**操作系统实验**

学 院 （系）： 计算机科学与技术学院

学 生 姓 名： 陆鑫

学 号： 20203015022

联 系 方 式： 18118115402

班 级： 电计2001

大连理工大学

Dalian University of Technology

目 录

[1 进程管理作业 - 1 -](#_Toc32588)

[1.1 实验目的 - 1 -](#_Toc29762)

[1.2 实验说明 - 1 -](#_Toc10911)

[1.3 实验过程 - 2 -](#_Toc30395)

[1.4 实验结果 - 9 -](#_Toc30719)

[1.5 实验中遇到的问题和解决办法 - 10 -](#_Toc27101)

[2 处理器调度作业 - 11 -](#_Toc32755)

[2.1 实验目的 - 11 -](#_Toc21655)

[2.2 实验说明 - 11 -](#_Toc17445)

[2.3 实验过程 - 11 -](#_Toc22137)

[2.4 实验结果 - 15 -](#_Toc16439)

[2.5 实验中遇到的问题和解决办法 - 18 -](#_Toc5571)

[3 进程同步和死锁作业 - 18 -](#_Toc15866)

[3.1 实验目的 - 18 -](#_Toc26416)

[3.2 实验说明 - 18 -](#_Toc28845)

[3.3 实验过程 - 19 -](#_Toc17607)

[3.3.1 编程模拟生产者/消费者问题 - 19 -](#_Toc14067)

[3.3.2 编程实现银行家算法 - 20 -](#_Toc5236)

[3.4 实验结果 - 23 -](#_Toc8596)

[3.4.1 编程模拟生产者/消费者问题 - 23 -](#_Toc1086)

[3.4.2 编程实现银行家算法 - 24 -](#_Toc3410)

[3.5 实验中遇到的问题和解决办法 - 25 -](#_Toc12625)

[4 内存管理作业 - 25 -](#_Toc20565)

[4.1 实验目的 - 25 -](#_Toc14391)

[4.2 实验说明 - 25 -](#_Toc16375)

[4.3 实验过程 - 25 -](#_Toc32754)

[4.4 实验结果 - 28 -](#_Toc14845)

[4.5 实验中遇到的问题和解决办法 - 29 -](#_Toc24282)

[5 磁盘调度作业 - 29 -](#_Toc29817)

[5.1 实验目的 - 29 -](#_Toc28519)

[5.2 实验说明 - 29 -](#_Toc12783)

[5.3 实验过程 - 30 -](#_Toc23631)

[5.4 实验结果 - 31 -](#_Toc9077)

[5.5 实验中遇到的问题和解决办法 - 32 -](#_Toc26277)

[6 文件管理作业 - 32 -](#_Toc20699)

[6.1 实验目的 - 32 -](#_Toc15835)

[6.2 实验说明 - 32 -](#_Toc24358)

[6.3 实验过程 - 33 -](#_Toc16590)

[6.4 实验结果 - 34 -](#_Toc4074)

[6.5 实验中遇到的问题和解决办法 - 37 -](#_Toc14412)

# 

# 1 进程管理作业

## 实验目的

加深对于进程并发执行概念的理解。实践并发进程的创建和控制方法。观察和体验进程的动态特性。进一步理解进程生命期期间创建、变换、撤销状态变换的过程。掌握进程控制的方法，了解父子进程间的控制和协作关系。练习 Linux 系统中进程创建与控制有关的系统调用的编程和调试技术。

## 实验说明

1. 与进程创建、执行有关的系统调用说明

进程可以通过系统调用fork()创建子进程并和其子进程并发执行。子进程初始的执行映像是父进程的一个复本。子进程可以通过exec()系统调用族装入一个新的执行程序。父进程可以使用wait()或waitpid()系统调用等待子进程的结束并负责收集和清理子进程的退出状态。

fork()系统调用语法：

#include <unistd.h>

pid\_t fork(void);

fork成功创建子进程后将返回子进程的进程号，不成功会返回-1。

exec函数族分别是：execl,execlp,execle,execv,execvp,execvpe

调用语法：

#include <unistd.h>

int execl(const char \*path, const char \*arg, ...);

int execlp(const char \*file, const char \*arg, ...);

int execle(const char \*path, const char \*arg,..., char \* const envp[]);

int execv(const char \*path, char \*const argv[]);

int execvp(const char \*file, char \*const argv[]);

int execvpe(const char \*file, char \*const argv[],char \*const envp[]);

参数说明：

path 要装入的新的执行文件的绝对路径/相对路径。

argv[] 要传递给新执行程序的完整的命令参数列表（可以为空）。

envp[] 要传递给新执行程序的完整的环境变量参数列表（可以为空）。

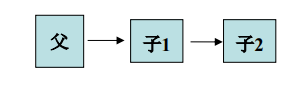
file 如果参数file中包含/，则就将其视为路径名，否则就按PATH环境变量，在它所指定的各目录中搜寻可执行文件。

exec执行成功后将用一个新的程序代替原进程，但进程号不变，它绝对不会再返回到调用进程了。如果exec调用失败，他会返回-1。

## 实验过程

1. 要求1：

每个进程都执行自己的独立的程序，打印自己的pid，每个父进程打印其子进程的pid。



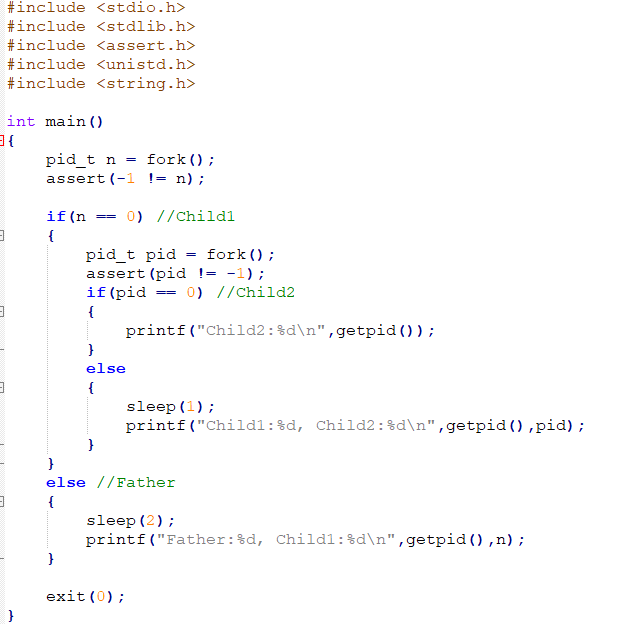
根据要求，首先需要创建子1进程，因此调用fork()创建子1进程。通过断言的方式检验是否创建成功。

pid\_t n = fork();

assert(-1 != n);

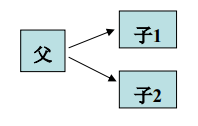
创建后父子进程是两个独立的进程，各自拥有一份代码。fork()方法调用之后，父，子进程都从fork()调用之后的代码开始运行。fork函数被调用一次，但是返回两次，子进程返回的是0.而父进程返回值则是新子进程的进程ID。故创建进程后进行判断，若n等于0，则为子1进程，否则为父进程。为保证父子进程的输出顺序，在父进程代码块前加入sleep(2)，从而保证子进程先于父进程执行。由子1进程创建子2进程同理。

程序源码test1.c



1. 要求2：

每个进程都执行自己独立的程序，打印自己的pid，父进程打印其子进程的pid。



要求1与要求2唯一的不同是，前者要求父进程创建子1进程，再由子1进程创建子2进程。后者要求父进程既创建子1进程，又创建子2进程。故只需将要求1中创建子进程代码从子1进程替换到父进程即可。

程序源码test2.c



1. 要求3：

编写一个命令处理程序，类似windows的命令行界面；主程序启动后，显示提示符，在提示符后输入命令，在命令结束后重新显示提示符，再次等待新命令的输入，一直等到输入待定字符串（如quit）才退出主程序，能处理max(m,m)、min(m,n)、average(m,n,l)这几个命令。（要求使用exec函数族实现）

首先在主进程中设置死循环while(1)用于重复显示提示符。接着将用户输入的命令存储为字符数组（字符串）：

printf("Please Input A Command:");

gets(cmd);

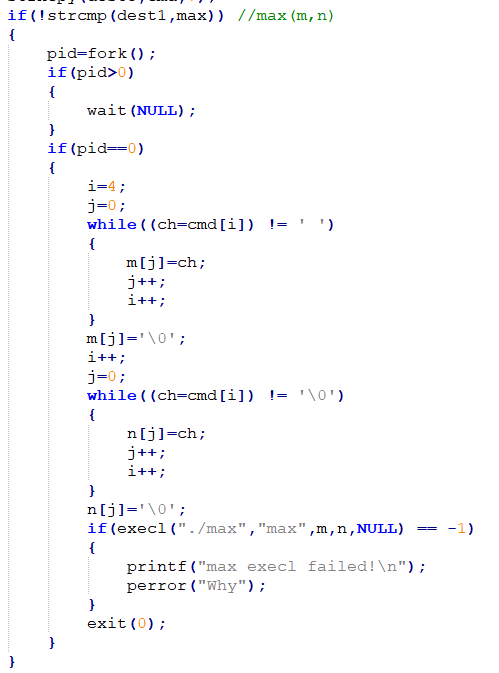
使用strncpy函数提取命令的前3、4、7个字符，用于判断指令类别。

strncpy(dest1,cmd,3);

strncpy(dest2,cmd,4);

strncpy(dest3,cmd,7);

若前三个字符为max，则判断为max命令，创建一个新的子进程，并使用execl函数执行max.c中的代码段，执行结束后，自我终止。而主进程调用wait(NULL)等待子进程调用结束后恢复执行。



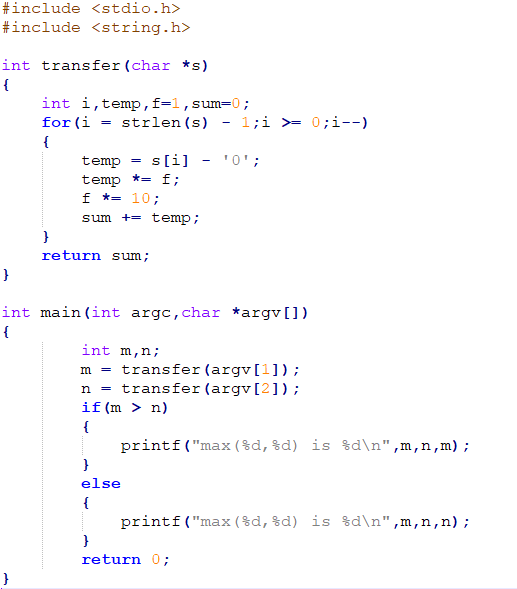
若执行语句execl时出错，则程序将会打印出错信息”max execl failed!”并调用perror显示出错原因，方便后续查找错误。

其他指令如min，average在主函数中的调用过程与max类似，而quit指令则会跳出死循环，结束整个主进程。若遇到非上述4条指令，则打印”Command Not Found!”

下面讨论max.c中的程序：

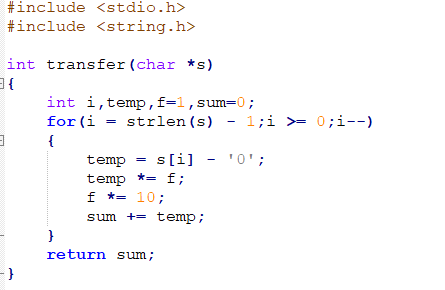
首先定义一个transfer函数，用于将接收到的字符串转换为整型数字。从字符数组的倒数第二位开始（倒数第一位是’\0’），依次减去’0’，得到整型数字，然后乘以倍数，倍数每次循环×10，然后累加便得到最后的整型数字。最后主函数中读取传进来的字符数组地址，依次转换为数字，最后比较大小后输出。

