



**实 验 报 告**

课程名称 操作系统实验

学生学院 计算机学院

专业班级

学 号 3119

学生姓名

指导教师

2021 年 6 月 12 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“时间片轮转”调度算法,给出运行结果。
2. 采用“时间片轮转”调度算法对进程进行调度。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。 显示进程运行过程,以及进程的带权周转时间和系统的平均带权周转时间。

三、实现思路

通过改写动态优先数调度程序实现。添加一个全局变量时间片长度timeslot,当前时间realativeTime,一个额外的数组来存放输入的PCB并排序,根据实时时间和此数组索引来判断进程的运行情况。模拟的初始状态在input()函数结束时的时候就调用operateReady()进行初始化了。改写了主函数,分为常规(就绪队列不为空),当前无正在运行的进程(空闲状态但就绪队列不为空)和当前无正在运行的进程(空闲状态且就绪队列为空)。在running()里加入提前结束的判断。

四、主要的数据结构

int timeSlot=0;

int relativeTime=0;

struct pcb { /\* 定义进程控制块PCB \*/

char name[10];

char state;

int arriveTime=0;

int finishTime;

int needTime;

int runTime;

struct pcb\* link;

}\*ready = NULL, \*processorInRunning=NULL;

typedef struct pcb PCB;

PCB\*\* Array4PCB;

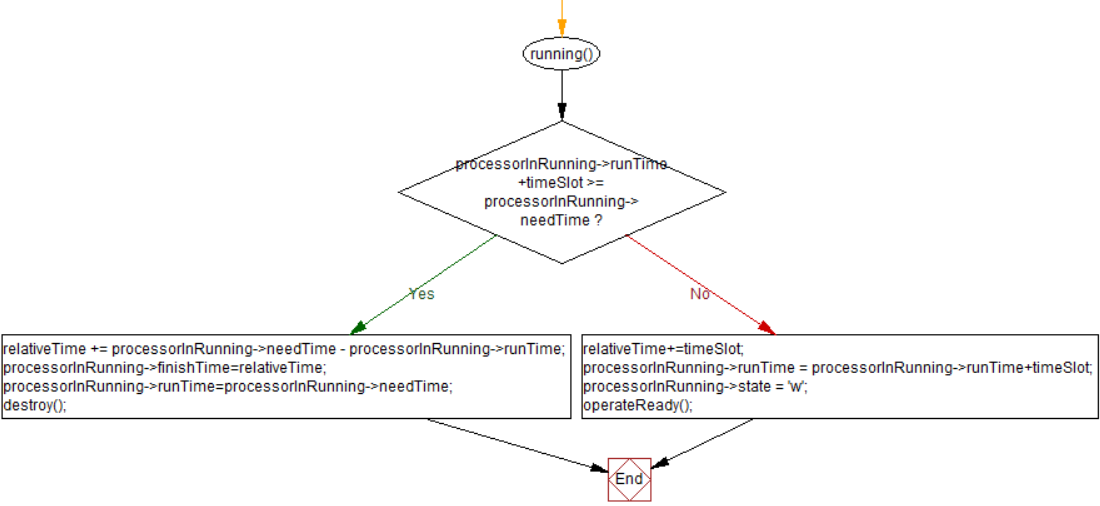
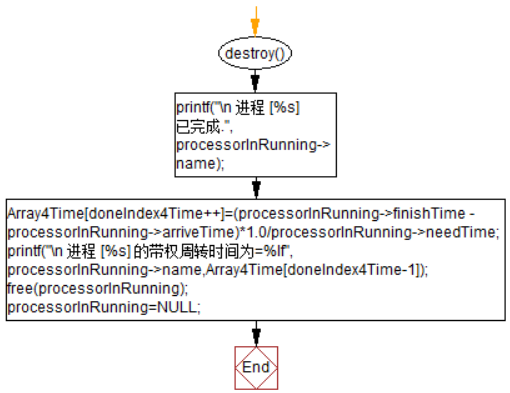
double\* Array4Time;

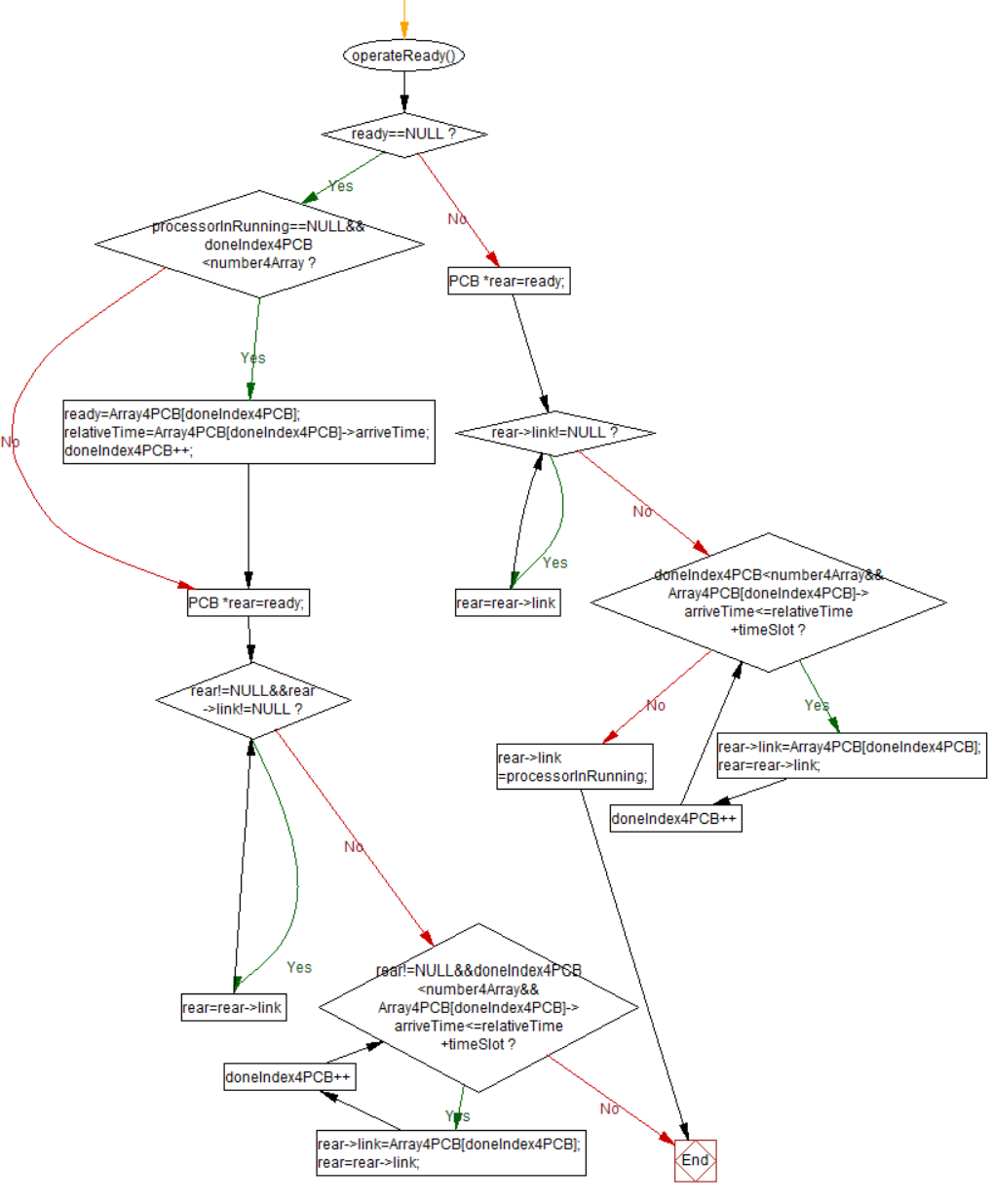
int doneIndex4Time=0;

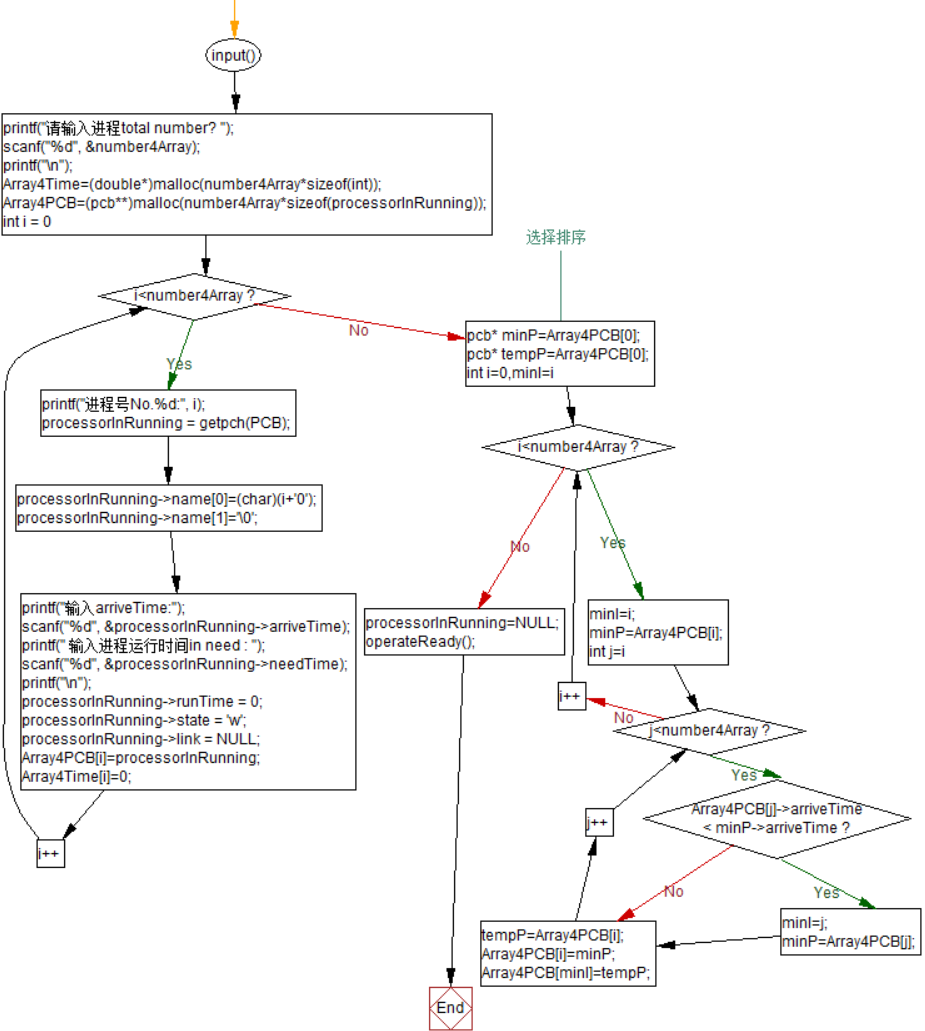
int doneIndex4PCB=0;

