实验一 进程调度

- 课程：操作系统
- 学院：信息工程学院
- 专业班级：计科2002
- 学号：2020010505
- 姓名：薛飞宇

一、实习目的

1. 模拟单处理器情况下的处理器调度
2. 加深了解处理器调度的工作原理
3. 理解进程控制块PCB的原理及数据结构
4. 理解“最高优先数优先”调度算法
5. 理解“轮转法”调度算法
6. 阅读Linux源码，加深进程调度的理解

二、实习内容

任务1：设计“最高优先数优先”调度算法

任务2：设计“简单轮转法”调度算法

任务3：阅读Linux源码

三、实习任务及完成情况

任务1：最高优先数优先

1. 阅读下面源代码，完善程序中填空处内容。
    - 填空处内容：
        1. 如果就绪队列为空或者当前进程优先级大于队头进程优先级的话，将当前进程PCB插入队头
        2. `first = second;`
    - 源码如下：

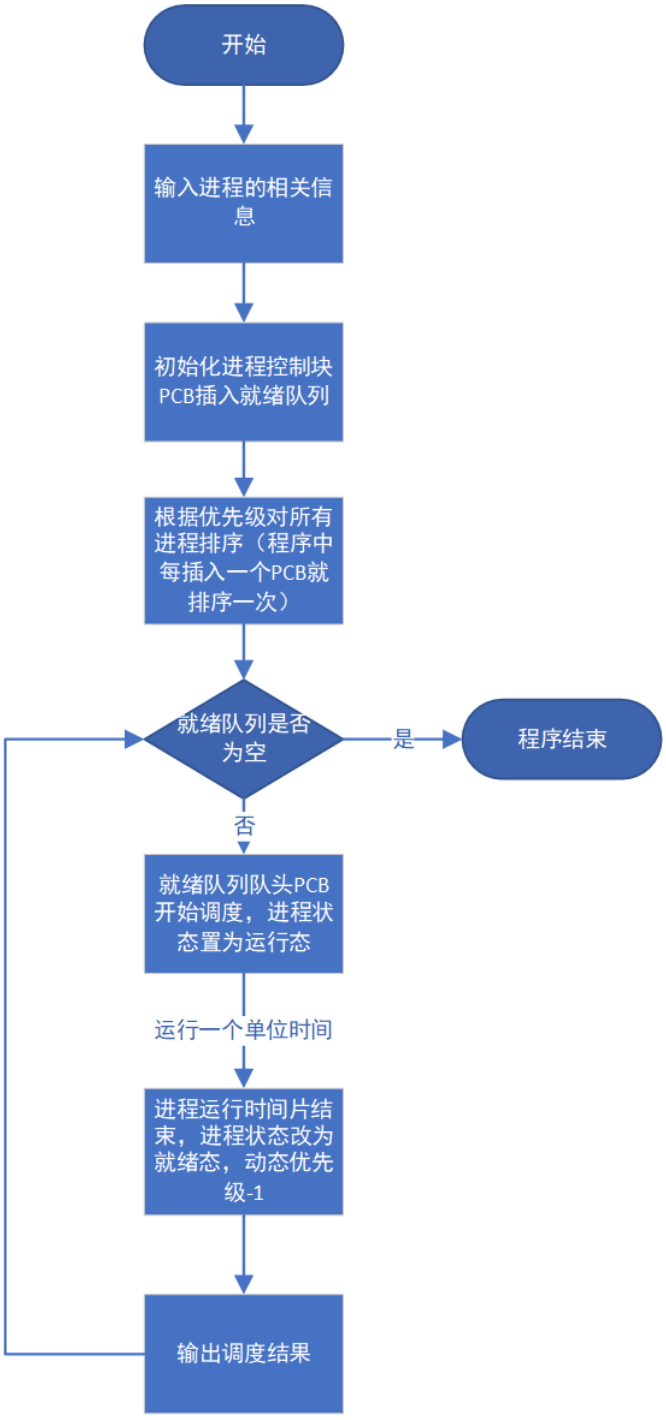
```
#include "stdio.h"
#include <stdlib.h>
#include <conio.h>
#define getpch(type) (type*)malloc(sizeof(type))
#define NULL 0

struct pcb {             /* 定义进程控制块PCB */
    char name[10];       /*进程名字*/
    char state;          /*进程状态*/
    int super;           /*进程优先级*/
    int ntime;           /*进程预计运行时间*/
    int rtime;           /*进程运行时间*/
    struct pcb* link;    /*指向下一个进程的指针*/
}*ready=NULL,*p;
```

```
                        /*ready为就绪进程队列，p为当前正在执行的进程*/

typedef struct pcb PCB;

/* 建立对进程进行优先级排列函数*/
sort()
{
    PCB *first, *second;
    int insert=0;
    if((ready==NULL)||((p->super)>(ready->super)))
    /*1如果就绪队列为空或者当前进程优先级大于队头进程优先级的话，将当前进程PCB插入队头 */
    {
        p->link=ready;
        ready=p;
    }
    else /* 进程比较优先级,插入适当的位置中*/
    {
        first=ready;
        second=first->link;
        while(second!=NULL)
        {
            if((p->super)>(second->super)) /*若插入进程比当前进程优先数大,*/
            {                              /*插入到当前进程前面*/
                p->link=second;
                first->link=p;
                second=NULL;
                insert=1;
            }
            else /* 插入进程优先数最低,则插入到队尾*/
            {
                /*2*/
                first = second;
                second=second->link;
            }
        }
        if(insert==0) first->link=p;
    }
}

/* 建立进程控制块函数*/
input()
{
    int i,num;
    system("cls"); /*清屏*/
    printf("\n 请输入进程数量?");
    scanf("%d",&num);
    for(i=0;i<num;i++)
    {
        printf("\n 进程号No.%d:\n",i);
        p=getpch(PCB);
        printf("\n 输入进程名:");
        scanf("%s",p->name);
        printf("\n 输入进程优先数:");
        scanf("%d",&p->super);
        printf("\n 输入进程运行时间:");
        scanf("%d",&p->ntime);
        printf("\n");
        p->rtime=0;p->state='w';
```

```
        p->link=NULL;
        sort(); /* 调用sort函数，将就绪队列排序*/
    }
}

/*获取就绪队列的长度*/
int space()
{
    int l=0; PCB* pr=ready;
    while(pr!=NULL)
    {
        l++;
        pr=pr->link;
    }
    return(l);
}

/*建立进程显示函数,用于显示当前进程*/
disp(PCB * pr)
{
    printf("\n qname \t state \t super \t ndtime \t runtime \n");
    printf("|%s\t",pr->name);
    printf("|%c\t",pr->state);
    printf("|%d\t",pr->super);
    printf("|%d\t",pr->ntime);
    printf("|%d\t",pr->rtime);
    printf("\n");
}

/* 建立进程查看函数 */
check()
{
    PCB* pr;
    printf("\n **** 当前正在运行的进程是:%s",p->name); /*显示当前运行进程*/
    disp(p);
    pr=ready;
    printf("\n ****当前就绪队列状态为:\n"); /*显示就绪队列状态*/
    while(pr!=NULL)
    {
        disp(pr);
        pr=pr->link;
    }
}

/*建立进程撤消函数(进程运行结束,撤消进程)*/
destroy()
{
    printf("\n 进程 [%s] 已完成.\n",p->name);
    free(p);
}

/* 建立进程就绪函数(进程运行时间到,置就绪状态)*/
running()
{
    (p->rtime)++;
    if(p->rtime==p->ntime)
        destroy(); /* 调用destroy函数*/
    else
```

```
    {
        (p->super)--;
        p->state='w'; /*进程状态改为就绪态*/
        sort(); /*调用sort函数*/
    }
}

/*主函数*/
main()
{
    int len,h=0; /*声明*/
    char ch; /*声明*/
    input(); /*输入进程相关信息*/
    len=space(); /*len获取就绪队列长度*/
    while((len!=0)&&(ready!=NULL)) /*当就绪队列长度不为0并且就绪队列不为空时*/
    {
        ch=getchar(); /*按任意键继续*/
        h++;          /*系统执行时间+1*/
        printf("\n The execute number:%d \n",h); /*打印当前系统已执行时间*/
        /*执行就绪队列队头的进程*/
        p=ready; /*指针p指向就绪队列队头*/
        ready=p->link; /*就绪队列队头指向下一个PCB*/
        p->link=NULL; /*原先队头的指针域置空*/
        p->state='R'; /*原先队头的进程状态改为运行态*/
        check(); /*调用进程查看函数*/
        running(); /*更新进程的优先级，就绪队列重新排序*/
        printf("\n 按任一键继续......");
        ch=getchar();
    }
    printf("\n\n 进程已经完成.\n");
    ch=getchar();
}
```

