



**实 验 报 告**

课程名称 操作系统

学生学院 材料与能源学院

专业班级 材料成型及控制工程1班

学 号 3117005505

学生姓名 蔡昂儒

指导教师 张伟文

2020 年 5 月 13 日

**实验一 进程调度**

一、实验目的

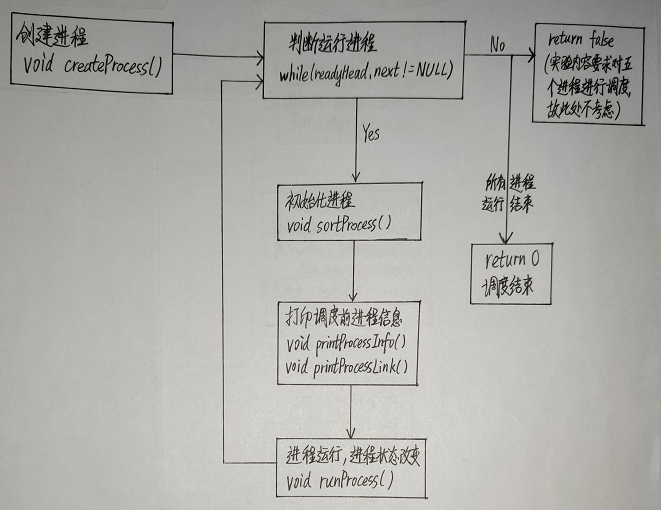
编写并调试一个模拟的进程调度程序，以加深对进程的概念及进程调度算法的理解．

二、实验内容

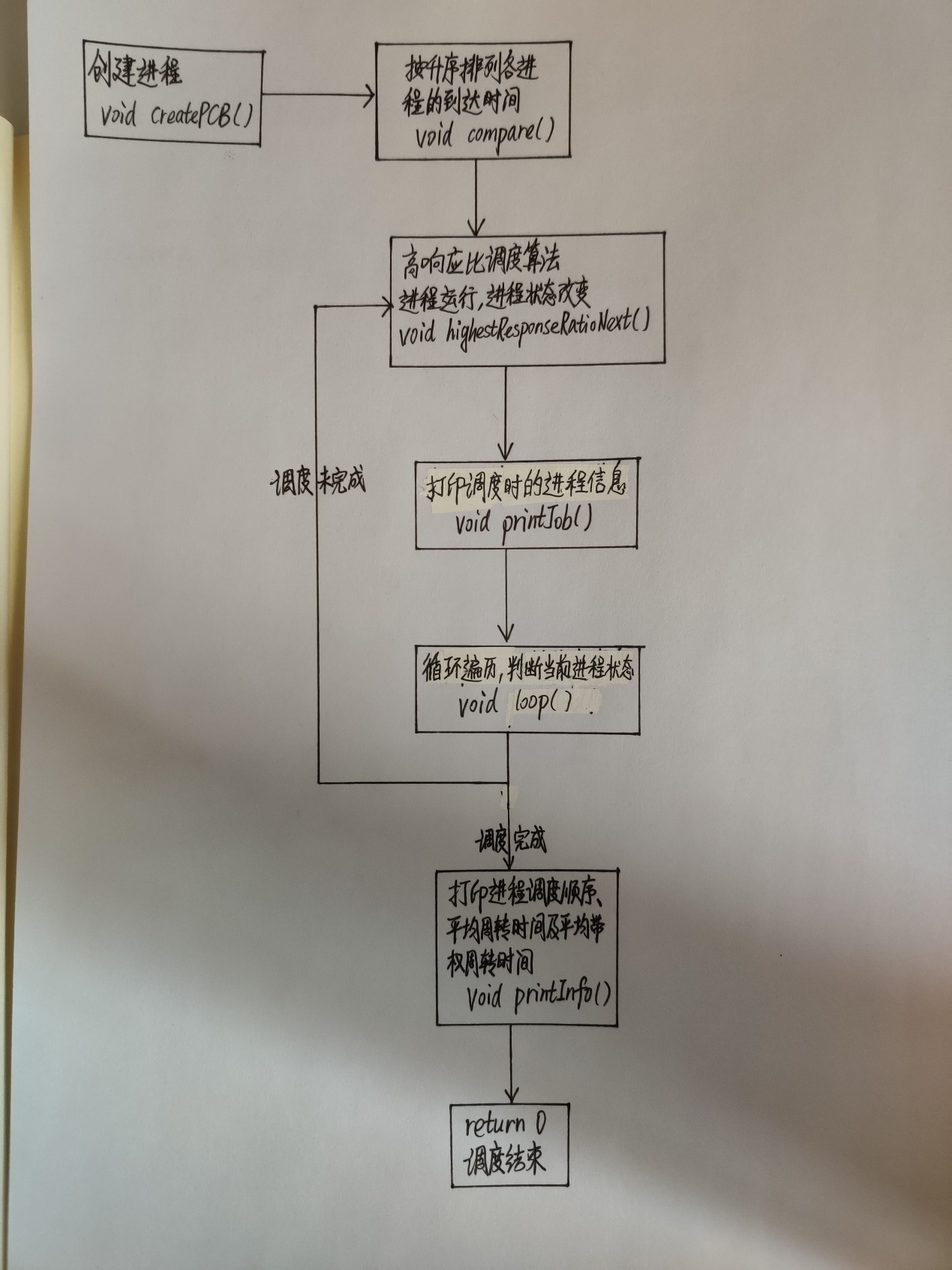
1. 调试运行“动态优先数”和“高响应比”调度算法，给出运行结果。
2. 采用“动态优先数”和“高响应比”调度算法对五个进程进行调度。每个进程有一个进程控制块（ PCB）表示。进程控制块可以包含如下信息：进程名、到达时间、需要运行时间、已用CPU时间、进程状态等等。
3. 每个进程的状态可以是就绪 W（Wait）、运行R（Run）、或完成F（Finish）三种状态之一。 每进行一次调度程序都打印一次运行进程、就绪队列、以及各个进程的 PCB，以便进行检查。重复以上过程，直到所要进程都完成为止。