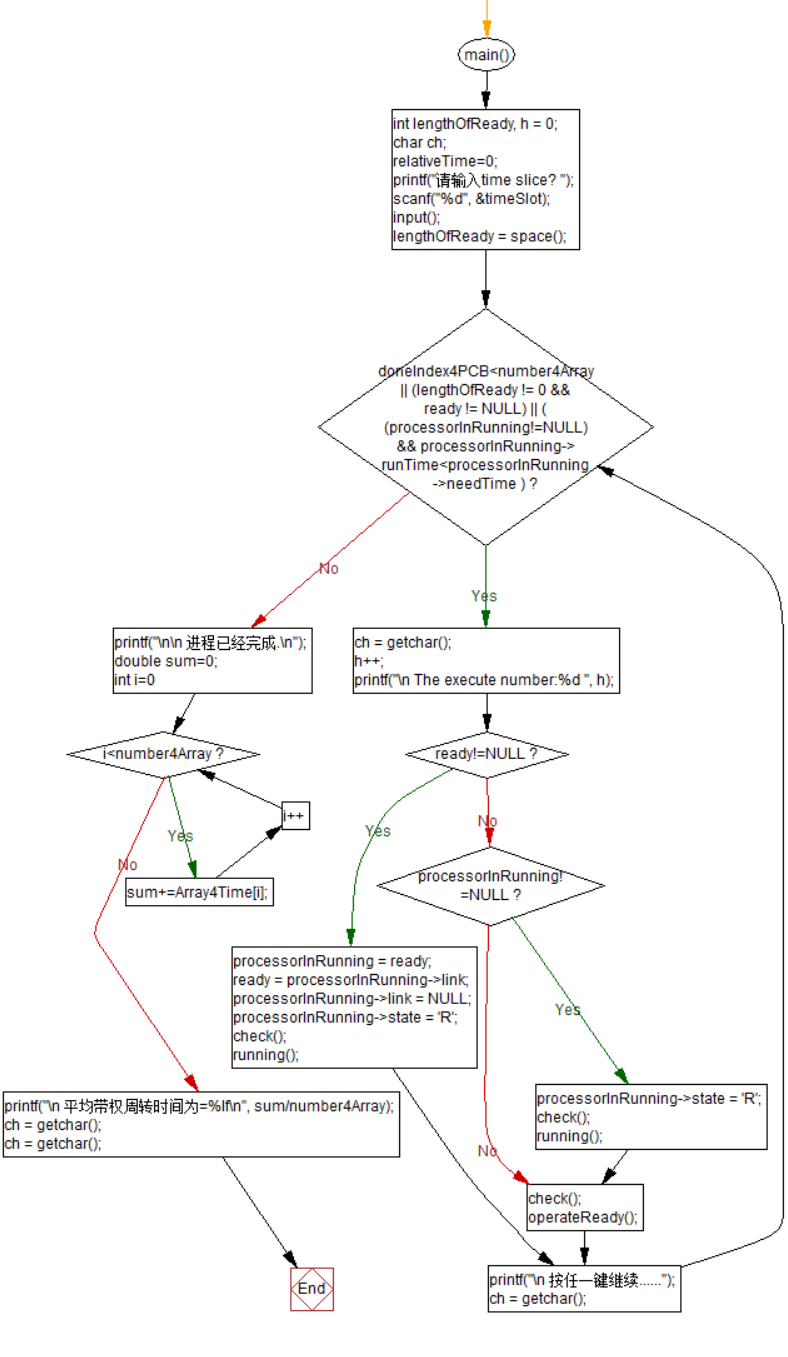
int number4Array=0;

五、算法流程图

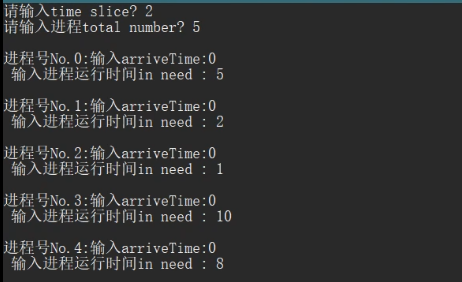


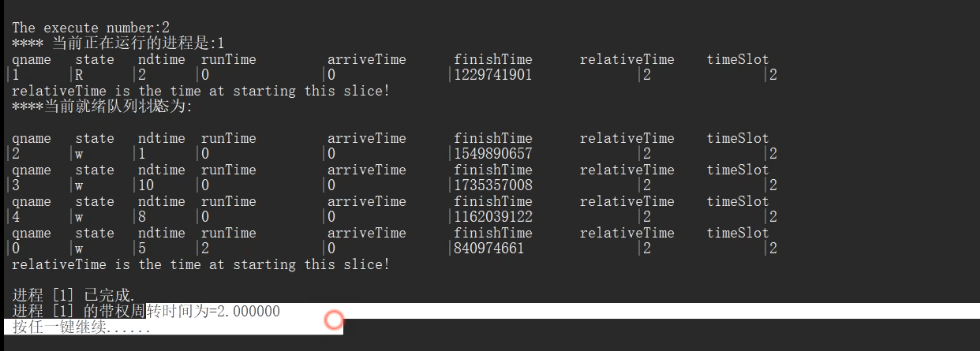


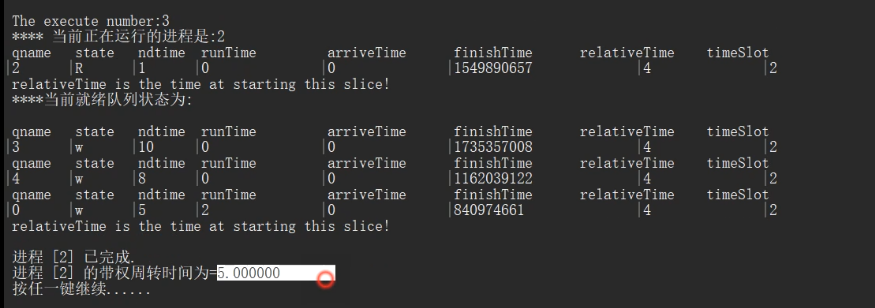


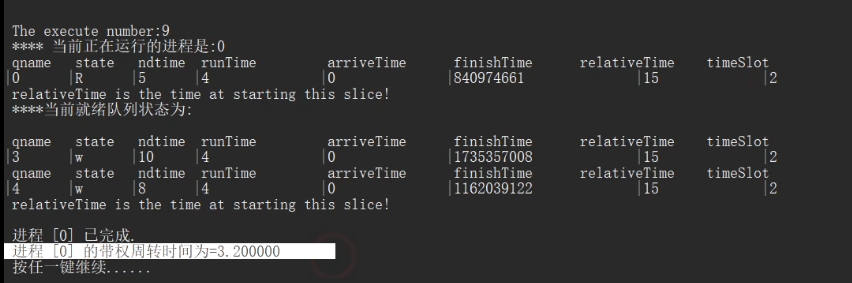


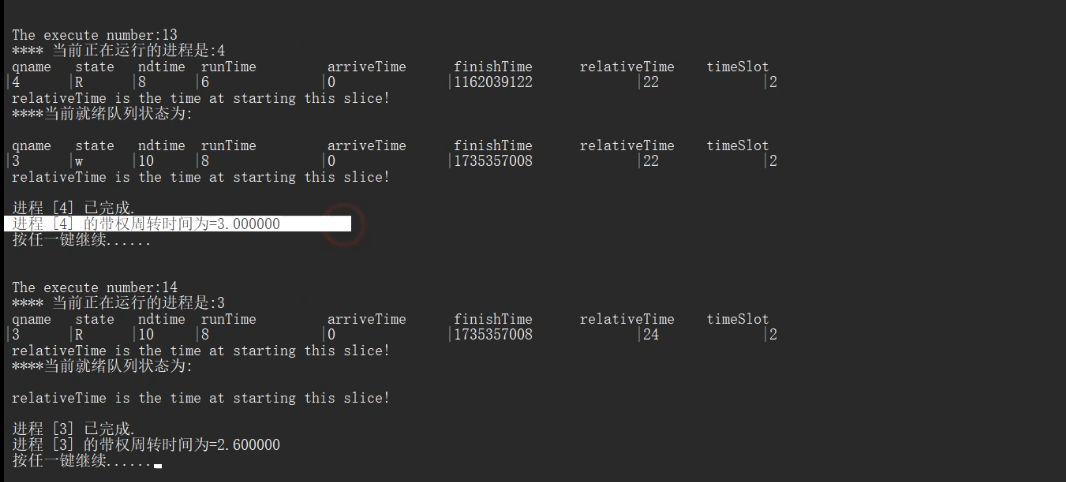
六、运行与测试*(系统运行截图)*

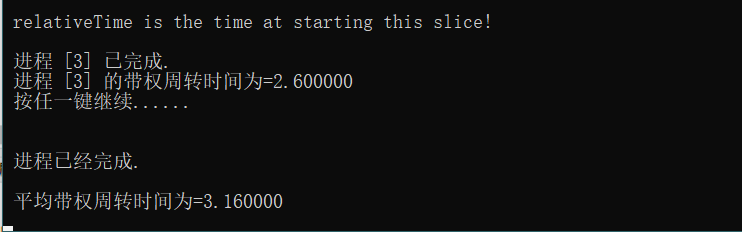












七、总结

此程序是在已有的简陋例程上修改的,因此程序的逻辑较为混乱。测试数据也用得很少。但是总归能较好地模拟出来真实的情况(比如一个pcb完成后下一个pcb仍要几个时间片才能到达的情况)。

还可以进一步优化,比如去掉存放周转时间的数组,改排序为效率更高的排序等。

通过本次实验我对时间片轮转(RR)有了更深的了解:

1,算法思想:公平地,轮流地为各个进程服务,让每个进程在一定时间间隔内都可以得到响应

2,算法规则:按照各个进程到达就绪队列的顺序,轮流让各个进程执行一个时间片(如100ms)如果进程未在一个时间片内执行完,则剥夺处理机,将进程放到就绪队列队尾重新排队

3,用于作业/进程调度:用于进程调度(只有专业放到内存中建立相应的进程后,才能分配处理机时间片)

4,是否可抢占:如果进程未在时间片 内运行完,将强行剥夺处理机的使用权,因此时间片轮转调度算法属于抢占式。由时钟装置发出时钟中断来通知CPU时间片已经到了

5,优缺点:

优点:公平,响应快,适用于分时操作系统

缺点:由于高频率的进程切换,因此会有一定的开销,不区分任务的紧急程度

6,是否导致饥饿:不会

7,补充:时间片分配太大就 退化成先来先服务算法,太小用于进程切换的开销就太大了

**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

用一个全局链表将空闲分区和已分配分配的分区连接在一起.

首次适应和最佳适应实现起来较为简单,遍历链表即可.分配完一个已被占用的分区后allocate()再根据用之前空闲分区”减去”此分区再得出一个空闲分区

空闲分区的回收可以根据前后有无紧邻的空闲分区划分.

四、主要的数据结构

struct partitionNode{

int processID;

int status;

int startingAddress;

int size;

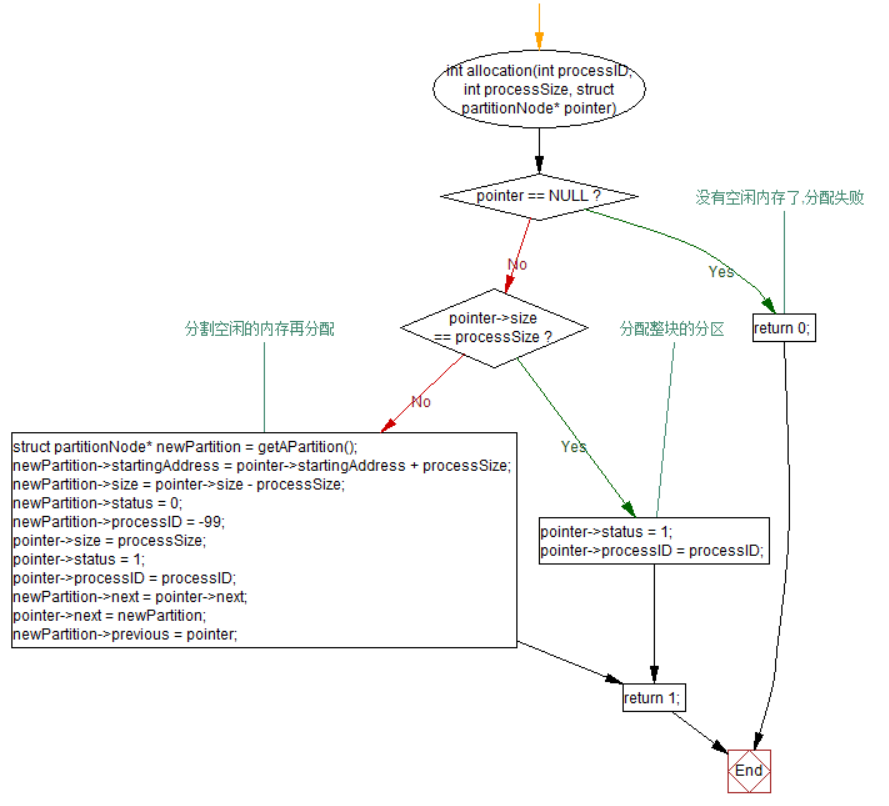
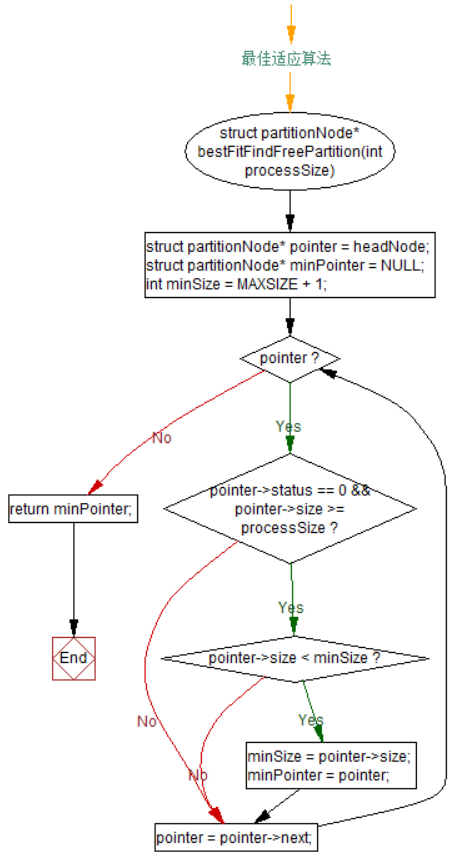
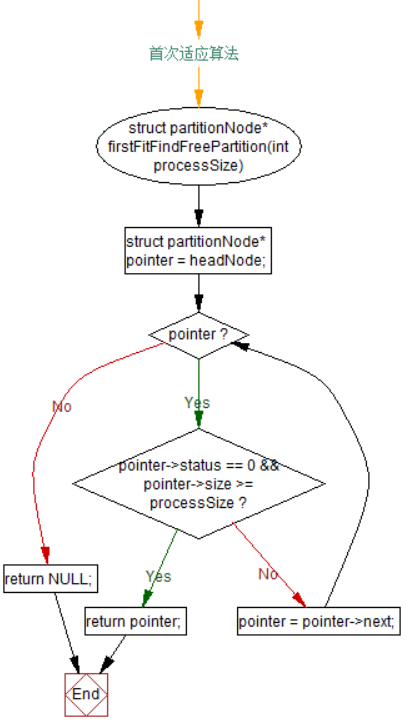
struct partitionNode\* previous;