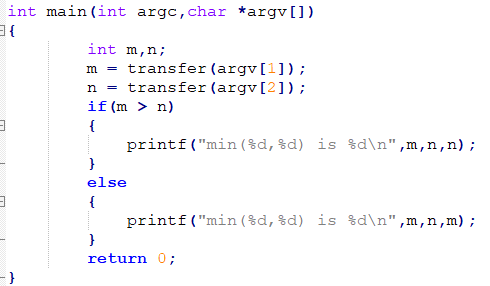
程序源码max.c



min.c与max.c程序非常相似，不过是把最后的输出反置即可。

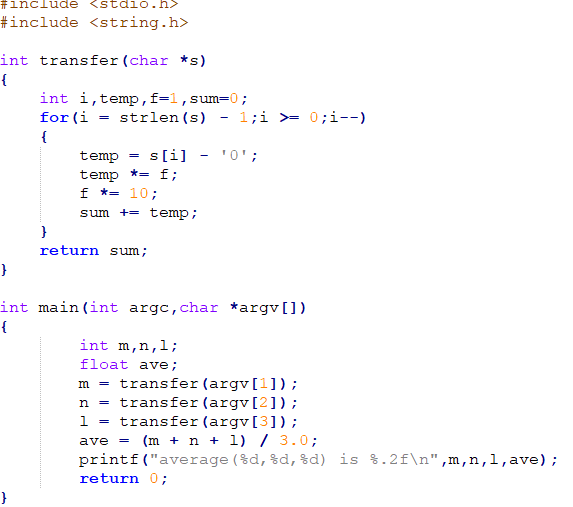
程序源码min.c



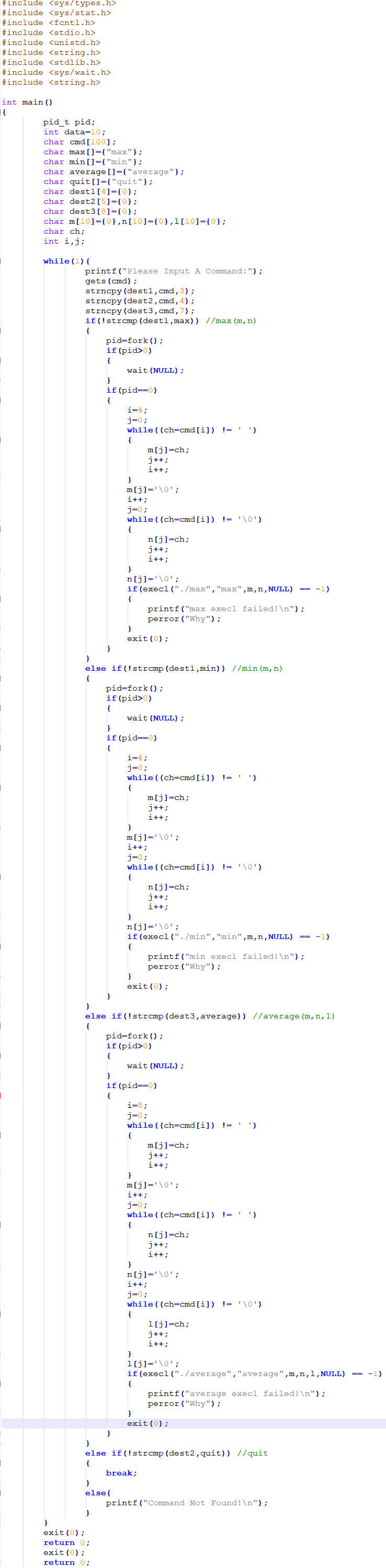


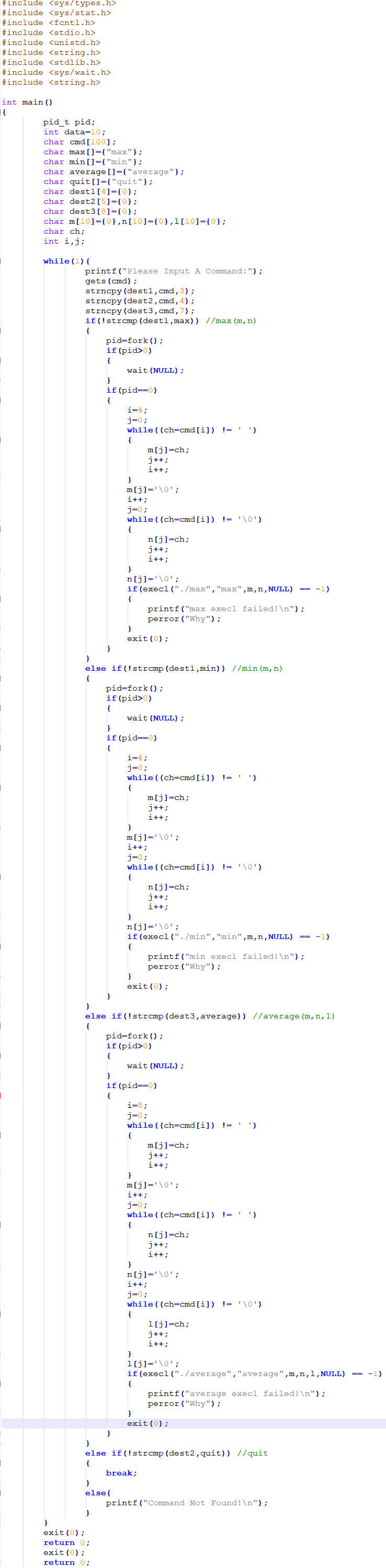
average.c与max.c也基本相同，只不过它接收三个参数，并求取平均值后输出。

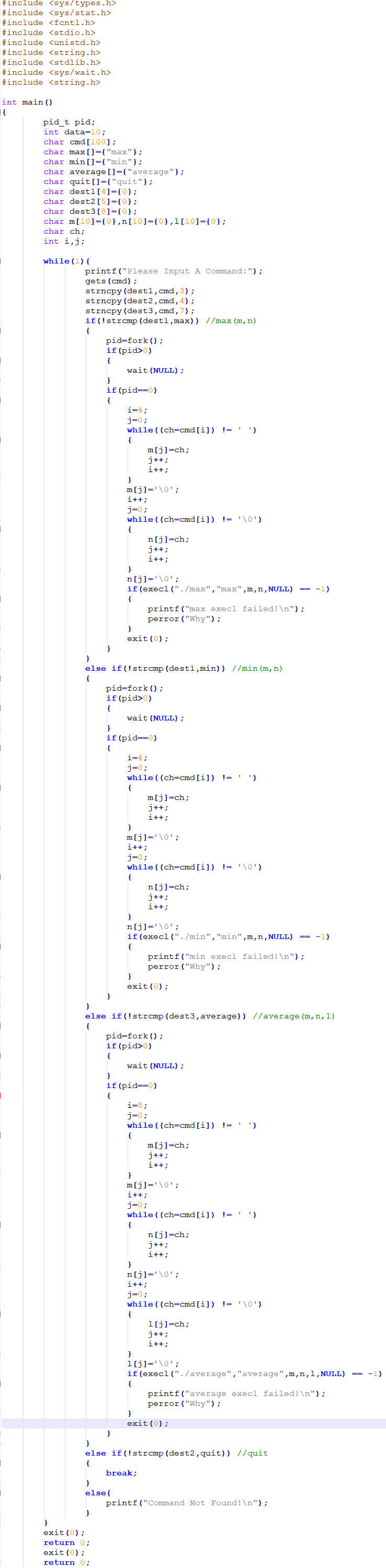
程序源码average.c



最后是主程序源码test3.c

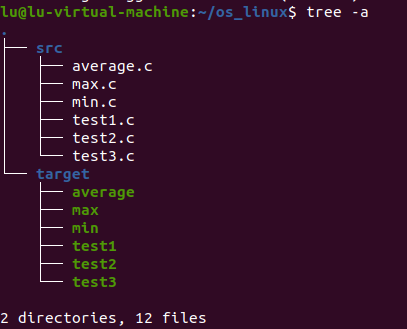




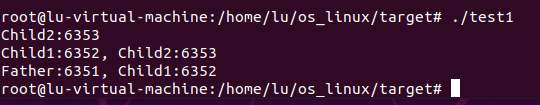


## 实验结果

本实验的所有代码均在Ubuntu20.04系统上运行。项目文件结构如下图所示，其中src文件夹保存.c源文件；target文件夹保存编译生成的可执行文件。

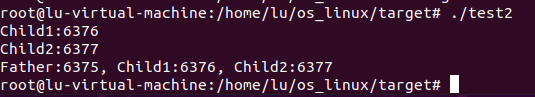


test1运行结果：



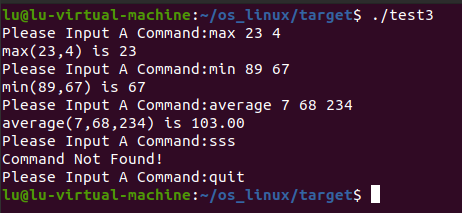
从打印出的进程id号可以看出，Child2的父进程是Child1，Child1的父进程是Father，同时，各个父进程均打印出了本进程和子进程的id号，满足实验的要求一。

test2运行结果：



从打印的进程id号可以看出，Child1和Child2的父进程均为Father，同时，各个父进程均打印出了本进程和子进程的id号，满足实验要求二。

test3 运行结果：



可以看出，当运行test3时，首先给出命令提示符：”Please Input A Command:”接着由用户输入指令，共支持四条指令，即max、min、average、quit，指令与参数之间，参数与参数之间用空格隔开。其中max和min指令仅支持两个参数，average支持仅支持三个参数。若遇到这四条指令之外的字符串，则会提示”Command Not Found!”。命令执行完毕后，程序继续询问下条指令，直到用户输入quit退出。满足实验的要求三。

## 实验中遇到的问题和解决办法

在要求一和要求二的实验中，一开始由于没有调用sleep()函数导致输出的时候，父进程和子进程输出的顺序不固定，解决办法是使用sleep()函数让父进程堵塞一段时间，这样就可以保证子进程先于父进程执行。

在要求三的实验中，由于没有搞清楚execl函数的第一个参数的具体含义，导致程序出现”Permission denied”警告，后查阅资料后发现该参数应该是完整的路径名，而不是所在目录的路径。重新添加文件的完整路径之后，问题得以解决。还有就是输入quit指令无法退出，后发现是因为每次创建子函数时最后没有调用exit函数自动终止，导致有许多未终止的子进程重复执行循环输入程序。于是在max、min、average程序的子进程程序段的最后都加上exit(0)语句，使子进程执行结束后自动终止，而不干扰其他进程。在quit程序段中直接加入break语句，意为跳出输入循环，最后执行exit(0)终止主进程。这样，问题便得到了解决。

# 2 处理器调度作业

## 2.1 实验目的

熟悉使用各种单处理器调度的各种算法，加深对于处理机调度机制的 理解。练习模拟算法的编程技巧，锻炼分析试验数据的能力。

## 2.2 实验说明

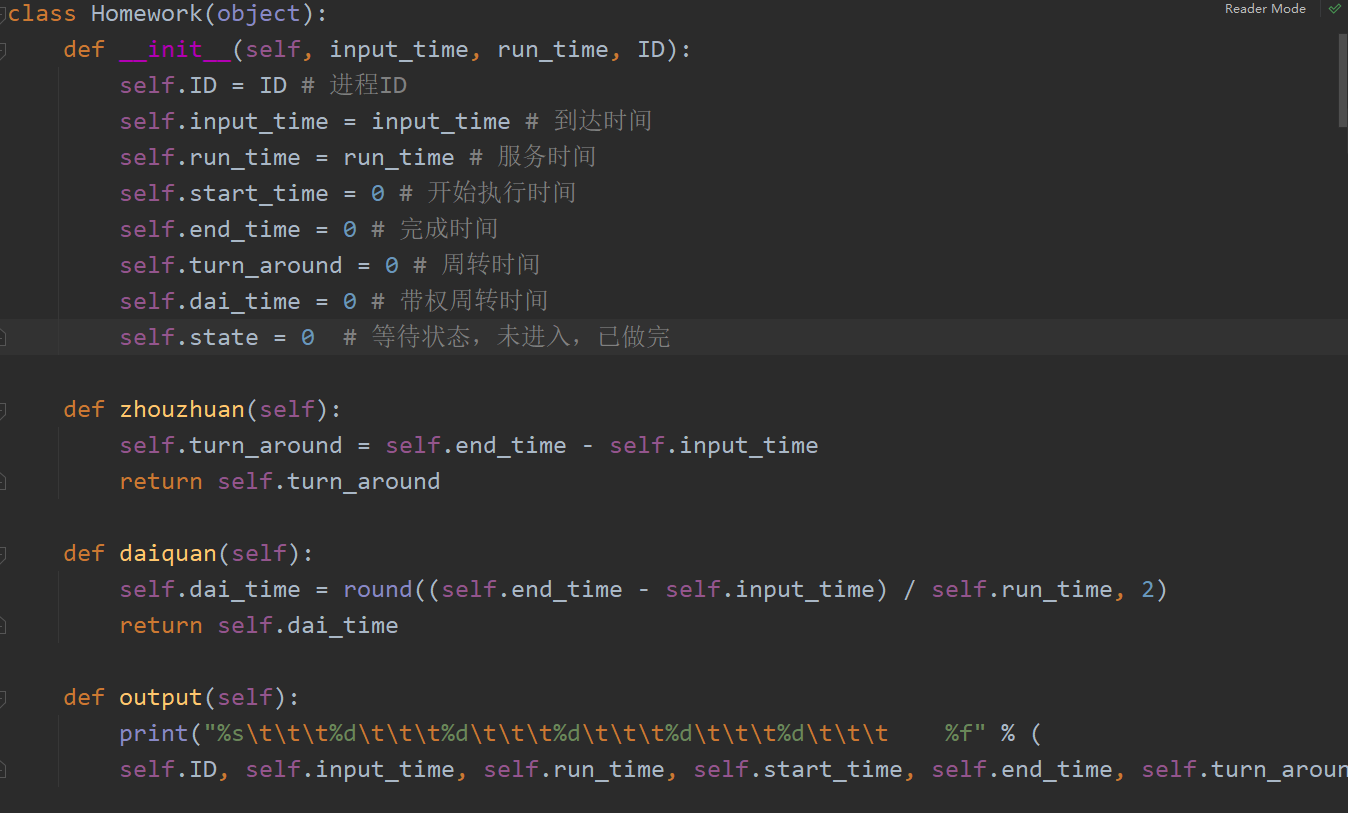
对随机给出的一个进程调度实例，如：

|  |  |  |
| --- | --- | --- |
| 进程 | 到达时间 | 服务时间 |
| A | 0 | 3 |
| B | 2 | 6 |
| C | 4 | 4 |
| D | 6 | 5 |
| E | 8 | 2 |