2. 阅读代码，写出调度算法、算法流程图和程序功能。
    - 调度算法：动态最高优先数优先调度算法
    - 算法流程图：



- 程序功能：模拟在单处理机环境下对并发执行的多个进程进行“最高优先数优先”算法的调度

3. 解释数据结构PCB的定义和作用

- 定义：进程控制块（Processing Control Block），是操作系统内核中一种数据结构，主要表示进程状态。

```
struct pcb {             /* 定义进程控制块PCB */
    char name[10];       /*进程名字*/
    char state;          /*进程状态*/
    int super;           /*进程优先级*/
    int ntime;           /*进程预计运行时间*/
    int rtime;           /*进程运行时间*/
    struct pcb* link;    /*指向下一个进程的指针*/
}
```

- 作用：使一个在多道程序环境下不能独立运行的程序（含数据），成为一个能独立运行的基本单位或与其它进程并发执行的进程。或者说，OS是根据PCB来对并发执行的进程进行控制和管理的。

4. 为main()写出每行的注释

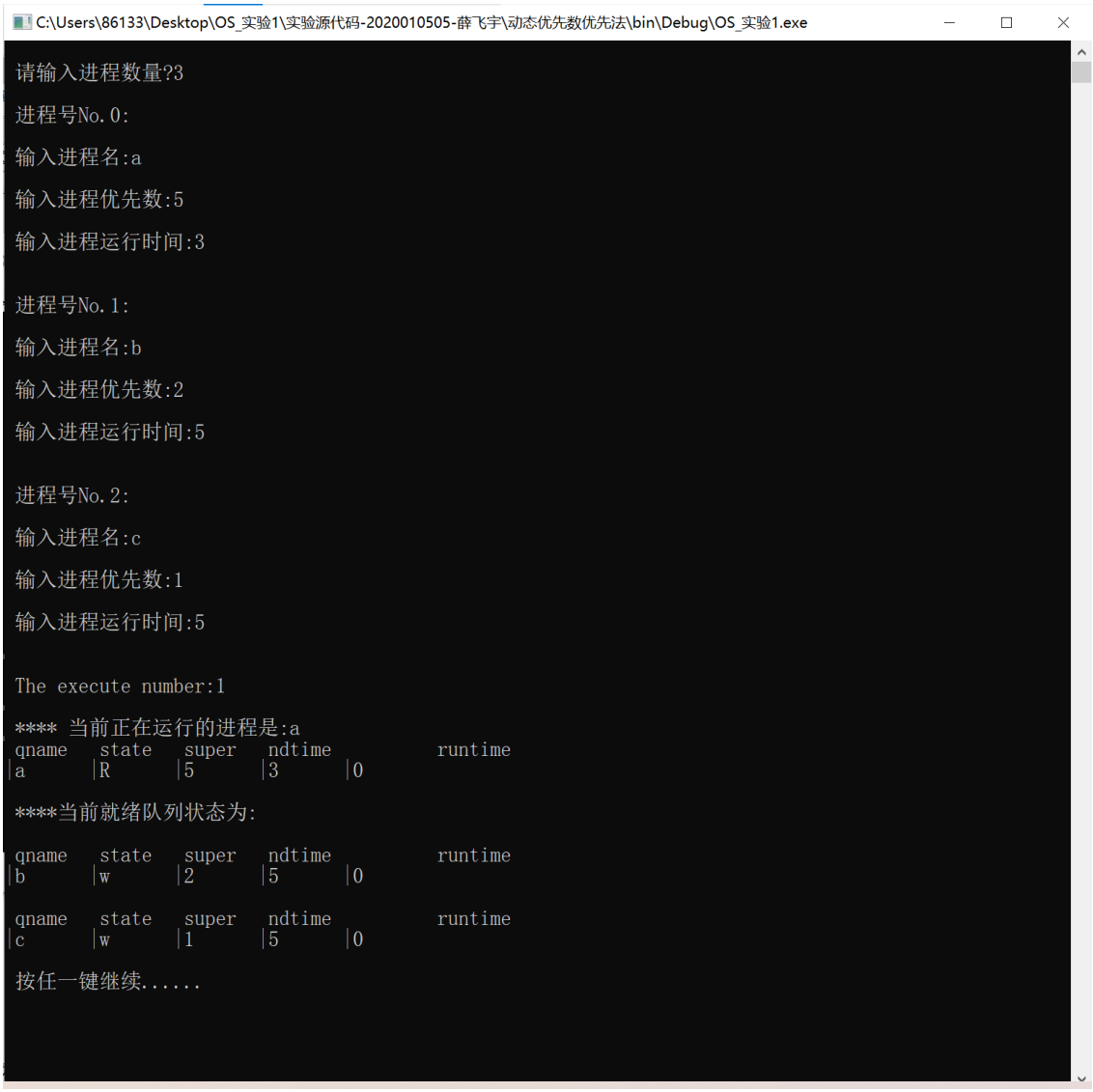
```
/*主函数*/
main()
{
    int len,h=0; /*声明*/
    char ch; /*声明*/
    input(); /*输入进程相关信息*/
    len=space(); /*len获取就绪队列长度*/
    while((len!=0)&&(ready!=NULL)) /*当就绪队列长度不为0并且就绪队列不为空时*/
    {
        ch=getchar(); /*按任意键继续*/
        h++;          /*系统执行时间+1*/
        printf("\n The execute number:%d \n",h); /*打印当前系统已执行时间*/
        /*执行就绪队列队头的进程*/
        p=ready; /*指针p指向就绪队列队头*/
        ready=p->link; /*就绪队列队头指向下一个PCB*/
        p->link=NULL; /*原先队头的指针域置空*/
        p->state='R'; /*原先队头的进程状态改为运行态*/
        check(); /*调用进程查看函数*/
        running(); /*更新进程的优先级，就绪队列重新排序*/
        printf("\n 按任一键继续......");
        ch=getchar(); /*使程序暂停，按任意键继续执行*/
    }
    printf("\n\n 进程已经完成.\n");
    ch=getchar();/*使程序暂停，按任意键继续执行*/
}
```