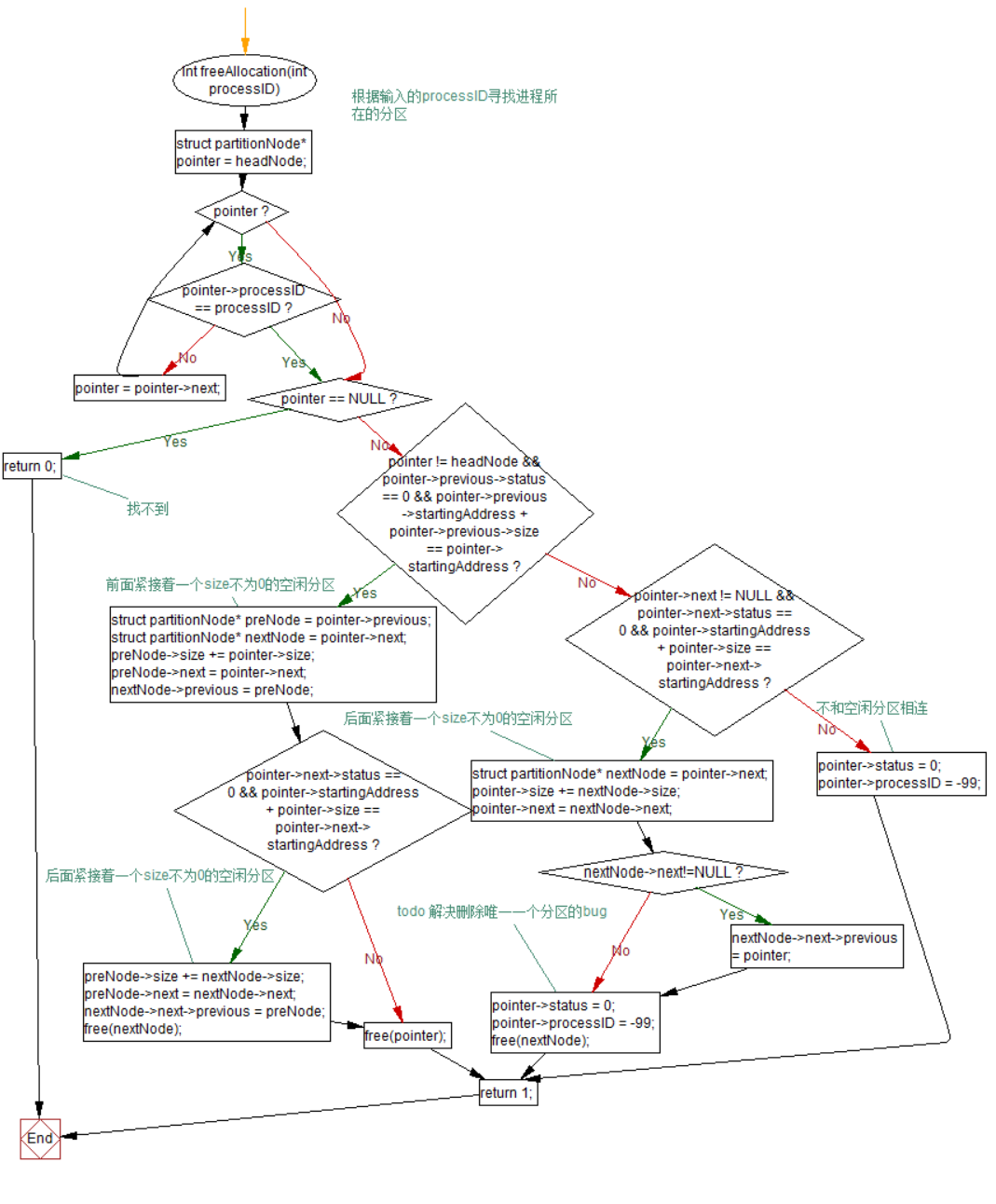
struct partitionNode\* next;

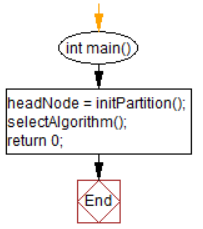
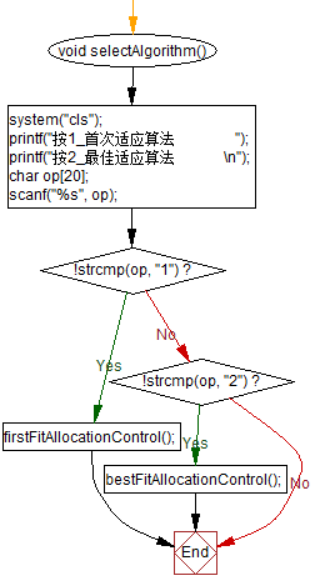
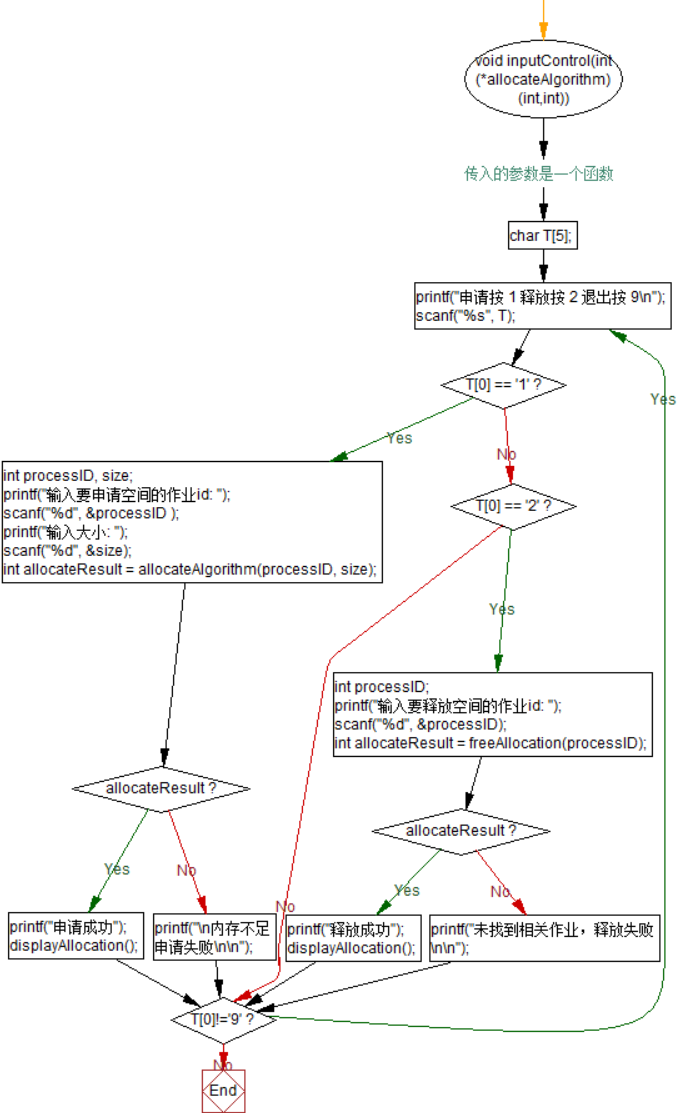
};

struct partitionNode\* headNode;

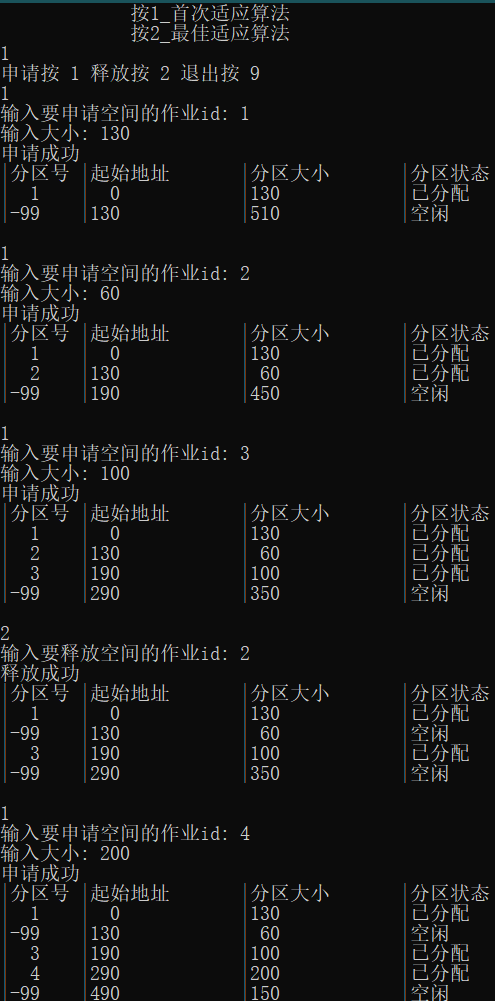
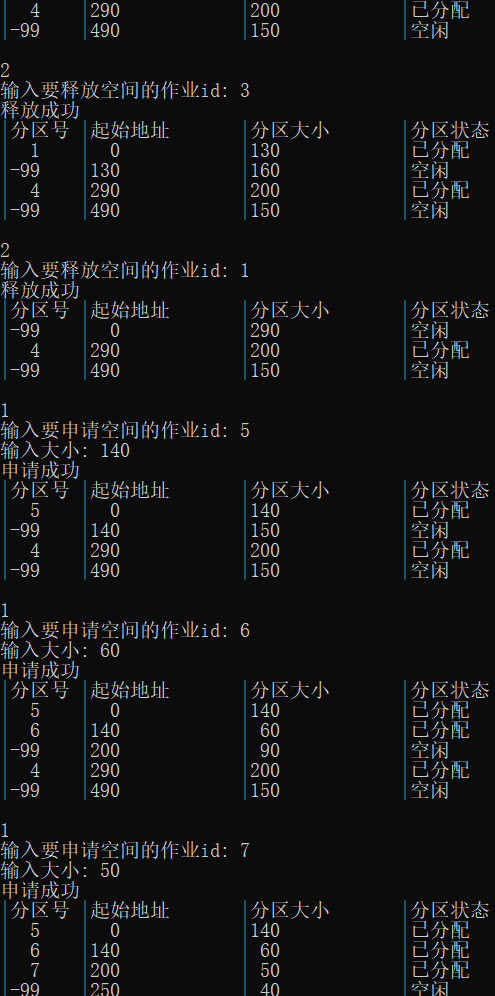
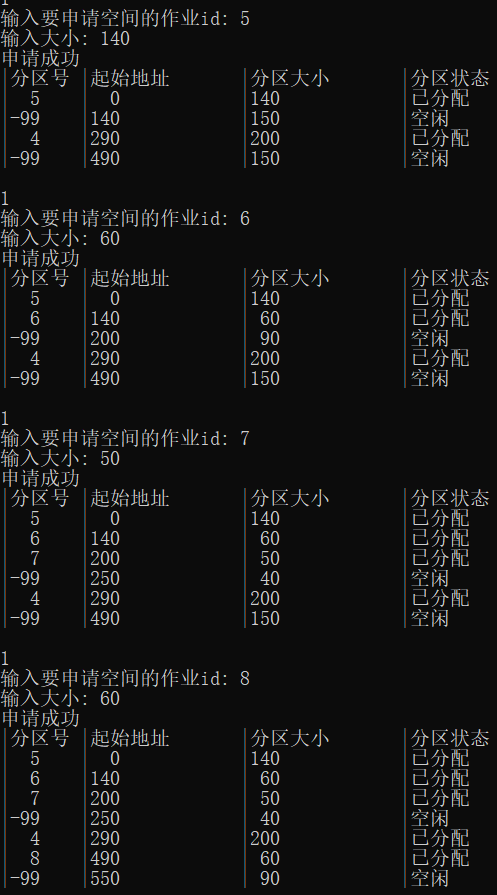
五、算法流程图