三、实现思路

动态优先数

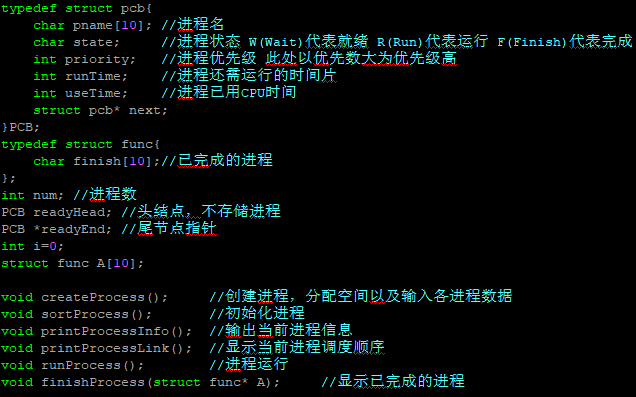


高响应比

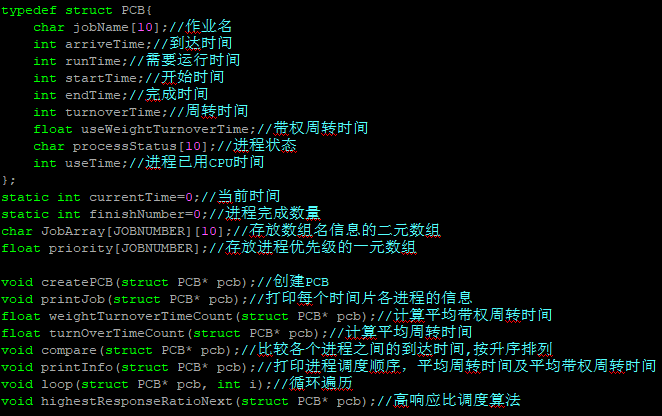


四、主要的数据结构

动态优先数

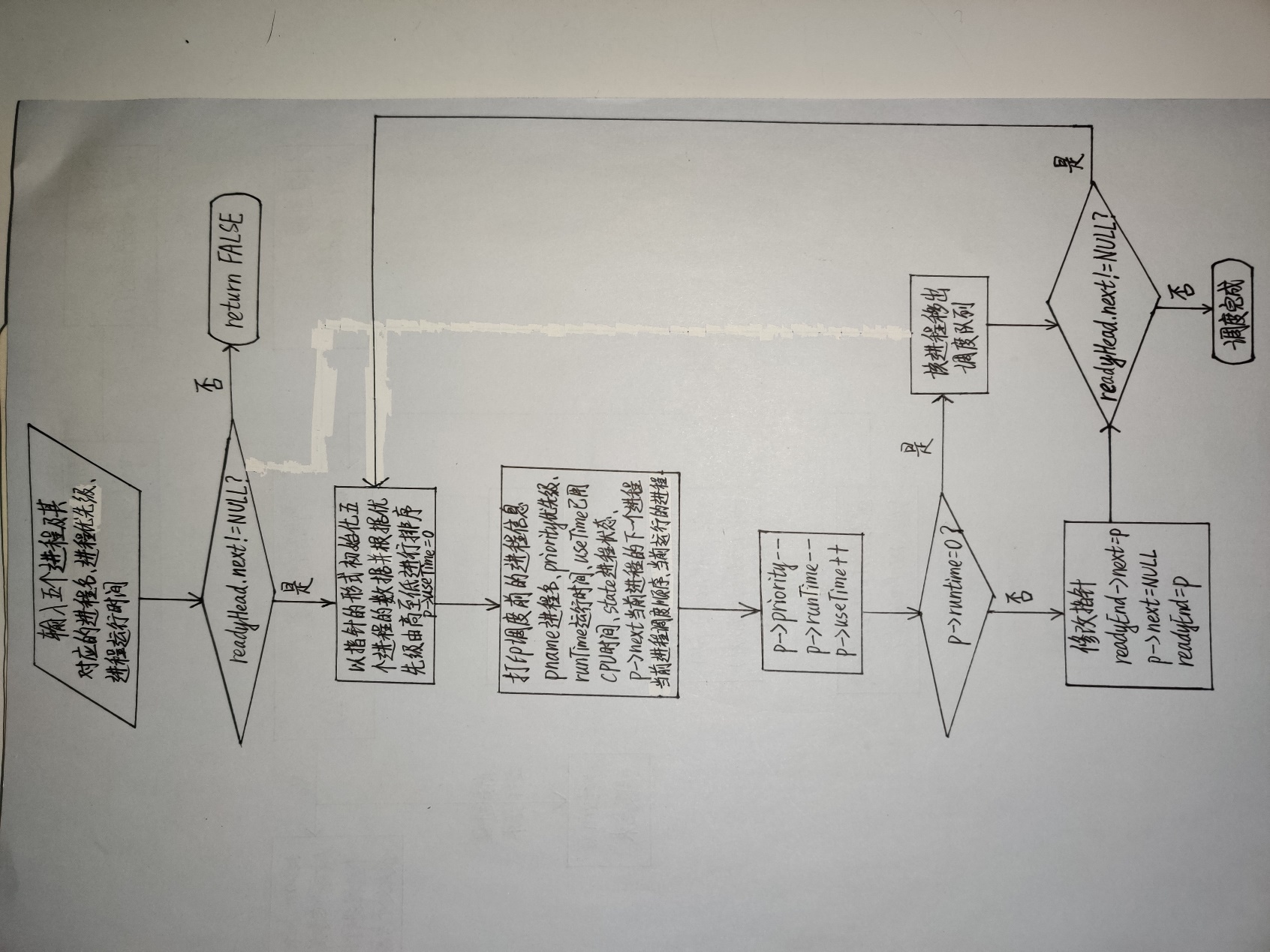


高响应比

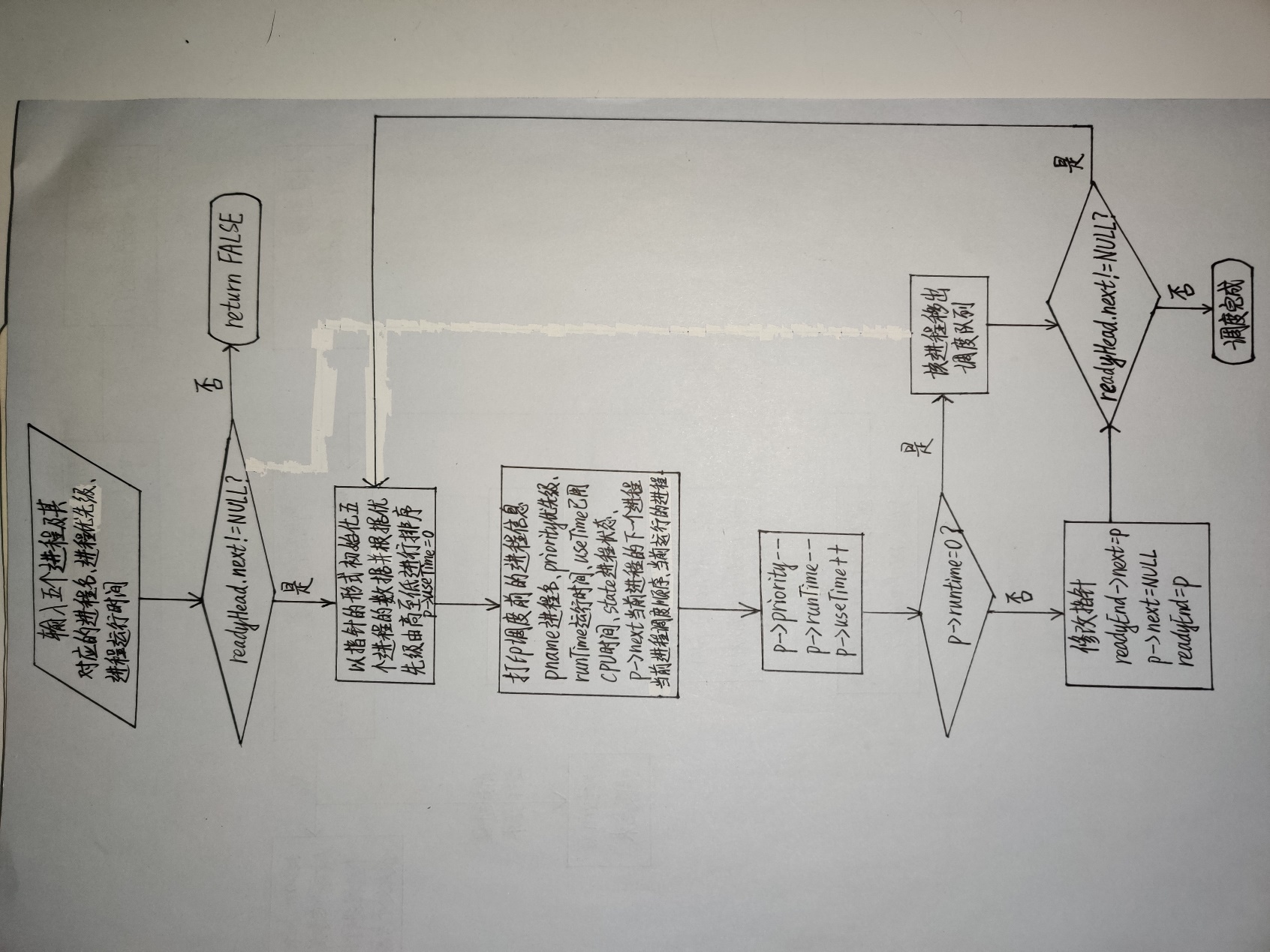


五、算法流程图

动态优先数



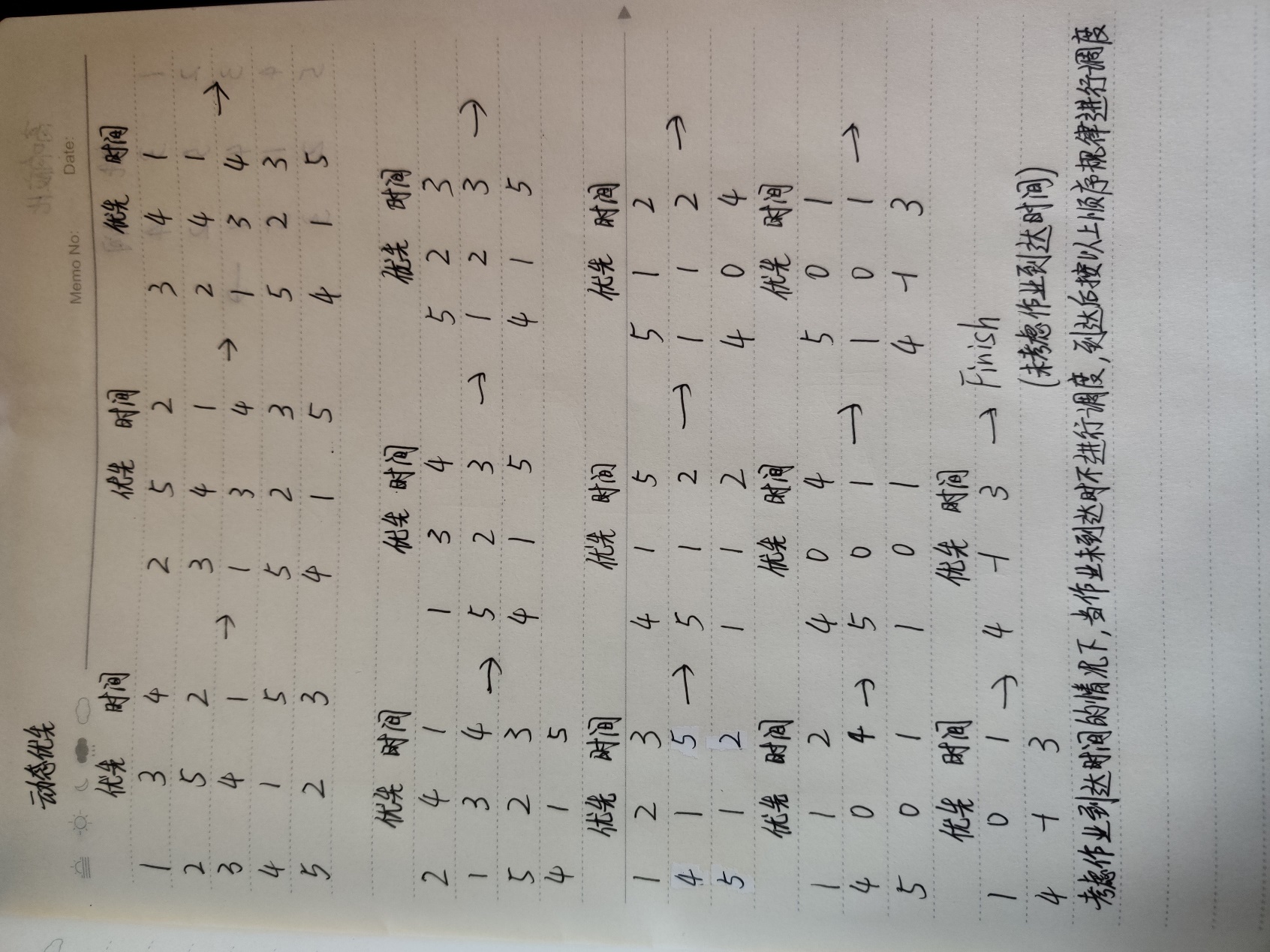
高响应比

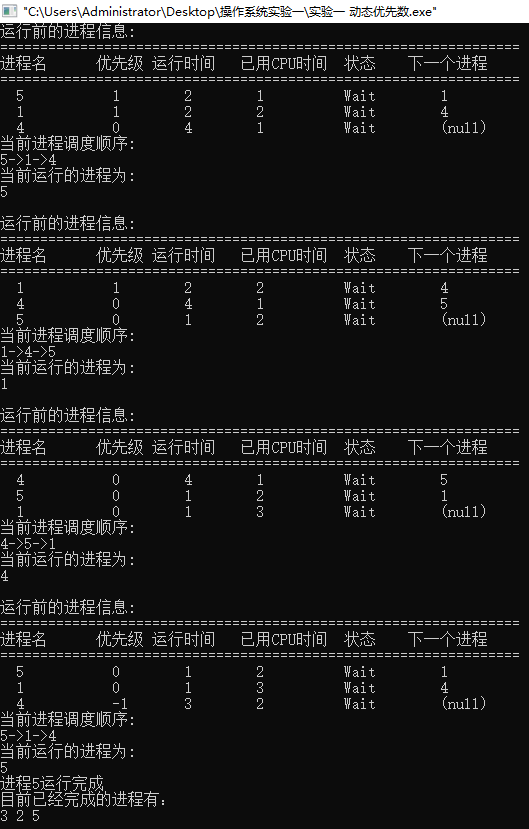