5. 调试并运行代码，写出结果

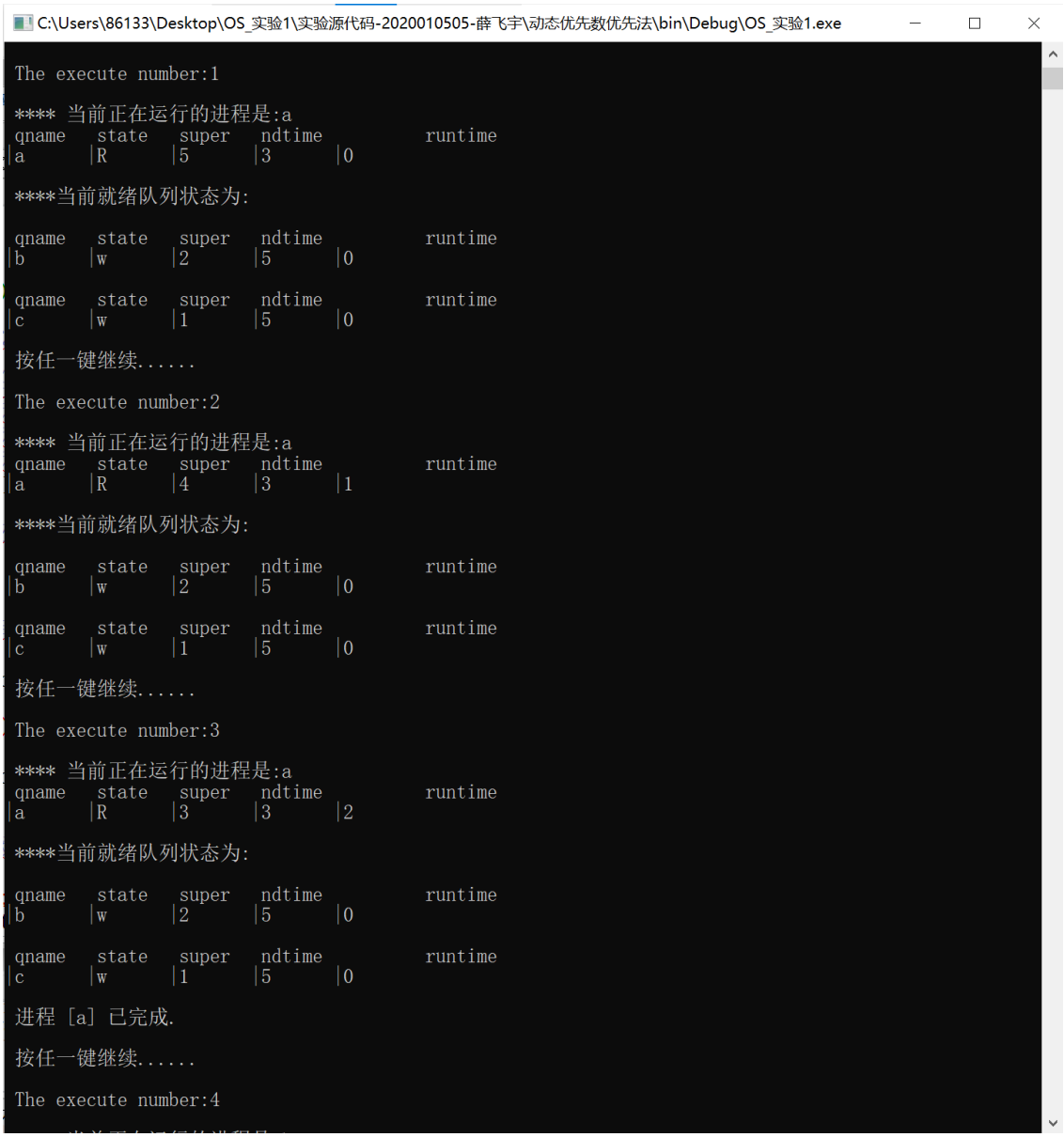
- 首先输入要测试的进程数量，这里为了方便演示，我填了3
- 然后给程序填入3个PCB的进本信息
- a进程的优先数最大，所以时刻1执行a进程，其他的进程放入就绪队列
- 每隔1个时刻，正在执行的进程优先数减1
- 每个时刻都判断一下当前执行的优先数和就绪队列中队头进程的优先数做出调整和调度
- 最终3个进程全部顺利执行完毕