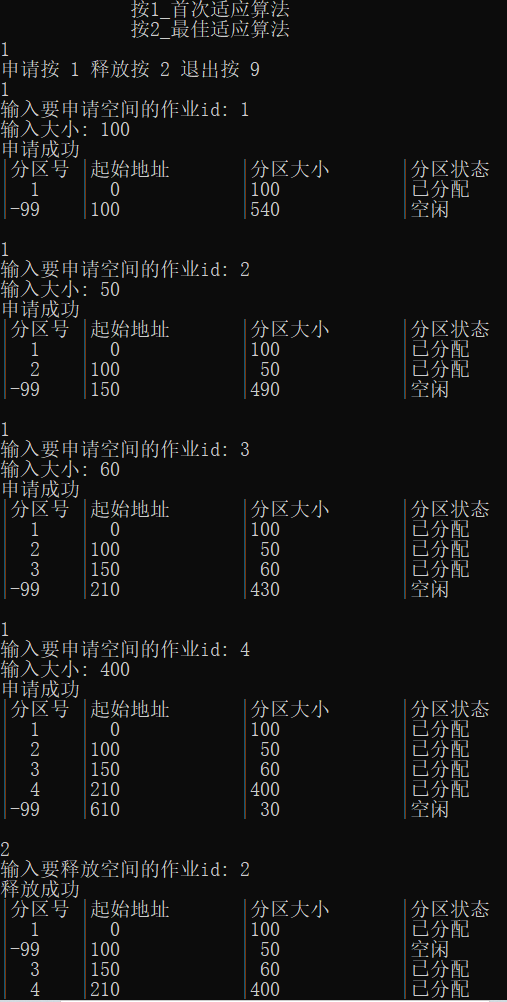
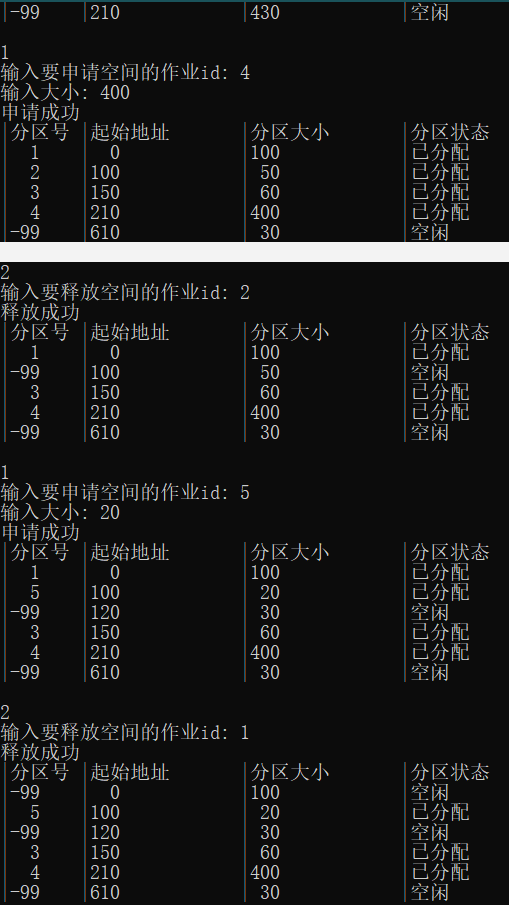




六、运行与测试 最佳适应算法:

首次适应算法:

七、总结

可以直接将一个函数的返回值当作参数输入另外一个函数.

可以利用函数指针实现模块化的过程聚合,达到复用inputControl()中的输入输出的目的.

通过这次实验,我体会到了以下两种算法的优缺点:

首次适应算法(从链首开始,查找可以用的空闲区)

优点:保留了高地址部分的大空闲区

缺点:低地址部分不断被划分,留下许多小的、很难利用的空闲区

最佳适应算法(每次查找时,将刚刚好的、稍微大一点点的空闲区分给作业)

优点:避免了“大材小用”

缺点:留下许多难以利用的小空闲区

**实验三 请求调页存储管理方式的模拟**

**一、实验目的**

**通过对页面、页表、地址转换和页面置换过程的模拟,加深对请求调页系统的原理和实现过程的理解。**

**二、实验内容**

**(1)假设每个页面中可存放10条指令,分配给作业的内存块数为4。**

**(2)用c语言模拟一个作业的执行过程,该作业共有85条指令,即它的地址空间为9页,目前它的所有页都还未调入内存。在模拟过程中,如果所访问的指令已在内存,则显示其物理地址,并转下一条指令。如果所访问的指令还未装入内存,则发生缺页,此时需记录缺页的次数,并将相应页调入内存。如果4个内存块均已装入该作业,则需进行页面置换,最后显示其物理地址,并转下一条指令。在所有85指令执行完毕后,请计算并显示作业运行过程中发生的缺页率。**

**(3)置换算法:采用先进先出(FIFO)、最近最久未使用(LRU)和最佳置换(OPT)算法置换算法。**

三、实现思路

声明主存块结构体,里面有首次到达的指令索引号和最后使用的指令索引号.

声明一个全局主存块链表,根据输入的数字决定块数,利用取余操作实现链表首尾相连.

**初始化函数initializeInstruction()通过随机数产生一个指令序列.**

**1)指令的地址按下述原则生成:**

**① 50%的指令是顺序执行的；**

**② 25%的指令是均匀分布在前地址部分；**

**③ 25%的指令是均匀分布在后地址部分；**

**具体的实施方法是:**

**① 在[0,319]的指令地址之间随机选取一起点m；**

**② 顺序执行一条指令,即执行序号为m+1的指令；**

**③ 在前地址[0,m-1]中随机选取一条指令并执行,该指令的序号为m1；**

**④ 顺序执行一条指令,其序号为m1+1的指令；**

**⑤ 在后地址[m1+2,319]中随机选取一条指令并执行,该指令的序号为m2；**

**⑥ 顺序执行一条指令,其序号为m2+1的指令；**

**重复上述步骤①～⑥,直到执行320次指令。**

**2)将指令序列变换为页地址流**

**声明一个commonPack()函数,指令进入主存后不用置换的情形下可以复用.**

四、主要的数据结构

#define MAXSIZE 85

int pageFaultCount = 0; //缺页次数

int instructionArray[MAXSIZE]; //指令地址流数组

int pageArray[MAXSIZE]; //页地址流数组

typedef struct Block{

int blockID; //主存物理块号

int pageID; //块中装进的虚存页号,初始值为-1

int firstInIndex;//首次调入内存的指令index,初始值为MAXSIZE+1

int leastRecentlyUsedIndex;//最后一次使用的指令index

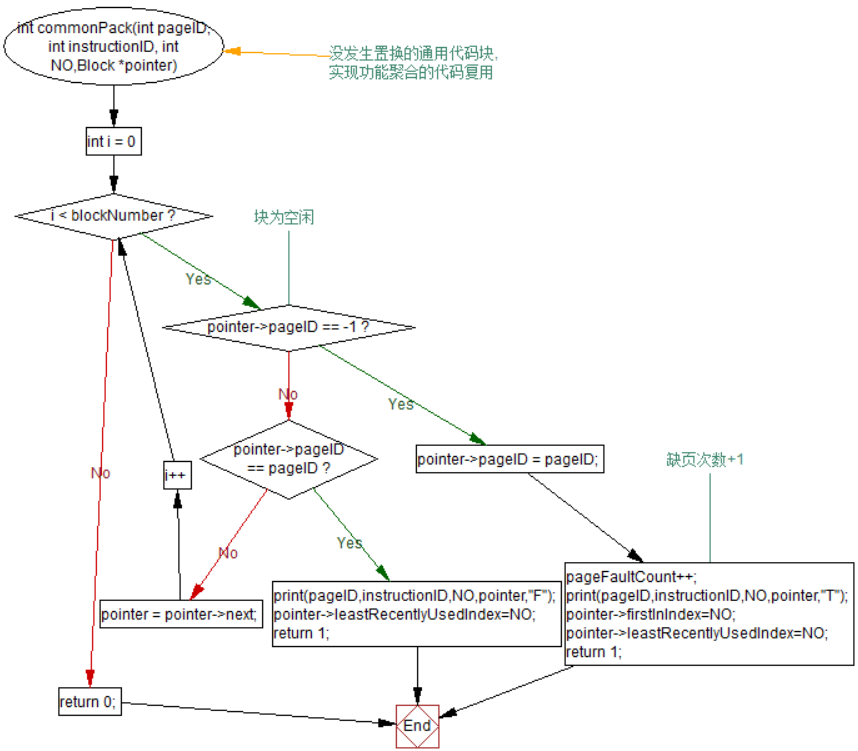
struct Block \*next;

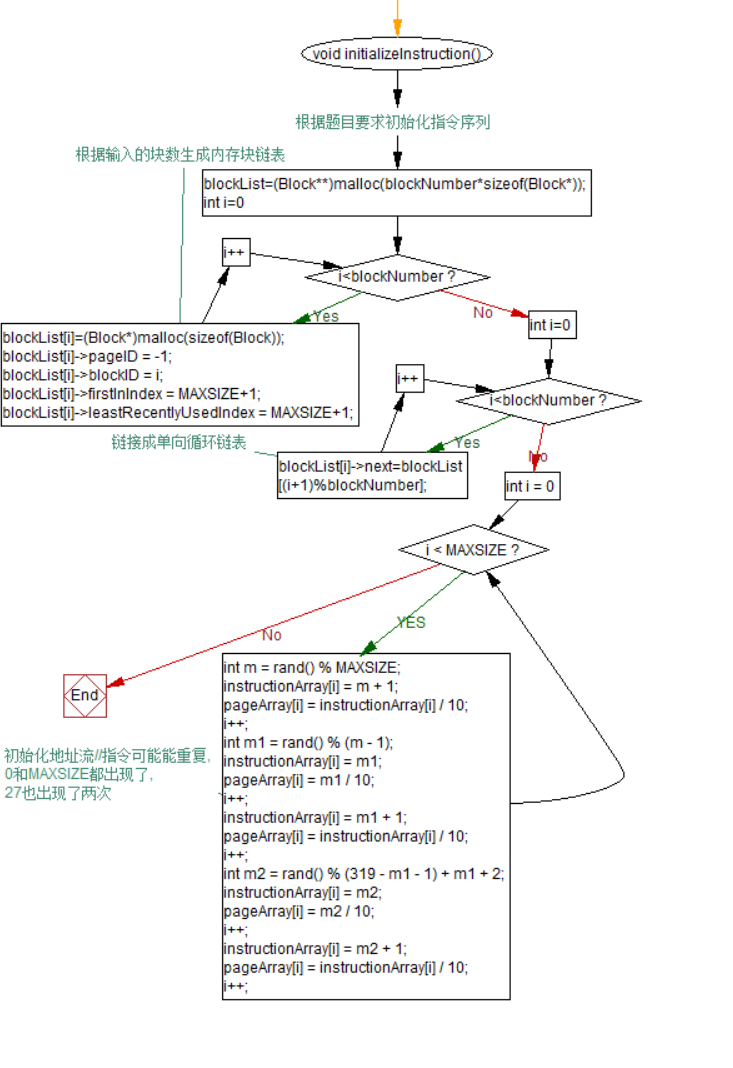
} Block;