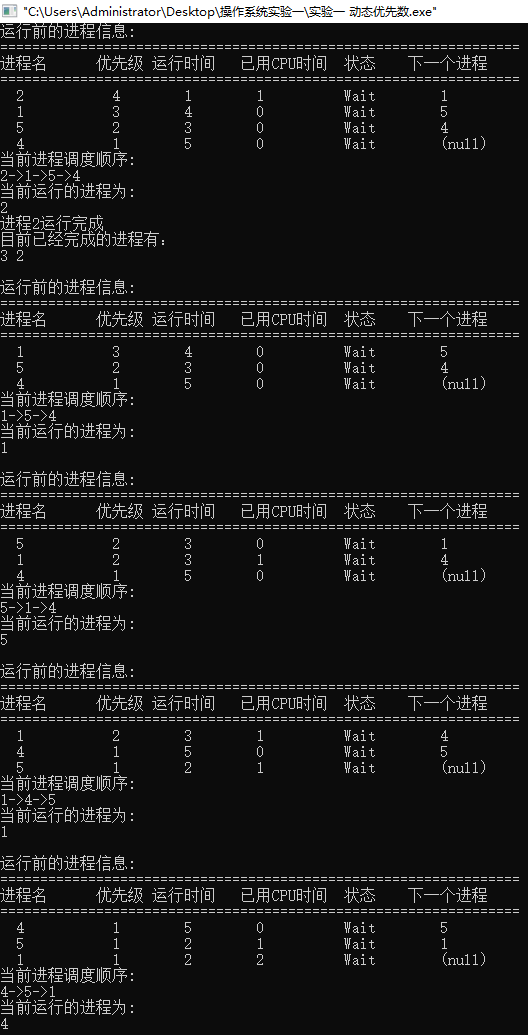
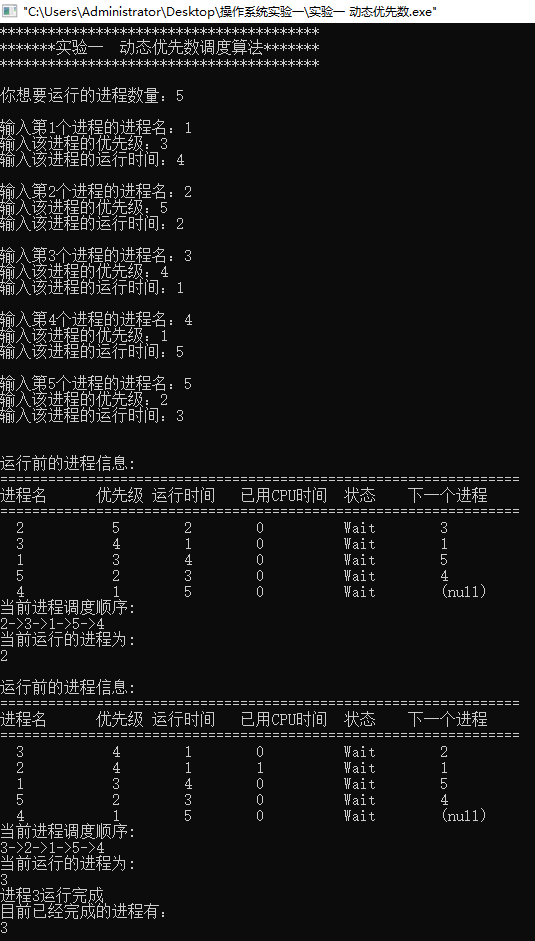


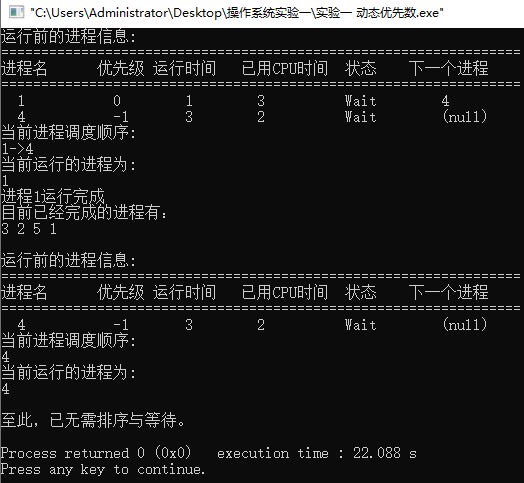
六、运行与测试（系统运行截图）

动态优先数(手算其中一次作为验算)

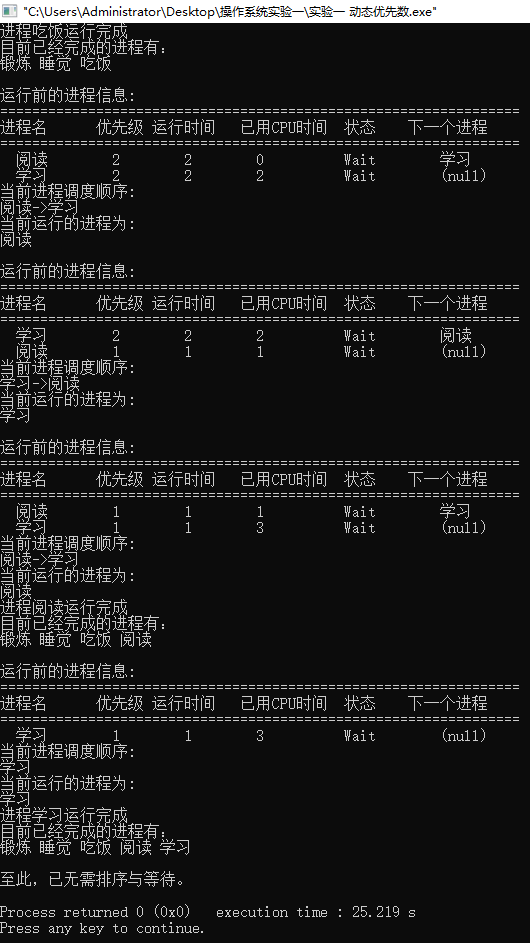
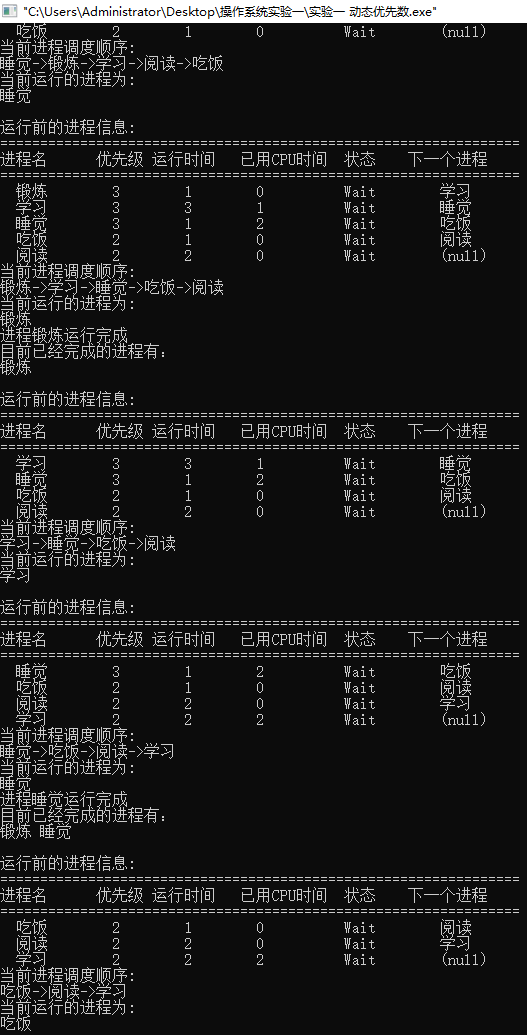
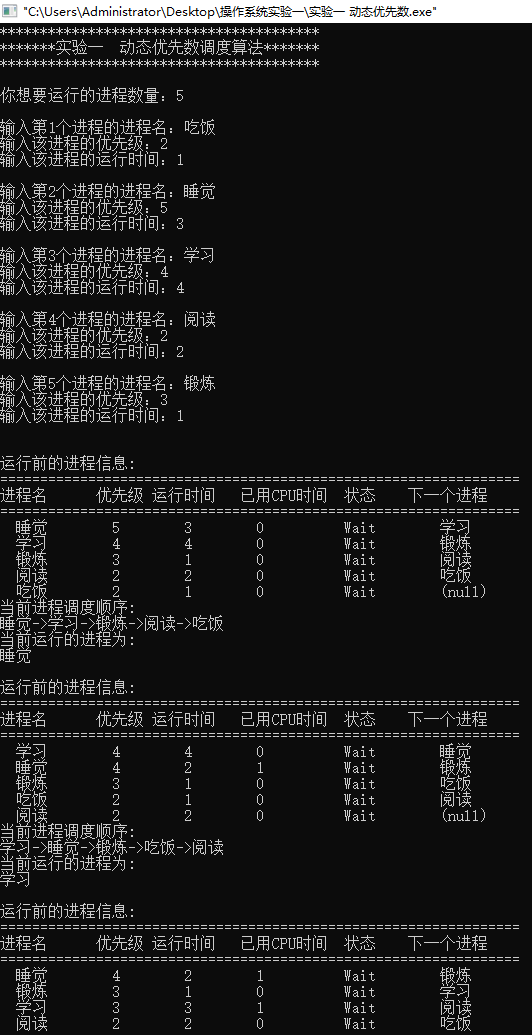
测试1：