1. 输入3个进程的PCB信息如下

    - 进程a 优先数5 运行时间3
    - 进程b 优先数2 运行时间5
    - 进程c 优先数1 运行时间5

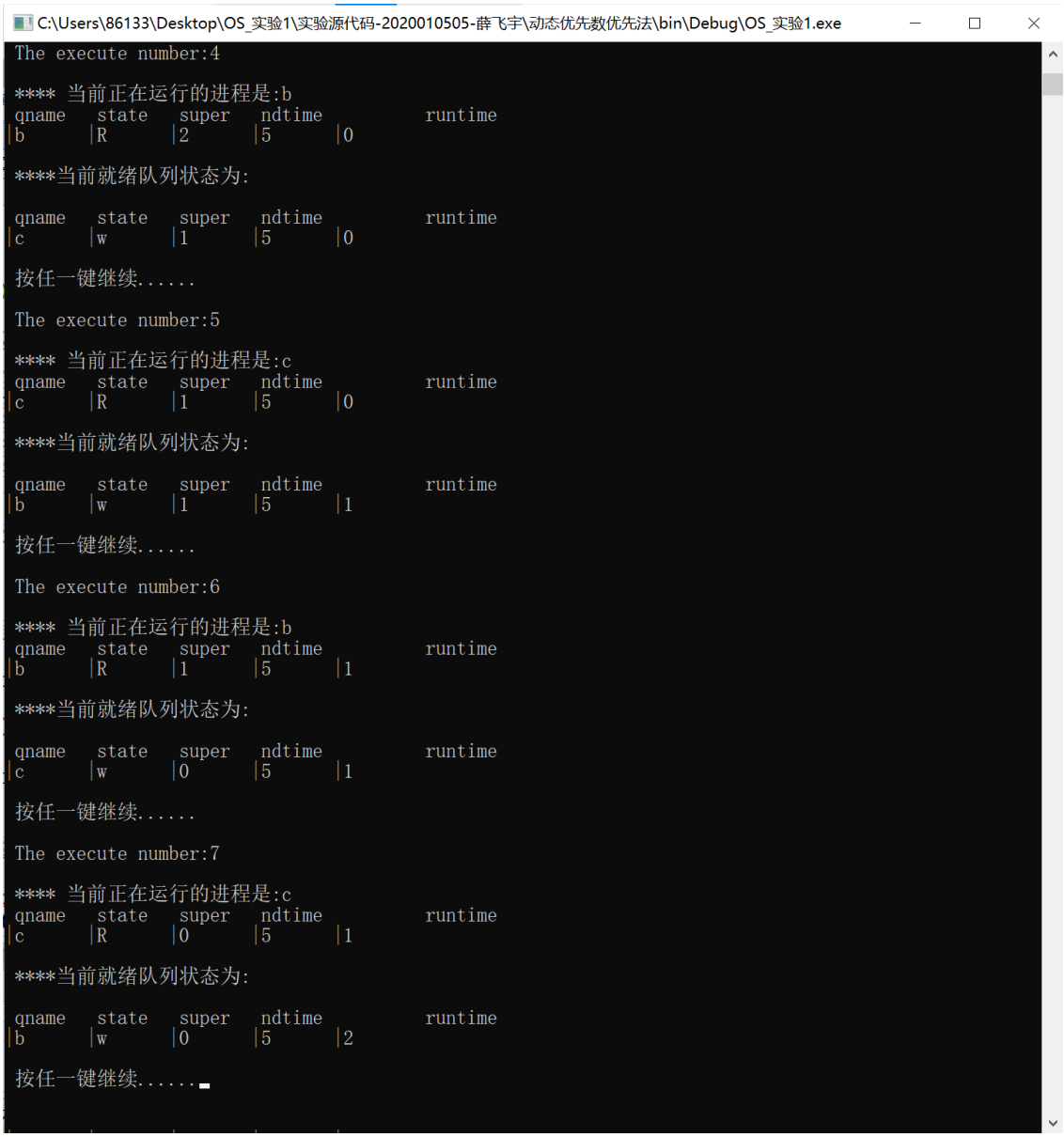


2. 执行时间1-3。进程a的优先数始终保持最高，在时刻3，进程a执行完毕

