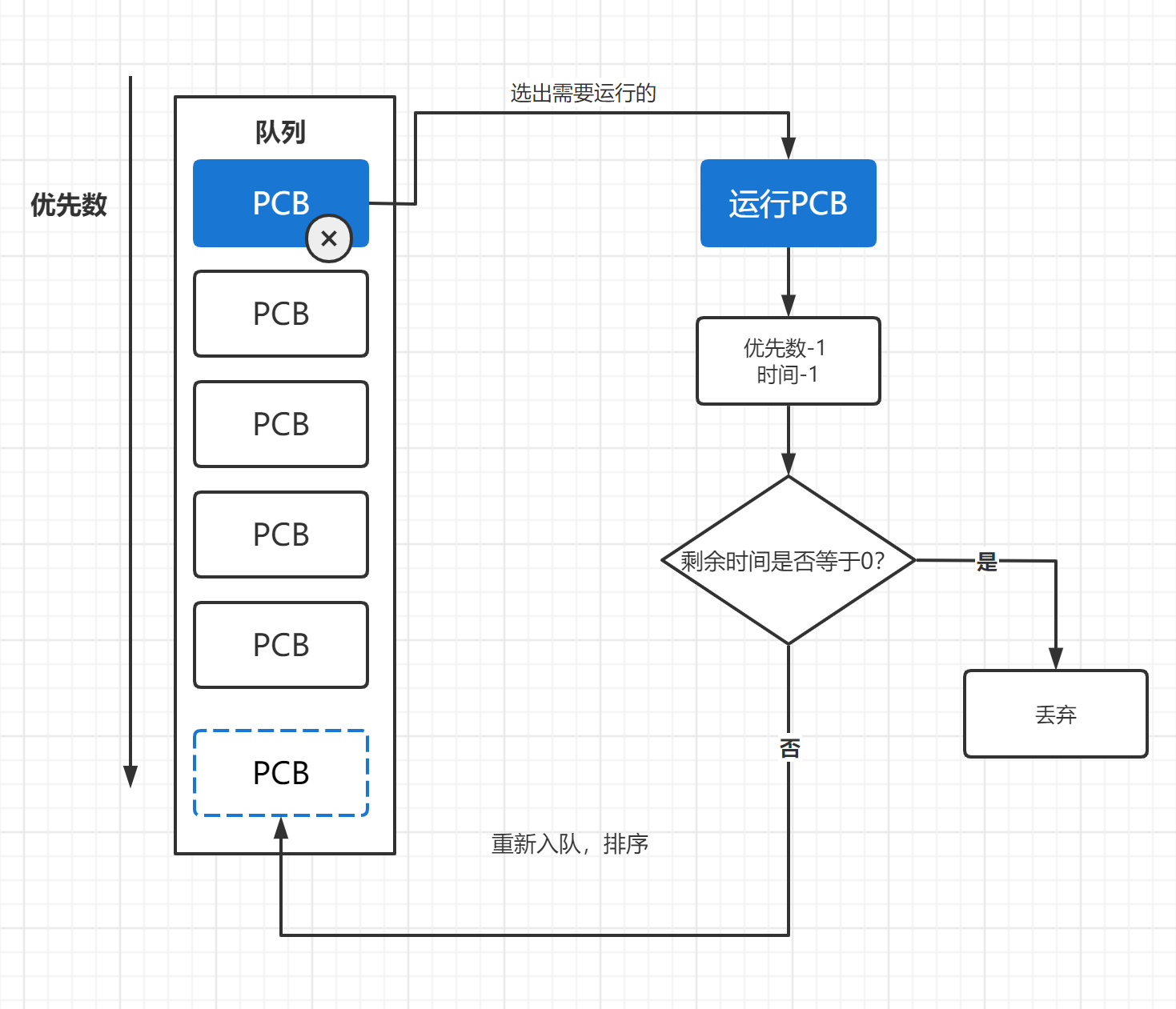
**问题分析与解决方案：**

假设系统中有n个进程，每个进程有一个进程控制块PCB，题目中指出优先数越高，优先级越低，调度时选择优先级最高的进程运行。

在处理机调度的过程中，每次从队列中选择一个优先数最高的PCB进行运行，将该PCB的优先数和时间减1，如果剩余时间大于0，再将其重新添加到队列中，否则直接跳出队列。

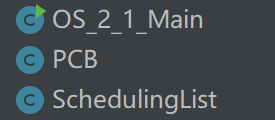
下图是处理机调度的流程图：

左侧是PCB的队列，为链表结构，当有新的PCB进入时，会进行自动排序；每次CPUTIME时，从队列中拿出优先数最高的PCB，进行运行，运行结束后需要判断该PCB是否还要返回到左侧的队列中，如果需要，则再次加入队列，排序。重复上述步骤，直到队列中没有PCB。