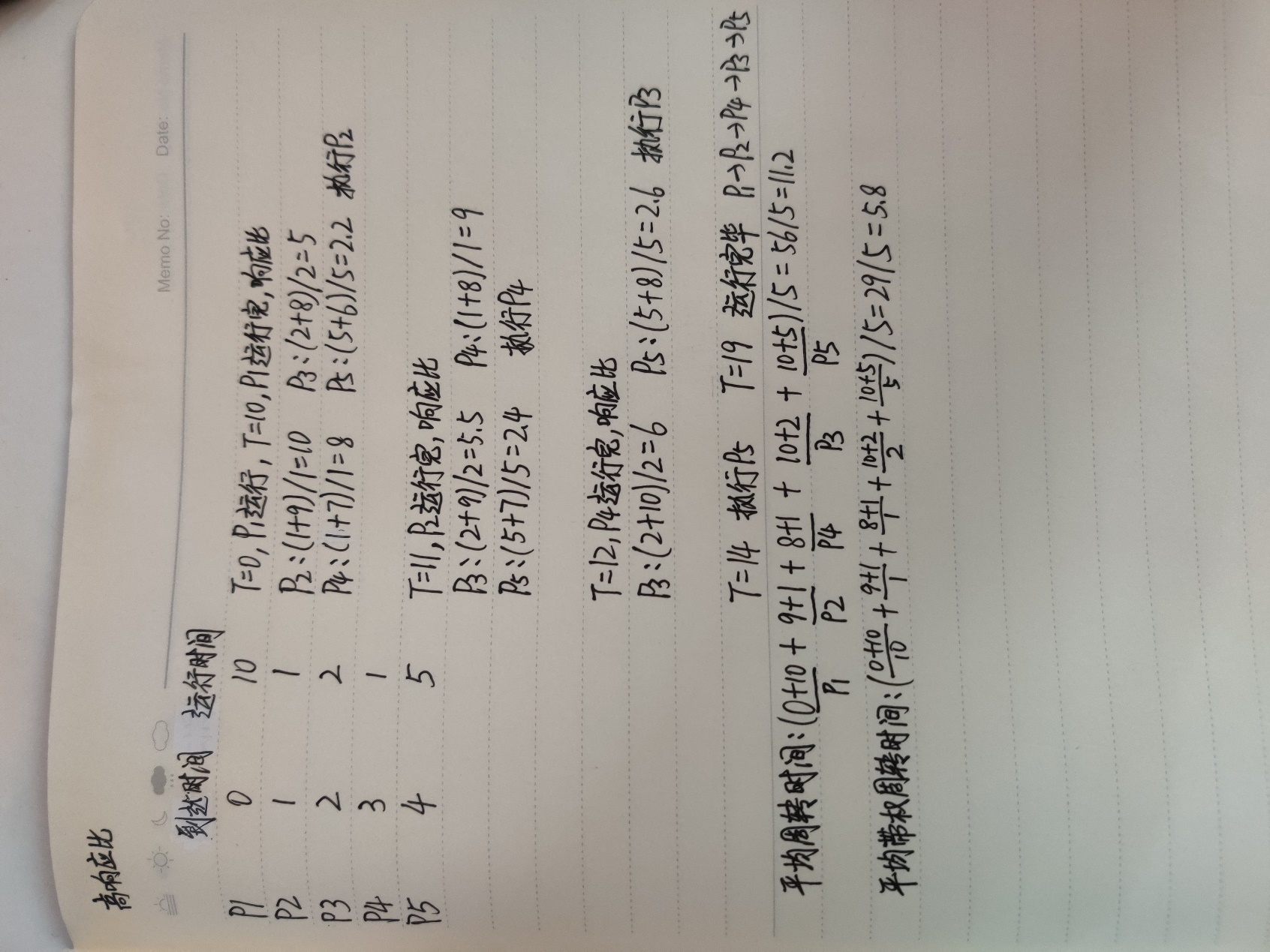


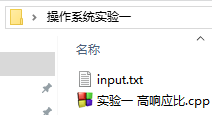
测试2：

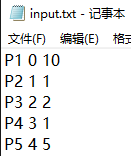


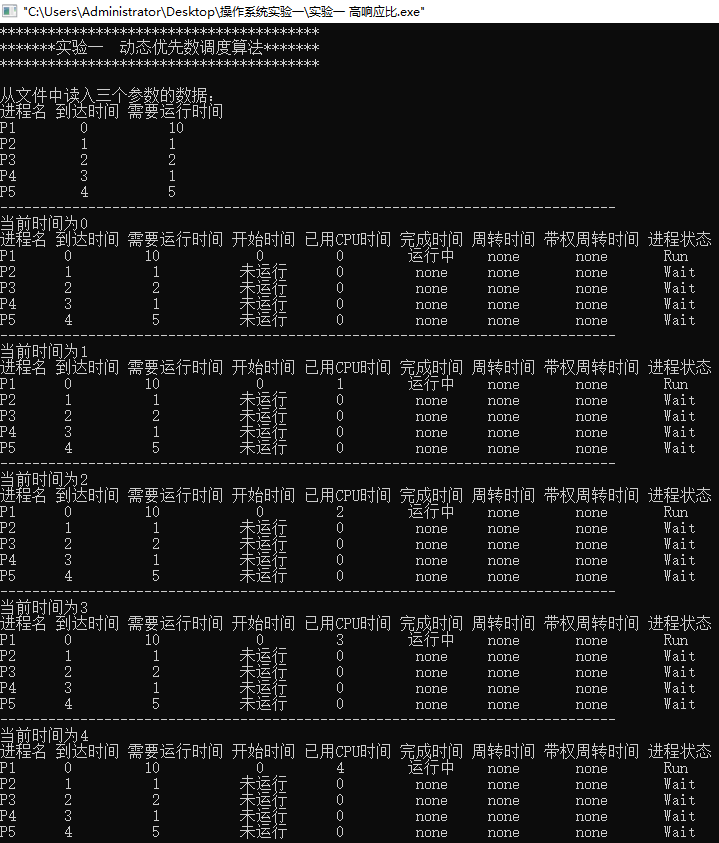
高响应比(手算其中一次作为验算)