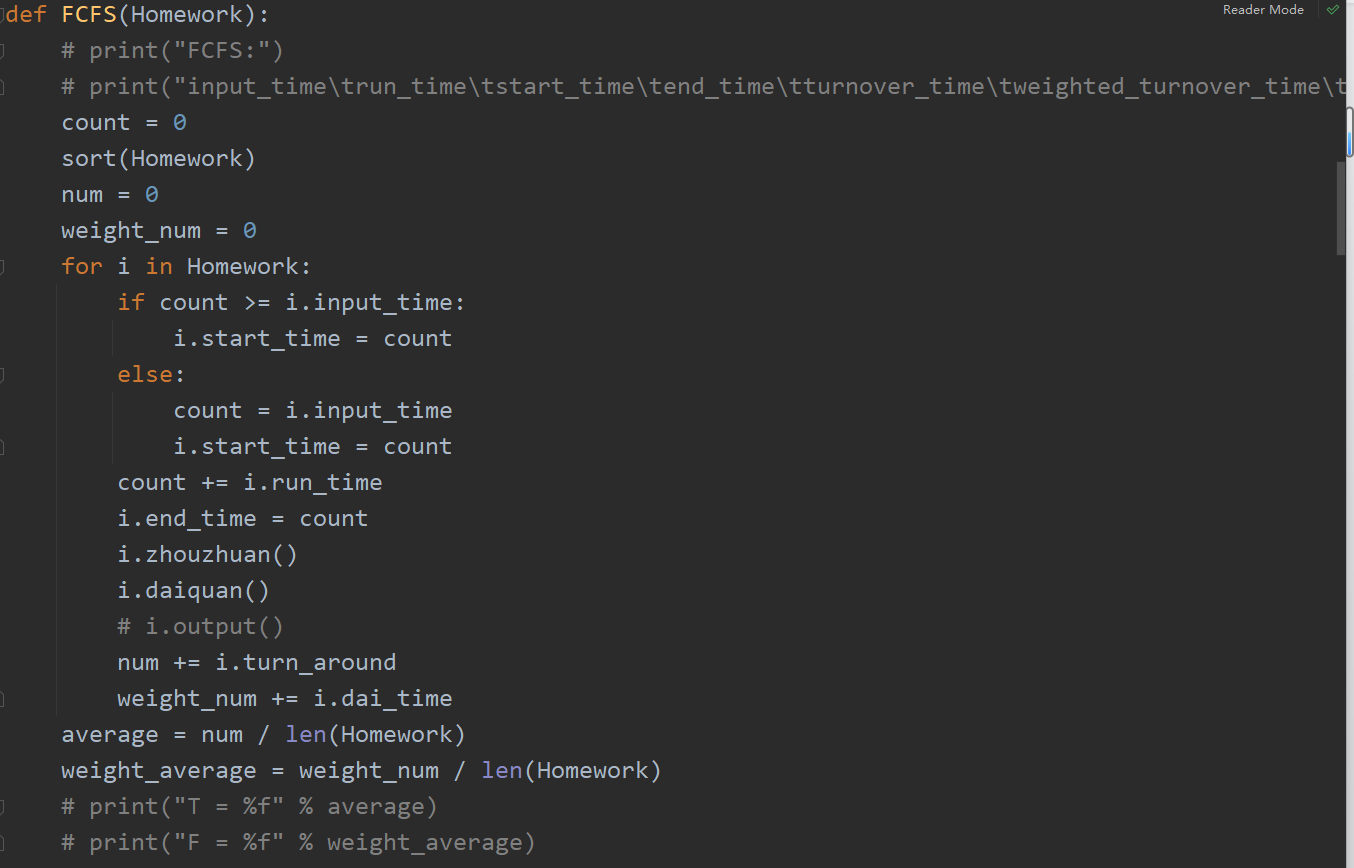
模拟进程调度，给出按照算法先来先服务 FCFS、轮转 RR（q=1）、最短进程优先 SJF、最高响应比优先 HRN 进行调度各进程的完成时间、周转时间、带权周转时间。

## 2.3 实验过程

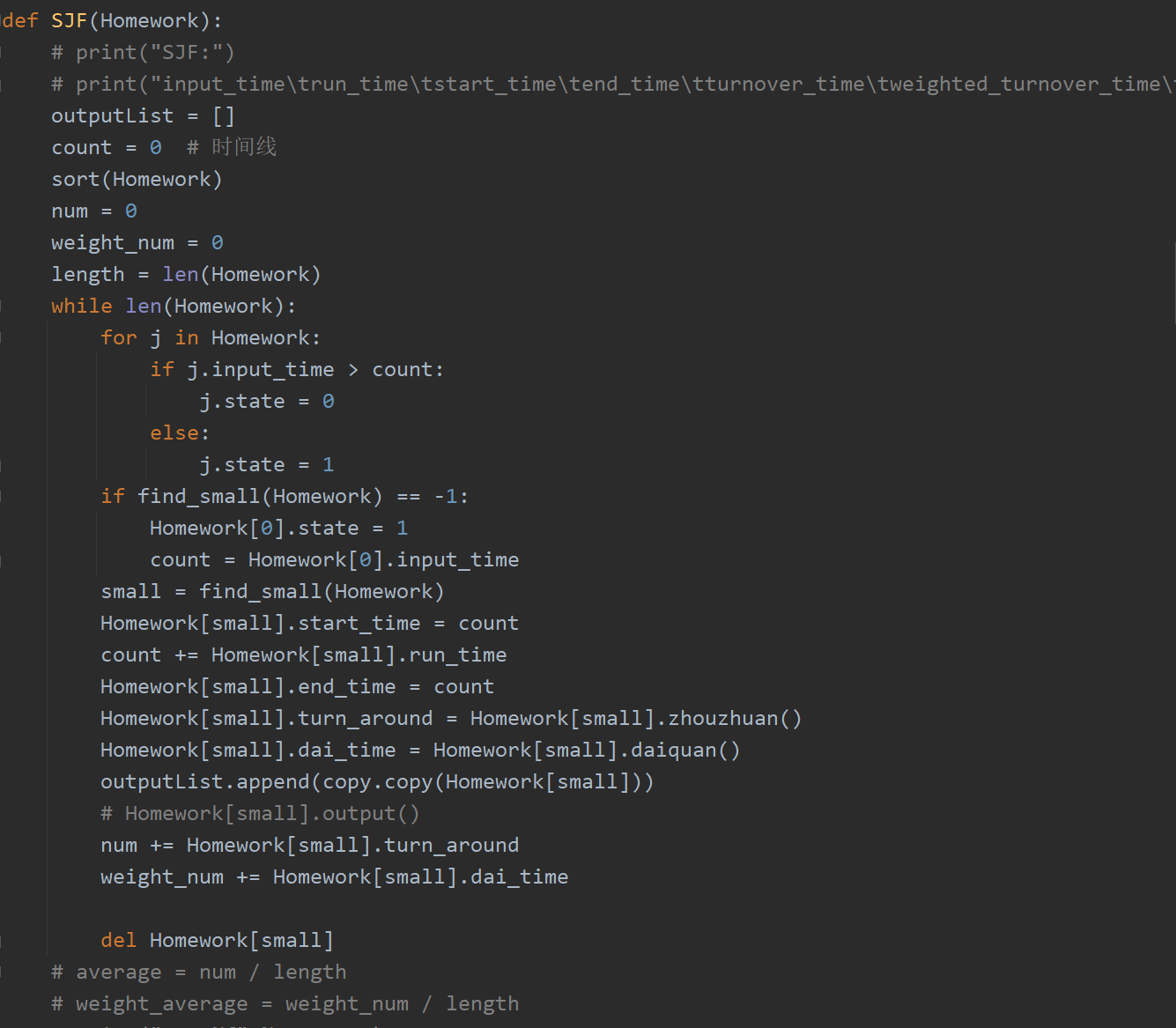
首先，为进程定义一个Homework类，初始化方法是输入进程的ID、到达时间和服务时间。Homework类有8个属性：进程ID、到达时间input\_time、服务时间run\_time、开始执行时间start\_time、完成时间end\_time、周转时间turn\_around、带权周转时间dai\_time。有三个方法：计算周转时间zhouzhuan()、计算带权周转时间daiquan()、输出自身值output()(该方法仅做调试用)。



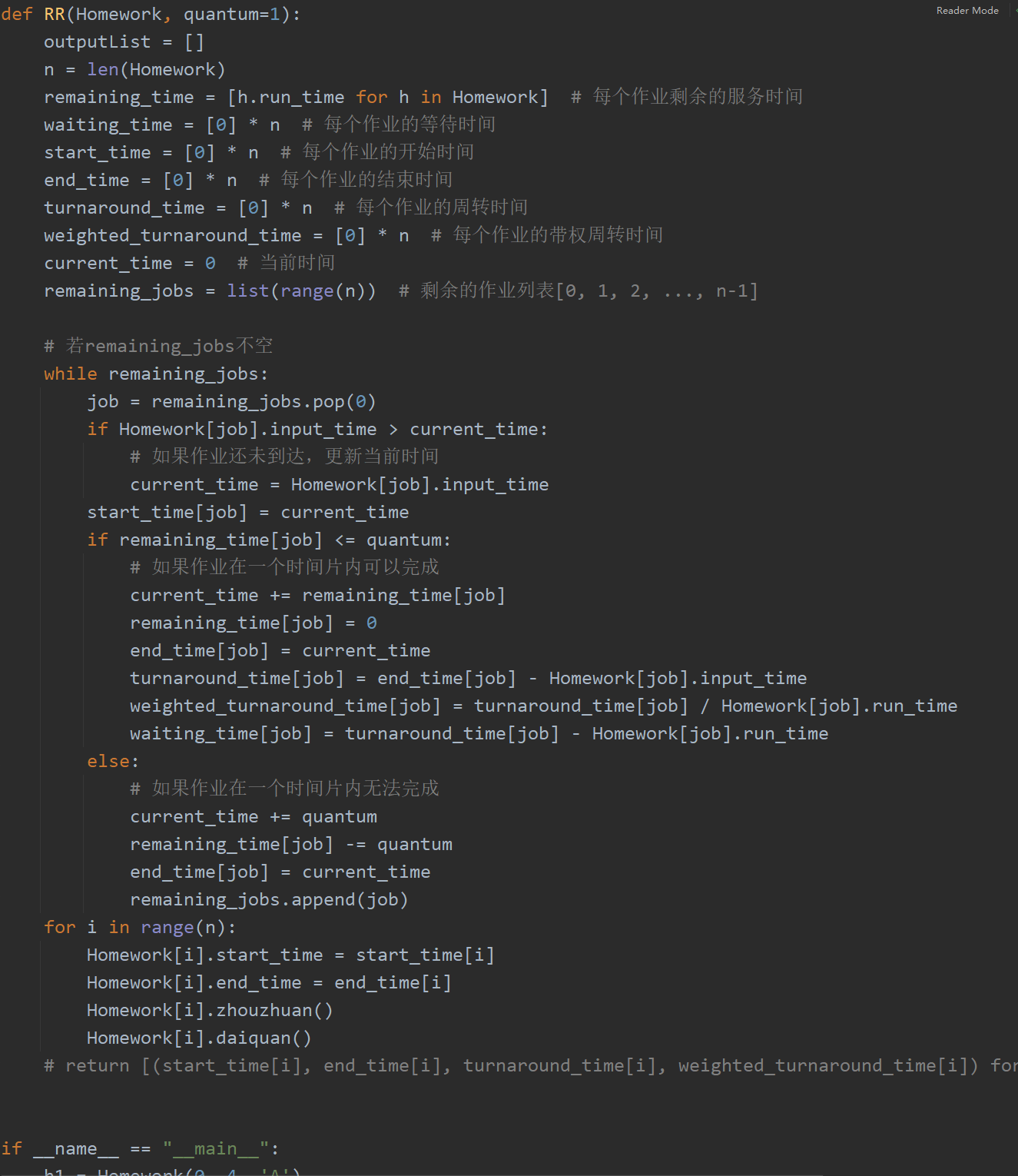
对于FCFS调度算法，首先根据进程的到达顺序进行排序，然后依次取出队列中的进程调度，并更新各个进程的各个属性参数。