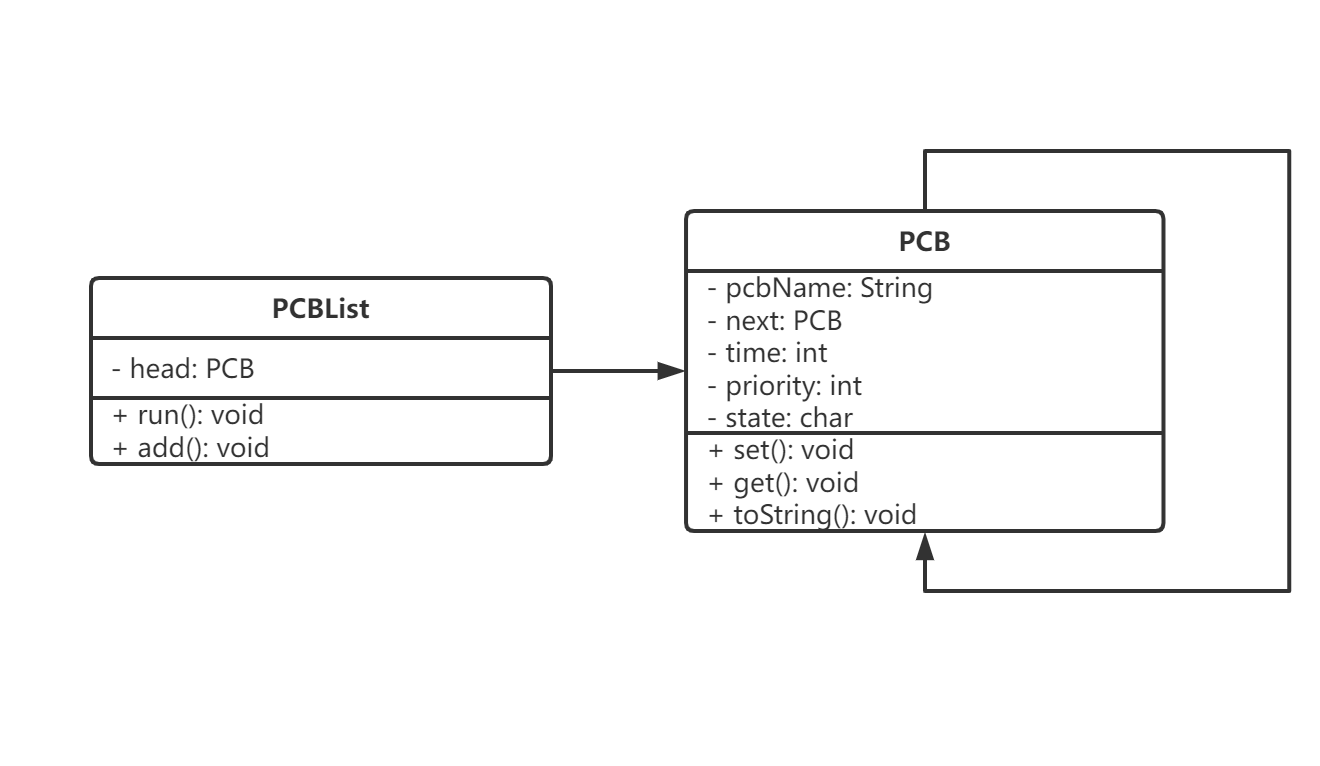


**三、实现过程与测试结果分析**

结构：



类图：

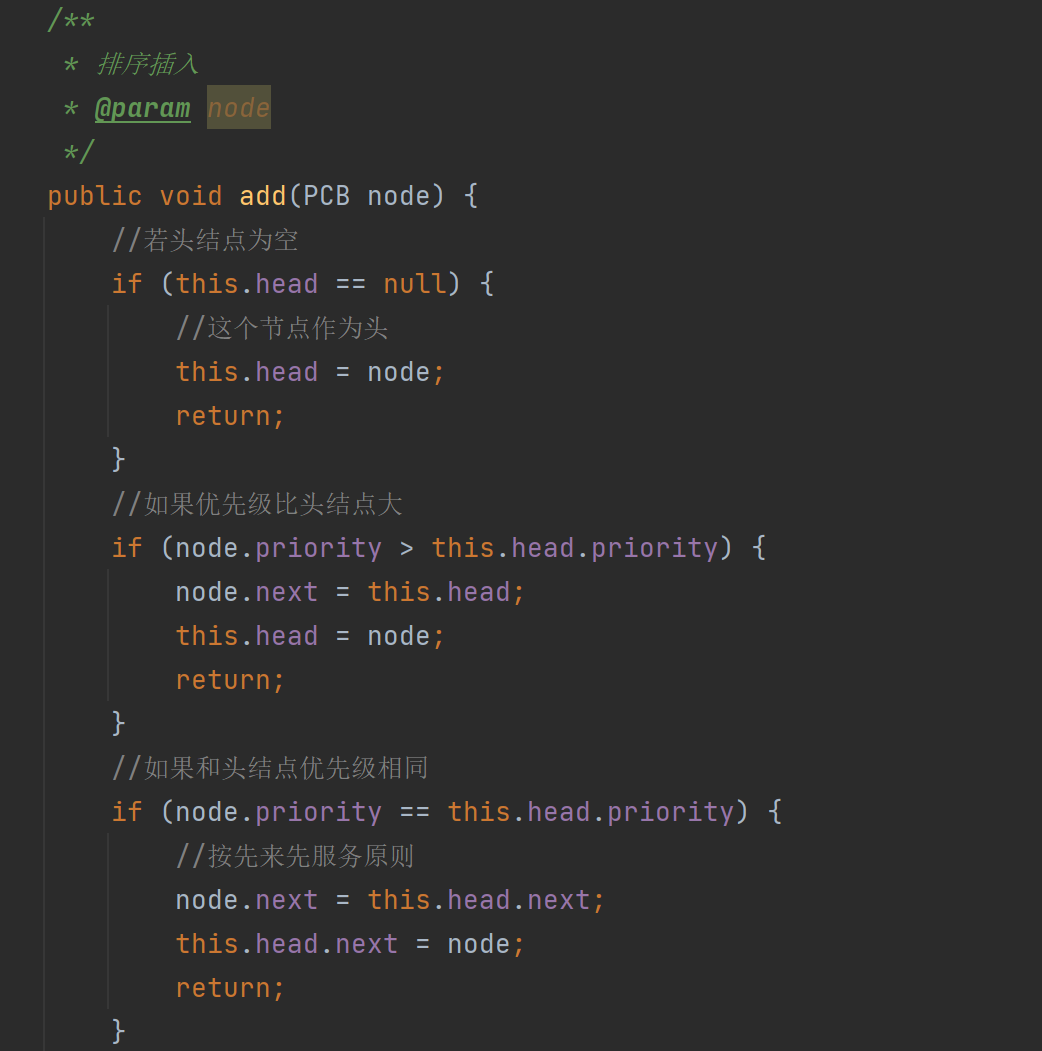


定义PCB：

next指向下一个PCB，通过优先级排序。

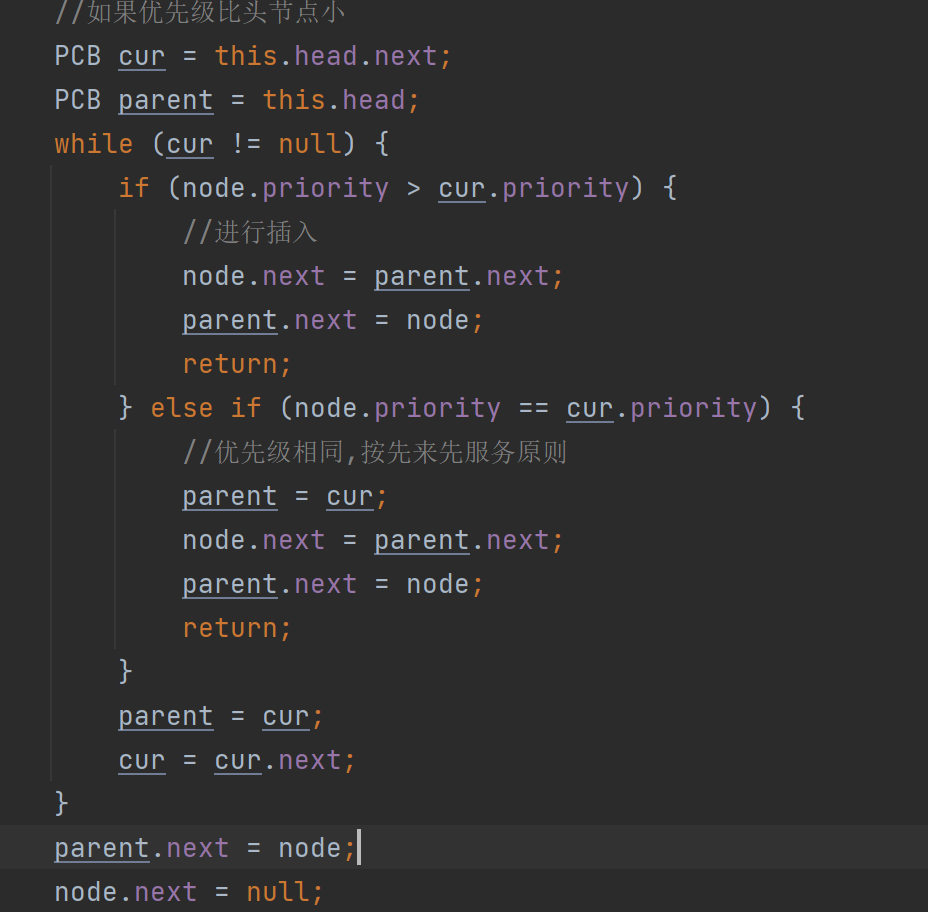
PCBList为自定义的数据结构，他包含一个PCB类型的head属性。Run方法会执行lisy中优先数最高的PCB，add方法会往List里添加PCB，同时对其进行排序。

关键代码：



插入算法：PCBlist中只有一个head，每一个PCB是通过next连接起来的，因此插入的方法和链表插入算法类似，经过排序插入后，head永远是优先级最高的：

1. 当head为空时，该节点作为head
2. 当head和该节点优先级相同时，因为该节点后来，所以该节点和head交换
3. 当head优先级比该节点小时，说明head之后的节点也比该节点小，此时将该节点作为head，原来的head连接在该节点的后一个节点上
4. 当head优先级比该节点高时，我们不能确定该节点的位置，需要进行循环，找到位置后插入。

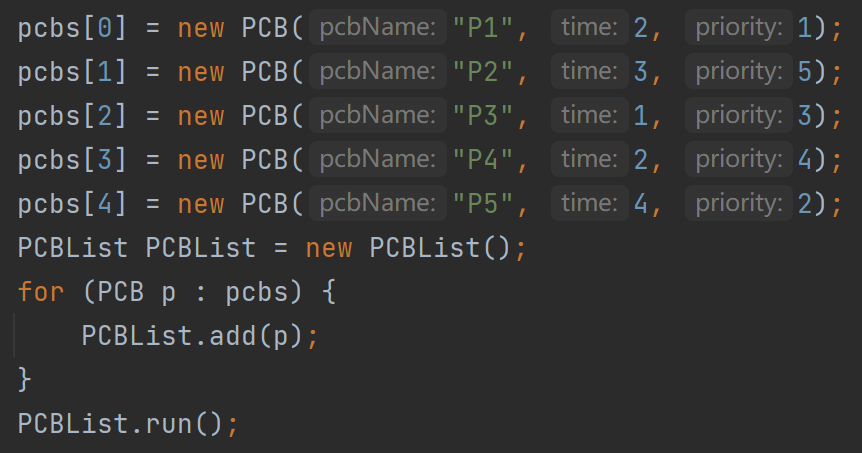


Run方法：

  /\*\*  
      \* 运行  
      \*/  
     public void run() {  
         int i =0;  
         while (this.head != null) {  
             System.out.println("\nCPUTIME:"+i++);  
             //运行优先级最高的第一个进程  
             PCB cur = this.head;  
             //设置状态为W  
             cur.state='W';  
             System.out.println("执行" + cur.pcbName + "进程");  
             System.out.println(cur);  
             cur.priority -= 1;  
             cur.time -= 1;  
 ​  
             //运行一次结束后遍历显示此时进程队列所有信息  
             if (this.head == null) {  
                 System.out.println("所有进程执行完毕");  
                 return;  
            }  
             //打印信息  
             System.out.println("进程队列:");  
             System.out.println("==================================");  
             display();  
             System.out.println("==================================");  
 ​  
             this.head = this.head.next;  
             //如果时间为0，不插入  
             if (cur.time == 0) {  
                 cur.state='E';  
                 System.out.println(cur.pcbName + "退出进程队列");  
                 System.out.println(cur);  
                 System.out.println();  
                 return;  
            }  
             else {  
                 //设置状态为R  
                 cur.state='R';  
                 //将cur再插入进程队列  
                 add(cur);  
            }  
 ​  
 ​  
        }  
    }