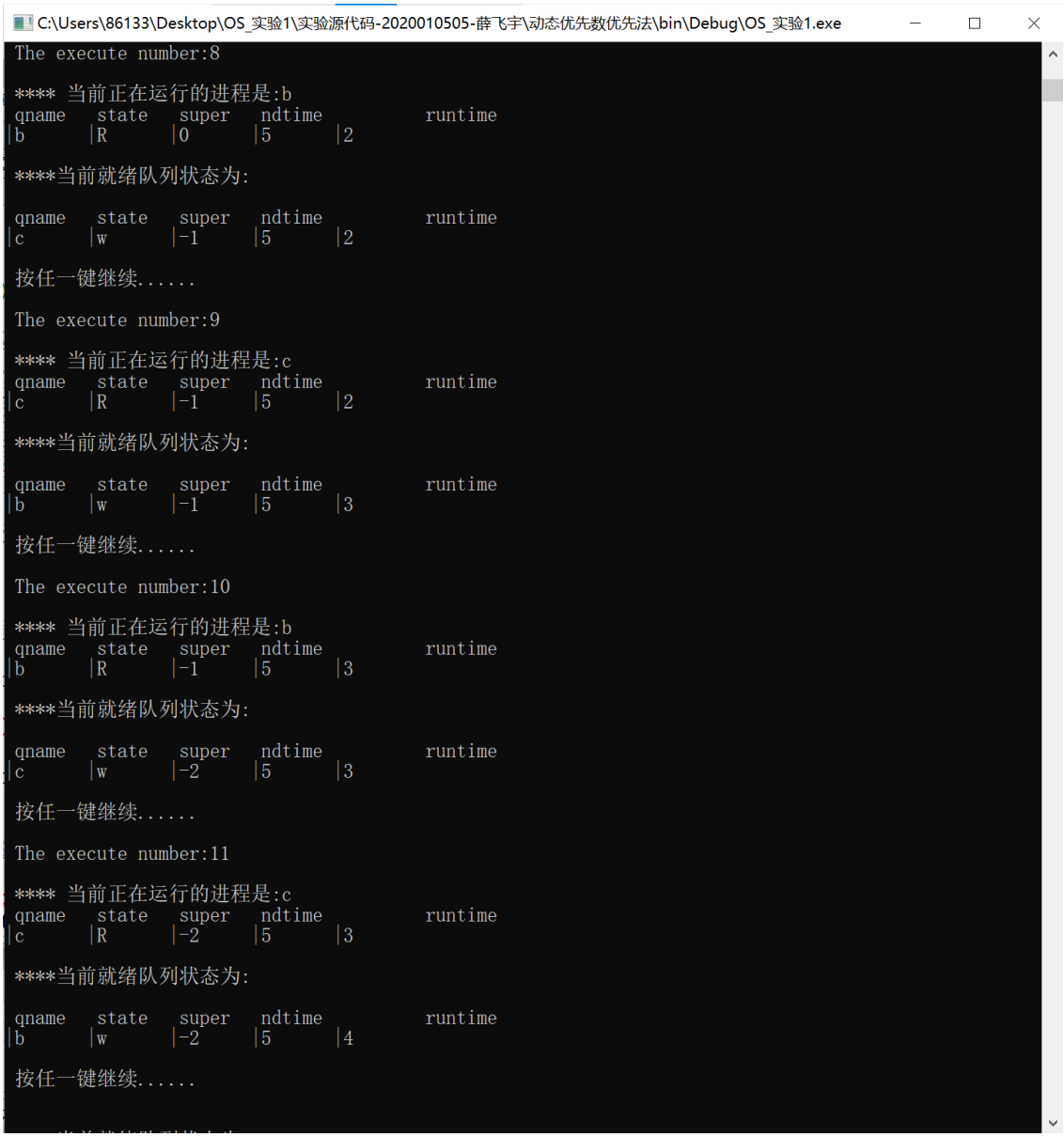


3. 执行时间4-7。

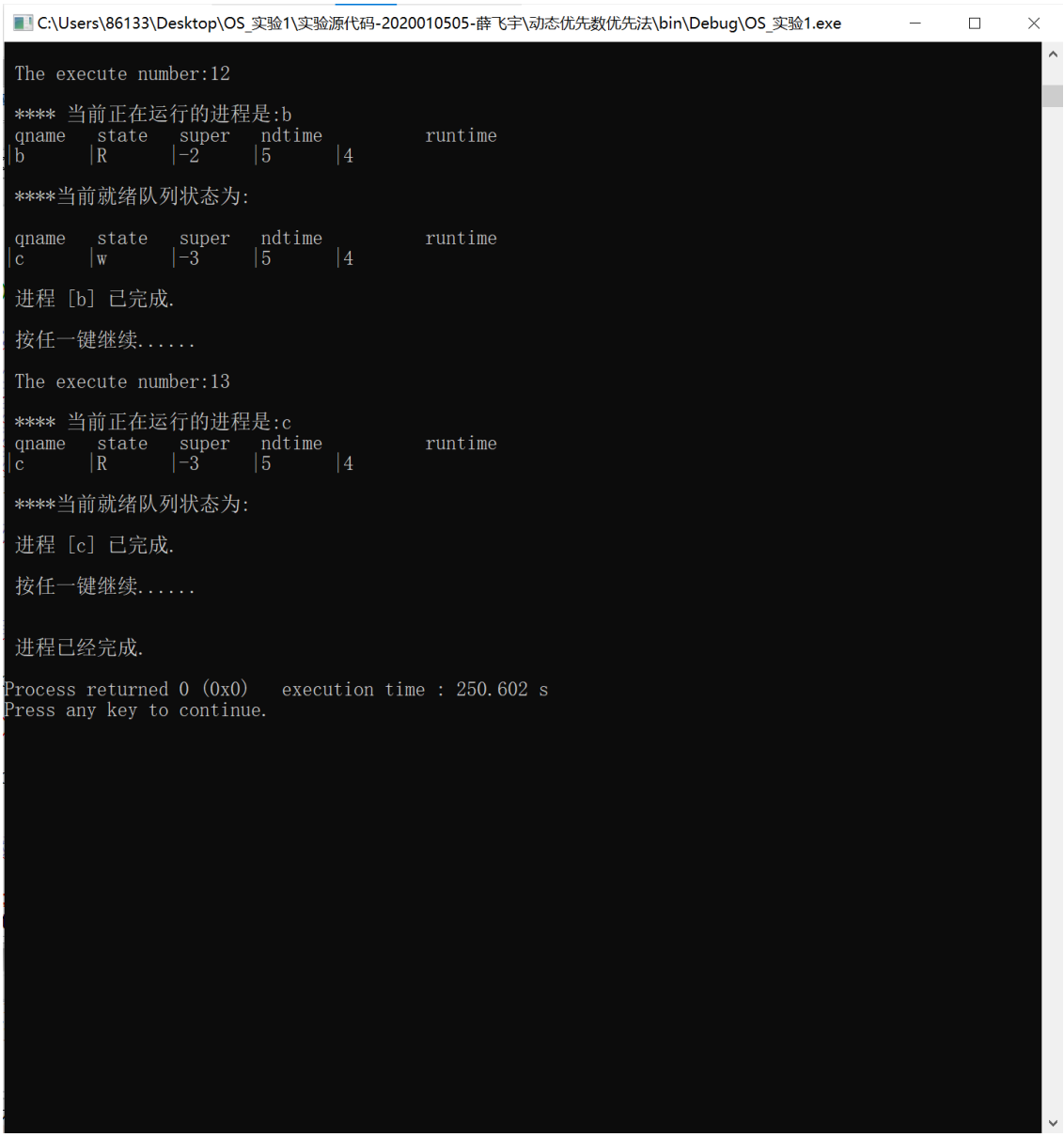
- 时刻4，进程b优先数高于进程c，故b执行；
- 时刻5，进程b优先数为1，和进程c相等，故c执行；
- 时刻6-7，进程b，c微观上交替执行，宏观上并发执行。