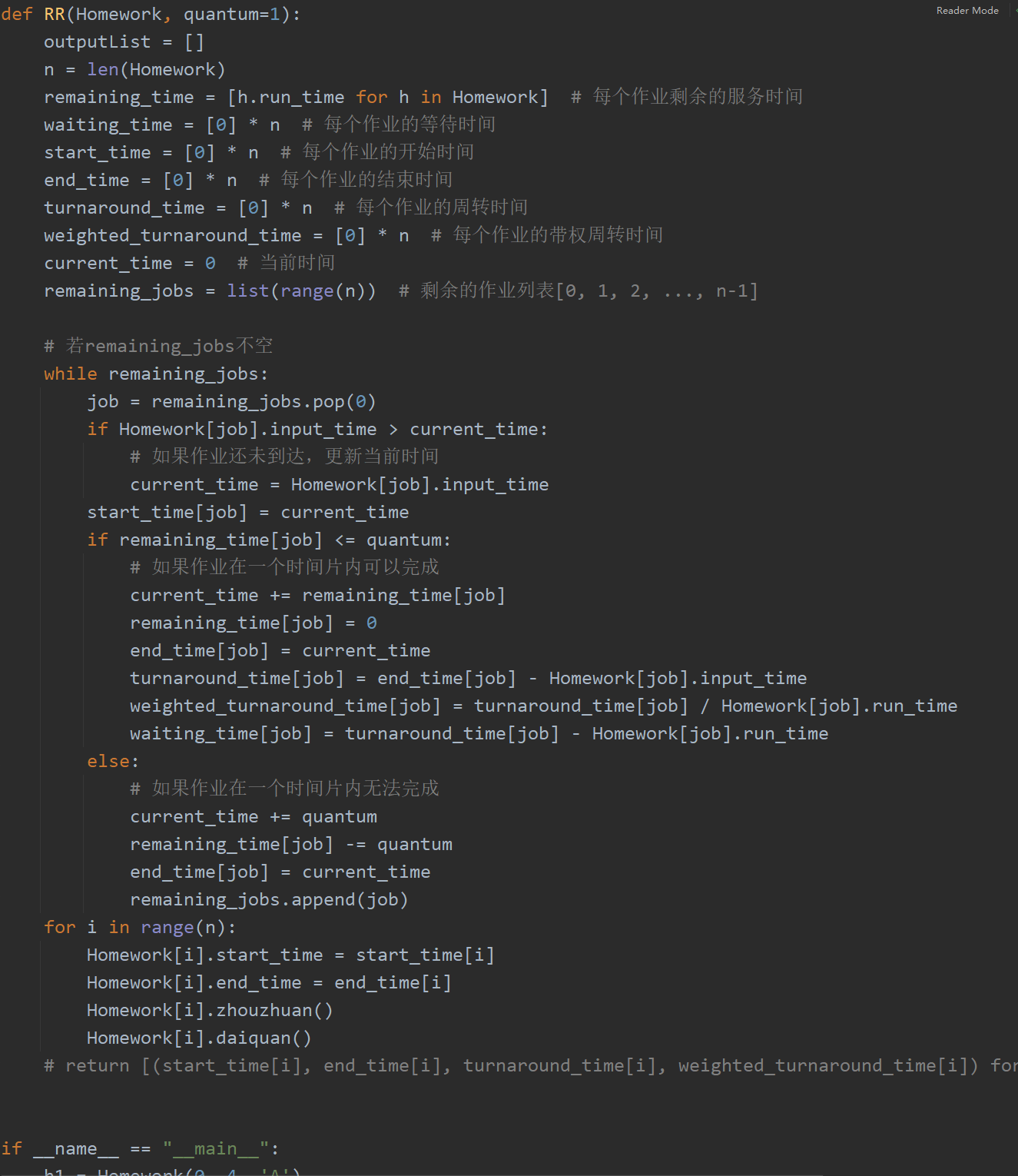


对于SJF调度算法，首先也是根据进程的到达顺序进行排序，但是在实际执行的过程中，根据实际参与进程的服务时间来调度，用state属性标识该进程是否已经进入到系统中。

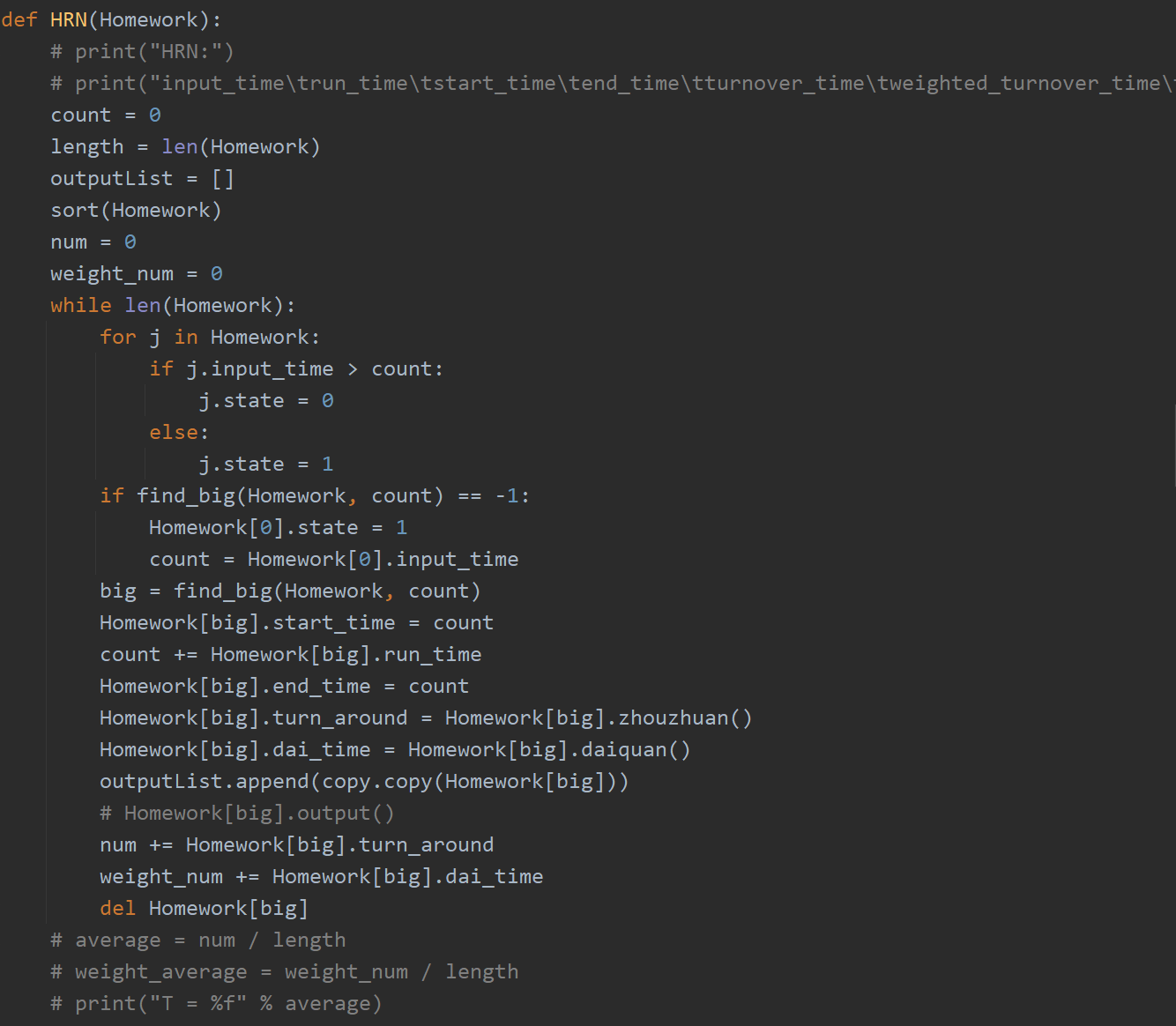


对于RR算法，设定默认时间片为1。如果进程还未到达，则更新当前时间；如果进程在一个时间片内可以完成，则直接计算结束时间；如果进程在一个时间片内无法完成，则立刻调度其他在队列中的进程。





对于HRN调度算法，首先也是根据进程的到达顺序进行排序，但是在实际执行的过程中，根据实际参与进程的响应比大小来调度，用state属性标识该进程是否已经进入到系统中。



## 2.4 实验结果

初始界面，随机生成5个进程序列：



若点击FCFS按钮，则在下方显示FCFS调度结果：



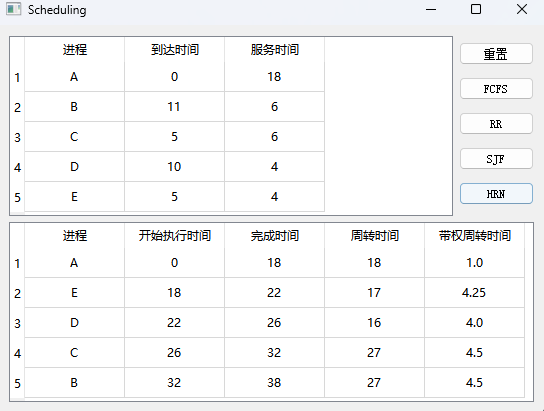
若点击RR按钮，则在下方显示RR调度结果：



若点击SJF按钮，则在下方显示SJF调度结果：



若点击HRN按钮，则在下方显示HRN调度结果：



若点击重置按钮，则重新随机生成进程序列：



## 2.5 实验中遇到的问题和解决办法

在SJF调度算法中，开始设计的时候每次输出一个进程就将其删除，但是这样不利于图形界面的后期显示。解决办法是在初始时定义一个列表类型的outputList，专门用于存放最终的输出结果，而每次计算出并更新进程的各属性时，就使用append方法将该进程的一个副本添加到outputList中，最后返回outputList即可。

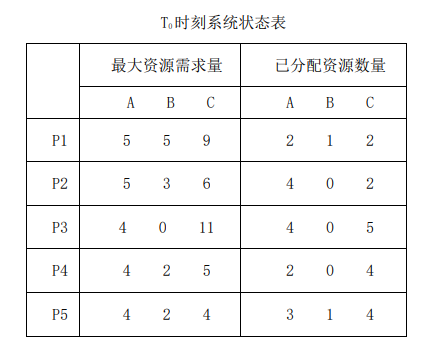
# 3 进程同步和死锁作业

## 3.1 实验目的

对于编程模拟生产者/消费者问题，目的是为了加深对于进程同步和互斥的了解，掌握使用信号量解决 进程同步和互斥的编程方法。对于编程实现银行家算法，加深对于进程死锁避免的了解，掌握使用银行家算法实 现死锁避免的编程方法。

## 3.2 实验说明

对于编程模拟生产者/消费者问题，要求以生产者/消费者模型为根据，编写一个图形界面程序，创建 n 个线程，使用记录型信号量机制，模拟生产者和消费者的工作流程。对于编程实现银行家算法，设系统中有 3 种类型的资源（A，B，C）和 5 个进程 P1、 P2、P3、P4、P5，A 资源的数量为 17，B 资源的数量为 5，C 资源的 数量为 20。在 T0时刻系统状态见下表所示。



编写一个图形界面程序，可以：

1. 判断 T0时刻是否为安全状态？若是，请给出安全序列。

2. 在 T0时刻，对进程 P2 请求资源（m，n，p），m、n、p 分别 是申请的 A、B、C 资源数（大于等于零的整型值，由程序提供接口， 让用户动态输入），程序可以判断是否能实施资源分配。