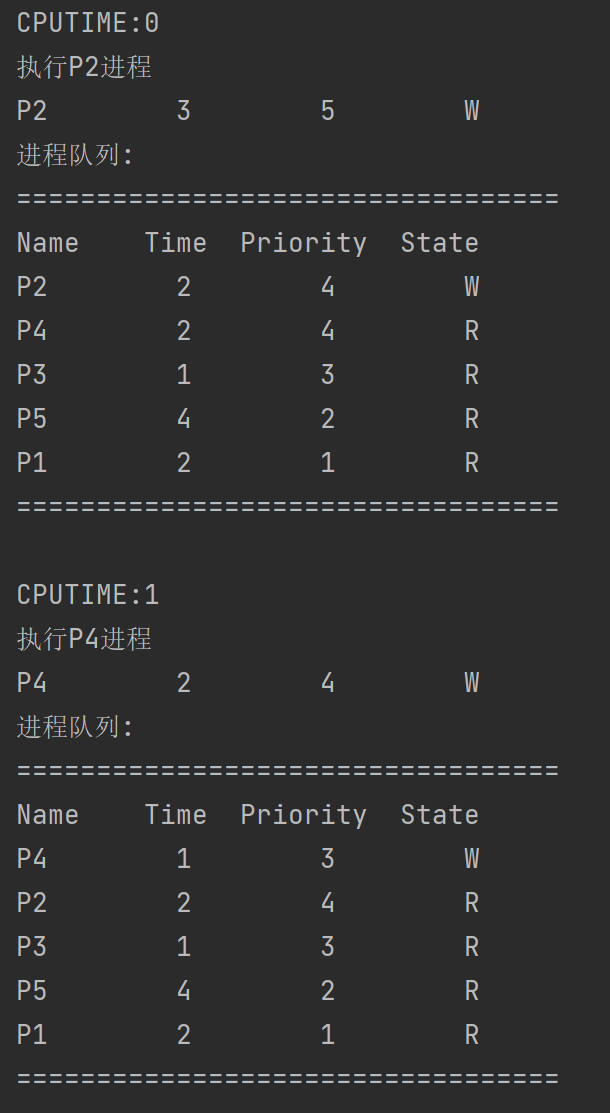
流程正如上文所述。

测试过程：

定义5个PCb，加入到PCBList中，然后执行run方法。

**四、实验结果总结**

CPUTIME0-1：

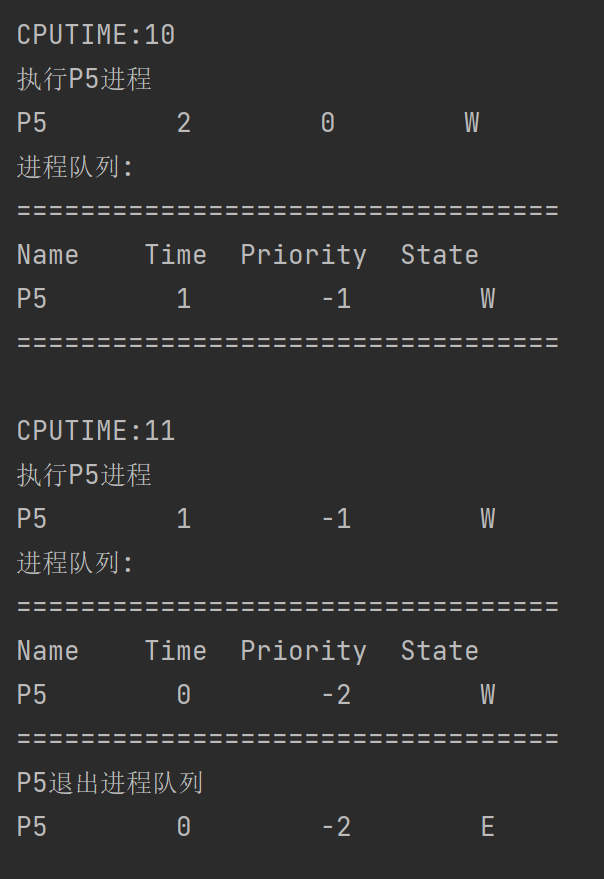


当CPUTIME为0时，执行的是P2进程，优先级最高为5；

当CPUTIME为0时，执行的是P4进程，优先级最高为4；

表格显示的内容为运行结束后当前所有进程。

运行结束：



最后一个进程退出，测试结束。

**五、对实验的意见与建议**

这次实验使我们加深对处理机进程调度机制的理解，对进程的概念有了更深一层次的认识。通过实现优先级调度算法，掌握进程状态转换过程和处理机调度策略的实现。在编程中我们不仅对课本内容有了更好的理解也提高了自己的编程能力。

实验二 处理机调度 (时间片轮转法)

**一、实验目的**

（1）加深对处理机调度的作用和工作原理的理解。

（2）进一步认识并发执行的实质。

**二、问题分析与程序设计**

**问题描述：**

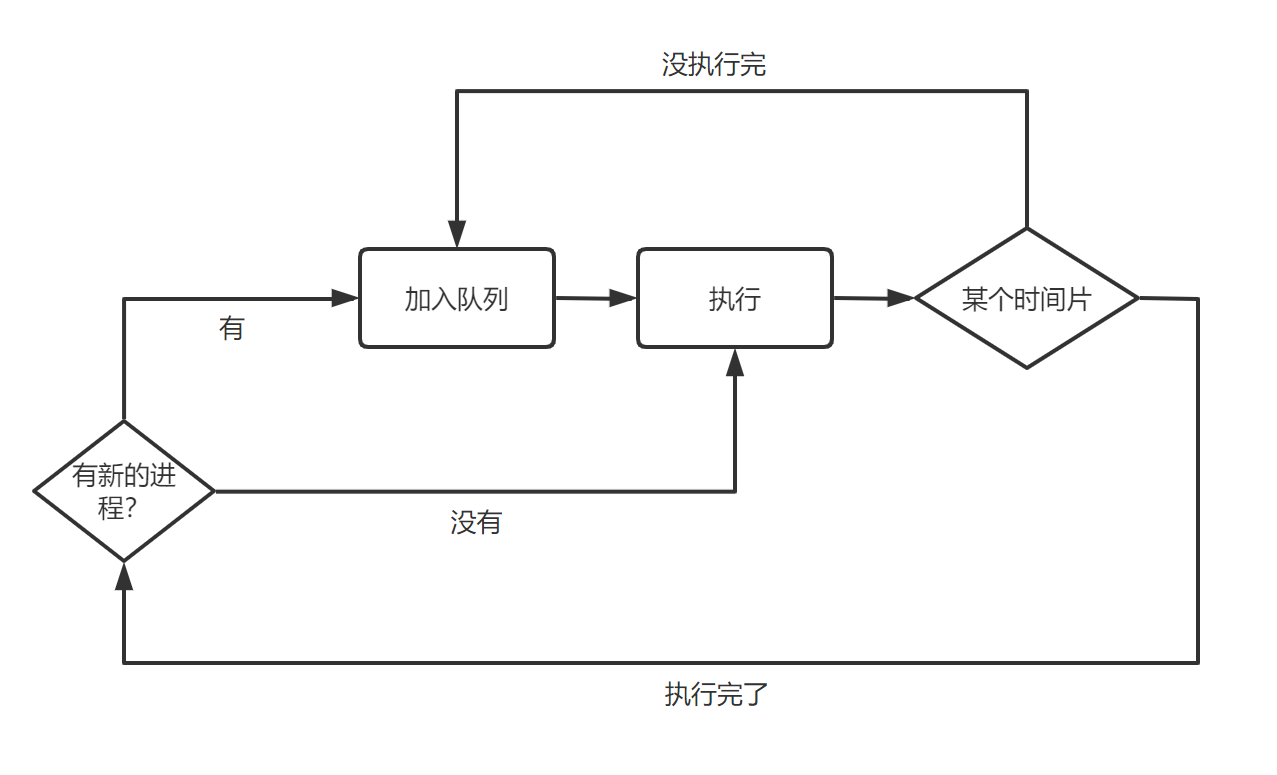
设计一个按时间片轮转法实现处理器调度的程序。

**问题分析与解决方案：**

首先，使用三个队列来存储进程。存储新进程的队列是一个优先级队列。它将按其到达时间对该过程进行排序。

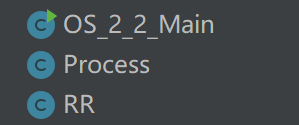
new队列存放的是所有未到达的进程，wait队列存放的是所有到达且准备执行的进程，finish队列存放所有完成的进程。开始执行时，从new队列取出一个进程执行，在执行的过程中，要判断该进程是否能在这个时间片内执行完毕，如果无法执行完毕，则将执行完一半的进程重新插入队尾。进程执行完毕后，继续遍历等待队列，按顺序执行其中的进程。同时，在每一次执行的过程中都要判断当前的时间是否有其他进程到达，并将其加入等待队列。