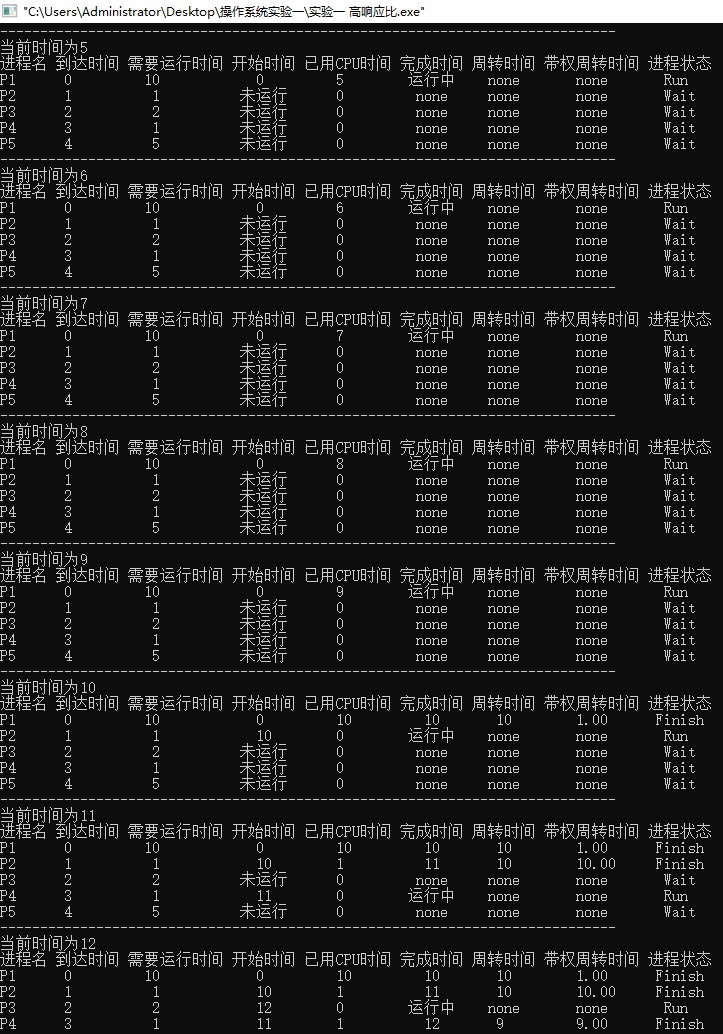
测试1：

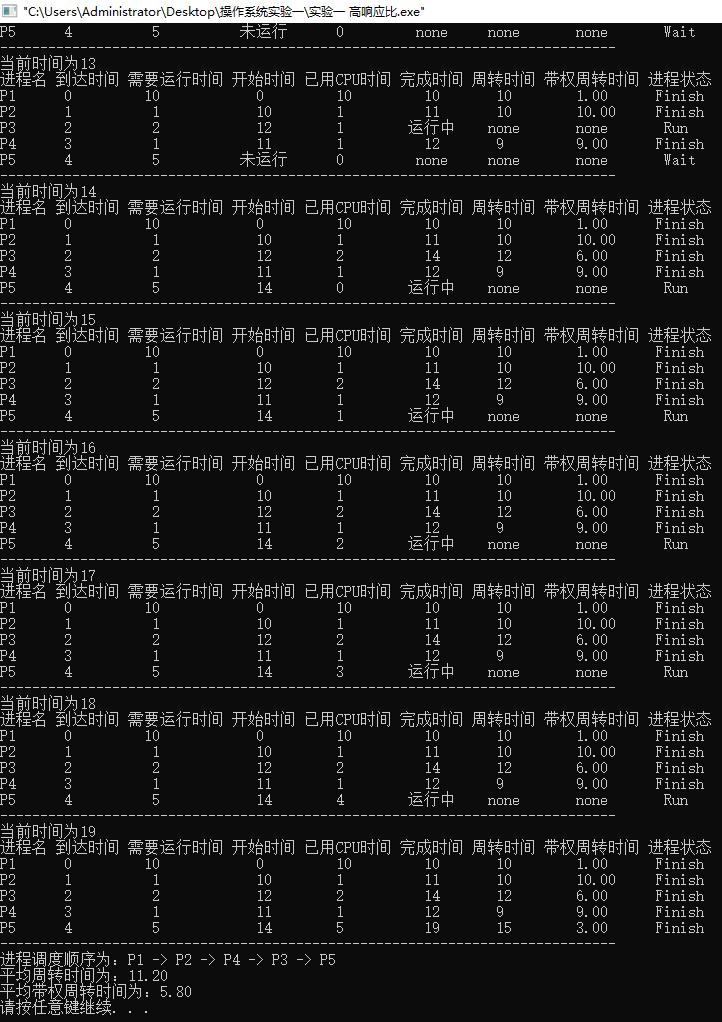








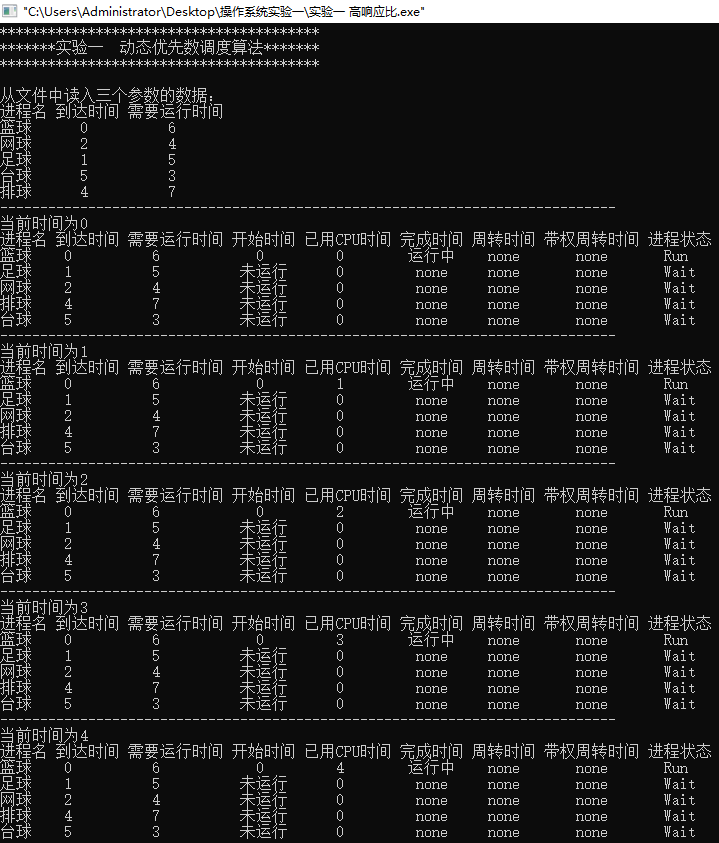


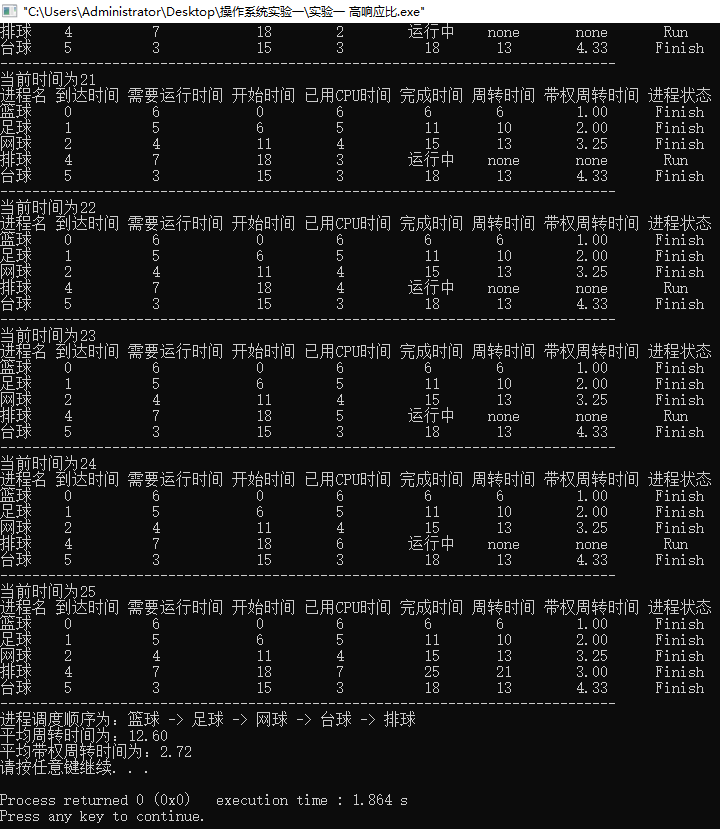
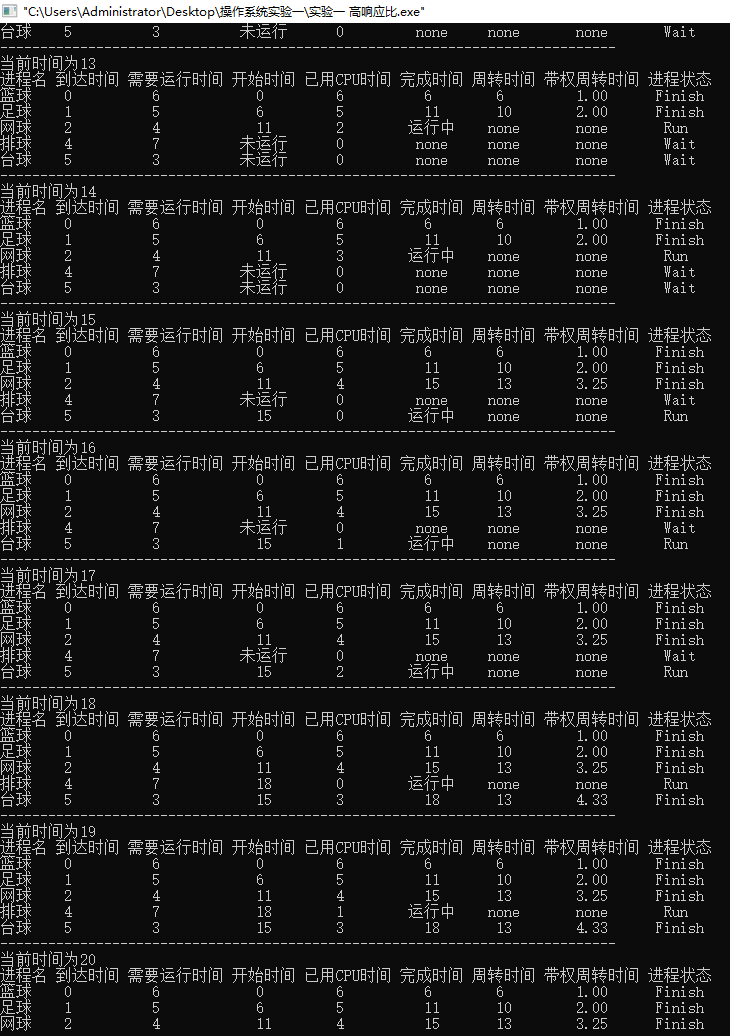
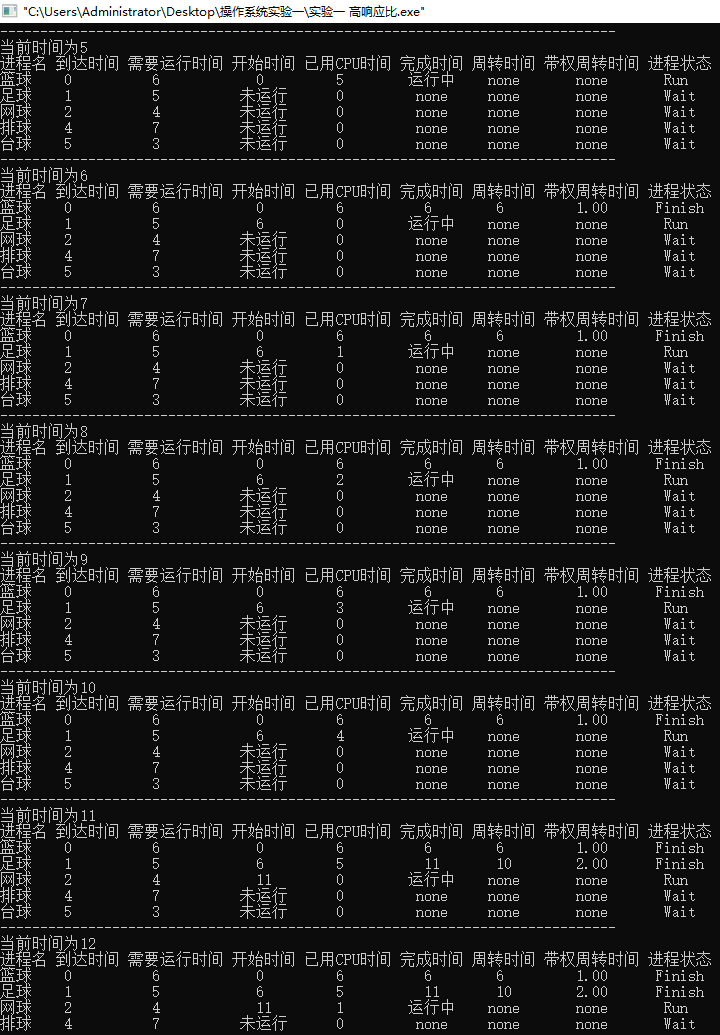