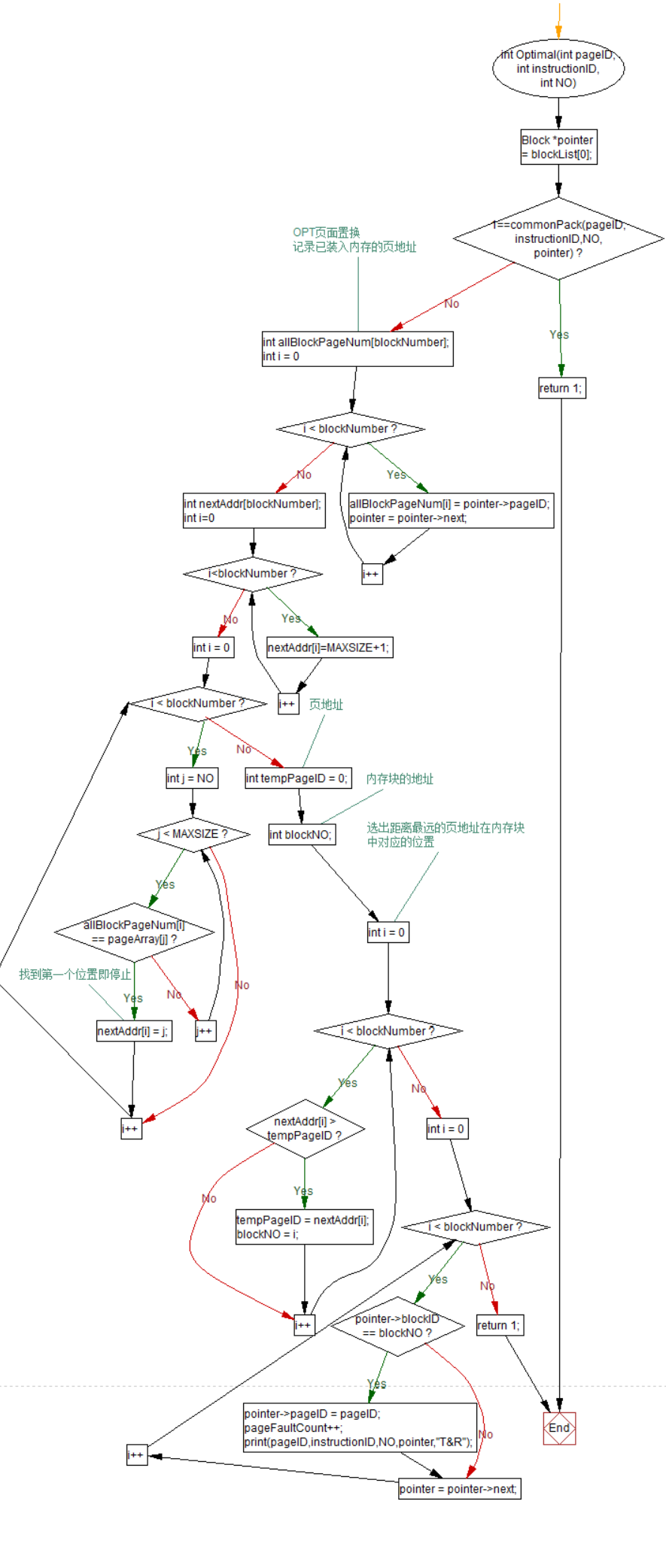
int blockNumber=3;

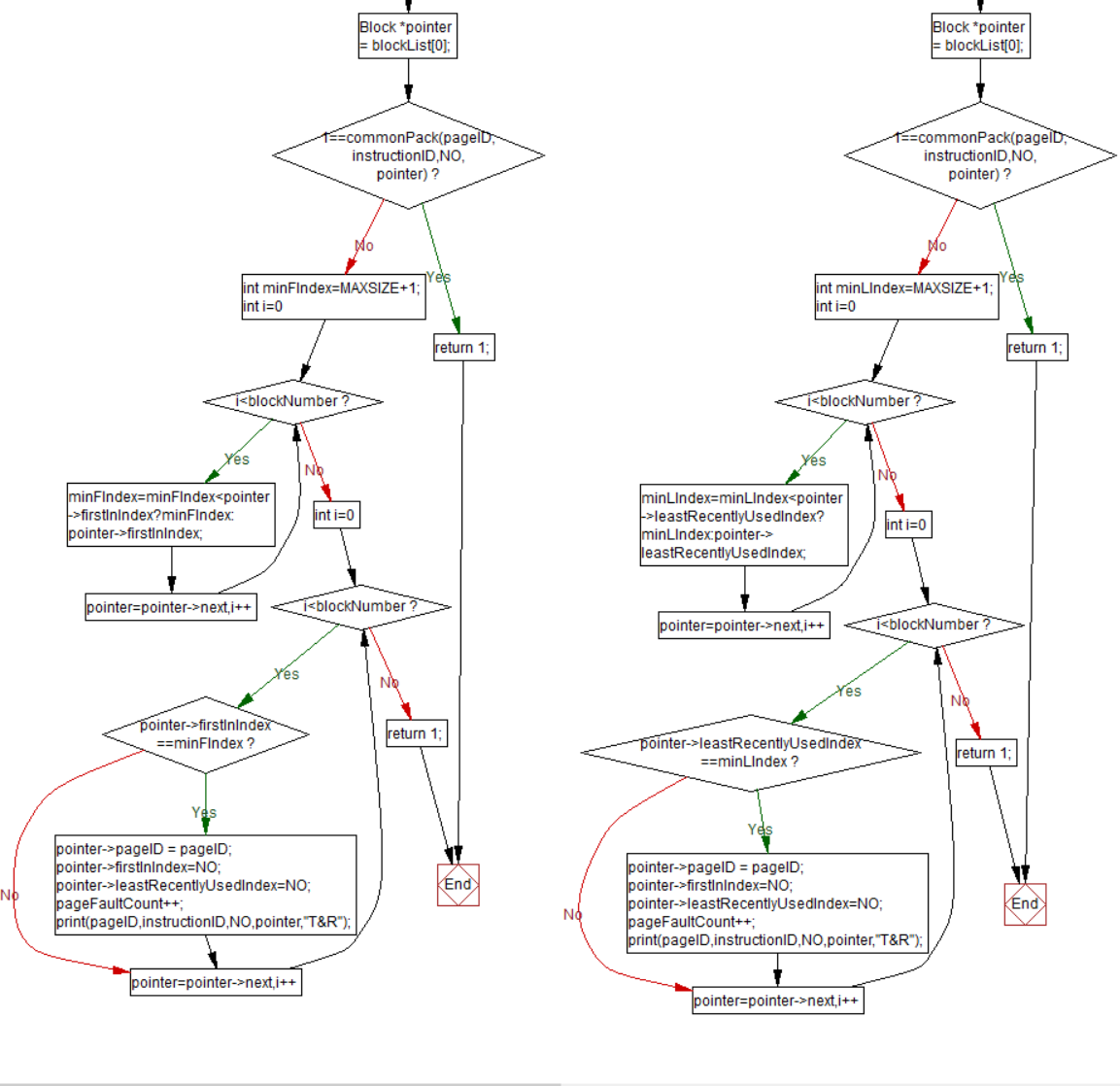
Block\*\* blockList;