## 3.3 实验过程

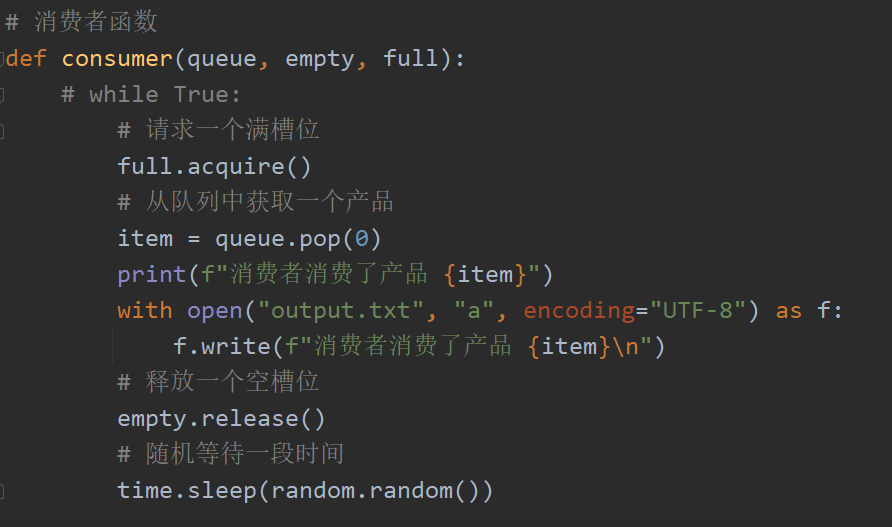
### 3.3.1 编程模拟生产者/消费者问题

首先创建ProducerAndConsumer.py文件，用于实现生产者消费者模型。而Window.py用于实现图形界面程序。

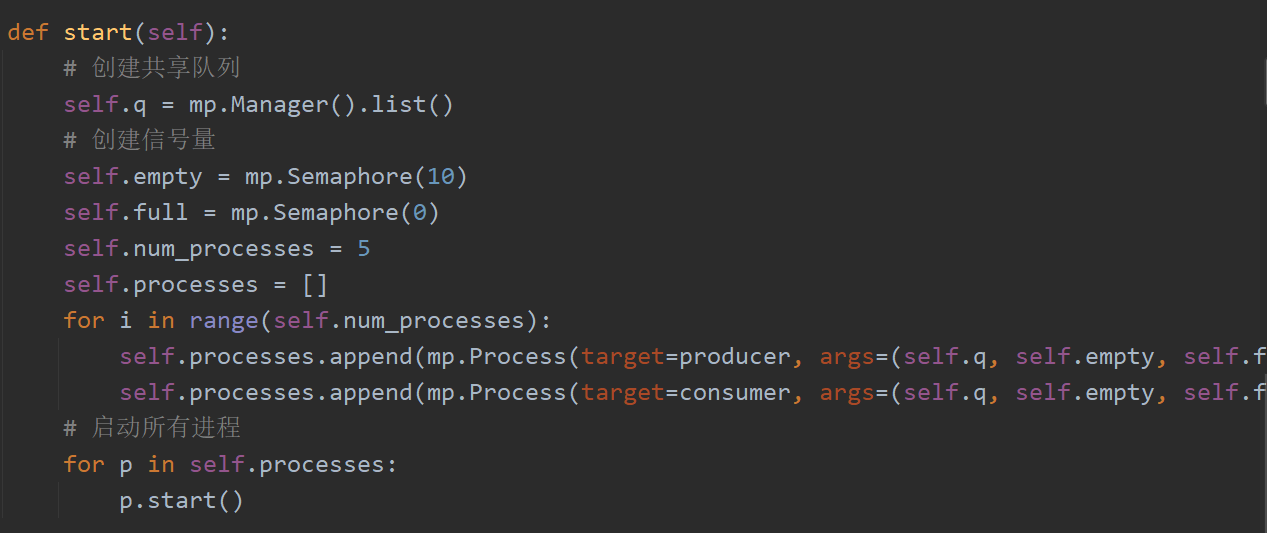
在生产者函数中，首先随机生成一个产品item，然后使用empty.acquire()请求一个空槽位，如果没有空槽位则会阻塞等待。接着使用queue.append(item)添加产品，queue'内部可以自动保证多进程的安全访问。最后使用full.release()释放一个满槽位，表示队列中有一个产品可供消费。



在消费者函数中，首先使用full.acquire()请求一个满槽位，如果没有满槽位则会阻塞等待。接着使用queue.append(item)添加产品，queue'内部可以自动保证多进程的安全访问。最后使用empty.release()释放一个空槽位，表示队列中有一个空槽位可供生产。

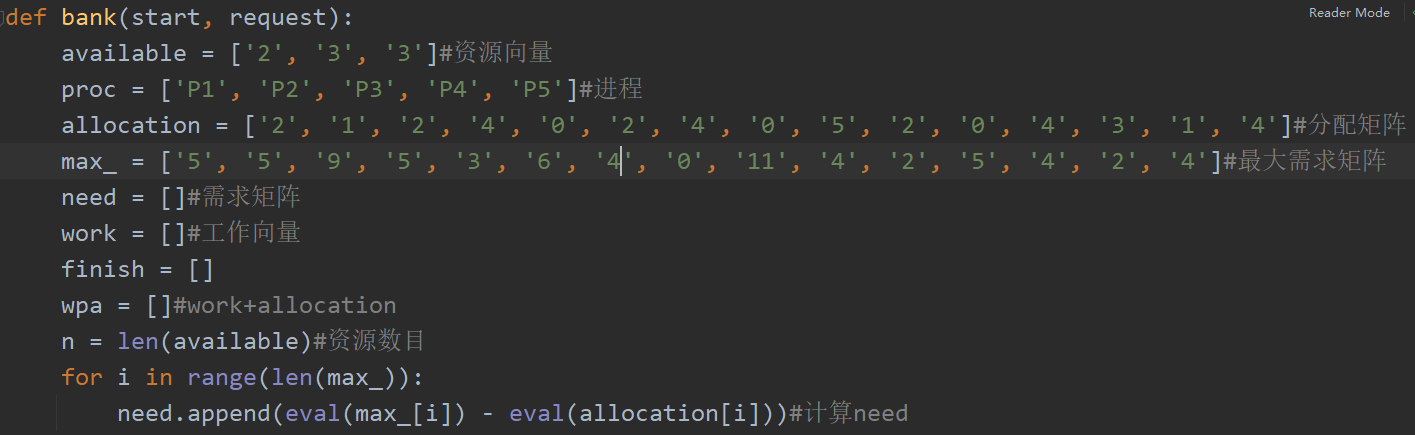


在图形界面程序中，每次开始时使用mp.Manager().list()创建一个共享列表q（最大容量为10），并使用mp.Semaphore()创建一个空槽位信号量empty（初值为10）和一个满槽位信号量full（初值为0）。随后创建5个生产者线程和5个消费者线程。

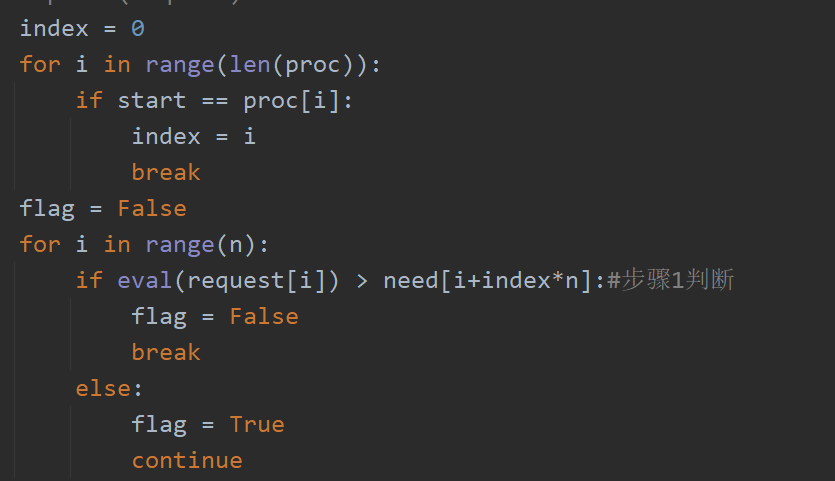


### 3.3.2 编程实现银行家算法

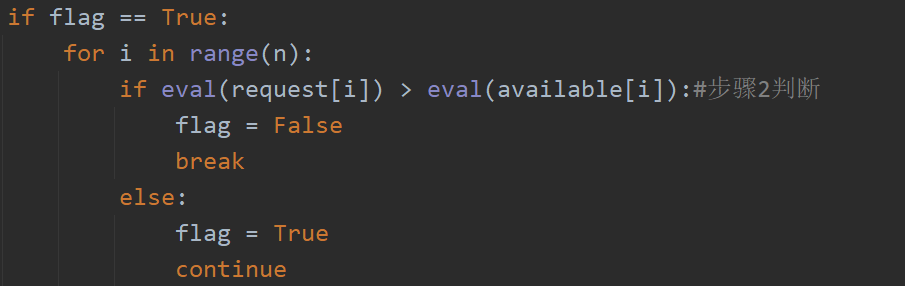
首先创建Bank.py文件，用于银行家算法。而Window.py用于实现图形界面程序。在Bank.py文件中，首先初始化资源向量、进程ID、分配矩阵、最大需求矩阵、需求矩阵和工作向量。初始时循环计算需求矩阵：



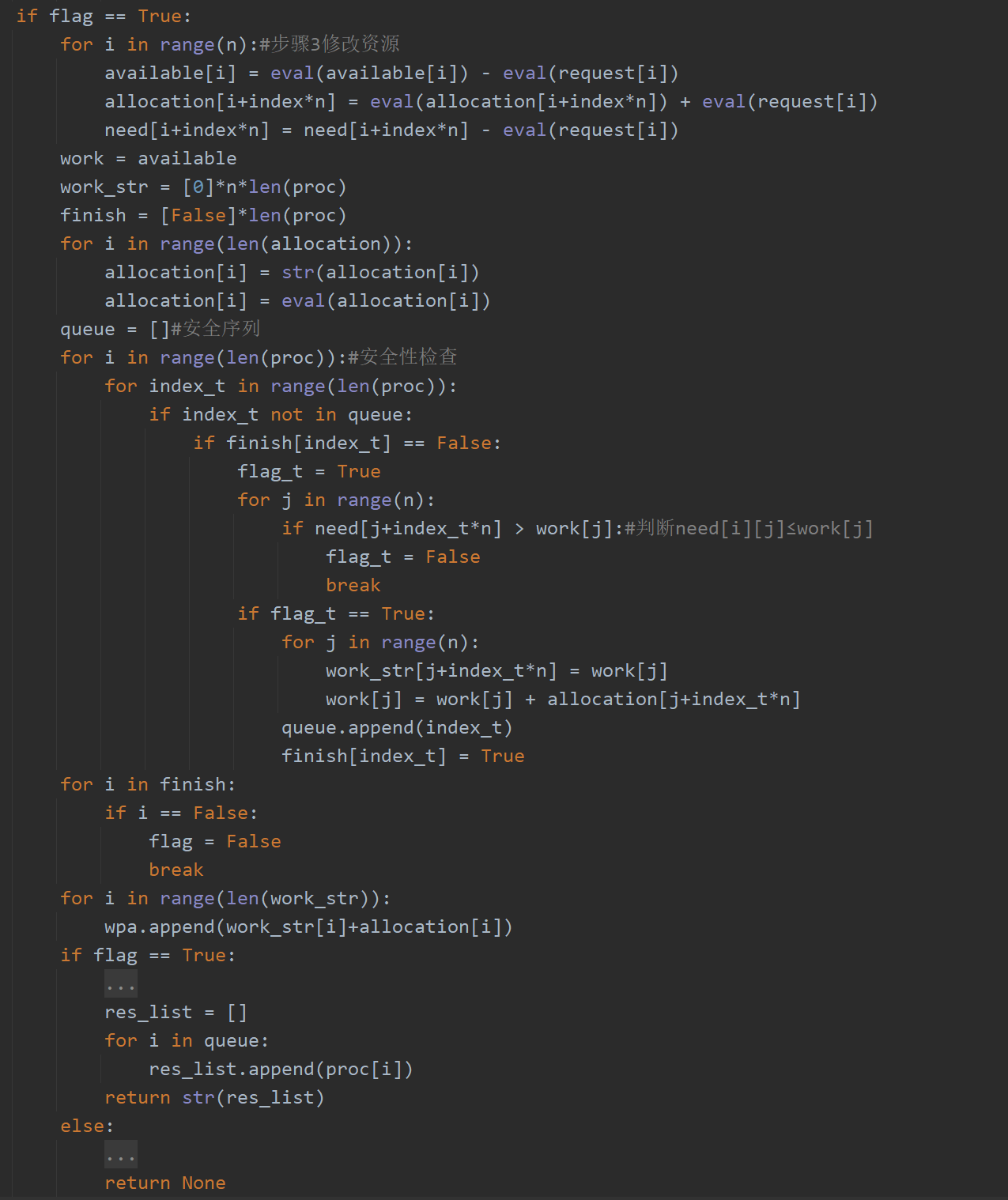
然后进行银行家算法的第一步判断（是否超过最大需求）：



银行家算法的第二步判断（是否超过可分配的资源数）：



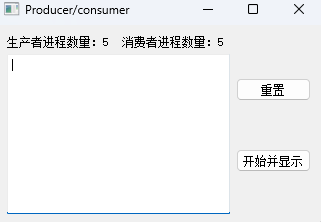
若上两步判断通过，则进行预分配，预先更新分配矩阵、资源矩阵、需求矩阵，然后通过执行安全性检查算法：



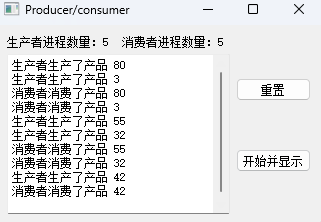
## 3.4 实验结果

### 3.4.1 编程模拟生产者/消费者问题

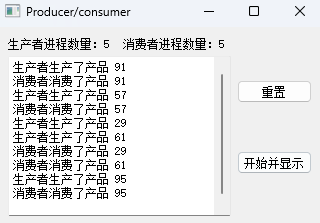
初始界面：



点击开始并显示，文本框中输出运行情况：



点击重置则会清空文本框，再点击开始并显示即可再次显示：



### 3.4.2 编程实现银行家算法

初始界面：



点击安全性检查按钮，输出当前是否处于安全状态，若是，则显示存在的安全序列：



在右侧输入框中依次输出请求资源的进程、申请A、B、C资源的数量，然后点击能否分配，输出分配判断，若能成功分配，则显示存在的安全序列：



## 3.5 实验中遇到的问题和解决办法

在编程模拟生产者/消费者问题中，遇到的问题是：使用互斥锁queue\_lock来保证队列的原子性操作，会导致pop from empty list错误，原因是共享数据结构（如列表）的访问不是线程安全的，因此可能会导致竞争条件和死锁的发生。解决方法是使用mp.Manager().list()创建一个共享列表，用于保证线程访问的安全性。