下图是等待队列的执行流程：

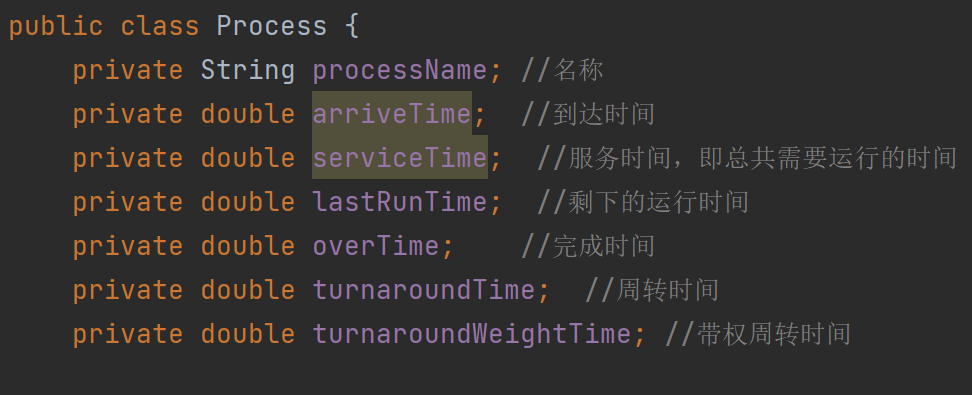


**三、实现过程与测试结果分析**

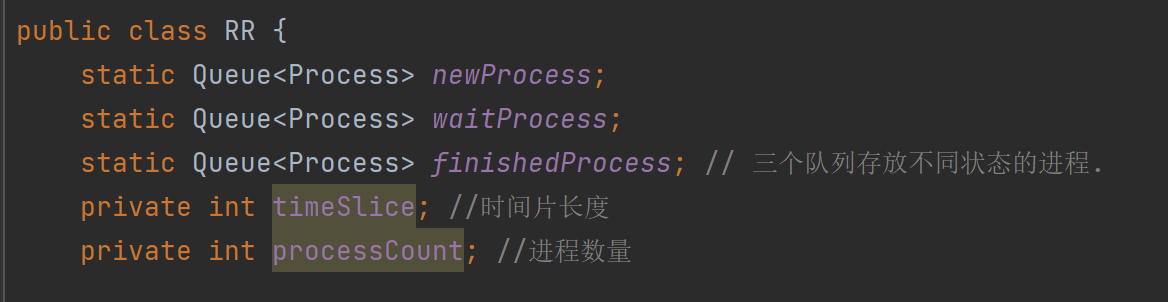
结构：



Process类：



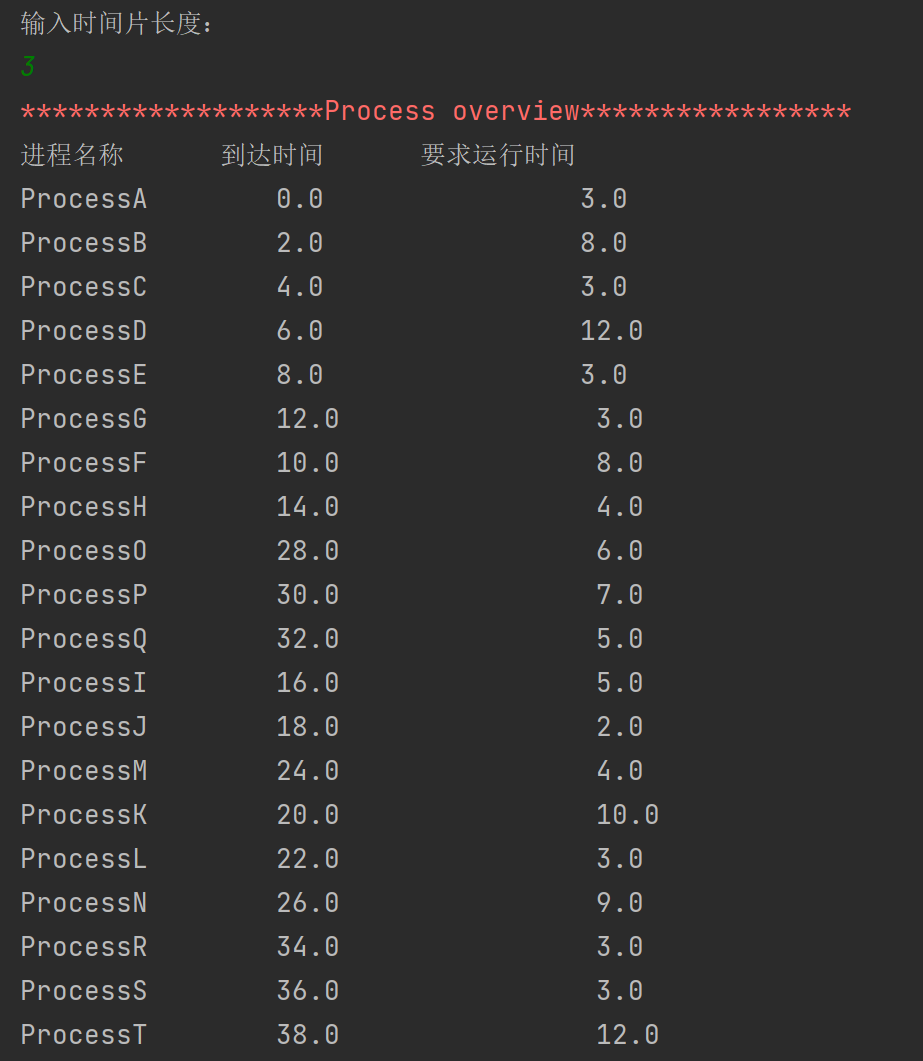
RR内部属性：



关键代码：

  /\*\*  
      \* RR算法  
      \*/  
     public void RRAlgorithm(){  
         //等待队列中新添加一个进程  
         waitProcess.add(newProcess.poll());  
         //获取这个进程  
         Process currProcess = waitProcess.poll();  
         //执行进程 初始时间为0  
         double currTime = executeProcess(currProcess,0);  
         while(!waitProcess.isEmpty() || !newProcess.isEmpty()) {  
             //拿当前时间和new队列里的进程的到达时间进行比较  
             while(!newProcess.isEmpty()) {  
                 //如果有进程到达，则将其加入等待队列  
                 if(newProcess.peek().getArriveTime() <= currTime) {  
                     waitProcess.add(newProcess.poll());  
                }  
                 else {  
                     break;  
                }  
            }  
             //如果当前进程还有剩余时间，继续加入队列  
             if(currProcess.getLastRunTime() > 0){  
                 waitProcess.add(currProcess);  
            }  
             //继续执行  
             if(!waitProcess.isEmpty()) {  
                 currProcess = waitProcess.poll();  
                 currTime = executeProcess(currProcess, currTime);  
            } else {  
                 currTime = newProcess.peek().getArriveTime();  
            }  
        }  
    }  
 ​  
 ​  
     /\*\*  
      \* 执行当前的进程并返回当前时间  
      \* @param currProcess  
      \* @param currTime  
      \* @return  
      \*/  
     public double executeProcess(Process currProcess,double currTime){  
         //如果该进程的剩余时间小于时间片，直接运行  
         if (currProcess.getLastRunTime() - timeSlice <= 0){  
             showExecuteMessage(currTime, currTime += currProcess.getLastRunTime(), currProcess.getProcessName());  
             //设定结束时间  
             currProcess.setOverTime(currTime);  
             //设定剩余时间为0  
             currProcess.setLastRunTime(0);  
             //计算该进程的周转时间和带权周转时间  
             calculateTurnaroundTime(currProcess);  
             calculateWeightTurnaroundTime(currProcess);  
             mTotalWholeTime += currProcess.getTurnaroundTime();  
             mTotalWeightWholeTime += currProcess.getTurnaroundWeightTime();  
             //将该进程加入完成队列中  
             finishedProcess.add(currProcess);  
        }  
         //如果该进程的剩余时间大于时间片，只能运行时间片大小  
         else {  
             showExecuteMessage(currTime, currTime += timeSlice, currProcess.getProcessName());  
             //重新设定剩余时间  
             currProcess.setLastRunTime(currProcess.getLastRunTime() - timeSlice);  
        }  
         return currTime;  
    }