4. 执行时间8-11，进程b，c交替执行。



5. 执行时间12-13。
    - 时刻12，进程b执行完成
    - 时刻13，进程c执行完成



任务2：简单轮转法

1. 基本思想

   简单轮转法的基本思想是：所有就绪进程按“先来先服务”排成一个队列，总是把处理机分配给队首的进程，各进程占用CPU的时间片相同。如果运行进程用完它的时间片后还未完成，就把它送回到就绪队列的末尾，把处理机重新分配给队首的进程。直至所有的进程运行完毕。

2. 源码

```
#include "stdio.h"
#include <stdlib.h>
#include <conio.h>
#define getpch(type) (type*)malloc(sizeof(type))
#define NULL 0

struct pcb {            /* 定义进程控制块PCB */
    char name[10];      /*进程名字*/
    char state;         /*进程状态*/
    int super;          /*进程优先级*/
    int ntime;          /*进程预计运行时间*/
    int rtime;          /*进程运行时间*/
    struct pcb* link;   /*指向下一个进程的指针*/
}*ready=NULL,*p;
```

```
                        /*ready为就绪进程队列，p为当前正在执行的进程*/

typedef struct pcb PCB;



/* 建立对进程进行优先级排列函数*/
/*简单轮转法中，只要将所有进程优先数*/
sort()
{
    PCB *first, *second;
    int insert=0;
    if((ready==NULL)||((p->super)>(ready->super)))
    /*1如果就绪队列为空或者当前进程优先级大于队头进程优先级的话，将当前进程PCB插入队头 */
    {
        p->link=ready;
        ready=p;
    }
    else /* 进程比较优先级,插入适当的位置中*/
    {
        first=ready;
        second=first->link;
        while(second!=NULL)
        {
            if((p->super)>(second->super)) /*若插入进程比当前进程优先数大,*/
            {                              /*插入到当前进程前面*/
                p->link=second;
                first->link=p;
                second=NULL;
                insert=1;
            }
            else /* 插入进程优先数最低,则插入到队尾*/
            {
                /*2*/
                first = second;
                second=second->link;
            }
        }
        if(insert==0) first->link=p;
    }
}



/* 建立进程控制块函数*/
input()
{
    int i,num;
    system("cls"); /*清屏*/
    printf("\n 请输入进程数量?");
    scanf("%d",&num);
    for(i=0;i<num;i++)
    {
        printf("\n 进程号No.%d:\n",i);
        p=getpch(PCB);
        printf("\n 输入进程名:");
        scanf("%s",p->name);
//      printf("\n 输入进程优先数:");
```

```
//      scanf("%d",&p->super);
/*简单轮转法中，只要将所有进程优先数置0即可*/

        p->super = 0;
        printf("\n 输入进程运行时间:");
        scanf("%d",&p->ntime);
        printf("\n");
        p->rtime=0;p->state='w';
        p->link=NULL;
        sort(); /* 调用sort函数，将就绪队列排序*/
    }
}

/*获取就绪队列的长度*/
int space()
{
    int l=0; PCB* pr=ready;
    while(pr!=NULL)
    {
        l++;
        pr=pr->link;
    }
    return(l);
}

/*建立进程显示函数,用于显示当前进程*/
disp(PCB * pr)
{
    printf("\n qname \t state \t ndtime \t runtime\n");
    printf("|%s\t",pr->name);
    printf("|%c\t",pr->state);
//  printf("|%d\t",pr->super);
    printf("|%d\t",pr->ntime);
    printf("|%d\t",pr->rtime);
    printf("\n");
}

/* 建立进程查看函数 */
check()
{
    PCB* pr;
    printf("\n **** 当前正在运行的进程是:%s",p->name); /*显示当前运行进程*/
    disp(p);
    pr=ready;
    printf("\n ****当前就绪队列状态为:\n"); /*显示就绪队列状态*/
    while(pr!=NULL)
    {
        disp(pr);
        pr=pr->link;
    }
}

/*建立进程撤消函数(进程运行结束,撤消进程)*/
destroy()
{
    printf("\n 进程 [%s] 已完成.\n",p->name);
    free(p);
}
```

```
/* 建立进程就绪函数(进程运行时间到,置就绪状态)*/
running()
{
    (p->rtime)++;
    if(p->rtime==p->ntime)
        destroy(); /* 调用destroy函数*/
    else
    {
//      (p->super)--;
        p->state='w'; /*进程状态改为就绪态*/
        sort(); /*调用sort函数*/
    }
}

/*主函数*/
main()
{
    int len,h=0; /*声明*/
    char ch; /*声明*/
    input(); /*输入进程相关信息*/
    len=space(); /*len获取就绪队列长度*/
    while((len!=0)&&(ready!=NULL)) /*当就绪队列长度不为0并且就绪队列不为空时*/
    {
        ch=getchar(); /*按任意键继续*/
        h++;          /*系统执行时间+1*/
        printf("\n The execute number:%d \n",h); /*打印当前系统已执行时间*/
        /*执行就绪队列队头的进程*/
        p=ready; /*指针p指向就绪队列队头*/
        ready=p->link; /*就绪队列队头指向下一个PCB*/
        p->link=NULL; /*原先队头的指针域置空*/
        p->state='R'; /*原先队头的进程状态改为运行态*/
        check(); /*调用进程查看函数*/
        running(); /*更新进程的优先级，就绪队列重新排序*/
        printf("\n 按任一键继续......");
        ch=getchar();
    }
    printf("\n\n 进程已经完成.\n");
    ch=getchar();
}
```