在编程实现银行家算法问题中，遇到的问题是：如何在现有的银行家分配算法上单独添加安全性检查算法，一个简单的解决方法是：任意选择一个进程，使其申请的所有资源数均为0，若能成功分配，则说明当前状态是安全状态。

# 4 内存管理作业

## 4.1 实验目的

加深对于存储管理的了解，掌握虚拟存储器的实现原理；观察和了解重要的页面置换算法和置换过程。练习模拟算法的编程技巧，锻炼分析试验数据的能力。

## 4.2 实验说明

1. 示例实验程序中模拟三种置换算法：OPTIMAL(最优)算法、LRU 算法和 FIFO 算法。

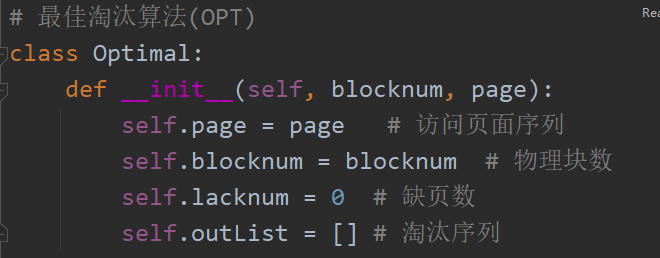
2. 给定任意序列不同的页面引用序列和任意分配页面数目，显示三种算法的页置换过程。

3. 能统计和报告不同置换算法情况下依次淘汰的页号、缺页次数（页错数）和缺页率。

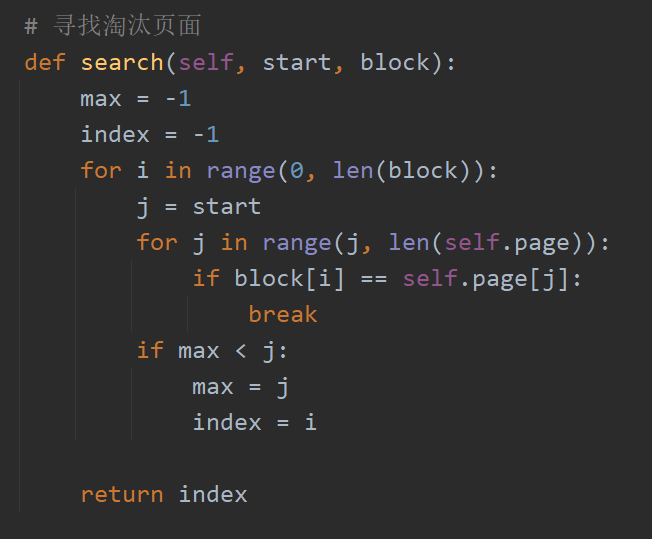
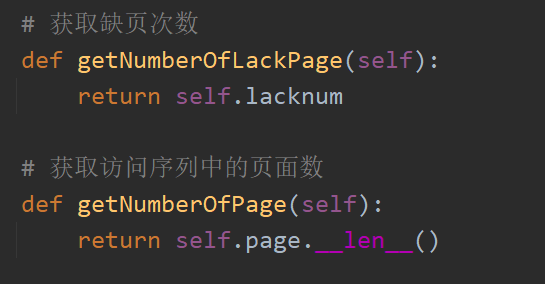
## 4.3 实验过程

首先创建RAM.py文件，用于实现内存管理。而Window.py用于实现图形界面程序。在RAM.py文件中，创建Optimal类、Fifo类和Lru类。分别表示OPTIMAL算法、FIFO算法和LRU算法。

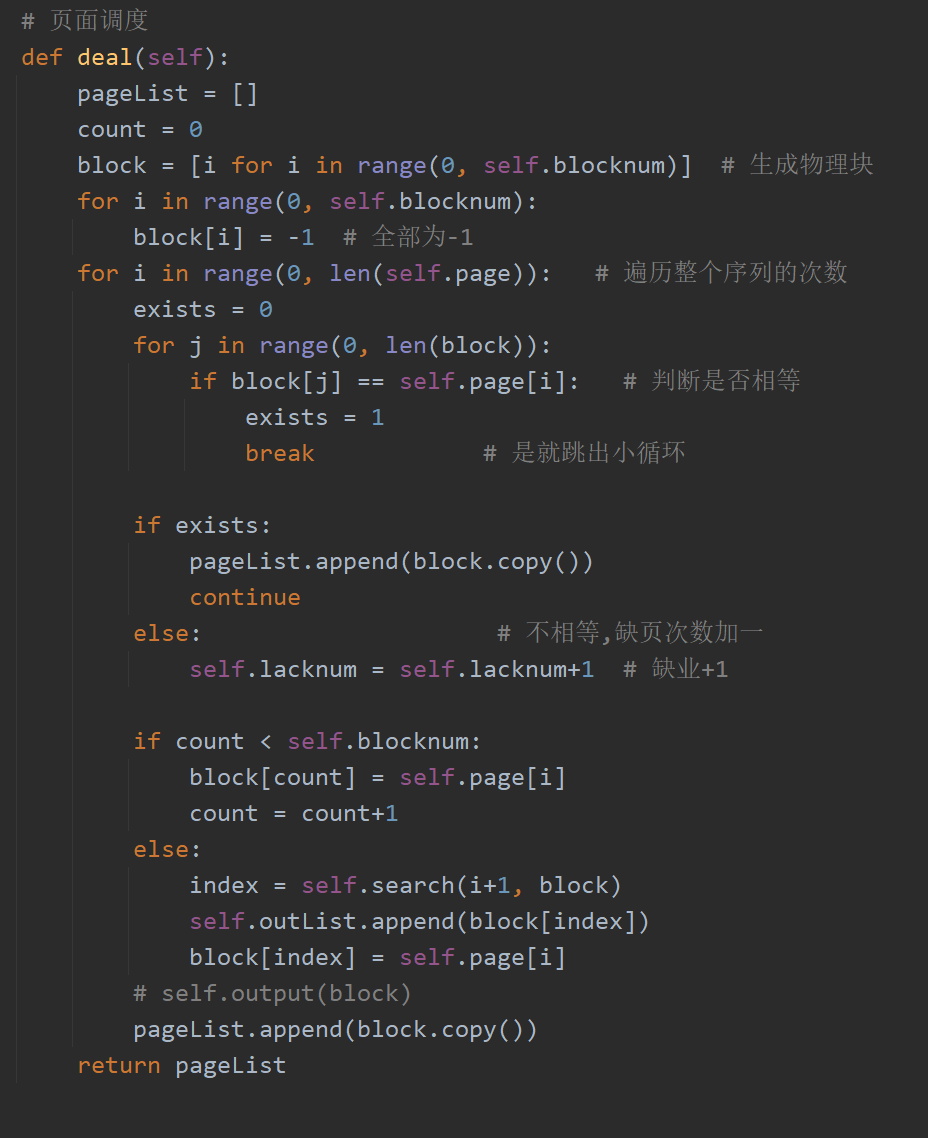
在Optimal类中，定义四个成员变量：page表示访问页面序列、blocknum表示物理块数、lacknum表示缺页数、outList表示淘汰序列。



定义getNumberOfLackPage方法来获取缺页次数、getNumberOfPage方法来获取访问序列中的页面数、search方法来寻找淘汰的页面。

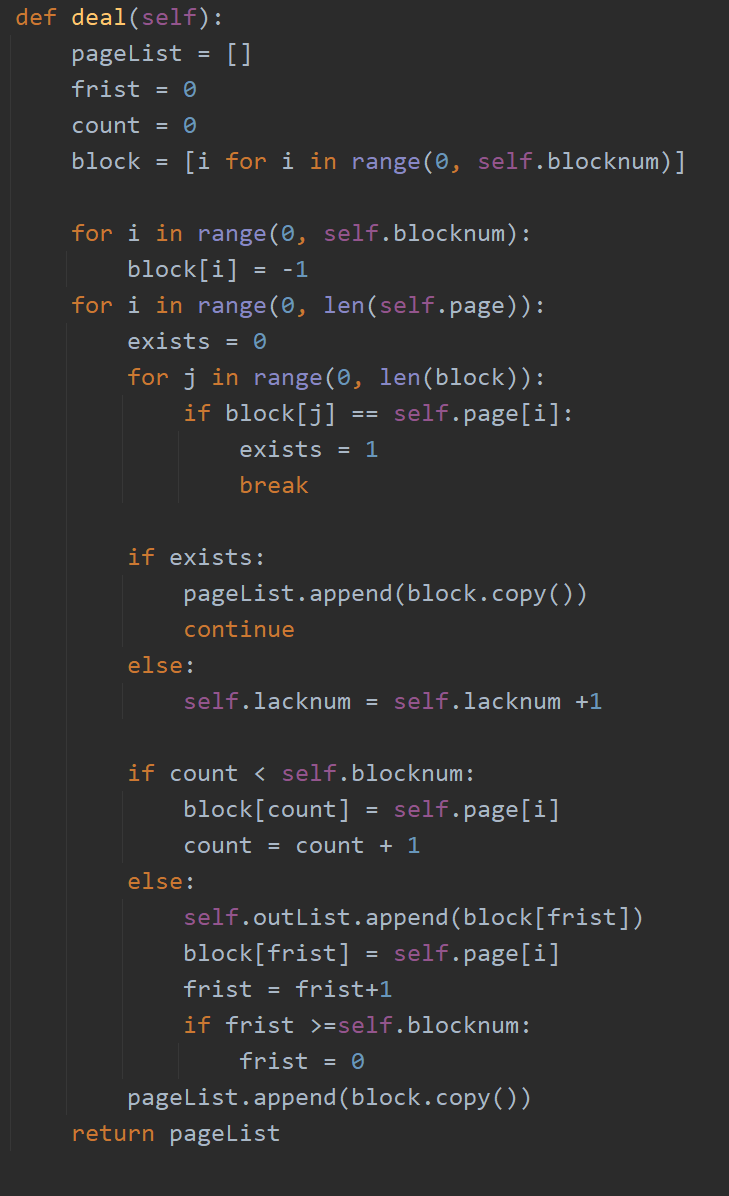


定义deal方法来实现页面调度：首先定义pageList来存放输出列表，block用于生成物理块，若遍历整个序列没有相等的页面，则缺页次数加一，否则跳过当前循环。每次结束前将当前block的副本保存。