测试过程：



输入时间片长度，进程信息被持久化存储在本地文件中，自动读取。

**四、实验结果总结**

时间片长度：5

时间：0.0--3.0 ProcessA 运行

时间：3.0--8.0 ProcessB 运行

时间：8.0--11.0 ProcessC 运行

时间：11.0--16.0 ProcessD 运行

时间：16.0--19.0 ProcessE 运行

时间：19.0--22.0 ProcessB 运行

时间：22.0--25.0 ProcessG 运行

时间：25.0--30.0 ProcessF 运行

时间：30.0--34.0 ProcessH 运行

时间：34.0--39.0 ProcessD 运行

时间：39.0--44.0 ProcessO 运行

时间：44.0--49.0 ProcessP 运行

时间：49.0--52.0 ProcessF 运行

时间：52.0--57.0 ProcessQ 运行

时间：57.0--62.0 ProcessI 运行

时间：62.0--64.0 ProcessJ 运行

时间：64.0--68.0 ProcessM 运行

时间：68.0--73.0 ProcessK 运行

时间：73.0--76.0 ProcessL 运行

时间：76.0--81.0 ProcessN 运行

时间：81.0--84.0 ProcessR 运行

时间：84.0--87.0 ProcessS 运行

时间：87.0--92.0 ProcessT 运行

时间：92.0--94.0 ProcessD 运行

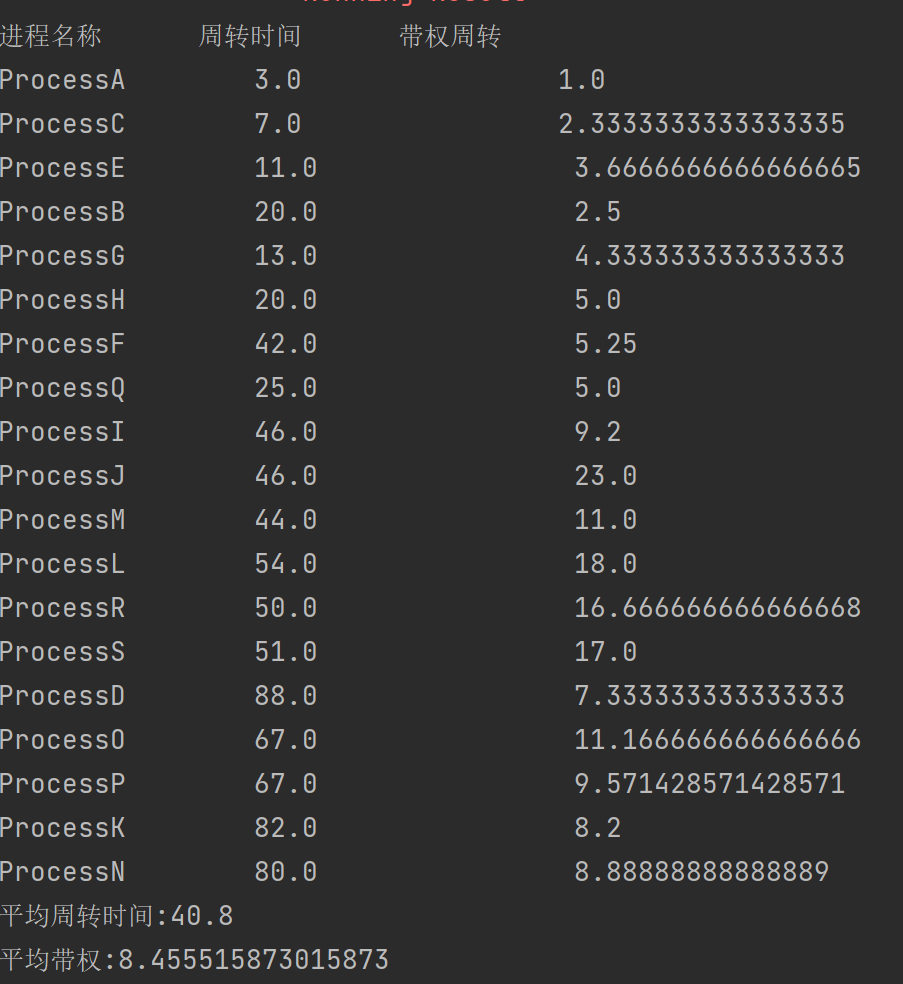
时间：94.0--95.0 ProcessO 运行

时间：95.0--97.0 ProcessP 运行

时间：97.0--102.0 ProcessK 运行

时间：102.0--106.0 ProcessN 运行

时间：106.0--111.0 ProcessT 运行



**五、创新的部分**

参考资料，加入了计算周转时间和带权周转时间的功能。

**六、对实验的意见与建议**

这次实验使我们加深对处理机进程调度机制的理解，对进程的概念有了更深一层次的认识。通过实现优先级调度算法，掌握进程状态转换过程和处理机调度策略的实现。在编程中我们不仅对课本内容有了更好的理解也提高了自己的编程能力。

**七、思考题**

（1）处理机调度的目的？

合理快速的处理计算机软件硬件资源。

（2）你实现优先权调度算法的思想？

运用链表的思想，在每次对进程执行相关操作前对优先级队列进行排序，优先级高的排在队列前端。

（3）你采用时间片轮转法实现处理机调度的思想？

将时间以一个cpu时间为单位，每次循环代表一次cpu时间，每次从等待队列中取出一个进程，在一个时间片内执行，如果能执行完毕，就执行队列的下一个进程，如果不能执行完毕，就把该进程放在队尾，再继续执行下一个进程。

（4）比较效率如何？

可以考虑使用平均周转时间的方法进行比较。

RR 算法的性能很大程度上依赖于时间片的大小。在极端情况下，如果时间片非常大，那么 RR 算法与 FCFS 算法一样。如果时间片很小，那么 RR 算法称为处理器共享。

实验三 存储管理 (第一题)

**一、实验目的**

（1）加深对存储管理的作用和工作原理的理解。

（2）进一步认识主存空间的分配和回收方法。

（3）进一步认识虚拟存储器的工作原理。

（4）进一步认识文件系统的内部功能及内部实现。

**二、问题分析与程序设计**

**问题描述：**

模拟分页式存储管理中硬件的地址转换和产生缺页中断。

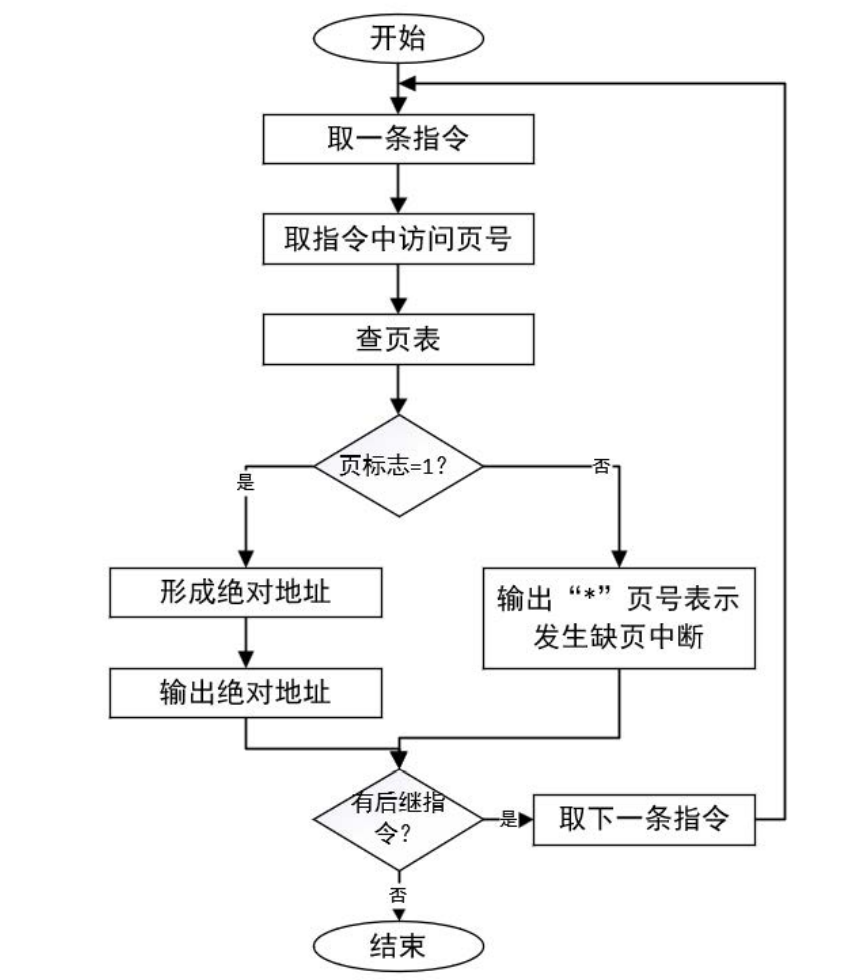
**问题分析与解决方案：**

模拟分页式储存管理硬件地址转换与缺页产生中断程序要分两种情况：地址转换与缺页产生中断。

作业在执行时，指令中的逻辑地址表示了参加运算的操作数存放的页号和单元号，通过按页号查询页表来检查该页的标志位是否为“1”。

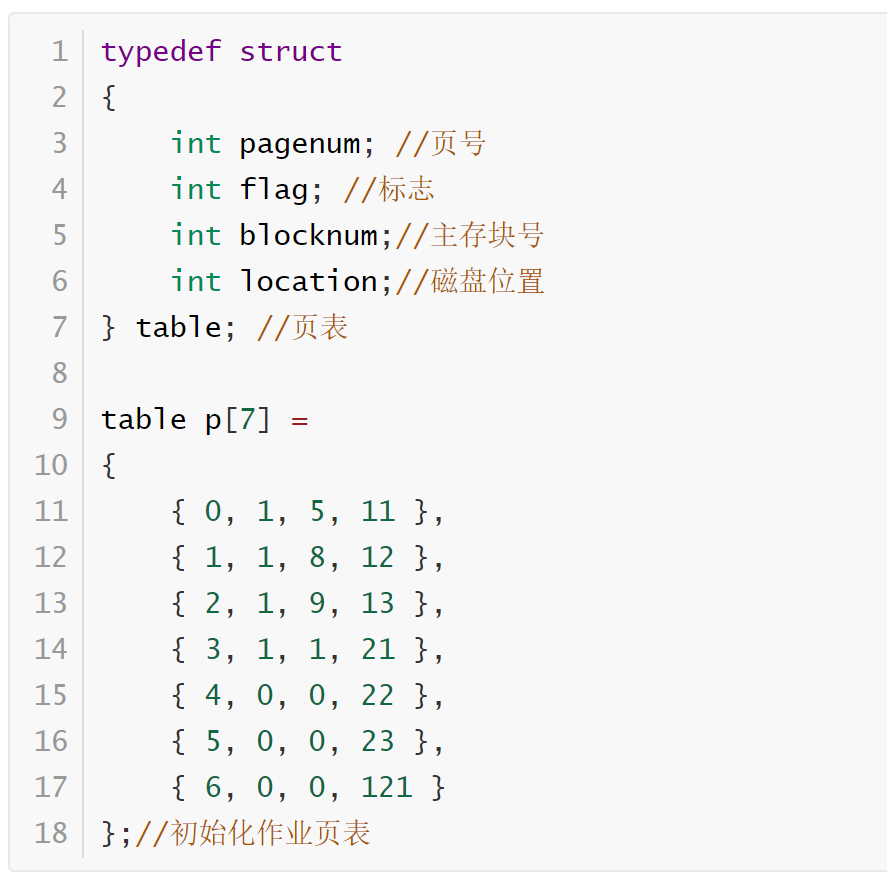
若该页对应标志为“1”，则表示该页已在主存，这时根据关系式：“绝对地址 = 块号 × 块长 + 单元号”计算出欲访问的主存单元地址。如果块长为2的幂次，则可把块号作为高地址部分，把单元号作为低地址部分，两者拼接形成绝对地址；