测试2：







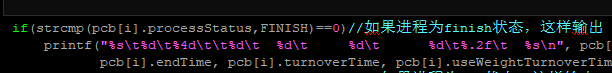
七、总结

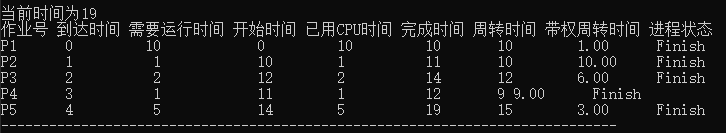
关于动态优先数，从运行的进程数量（实验要求为五个进程）到每个进程对应的进程名、进程优先级与进程运行时间都是在exe执行窗口中手动输入，在每个时间片，程序都会打印出一个当前所有未运行完程序信息的列表，排序为进程优先级降序（从高到低），当有两个或多个进程优先级相同时，相同优先级的进程将会根据程序剩余运行时间进行降序排序，目的是为了保证抢占式动态优先权的实行，使得长作业能够有更多的机会被处理机运行。同时，此代码中规定当前执行进程的优先级以速率β（β取1）下降，以保证多个进程能够不断轮转，从而避免一个长作业长期垄断处理机（PPT 3 处理机调度与死锁 P20）。为了验证

各未完成进程的有序调度，每一次都打印了当前进程调度顺序。这次动态优先数的主要运算是由指针完成的，十分繁杂且难以维护，容易出问题。最初的基本框架只有进程的几个信息，随后在往里面加入更多内容时，出现了一个难以解决的困难，即同时保持执行进程的优先级下降与进程的到达时间会出现很多排序的混乱和指针指向的错误，这一点超出了我的能力范围，经过长时间的代码修改以及个人取舍之后，我最终决定将各个进程的到达时间都设为0，以保证每个时间片的执行进程的优先级下降的同时能够保证调度稳定。在代码后期的修改中，我改进了两个点，一是增加了多个进程优先级相同的情况下根据程序剩余运行时间进行降序排序这一功能，二是打印中每一步显示的已经完成的进程，这一功能我起初犯了一个低级错误，我最初用的是一个普通的字符数组去存储各进程名，这个问题在测试1中没有得以体现，但在测试2中出现了，普通的字符数组每一位都只能存储一个字符，测试1里进程名为1、2、3、4、5时是没有问题的，但是测试2里进程名为吃饭、学习、睡觉、阅读、锻炼时打印就会出现乱码，所以后来代码被修改了，增加了结构体func，该结构体内含有字符数组finish，用于存放已完成进程的进程名，并且定义了结构体数组A，在函数runProcess里通过strcpy将已经运行完成的进程名复制到数组A中，并通过函数finishProcess打印出来。

总的来说，动态优先数是在创建进程时赋予该进程一个初始优先级，然后其优先级随着进程的执行情况的变化而改变，以便获得更好的调度性能。在我的代码中表现为执行中的进程的优先级会减1，这样的好处是能让更多的进程参与到轮转并且被运行，同时各进程被执行完毕的时间差会减小，坏处是有的原本能被更早运行完的进程会因为多次调度轮转而更晚被执行完。

关于高响应比，代码相比起前面的动态优先数有了很大的优化。首先是运用了freopen函数，访问了代码同路径下的文本文档，以及stdin标准输入，读取文本文档里的各进程名、到达时间及需要运行的时间；其次，相比起前面的进程完成后移出调度队列，这次每个单独完成的进程都没有移出队列，并且在进程完成后会即时给出完成时间、周转时间及带权周转时间；同时，进程状态的显示更多了，有Wait（就绪）、Run（运行）和Finish（完成）三种；这次代码也改变了计算方式，避免了大量指针的使用，降低了代码的繁杂度。主函数开头就定义了结构体数组，并且在函数中大量增加了形参，便于数组的引用。在打印函数printJob中，改变了原来p->pname输出的格式，变成了pcb[i].jobName的形式。考虑到所有进程运行完需要较长时间，故在打印中增加显示了每次的时间。这次代码的完成相比起动态优先数也用了更长的时间，在不断测试的过程中，我发现了一件很有趣的事情：关于\t的输出，出现了一个我一直都没想明白的问题：



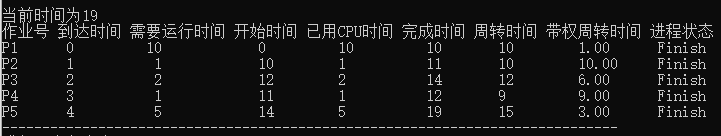


不同于其他四个数据，此处P4的周转时间和带权周转时间的数据中间的\t变为了一个空格，5个数据都是这个Finish语句输出的，但是唯独P4的格式输出有所不同，看起来是周转时间为一位数而其它进程的周转时间为两位数而导致的问题，但是这也不是\t变为单个空格的合理原因，询问了同学，说是这个\t可能刚好在输出是卡在了第8n个单位上(\t为8个空格)，由此导致的问题，我认为有这种可能，但是也不理解其中的原因，随后我在中间多加了一个空格：

C:\Users\Administrator\Documents\Tencent Files\791847873\Image\C2C\0{9RR{VFOTCLU9[3_WLXQPL.png

C:\Users\Administrator\Documents\Tencent Files\791847873\Image\C2C\WQ~NUJ{}@@}FU]_ILF9]ZGI.png

而\t的问题也因此迎刃而解：



总的来说，高响应比优先算法是一个比较综合的调度算法，它既有先来先服务的特点，也有短作业优先的特点，同时，长作业也能得到服务。一方面，先来的进程先服务可以避免时间的浪费，短作业优先可以保证短时间内有更多的进程被运行完，另一方面，长作业的优先级也会随着等待而不断增加优先级，以避免长作业无法被运行的情况，我认为是一个综合性较强的好算法。

实验二 动态分区分配方式的模拟

一、实验目的

了解动态分区分配方式中的数据结构和分配算法，并进一步加深对动态分区存储管理方式及其实现过程的理解

二、实验内容

1. 用C语言分别实现采用首次适应算法和最佳适应算法的动态分区分配过程和回收过程。其中，空闲分区通过空闲分区链（表）来管理；在进行内存分配时，系统优先使用空闲区低端的空间。
2. 假设初始状态下，可用的内存空间为640KB，并有下列的请求序列：