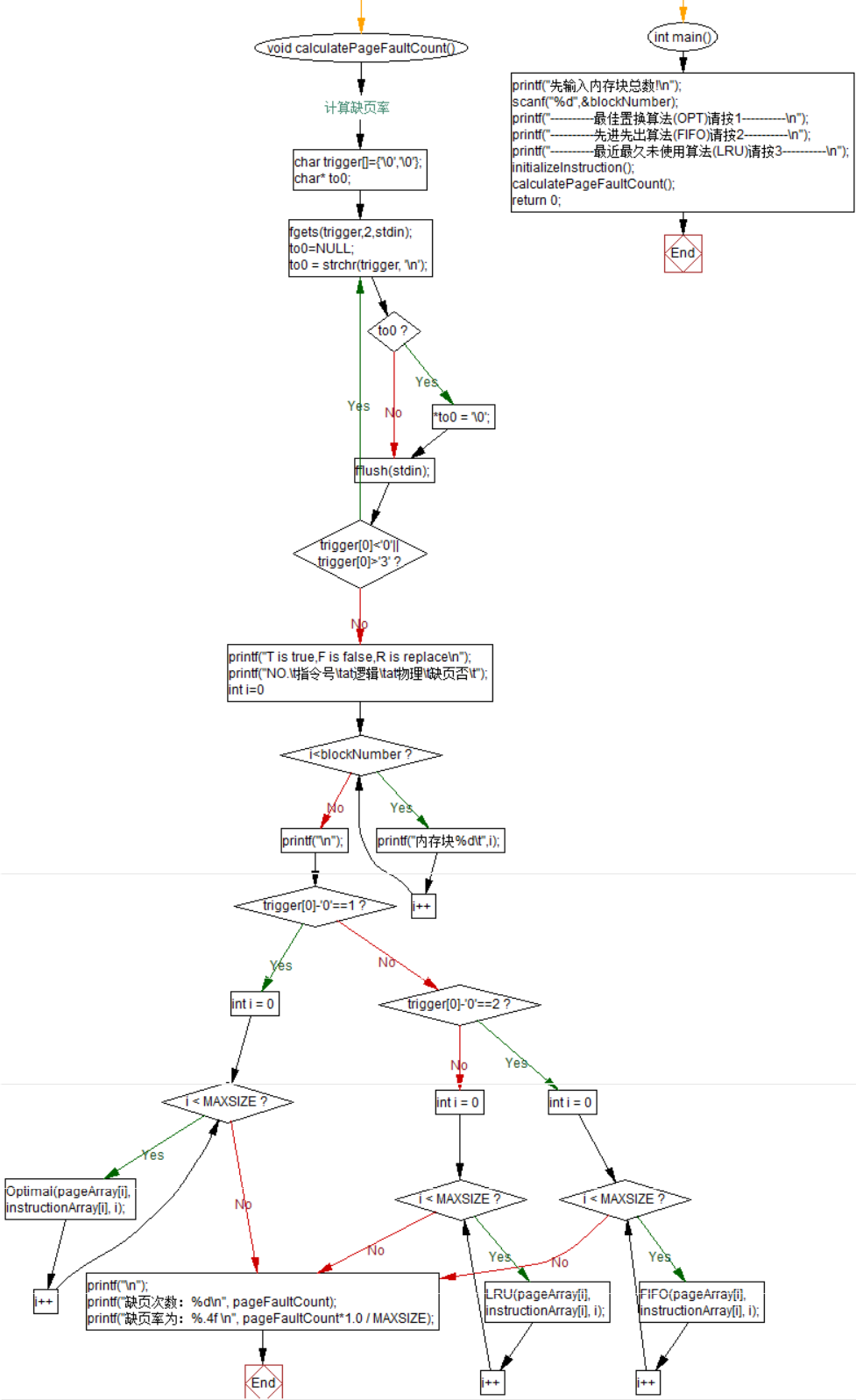
五、算法流程图



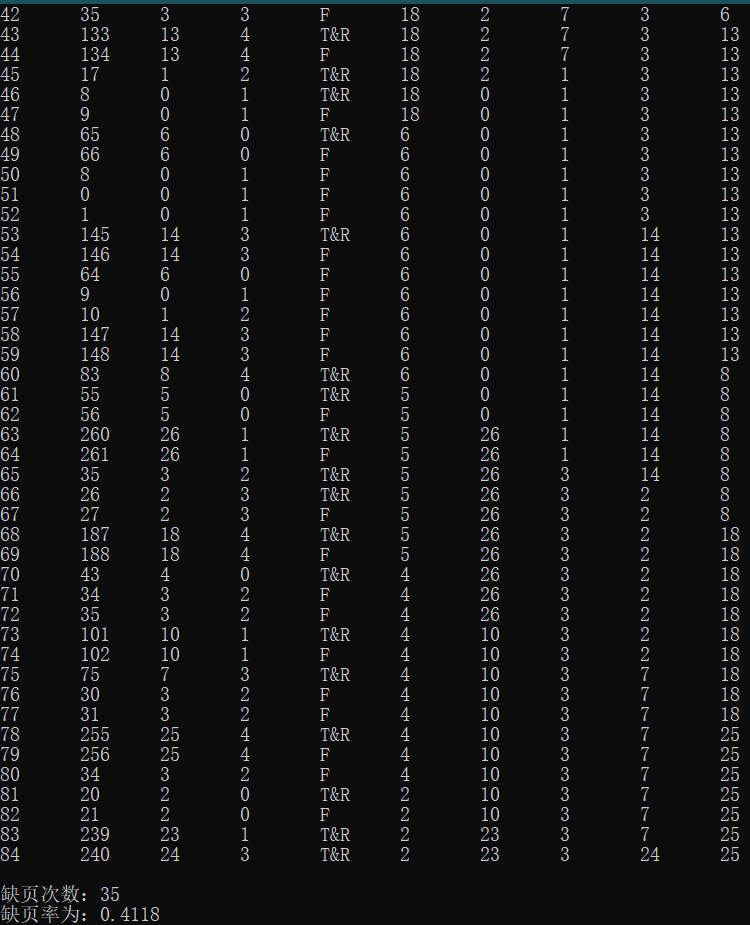
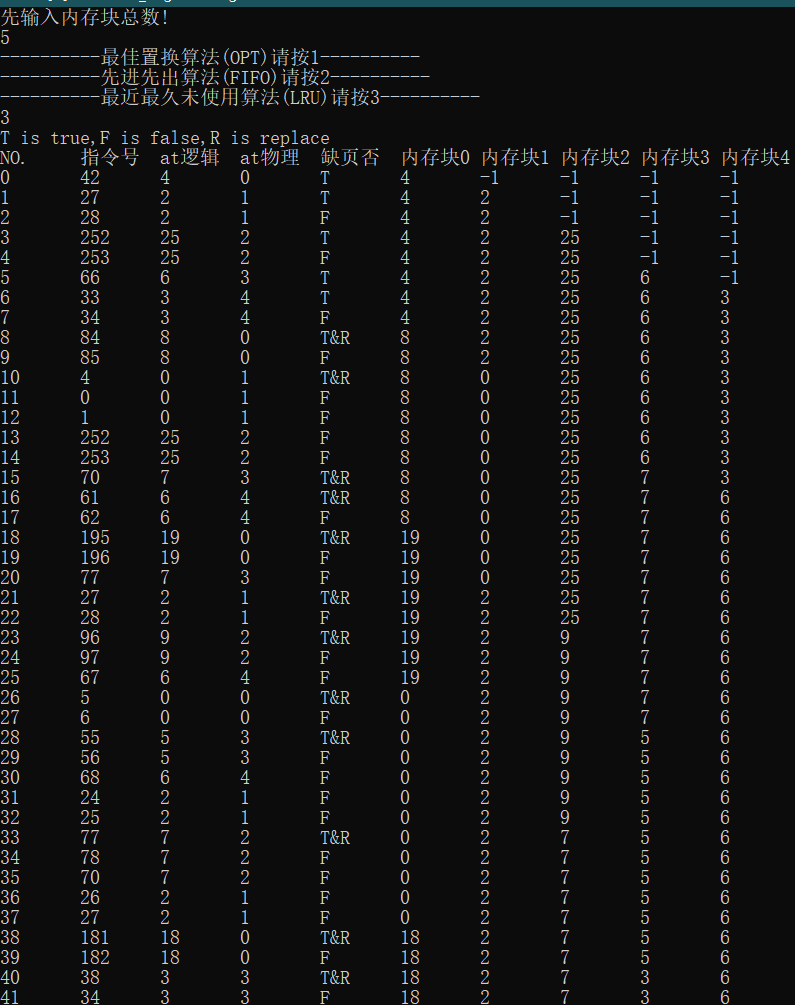
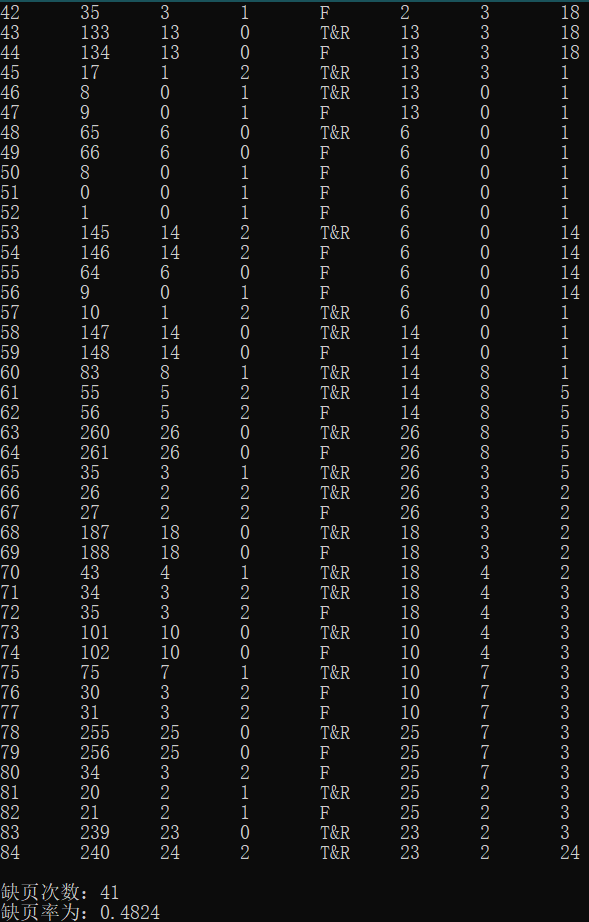








六、运行与测试



七、总结

(1)一开始我用Gets输入字符串,结果在一次测试输入时无意中一直长按键盘的某个结果发生了Buffer Overflow。,查阅之后才知道gets十分不安全, 曾在1988年制造出“Morris 蠕虫”。后全部改用fgets。但fgets也有很多坑,比入没输入字符串的字符会留在缓冲区。这时候就要fflush(stdin)。而且换行符也会被存入字符串。这时候就要用一个循环把这个换行符换为’\0’。

(2)每个指针访问前必须要先malloc！c语言自由度太高,编译时不检查这个错误,而我有时候会忘了malloc导致查半天bug查不出来。

(3)free(p)之后p成为悬垂指针,必须使p=NULL,其他有相同指向的指针也一样。

(4)一开始我没有做好模块化设计,各种输入输出和其他功能函数混一起,经过指点我才知道模块化的重要性。模块化之后的却更易于增写代码,删减冗余,查看理解代码,改bug。

**拓展实验**

1. 实验目的:

分析操作系统的核心功能模块,理解相关功能模块实现的数据结构和算法,并加以实现,加深对操作系统原理和实现过程的理解。

1. 实验内容:多级反馈调度队列调度算法

三、实现思路:

先写好moveDown();//processorInRunning进程完成一个时间片后的移动函数

int judgeNULL();//找出首个应该执行的进程所在队列级别

PCB\* extract();//从多级队列中抽出应该执行的进程

通过改写实验一实现,主要考虑在时间片中间插入的不同情况写operateReady(),分为在0级队列插入(可能一次插入多个)和其他队列插入(一次只能插入一个),以及跳过中间空白的时间片.running()分为没发生插入或发生插入但是是0级队列,和在非0级队列插入的情况,对processInRunning进行处理.

四、主要的数据结构

int relativeTime=0;//实时时间

struct pcb { /\* 定义进程控制块PCB \*/

char name[10];

char state;

int arriveTime=0;

int finishTime;

int needTime;

int runTime;

struct pcb\* link= NULL;

int belong2Index;//属于第几级队列

int timeSlot;//该级队列的时间片长度

}\*processorInRunning=NULL;

typedef struct pcb PCB;

PCB\*\* array4PCB;

int amount4PCBArray=0;//总共几个进程

int doneIndex4PCB=0;//当前实时时间到达了多少个进程

double\* array4Time;//用于记录周转时间的数组

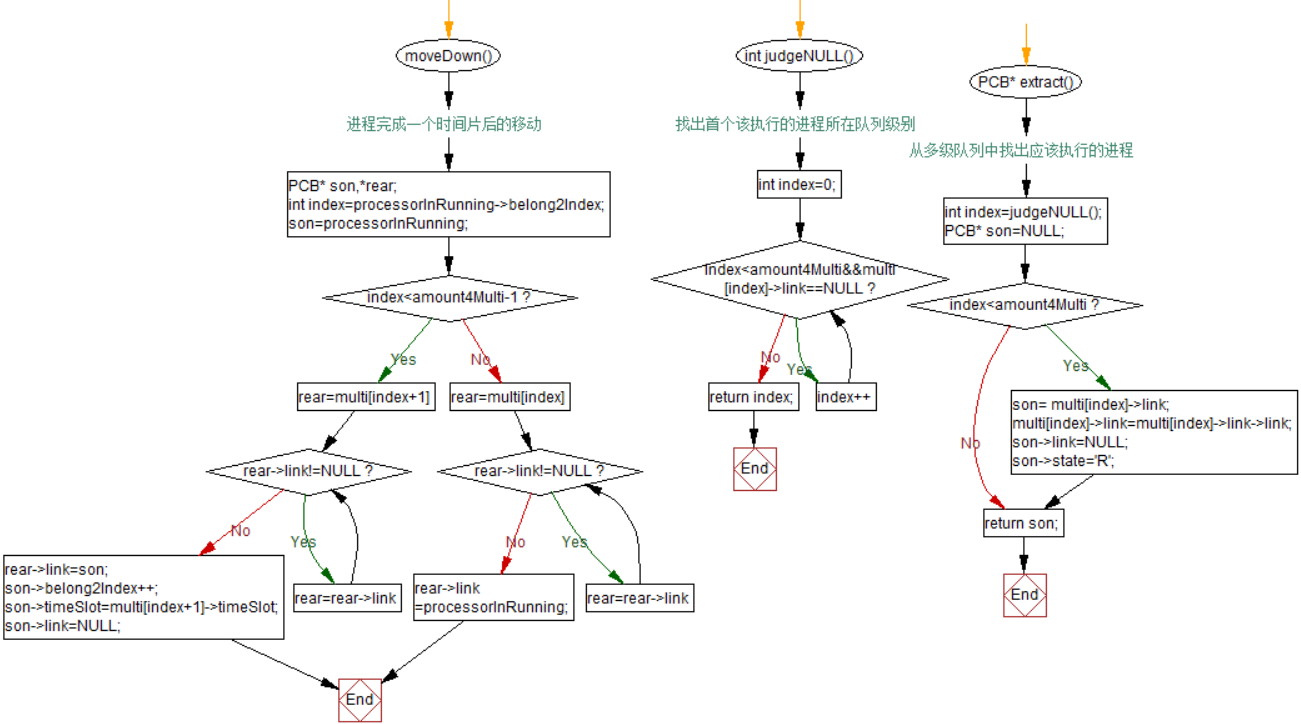
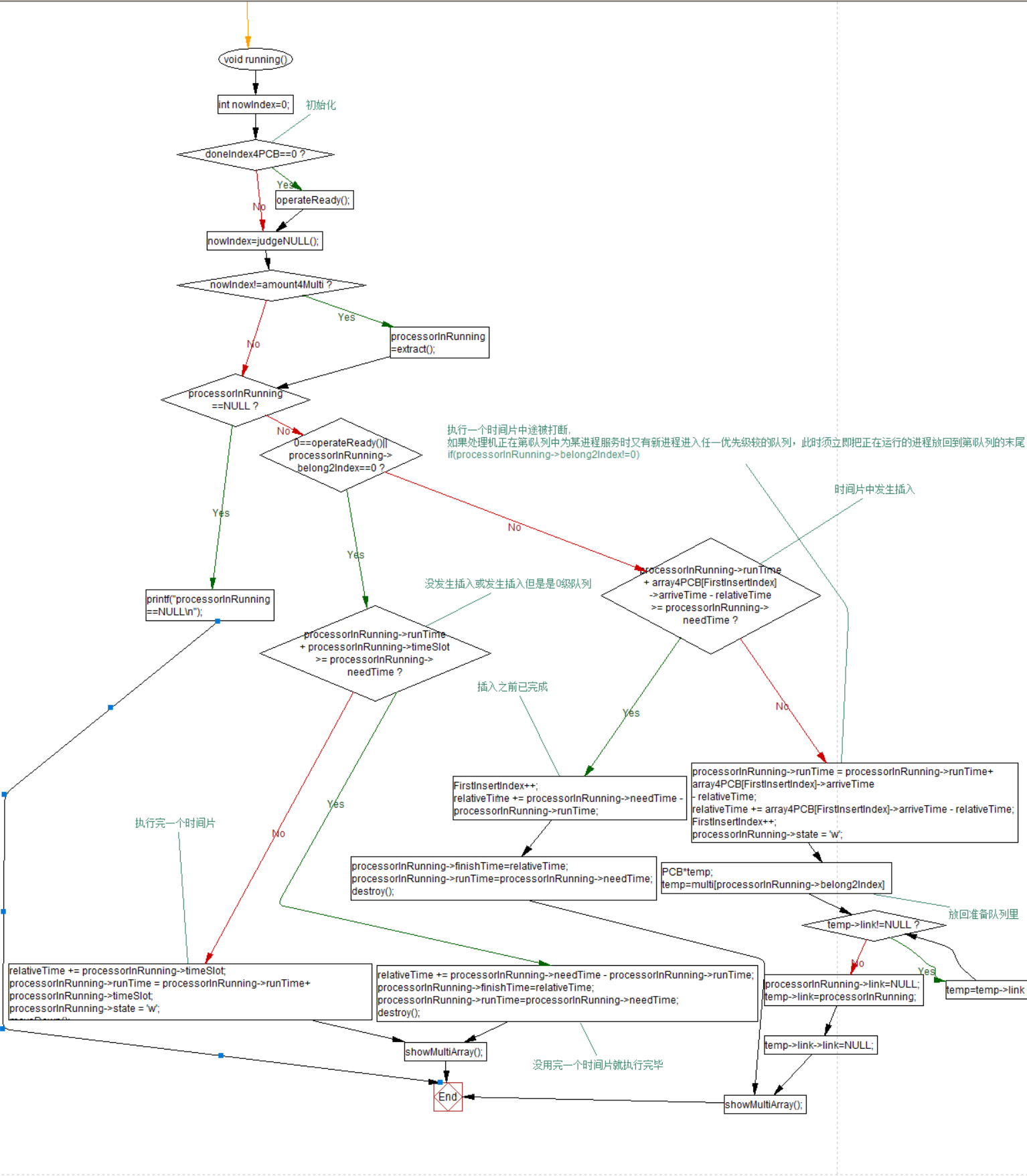
int doneIndex4Time=0;//记录第n个周转时间的索引