若访问页对应标志位“0”，则表示该页不在主存，这时硬件发送缺页中断的信号，操作系统按该页在磁盘的位置，将页信息从磁盘中读出装入主存，重新执行命令。



**三、实现过程与测试结果分析**

定义页表结构：

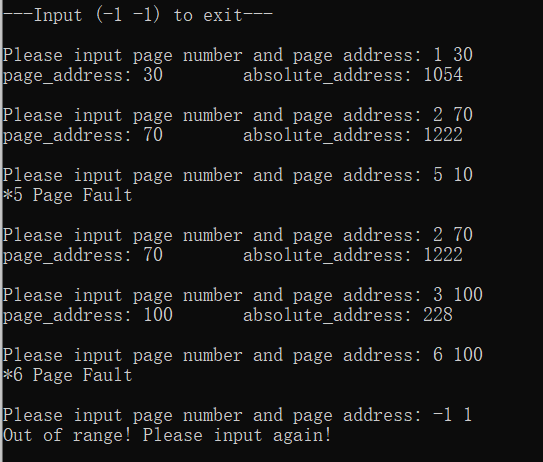


关键代码：

Main函数中通过简单的循环判断即可实现。

     int page;  
     int offset;  
     int memaddress;  
     printf("\n---Input (-1 -1) to exit---\n\n");  
 ​  
     while (1)  
    {  
         setbuf(stdin, NULL);  
         printf("Please input page number and page address: ");  
         scanf("%d %d", &page, &offset);  
         if (page == -1 && offset == -1)  
        {  
             printf("\n");  
             exit(1);  
        }  
         else if (page < 0 || page > 6 || offset < 0)  
        {  
             printf("Out of range! Please input again!\n\n");  
        }  
         else  
        {  
 ​  
             if (p[page].flag != 0)   //该页已经装入主存  
            {  
                 memaddress = p[page].blocknum \* block\_size + offset;  
                 printf("page\_address: %d\tabsolute\_address: %d\n\n", offset, memaddress);  
            }  
             else  
            {  
                 printf("\*%d Page Fault\n\n", page);  
            }  
 ​  
        }  
    }  
 ​

**四、实验结果总结**



**五、对实验的意见与建议**

无。

实验三 存储管理(第二题)

**一、实验目的**

同上。

**二、问题分析与程序设计**

**问题描述：**

用先进先出（FIFO）页面调度算法处理缺页中断。

**问题分析与解决方案：**

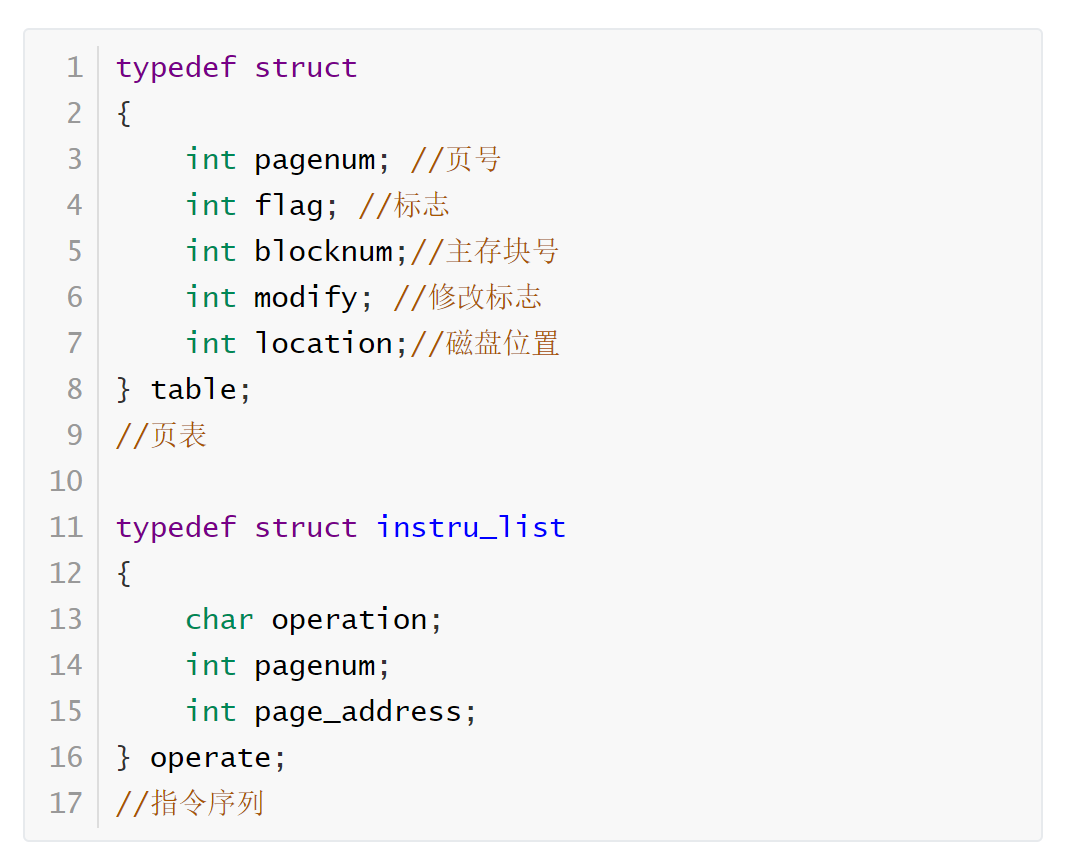
在分页式虚拟存储系统中，当硬件发出“缺页中断”后，引出操作系统来处理这个中断事件。如果主存中已经没有空闲块，则可用 FIFO 页面调度算法把该作业中最先进入主存的一页调出，存放到磁盘上。然后再把当前要访问的页装入该块。调出和装入后都要修改页表中对应页的标志。

过程：

取一条指令，取出指令中访问的页号（为L），再查页表，判断页标志是否为1，若不为1，则产生页中断，模拟硬件地址转换模拟FIFO页面调度；若为1，则形成绝对绝对地址，判断是否为“存”指令，若是则修改标志为1，若不是则输出绝对地址。输出后，判断是否有后继指令，若有重复上述过程，若没有，则结束程序。