•作业1申请130KB

•作业2申请60KB

•作业3申请100KB

•作业2释放60KB

•作业4申请200KB

•作业3释放100KB

•作业1释放130KB

•作业5申请140KB

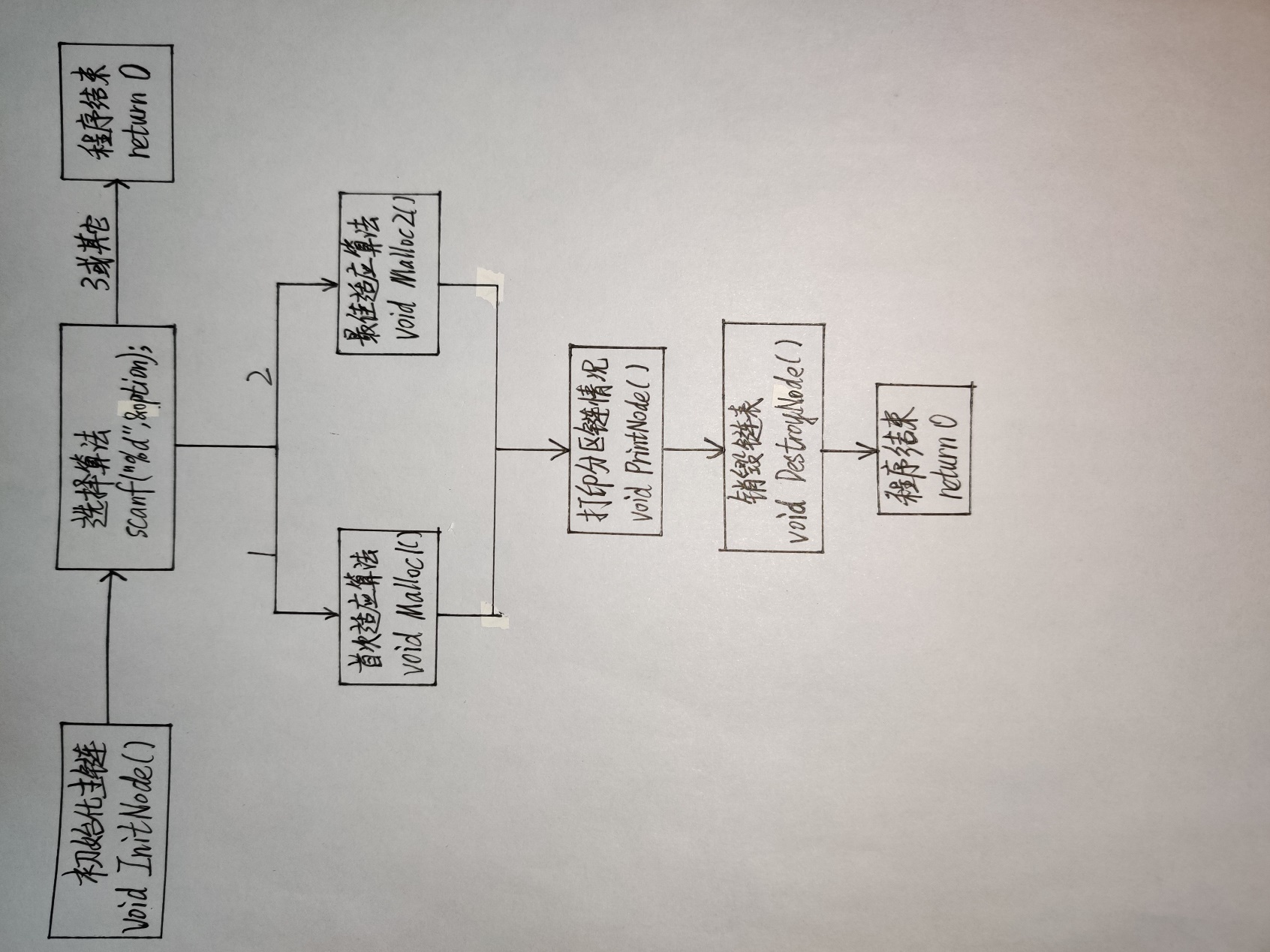
•作业6申请60KB

•作业7申请50KB

•作业8申请60KB

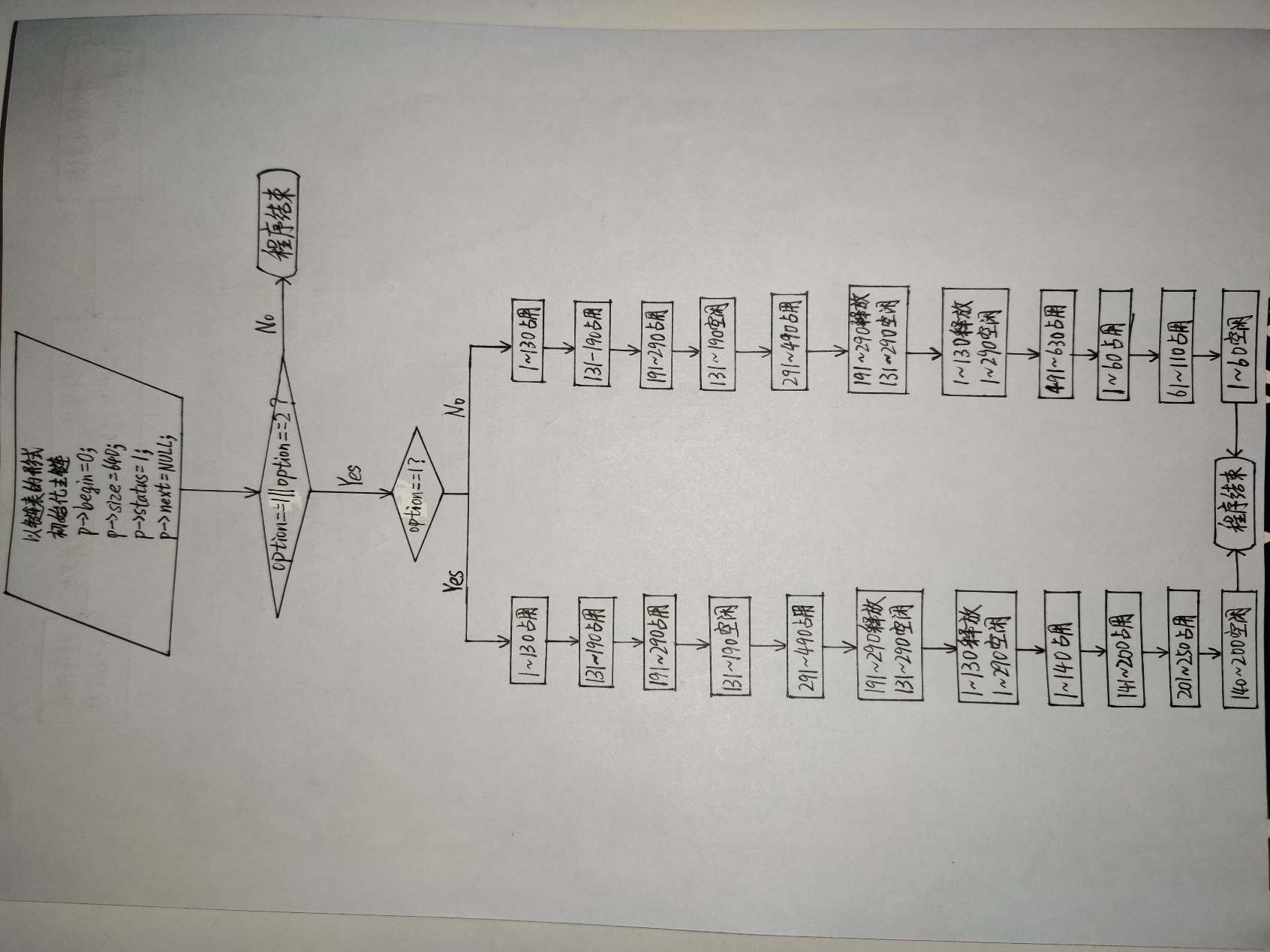
请分别采用首次适应算法和最佳适应算法进行内存的分配和回收，要求每次分配和回收后显示出空闲内存分区链的情况。