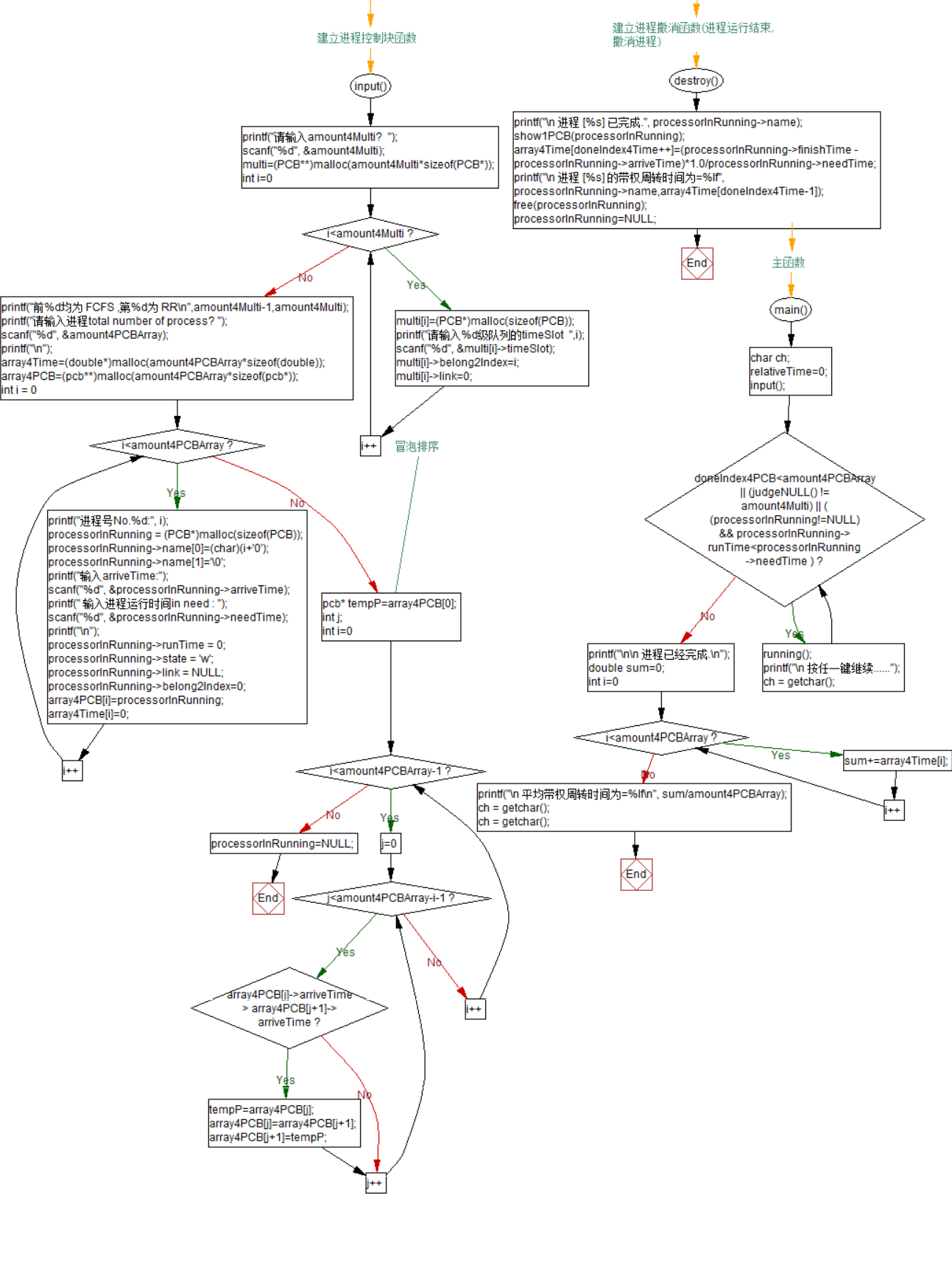
int FirstInsertIndex=-1;//和doneIndex配合用于判断有无发生插入