**三、实现过程与测试结果分析**

结构：

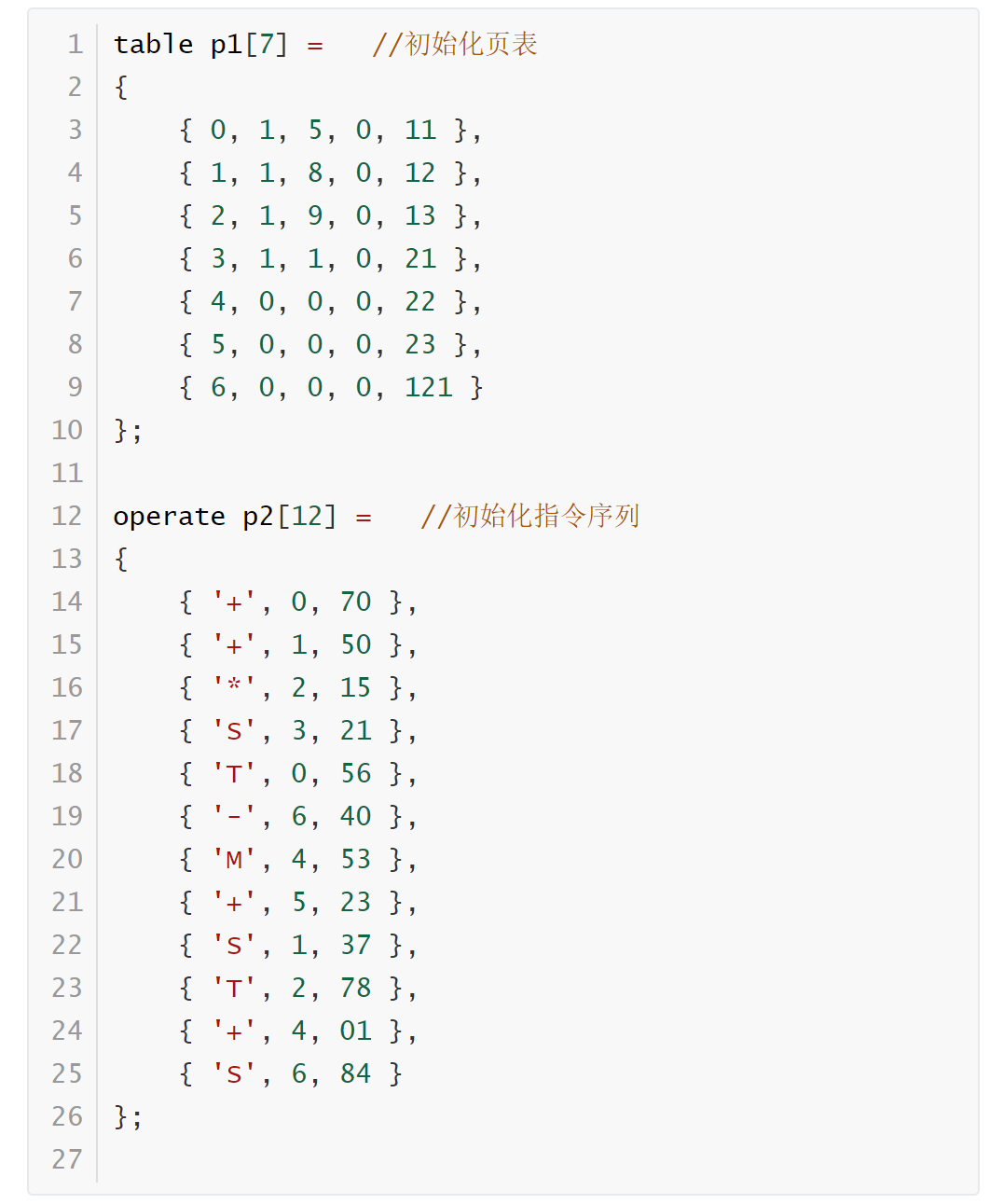


定义页表和指令结构。

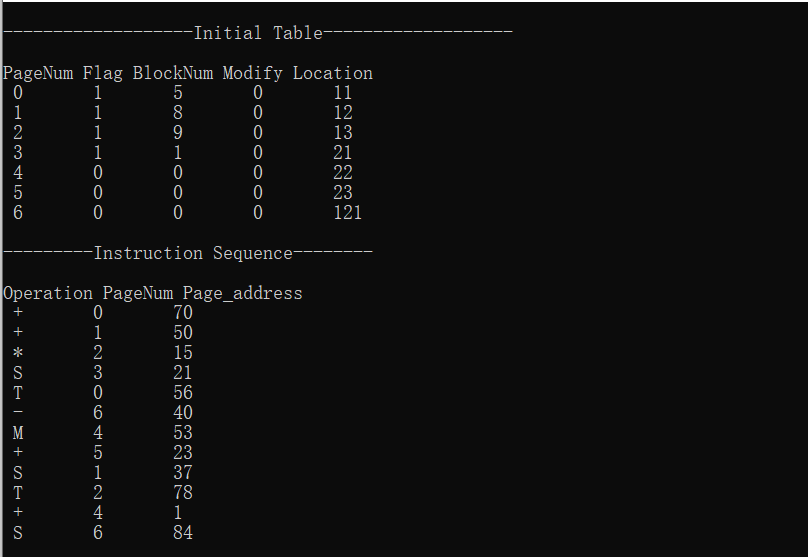
关键代码：使用main函数直接测试：

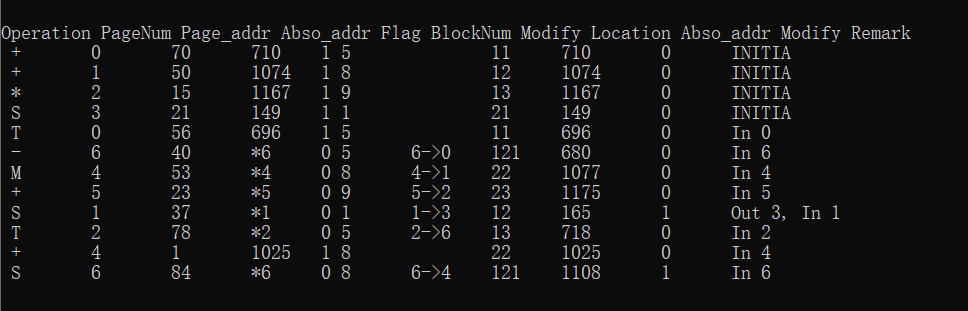
 int i, j, page, rep\_page, modify, flag, memaddress;  
     int k = 0;  
     int m = 4;  
     int p[4] = { 0, 1, 2, 3 };  
 ​  
     printf("\n-------------------Initial Table-------------------\n\n");  
     printf("PageNum Flag BlockNum Modify Location\n");  
     for (i = 0; i < 7; i++)  
    {  
         printf(" %d\t %d\t %d\t %d\t %d\n", p1[i].pagenum, p1[i].flag, p1[i].blocknum, p1[i].modify, p1[i].location);  
    }  
     printf("\n---------Instruction Sequence--------\n\n");  
 ​  
     printf("Operation PageNum Page\_address\n");  
     for (i = 0; i < 12; i++)  
    {  
         printf(" %c\t %d\t %d\n", p2[i].operation, p2[i].pagenum, p2[i].page\_address);  
    }  
 ​  
     printf("\n------------------------------------------------------Outcome------------------------------------------------------\n\n");  
     printf("Operation PageNum Page\_addr Abso\_addr Flag BlockNum Modify Location Abso\_addr Modify Remark\n");  
 ​  
     for (i = 0; i < operation\_num; i++)  
    {  
         page = p2[i].pagenum; //待处理的页号  
         flag = p1[page].flag; //待处理的页标志，判断该页是否已经装入主存  
         if (flag == 0)   //该页未装入主存  
        {  
             rep\_page = p[k]; //将被替换的页号  
             modify = p1[rep\_page].modify; //将被替换的页的修改标志  
             p1[page].blocknum = p1[rep\_page].blocknum; //装入主存  
             p1[page].flag = 1; //该页装入主存，标志位改为１  
             p1[rep\_page].flag = 0;//被替代的页标志位改为０  
             p1[rep\_page].modify = 0; //被替代的页未被修改  
             p1[rep\_page].blocknum = -1; //移出主存，无对应块号  
             p[k] = page;  
             k = (k + 1) % m;  
        }//待处理页已经装入主存  
         memaddress = p1[page].blocknum \* block\_size + p2[i].page\_address;  
         if (p2[i].operation == 'S')  
             p1[page].modify = 1;  
         printf(" %c\t %d\t %d\t ", p2[i].operation, p2[i].pagenum, p2[i].page\_address);  
         if (flag == 1) //待处理页已装入主存  
             printf("%d \t%d %d \t\t %d \t%d\t ", memaddress, flag, p1[page].blocknum, p1[page].location, memaddress);  
         else  
             printf("\*%d \t%d %d\t %d->%d\t %d \t%d\t ", p2[i].pagenum, flag, p1[page].blocknum, p2[i].pagenum, rep\_page, p1[page].location, memaddress);  
 ​  
         if (i < 4)  
             printf(" 0\t INITIA");  
         else  
        {  
             if (modify == 1)  
                 printf(" %d\t Out %d, In %d", p1[page].modify, rep\_page, page);  
             else  
                 printf(" %d\t In %d", p1[page].modify, page);  
        }  
         printf("\n");  
    }  
     printf("\n");  
 ​

测试数据：



**四、实验结果总结**





**五、对实验的意见与建议**

无。

实验三 存储管理(第四题)

**一、实验目的**

用高级语言编写和调试一个简单的文件系统，模拟文件管理的工作过程。从而对各种文件操作命令的实质内容和执行过程有比较深入的了解。

**二、问题分析与程序设计**

**问题描述：**

设计一个按时间片轮转法实现处理器调度的程序。

设计一个10个用户的文件系统，每次用户可保存 10 个文件，一次运行用户可以最多打开 5 个文件程序。采用二级文件目录（即设置主目录[MFD]）和用户文件目录（UFD，即文件目录，包括文件名，保护码，文件长度）。另外， 为打开文件设置了运行文件目录（AFD，文件执行读命令或写命令之前，把相关文件目录信息调入AFD）。 在执行读写命令时，只需改读写指针。

（1）文件目录的检索使用线性搜索

（2）文件保护简单使用了三位保护码：允许读写执行、对应位为1；不允许读写、执行，对应位为0，

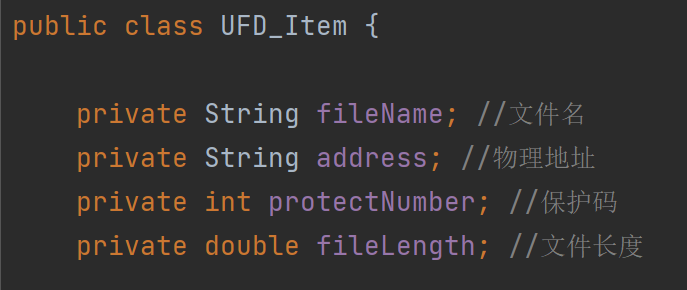
（3）程序中使用的主要设计结构如下：主文件目录和用户文件目录(MFD、UFD)、打开文件目录(AFD)(即运行文件目录)

**三、实现过程与测试结果分析**

结构：

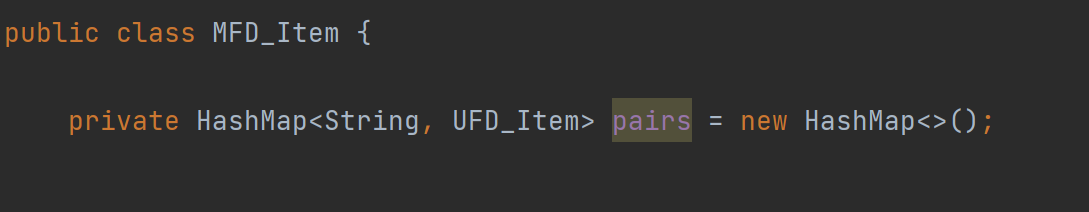


设计UFD和MFD结构：



这里UFD不是整个用户文件目录，而是其中的一个最小单元；类似的MFD也是主文件目录的一个最小单元，而主文件目录的对应关系是用户名和UFD目录（这里是多个UFD\_Item）