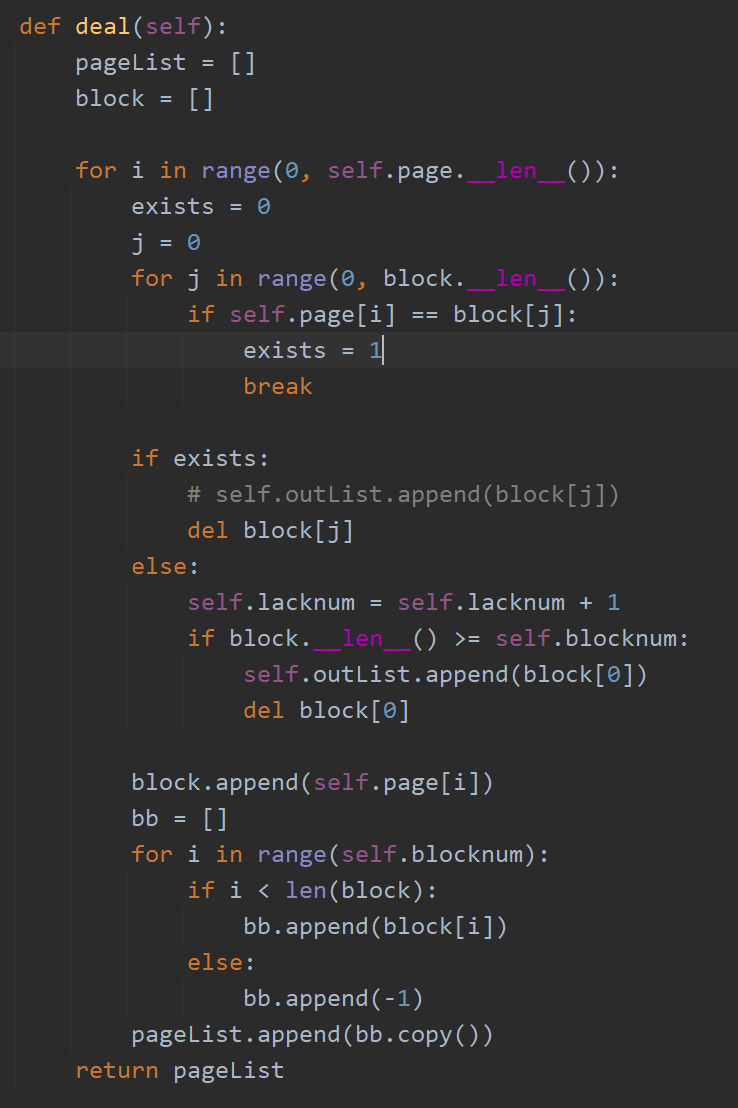


其他两类的定义与Optimal类差不多，只是在deal方法上存在区别。对于Fifo算法，

需要始终保持一个最久的队列，每次置换时总是换出第一个队列的元素（即最老的页面）。

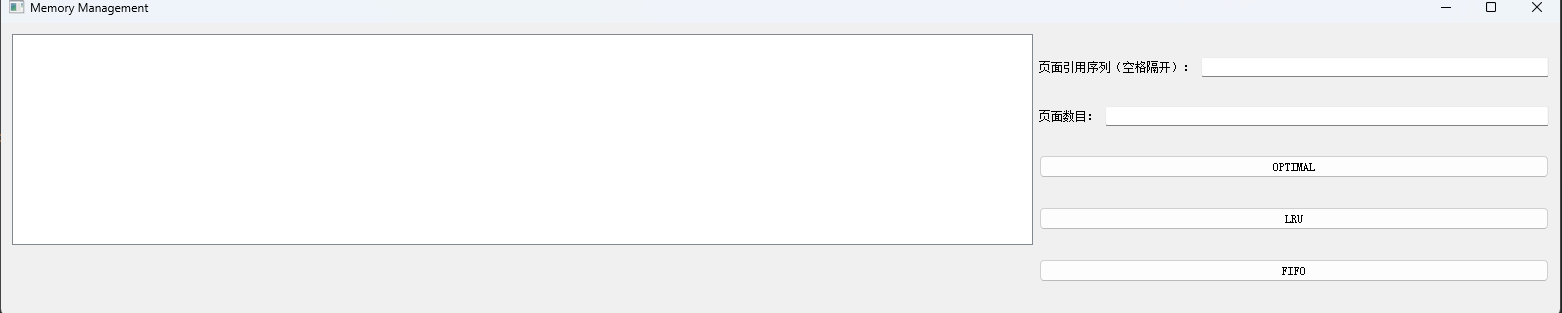


对于LRU算法，在deal方法的实现上，始终维护一个栈，每次访问页面时，都将改页面置顶，这样缺页时最先换出栈底的物理块（即最近最久未用的物理块）。

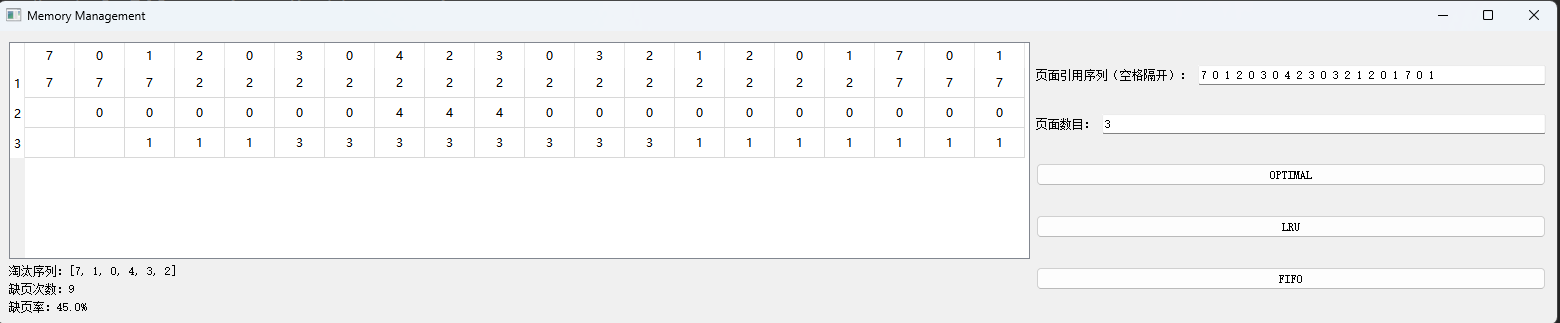


## 4.4 实验结果

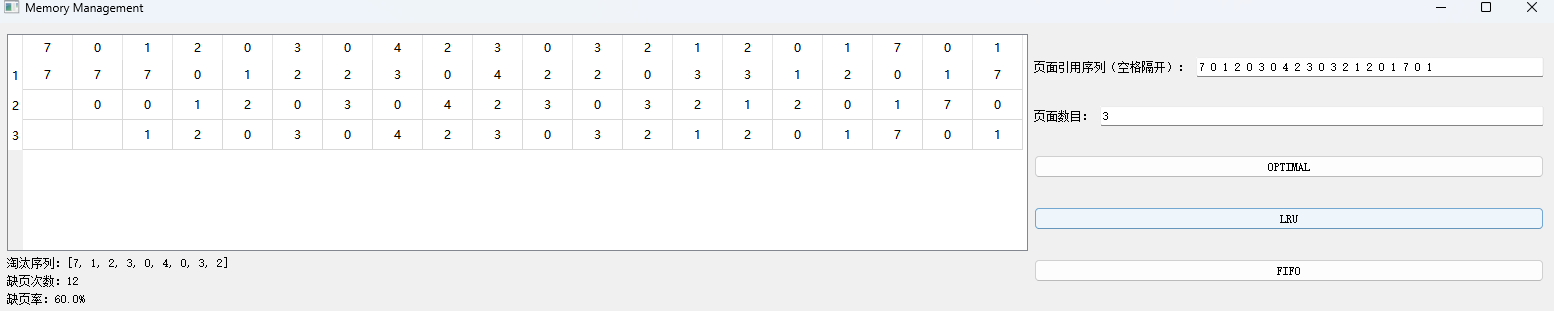
初始界面，要求输入页面引用序列（空格隔开）和页面数目：



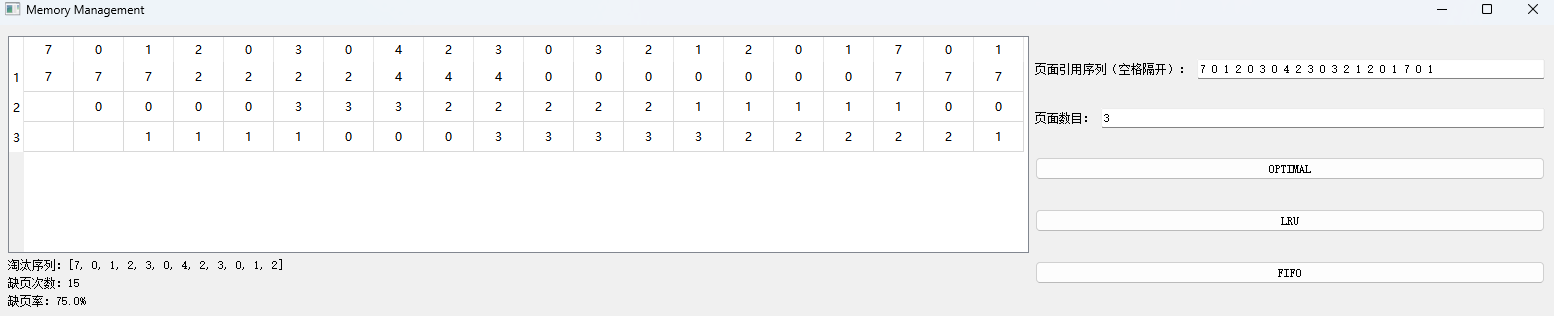
输入页面引用序列和页面数目，然后点击OPTIMAL按钮，即可得到OPTIAML算法的输出结果。



按下LRU按钮，即可得到LRU算法的输出结果：



按下FIFO按钮，即可得到FIFO算法的输出结果：



## 4.5 实验中遇到的问题和解决办法

在本实验中，我遇到的问题是：在实现LRU算法时，由于需要将最新的页面换到第一个，会删除原有位置的元素，导致页面刚开始的时候只有1-2个物理块，并导致最终图形界面时访问列表越界，解决的方法是若页面数小于物理块数则把缺省的页面置为-1，表示空。

# 5 磁盘调度作业

## 5.1 实验目的

加深对于操作系统设备管理技术的了解，体验磁盘移臂调度算法的重要性；掌握几种重要的磁盘移臂调度算法，练习模拟算法的编程技巧，锻炼研究分析试验数据的能力。

## 5.2 实验说明

1.实验程序中模拟两种磁盘移臂调度算法：SSTF 算法和 SCAN 算法

2.能对两种算法给定任意序列不同的磁盘请求序列，显示响应磁盘

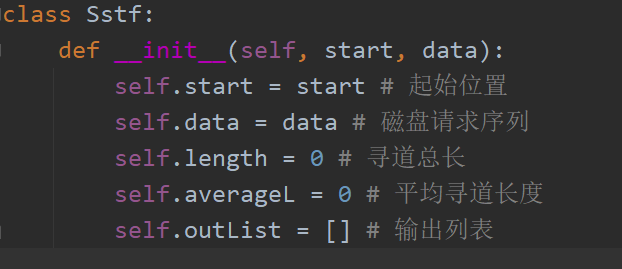
请求的过程。

3.能统计和报告不同算法情况下响应请求的顺序、移臂的总量。

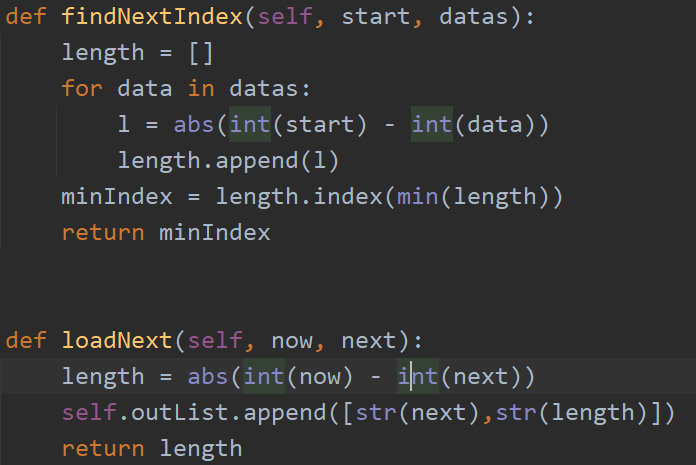
## 5.3 实验过程

首先创建SSTF.py文件和SCAN.py文件，用于实现SSTF和SCAN算法，而Window.py用于实现图形界面程序。

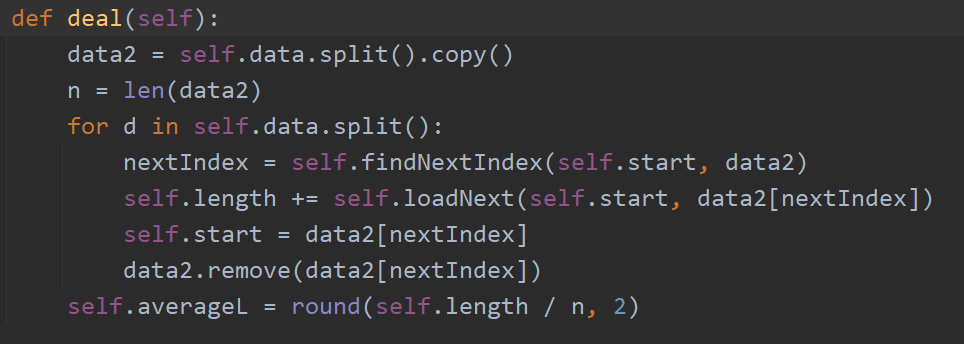
在SSTF.py文件中，创建Sstf类，该类的成员变量有：start表示起始位置、data表示磁盘请求序列、length表示寻道总长、averageL表示平均寻道长度、outList表示输出列表。



然后定义findNextIndex方法用于寻找下一个寻道块（通过计算各个磁道的距离来判断）、定义loadNext方法用于计算寻道长度。



最后定义deal方法用于实现SSTF算法。首先生成一个磁道请求序列的副本data2，得到总的任务数，然后遍历磁道请求序列data，调用findNext方法寻找下一个磁道，并更新新的开始位置。

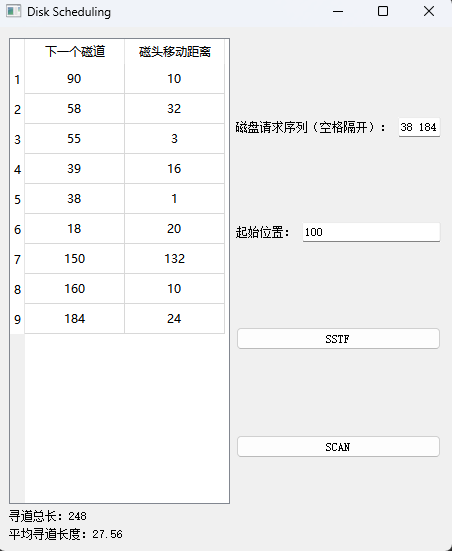


## 5.4 实验结果

初始界面，要求输入磁道请求序列（空格隔开）和起始位置：



输入磁道请求序列和起始位置，然后点击SSTF按钮，即可得到SSTF算法的输出结果。点击SCAN按钮，即可得到SCAN算法的输出结果。



## 5.5 实验中遇到的问题和解决办法

在本次实验中，我遇到的问题是：在输出SCAN算法的结果时，把内外循环的循环次数颠倒了，导致访问输出列表outList时越界，索引访问顺序也错误。解决方法是将内外循环的次数互换，并将索引访问顺序互换。