3. 解释

   简单轮转法十分简单，只需要在用户输入PCB时，默认进程优先数都为0即可。此外，在输入和输出的过程中不显式地指定优先数。

4. 调式运行

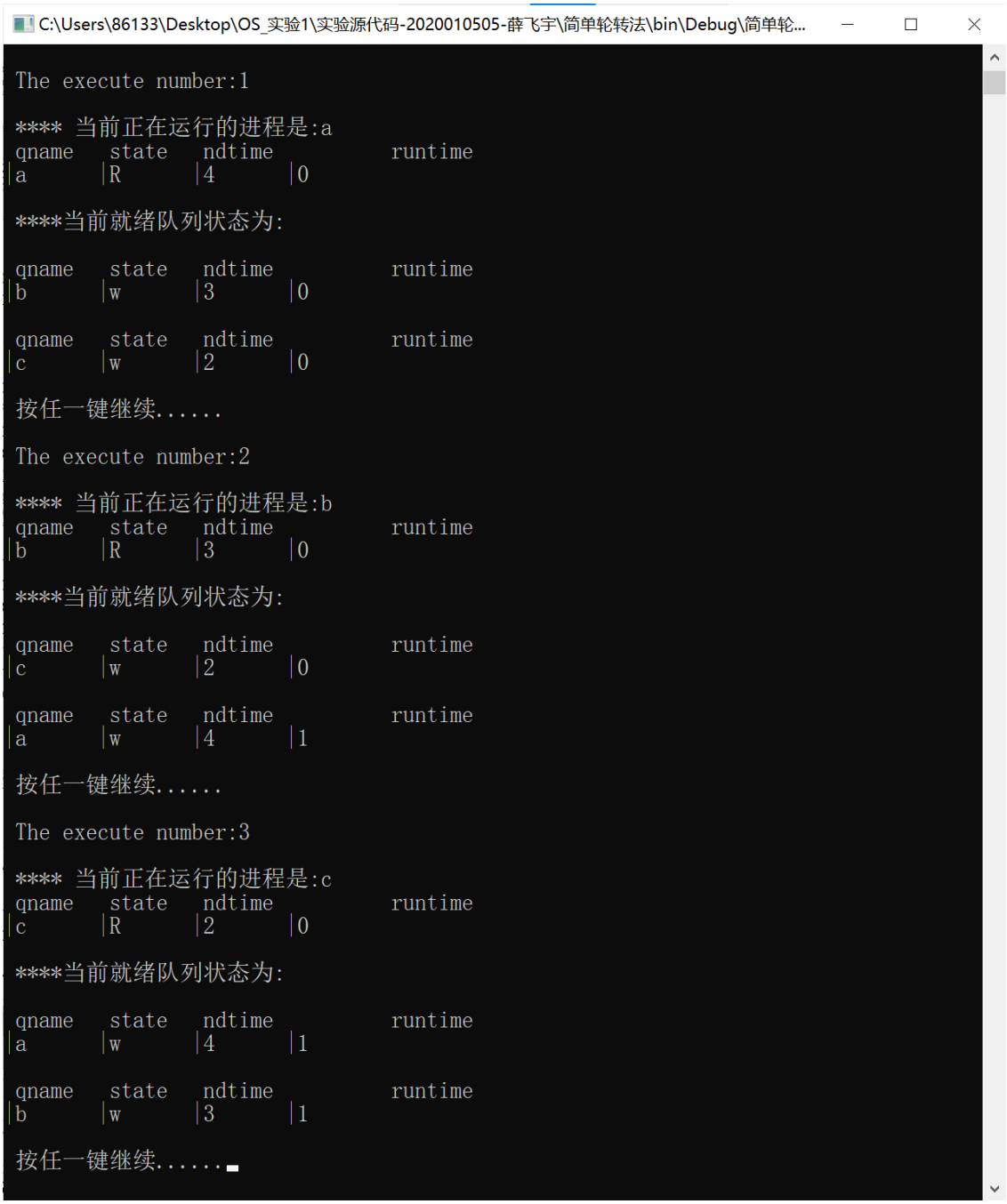
   - 在每个时刻，当前运行的进程都重新放入就绪队列队尾
   - 就绪队列队头的程序分配处理机