三、实现思路

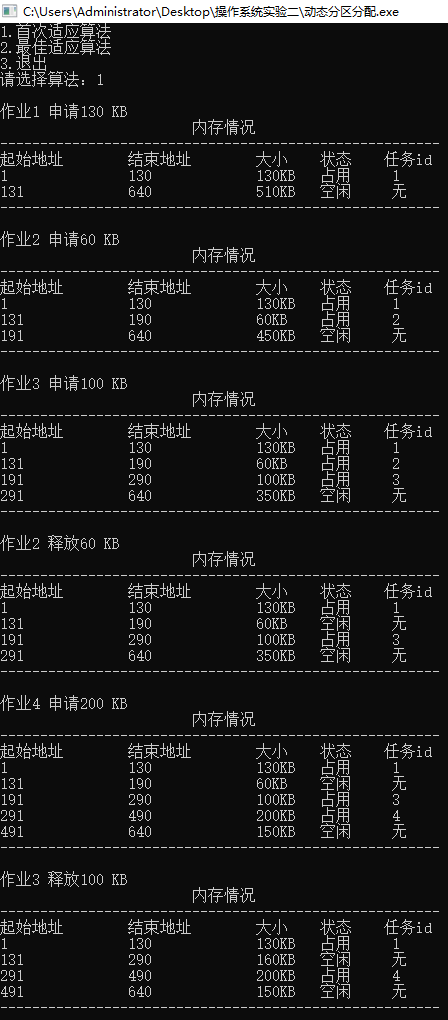


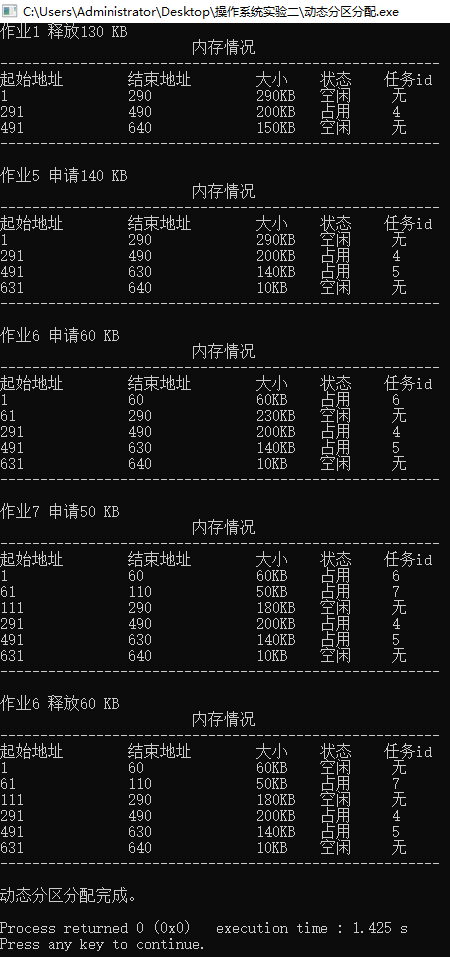
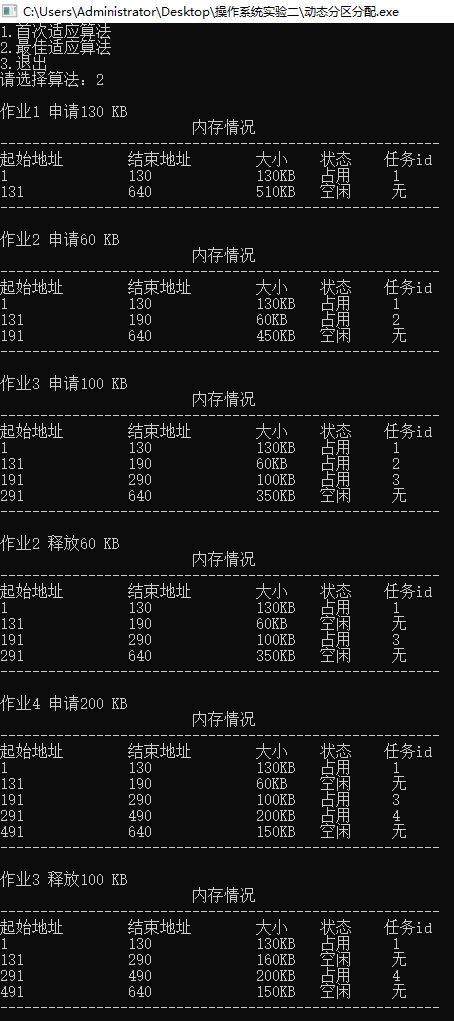
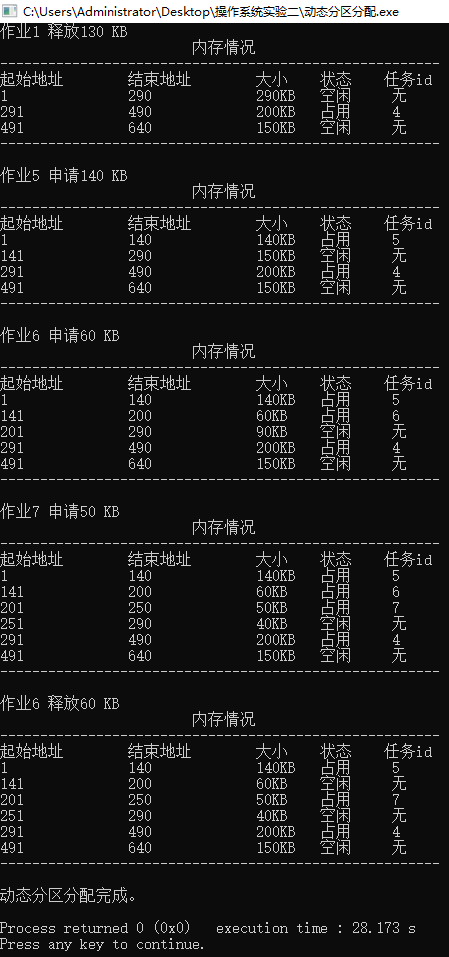
四、主要的数据结构

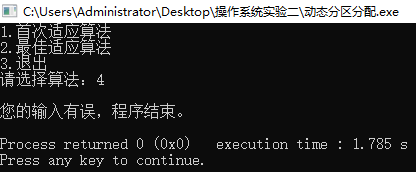
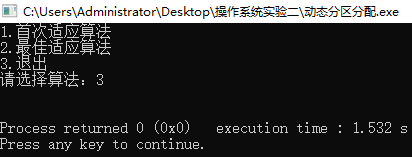


五、算法流程图

六、运行与测试（系统运行截图）



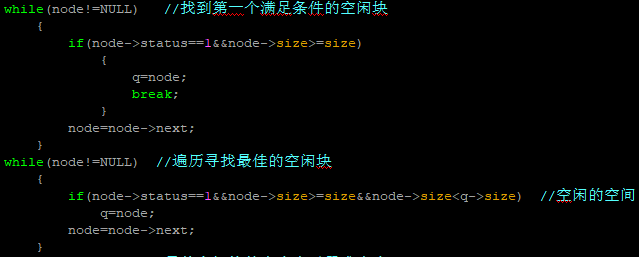




七、总结

关于这两种分配算法的特点，对于首次适应算法，该算法倾向于优先利用内存中低地址部分的空闲区，以便保留高地址部分的大空闲区，这为以后到达的大作业分配大的内存空间创造了条件。首次适应算法的缺点在于为低地址部分不断地划分，会留下许多难以利用的，很小的空闲分区，称为碎片。而每次查找又都是从低地址部分开始的，这会增加查找可用空闲分区时的开销；对于最佳适应算法，该算法是从全部空闲区中找出能满足作业要求的、且大小最小的空闲分区，这种方法能使碎片尽量小。为了适应这种算法，空闲分区表(空闲区链)中的空闲分区要按从小到大进行排序，从表头开始查找到第一个满足要求的自由分区分配。这种算法能够保留大的空闲区，但是会造成许多小的空闲区，这些小的空闲区不能满足大多数进程请求内存的大小，所以会造成内存浪费。这两种分配算法的主要区别在于，首次适应算法容易造成的碎片主要集中在低地址部分，而最佳适应算法容易造成的碎片主要集中在较小的空闲区。从本次实验的实验结果来看，首次适应算法最终剩下3个空闲区，大小分别为60KB、40KB、150KB；最佳适应算法最终也剩下3个空闲区，大小分别为60KB、180KB、10KB。两种算法都得出60KB空闲区的原因在于作业6所占的60KB空间是与其他被占用空间相邻的，最后一步的释放作业6，在两种算法下都会得出60KB的空闲区。相比起首次适应算法，最佳适应算法得出了180KB的大空闲区，更能够满足大作业所需的大内存空间，但是同时也有很难被利用的10KB空闲区，也就是碎片。

从代码来看，首次适应算法更为简单，这种算法自始至终都是从头开始寻找能够满足作业要求的空闲区，但是随着作业量的增加，碎片会逐渐在低地址部分堆积，后面寻找空闲区所需的时间会不断增加。最佳适应算法相比起首次适应算法，主要增加了这一步：



这一步先是要找到第一个满足条件的空闲块，用于保证当前作业不会找不到空闲区，然后再遍历寻找最佳的空闲块，即找出不小于作业所需内存空间的最小空闲区，这种算法起初查找时间通常会高于首次适应算法，但是其查找时间递增程度不会像首次适应算法那么明显。

空闲空间的合并在Free函数中实现，当一个空间被释放之后，会查找当前空间的上一个空间是否同样为空闲，如果是的话，会将这两个空闲的空间合并，以便于满足占用空间大的作业。

本次实验所给的作业请求序列对于两种算法来说都没有出现找不到空闲区的作业的情况，故代码中也没有写入这一情况所应该给出的警告。当遇到更长更复杂的作业请求序列时，可能会出现这种情况。

这次写代码的中途也遇到了问题，问题如下：



虽然前面空间分配的过程和结果都是正确的，但是returned -1073741819而不

是returned 0，说明不是代码语法的错误或逻辑的错误，而是其他的错误。查阅资料后得知这个结果有可能是因为以下几点：1.内存分配未成功，却使用了它。2.内存分配虽然成功，但是尚未初始化就引用它。3.使用free或delete释放了内存后，没有将指针设置未NULL。经过反复检查，我发现我的问题出在第三点，其实这点错误我时常会忽视，一个完整的程序不仅需要代码正确地运行得出正确的结果，也需要在运行结束后正确地释放空间并且修改指针地指向，否则就会出现return的错误。这个问题在我修改了DestroyNode函数之后得以解决，该函数通过while循环依次将node指向的空间释放并最终使node指向NULL。