# 6 文件管理作业

## 6.1 实验目的

通过模拟文件的创建、删除操作，加深对于操作系统文件管理功能的了解，练习模拟算法的编程技巧，锻炼研究分析试验数据的能力。

## 6.2 实验说明

给出一个磁盘块序列：1、2、3、……、500，初始状态所有块为空的，每块的大小为 2k。选择使用空闲表、空闲盘区链、位示图三种算法之一来管理空闲块。对于基于块的索引分配执行以下步骤：

（1）随机生成 2k-10k 的文件 50 个，文件名为 1.txt、2.txt、……、50.txt，按照上述算法存储到模拟磁盘中。

（2）删除奇数.txt（1.txt、3.txt、……、49.txt）文件

（3）新创建 5 个文件（A.txt、B.txt、C.txt、D.txt、E.txt），大小为：7k、5k、2k、9k、3.5k，按照与（1）相同的算法存储到模拟磁盘中。

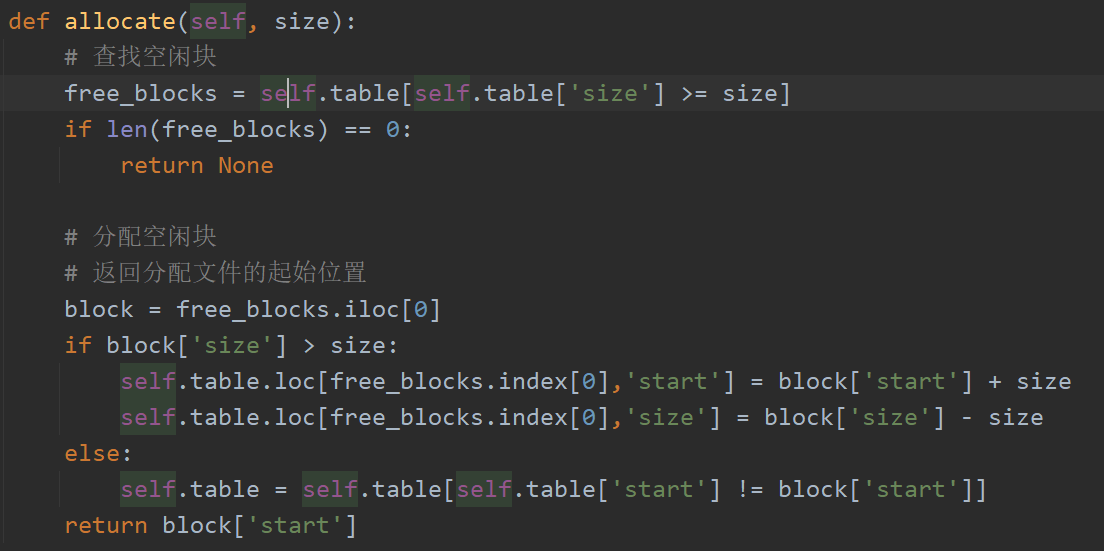
（4） 给出文件 A.txt、B.txt、C.txt、D.txt、E.txt 的盘块存储状态和所有空闲区块的状态。

## 6.3 实验过程

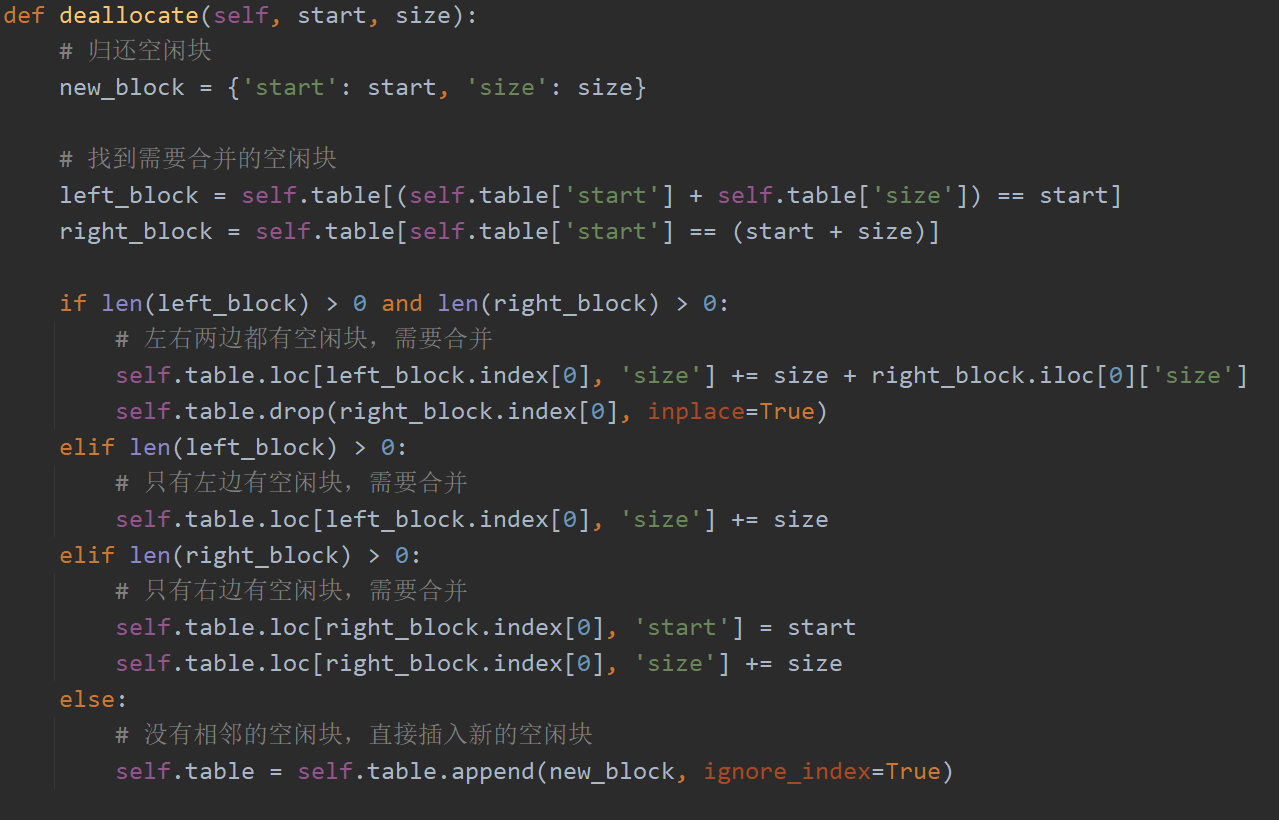
首先创建FM.py文件，用于实现文件管理，而Window.py用于实现图形界面程序。本实验选择用空闲表法实现文件管理。在FM.py中创建FreeBlockTable类，该类有三个成员变量，分别是：start表示磁盘起始存储位置、size表示磁盘总大小、table表示空闲表。



定义allocate方法实现空闲磁盘的分配，首先查找是否有满足大小的空闲块，若有则为之分配空闲块并修改空闲表的相应参数。若没有满足大小的空闲块，则分配失败。

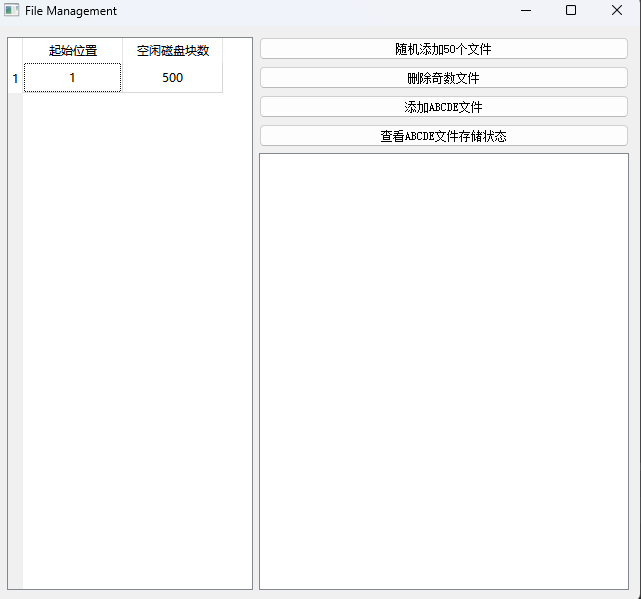


定义deallocate方法实现磁盘块的回收，首先定位到文件存储的起始位置和占用磁盘块大小，然后查看可能需要合并的左右相邻空闲块。若左右两边都有空闲块，需要将三个相邻的块合并。若只有左边有空闲块，则和左边空闲块合并。若只有右边有空闲块，则和右边的空闲块合并。若没有相邻的空闲块，则直接插入新的空闲块。

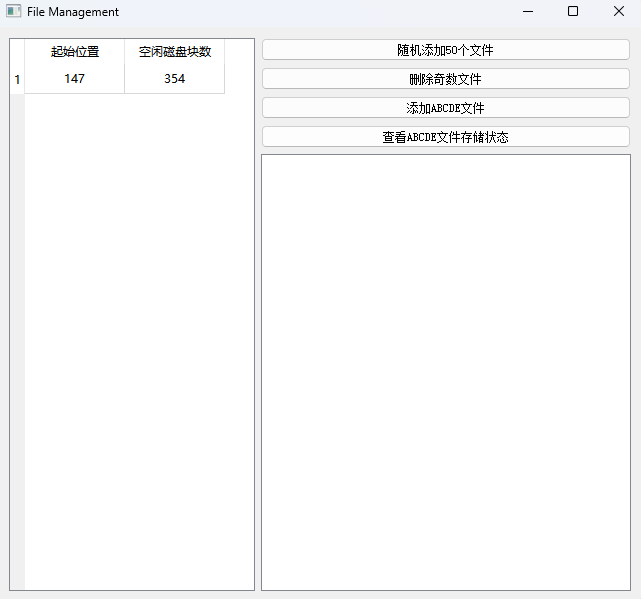


## 6.4 实验结果

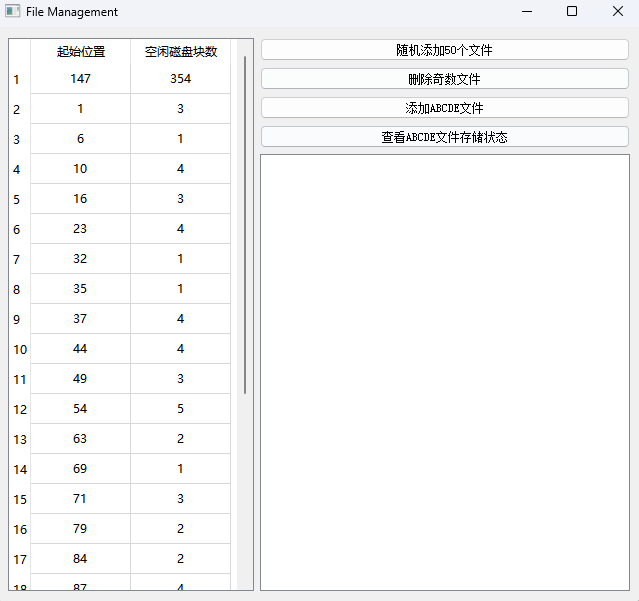
初始界面，左侧显示初始的空闲表：



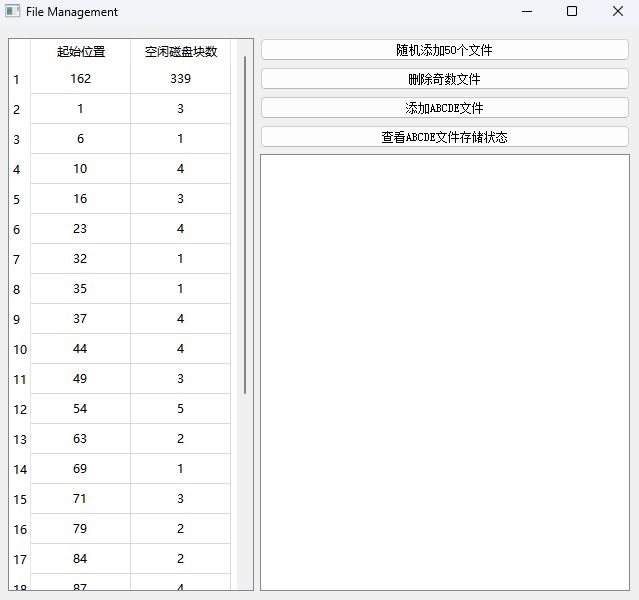
点击“随机添加50个文件”，左侧显示更新后的空闲表。



点击“删除奇数文件”，左侧显示更新后的空闲表。



点击“添加ABCDE文件”，左侧显示更新后的空闲表。



点击“查看ABCDE文件存储状态”，右侧显示各个文件的文件名、存储起始位置和占用磁盘块数。



## 6.5 实验中遇到的问题和解决办法

在本次实验过程中，我遇到的问题是：在“添加ABCED文件”步骤和“查询ABCDE文件状态”步骤之间，需要共享blockList变量和letterStartList变量，而这两个步骤又在不同的方法中。解决方法是将这两个共享变量设置为FreeBlockTable类的成员变量，这样就能在类作用域中共享这两个变量了。