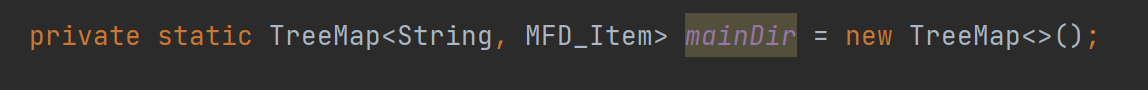
MFD\_Item：



包含一个Map，对应用户名和该用户名的文件。

MFD整体结构：

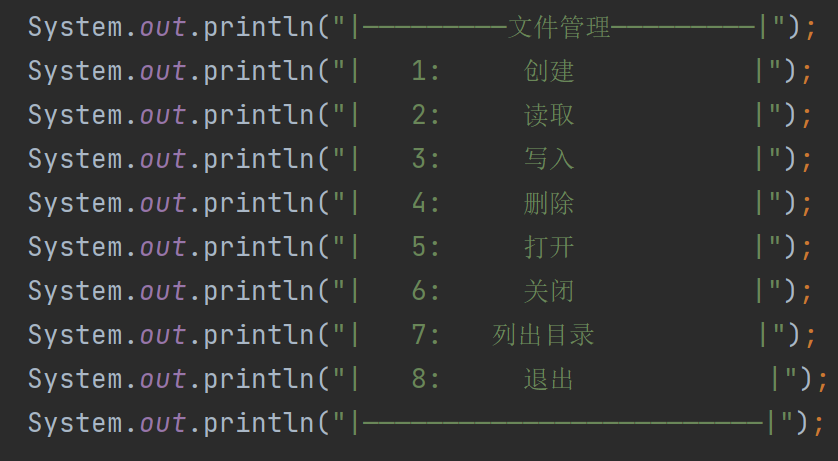
采用TreeMap：



存放用户名和对应的MFD\_Item

关键代码：

通过登录后输出menu，通过switch进行选择。



传入MFD和用户名，对文件的操作就是简单的增删改查，操作时要注意一些细节，比如说同名文件，文件是否在打开状态，文件的保护码。

打开文件和关闭文件的代码：

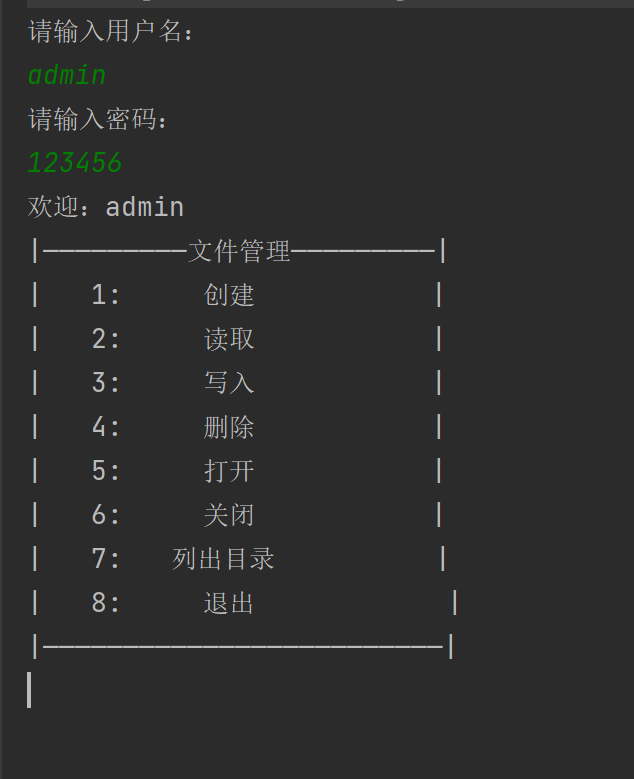
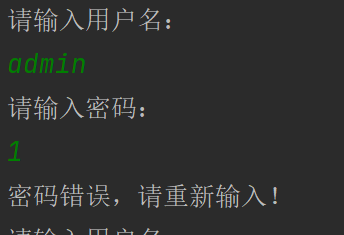
 case 5:  
                 Scanner input7 = new Scanner(System.in);  
                 fileName = input7.next();  
 ​  
                 if (!mainDir.get(address).find(fileName))  
                     System.out.println("文件不存在");  
                 else if (mainDir.get(address).get(fileName).getProtectNumber() == 0)  
                     System.out.println("文件不可读");  
                 else if (openNumber >= maxOpen)  
                     System.out.println("打开文件数已达到本次访问上限");  
                 else {  
                     openTable[openNumber++].addItem(fileName, mainDir.get(address).get(fileName).getProtectNumber(), 0);  
                     System.out.println("文件已打开，AFD表写入成功！");  
                }  
                 try {  
                     TimeUnit.SECONDS.sleep(3);  
                } catch (InterruptedException ignored){}  
 ​  
                 fileOperate(address, mainDir);  
                 break;  
 ​  
             case 6:  
                 System.out.println("输入文件名");  
 ​  
                 Scanner input8 = new Scanner(System.in);  
                 fileName = input8.next();  
 ​  
                 boolean found = false;  
                 for (int i = 0; i < openTable.length; i++){  
                     if (openTable[i].getFileOpen().equals(fileName)) {  
                         for (int j = i; j < openTable.length - 1; j++)  
                             openTable[j] = openTable[j + 1];  
                         openNumber--;  
                         found = true;  
                    }  
                }  
                 if (!found)  
                     System.out.println("文件未打开");  
                 try {  
                     TimeUnit.SECONDS.sleep(3);  
                } catch (InterruptedException ignored){}  
 ​  
                 fileOperate(address, mainDir);  
                 break;

**四、实验结果总结**

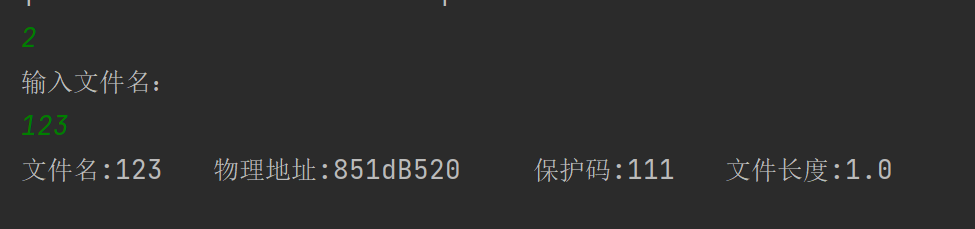
实现功能：login 用户登录；displayUserFile列用户文件目录；create 创建文件；delete 删除文件；open 打开文件；close 关闭文件；read 读文件；write 写文件指令。

运行截图：

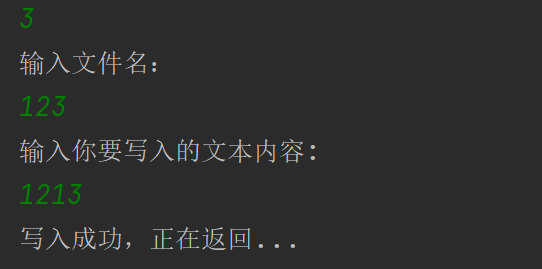
登录：

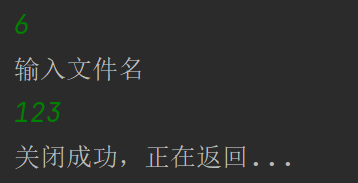
读取：



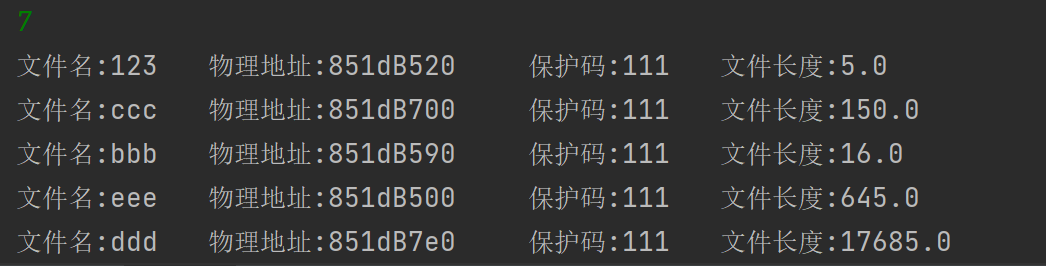
写入：



关闭文件：



文件目录：



**五、创新的部分**

加入了退出系统的功能。

**六、对实验的意见与建议**

通过本实验，我们对各种文件操作命令的实质内容和执行过程有了比较深入的了解。能够用高级语言编写和调试一个简单的文件系统，模拟文件管理的工作过程。

**七、思考题**

（1）先进先出页面调度算法的思想？

先进先出页面调度算法就是当需要置换一页时，根据页面进入内存的时间先后选择淘汰页面，先进入内存的页面先淘汰，后进入内存的后淘汰。

（2）最近最少用（LRU）页面调度算法思想？

最近最少用（LRU）页面调度算法就是当需要置换一页时，在内存块中找到一个最久没有使用的页面予以淘汰并进行置换的算法。

1. 比较两种调度算法的效率（哪种调度算法使产生缺页中断的次数少）？

LRU算法的性能较好，但需要寄存器和硬件的支持，LRU是堆栈类的算法，理论上不会出现Belady异常。

1. 分析在什么情况下采用哪种调度算法更有利？

如果页访问高度随机的话，LRU不一定有利，LUR是根据最近最少使用的决定哪个页被替换，但过去很多时候不能代表未来。

（5）文件管理和目录管理的思想？

文件管理：根据不同单位计算机技术在现行文件和文档管理中应用情况，可以分为三种模式，一是现代文件和档案管理分别应用计算机技术模式；二是现行文件和档案管理综合应用技术模式；三是电子文件模式。要结合本单位的特点，采用相应的管理模式。

目录管理：管理和操作索引，不直接对文件进行操作。

评分标准：

|  |  |
| --- | --- |
| 考核标准 | 得分 |
| （1）正确理解和掌握实验所涉及的概念和原理（20%）； |  |
| （2）按实验要求合理设计数据结构和程序结构（20%）； |  |
| （3）认真记录实验数据，原理及实验结果分析准确（20%）； |  |
| （4）实验过程中，具有严谨的学习态度和认真、踏实、一丝不苟的科学作风（10%）； |  |
| （5）所做实验具有一定的创新性（10%）； |  |
| （6）实验报告规范（20%）。 |  |