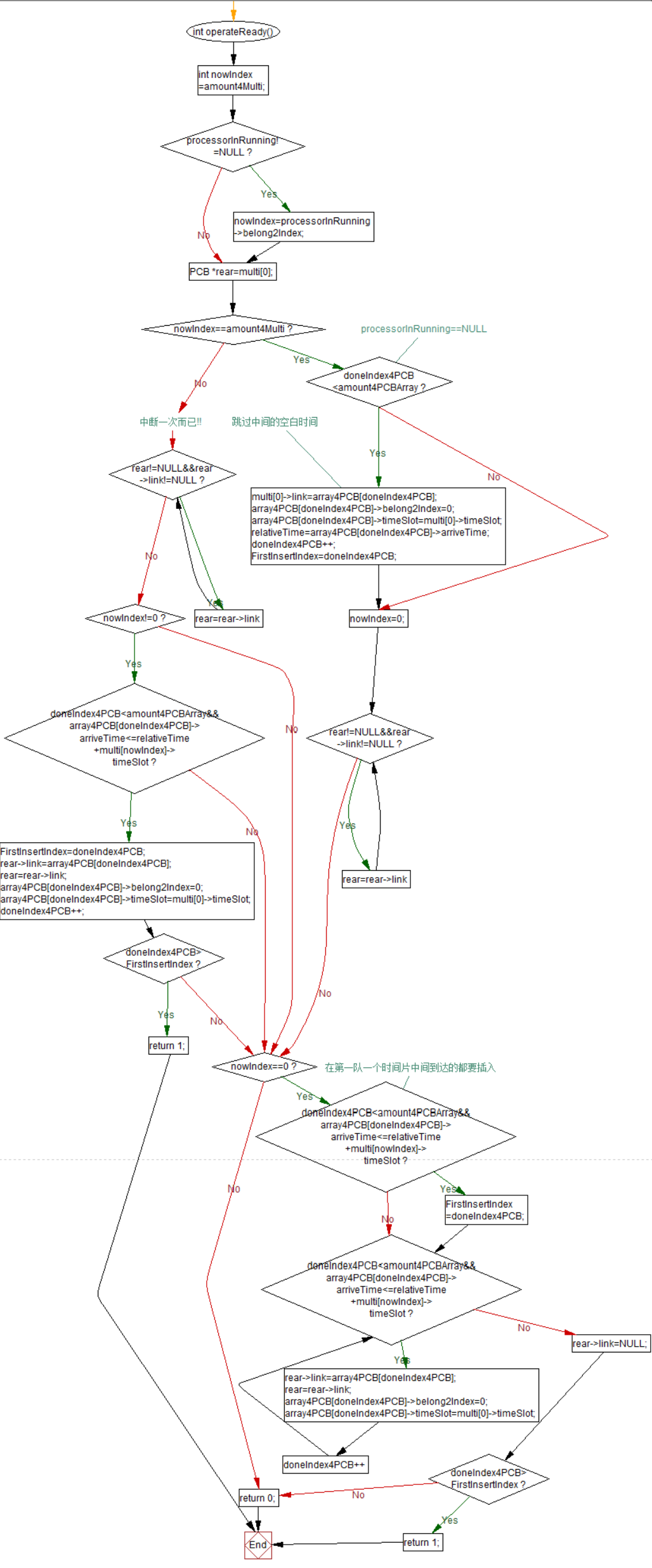
PCB\*\* multi;//多级反馈队列队头指针

int amount4Multi=0;//共多少级队列

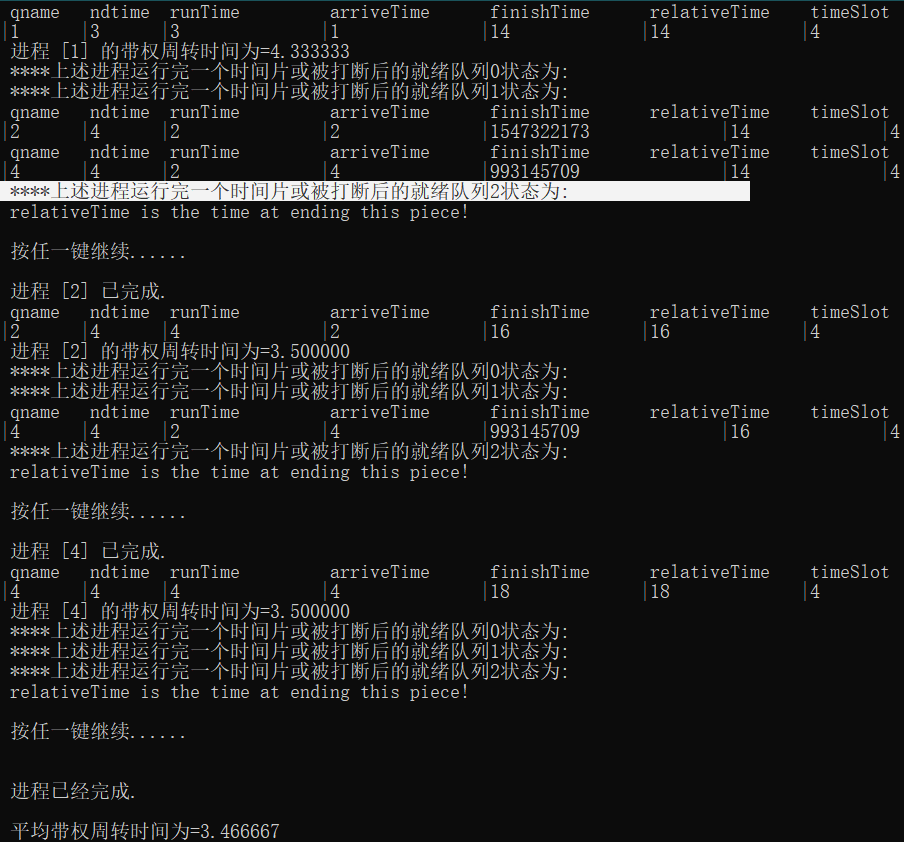
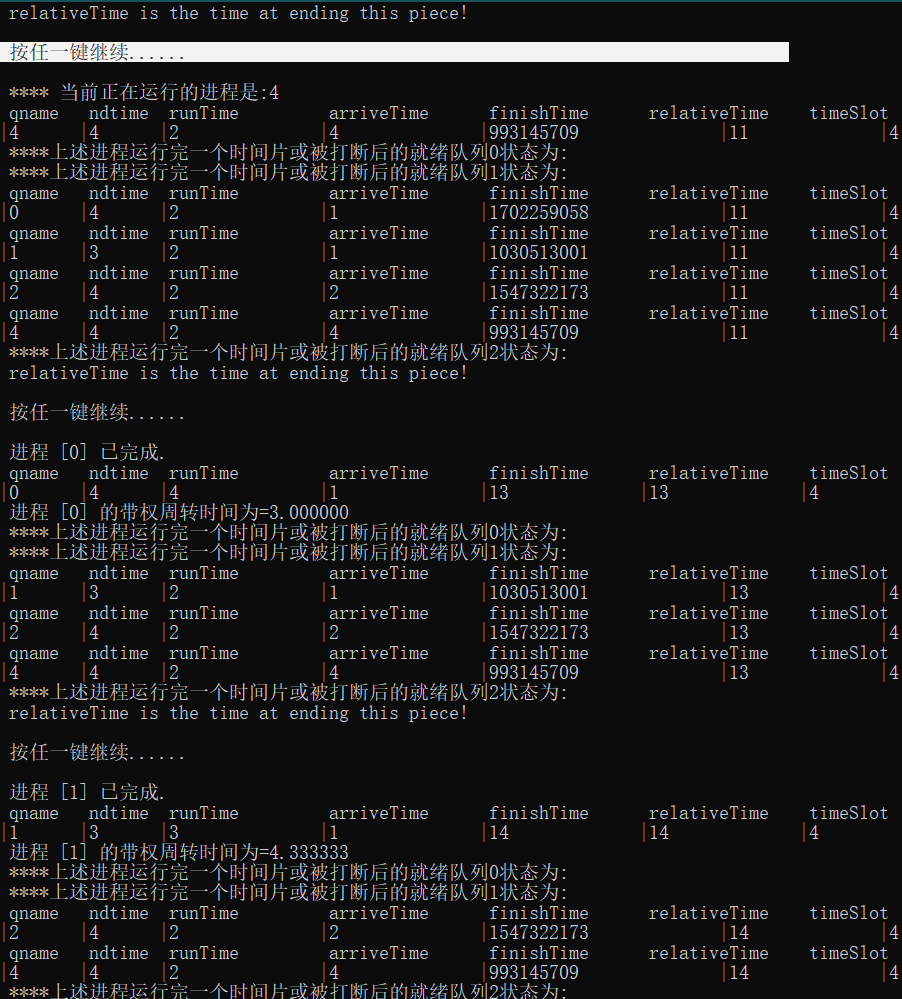
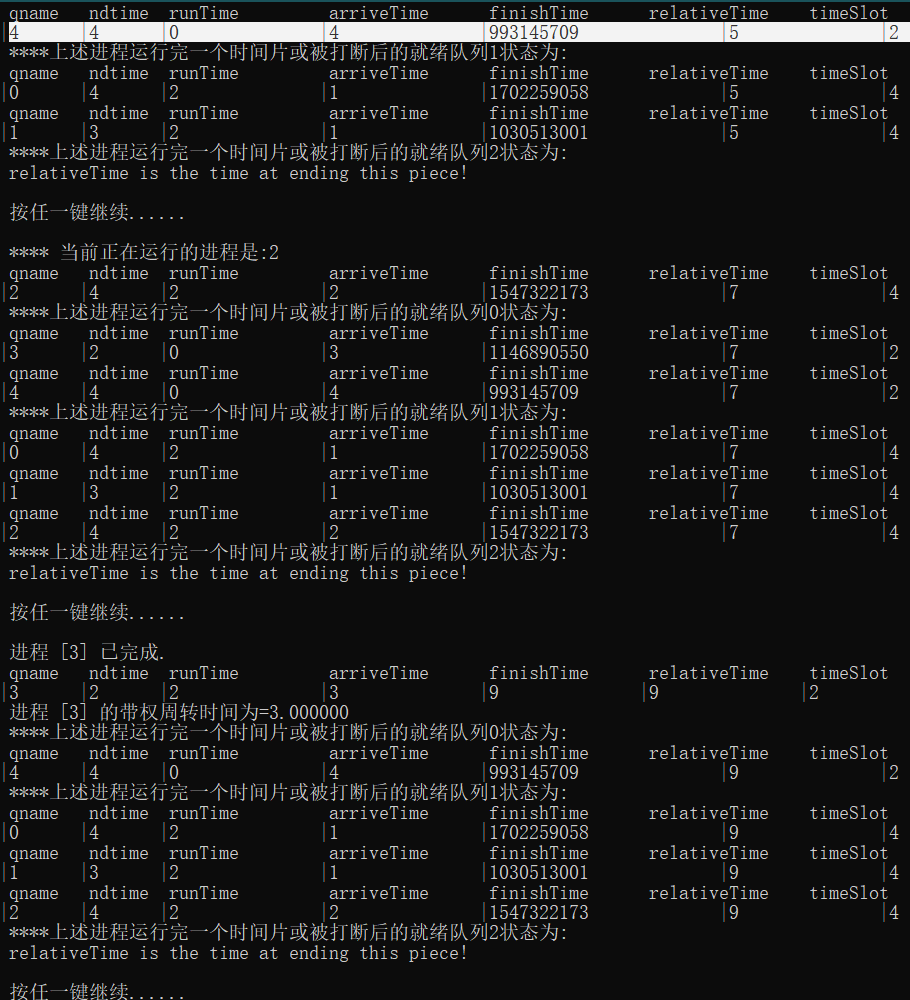
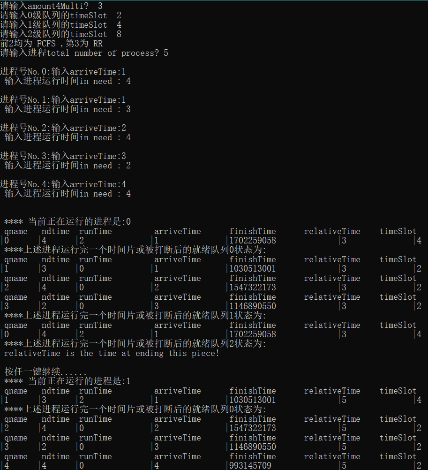
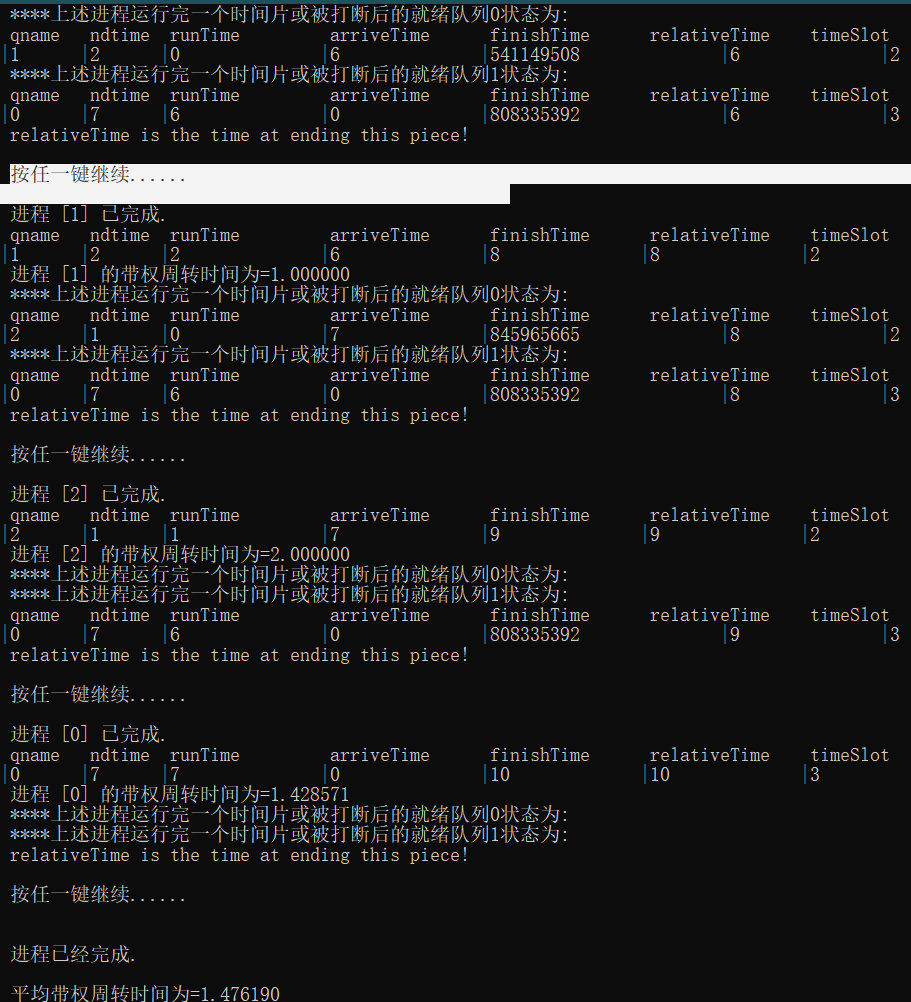
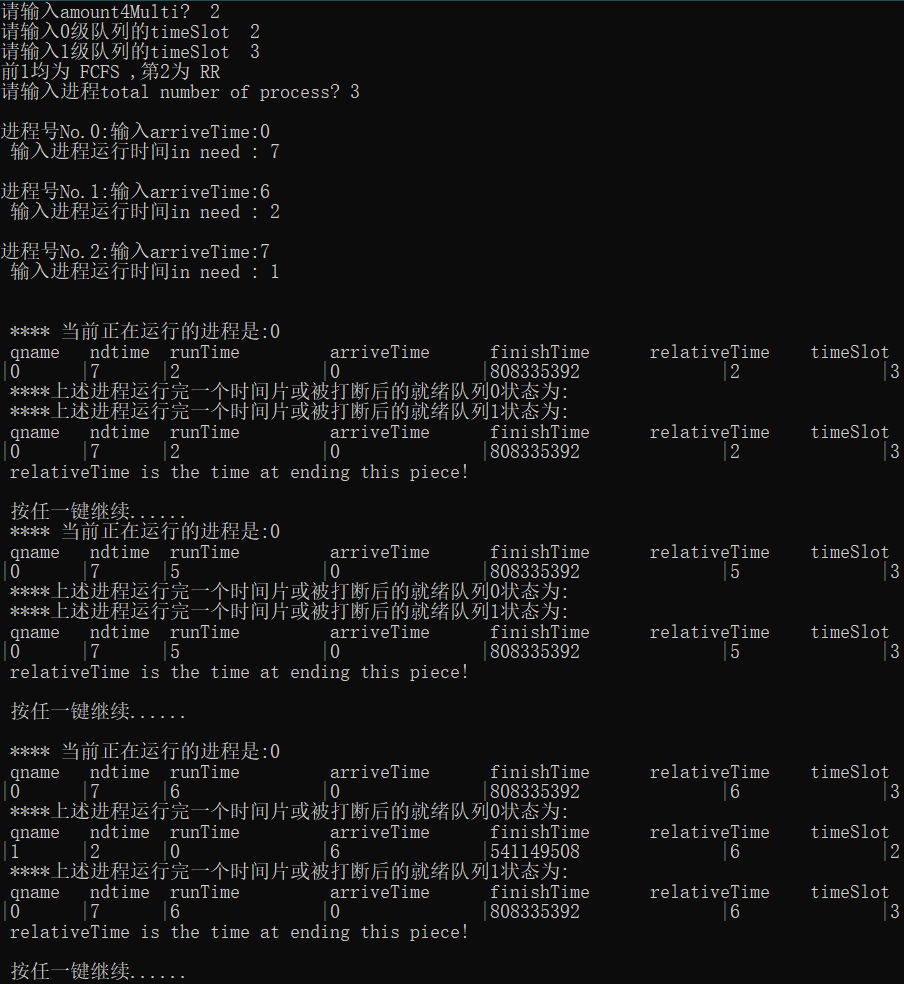
五、算法流程图





六、运行与测试



七、总结

模拟多级反馈队列需要考虑很多情况,如当前进程是否在0级队列,是否在最末级队列,运行本时间片中是否发生插入,插入前是否完成等等,处理这不好这几个条件之间的关系,举测试用例总会出错.先写好通用的工具函数如movedown(),show1PCB()等会使得考虑主要的控制逻辑operateReady()和runnnig()时思路更清晰.

通过这次实验,我体会到了多级反馈队列是一种性能较好的作业低级调度策略,能够满足各类用户的需要。对于分时交互型短作业,系统通常可在第一队列(高优先级队列)规定的时间片内让其完成工作,使终端型用户都感到满意；对短的批处理作业,通常,只需在第一或第一、第二队列(中优先级队列)中各执行一个时间片就能完成工作,周转时间仍然很短；对长的批处理作业,它将依次在第一、第二、……,各个队列中获得时间片并运行,决不会出现得不到处理(饥饿)的情况.