   1. 输入3个进程的PCB信息如下

      - 进程a 运行时间4
      - 进程b 运行时间3
      - 进程c 运行时间2

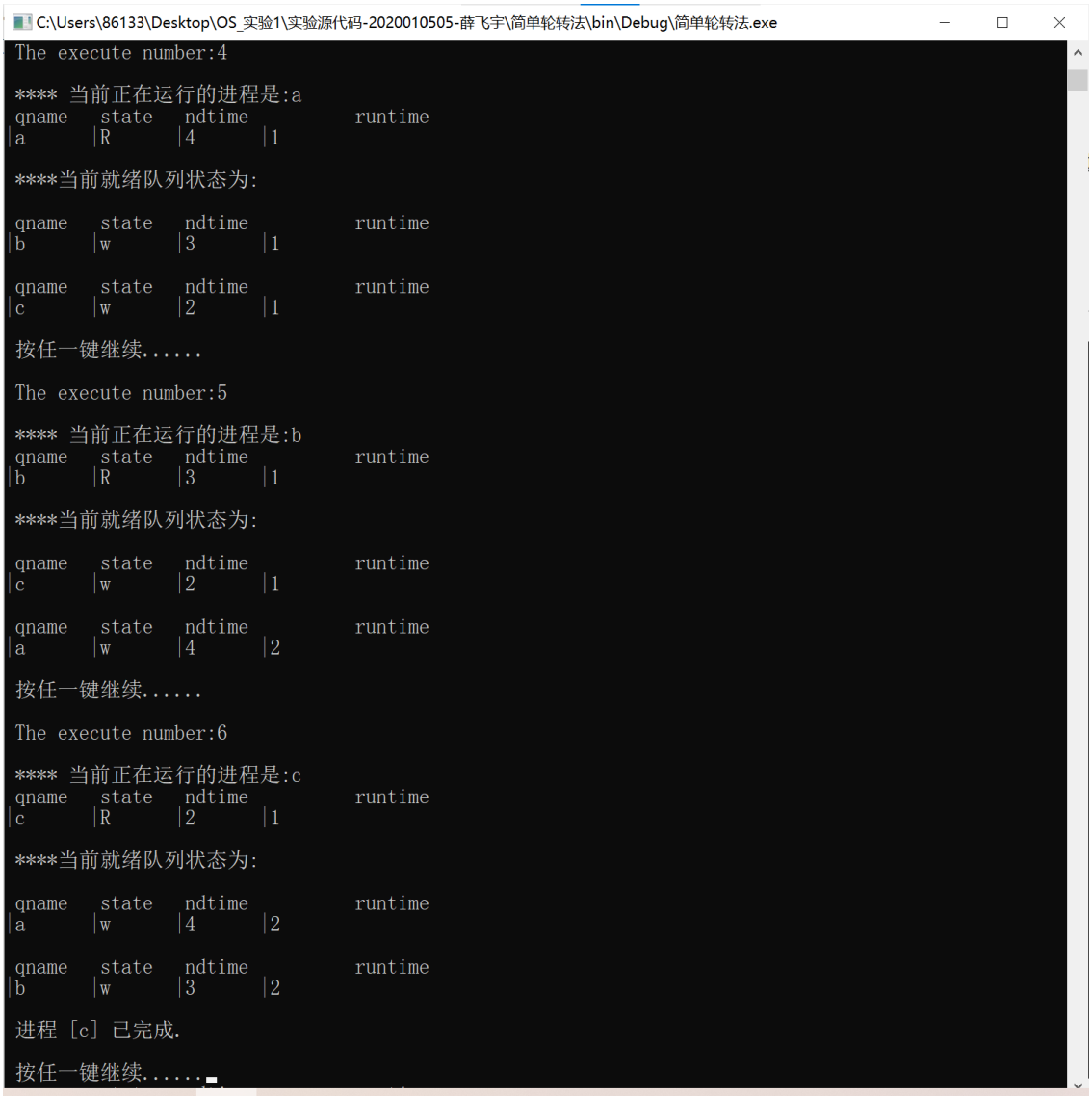
2. 执行时间1-3。

- 时刻1，进程a执行
- 时刻2，进程b执行
- 时刻3，进程c执行



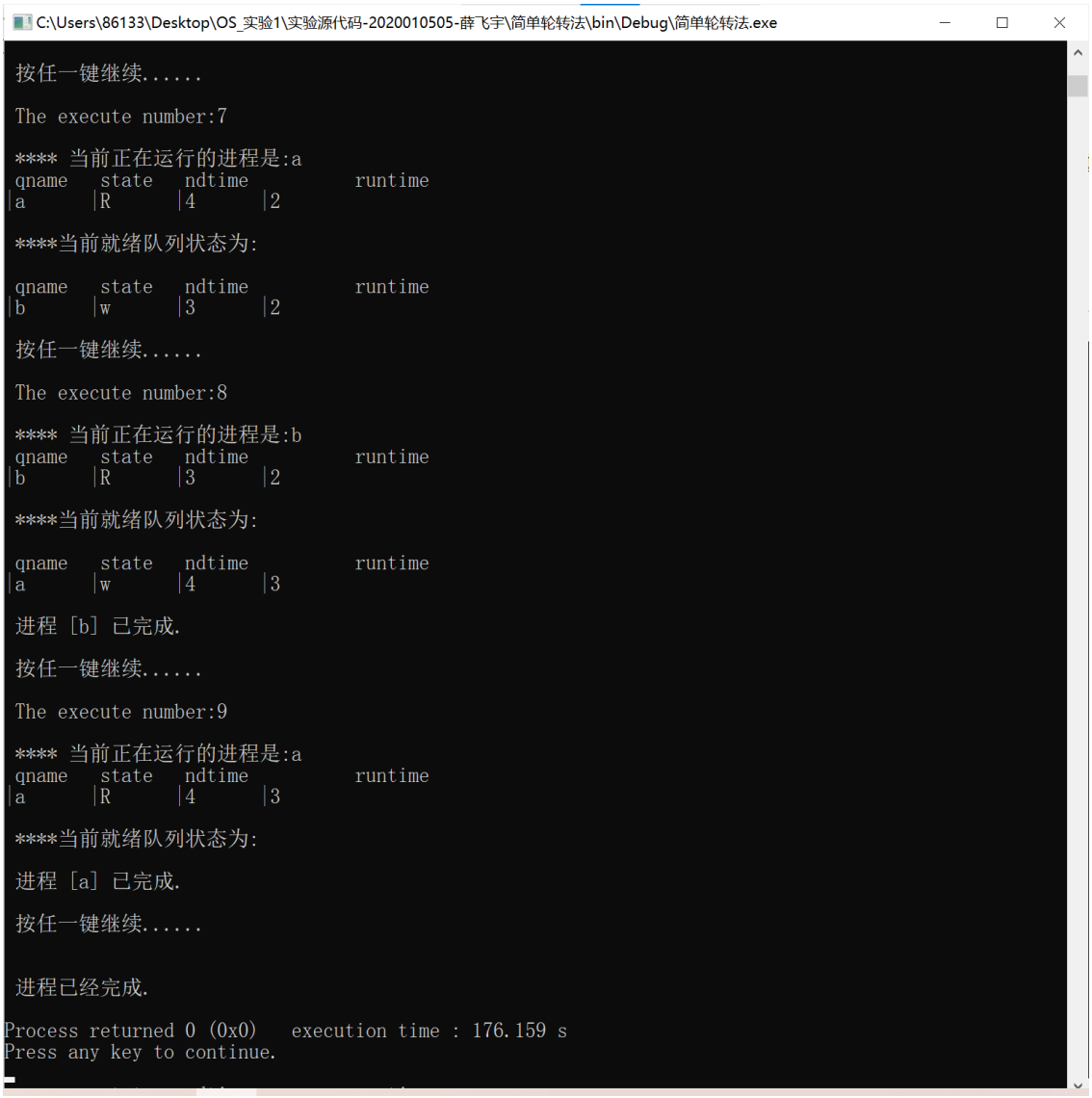
3. 执行时间4-6

- 时刻4，进程a执行
- 时刻5，进程b执行
- 时刻6，进程c执行完毕



4. 执行时间7-9

- 时刻7，进程a执行
- 时刻8，进程b执行完毕
- 时刻9，进程a执行完毕



任务3：阅读Linux源码

schedule() 是 Linux 调度器中最重要的一个函数。它没有参数，没有返回值，却实现了内核中最重要的功能，当需要执行实际的调度时，直接调用 shedule()。

schedule 函数定义在 kernel/sched/core.c 中，源码如下:

```
void __sched schedule(void)
{
    struct task_struct *tsk = current;
    ...
    do {
        preempt_disable();
        __schedule(false);
        sched_preempt_enable_no_resched();
    } while (need_resched());
}
```

在*Linux*中，进程（*Linux*中用轻量级的进程来模拟线程）使用的核心数据结构。一个进程在核心中使用一个*task_struct*结构来表示，包含了大量描述该进程的信息，其中与调度器相关的信息主要包括以下几个:

1. state

volatile long state; / -1 unrunnable, 0 runnable, >0 stopped */*

*Linux*的进程状态主要分为三类：可运行的（*TASK_RUNNING*，相当于运行态和就绪态）；被挂起的（*TASK_INTERRUPTIBLE*、*TASK_UNINTERRUPTIBLE*和*TASK_STOPPED*）；不可运行的（*TASK_ZOMBIE*），调度器主要处理的是可运行和被挂起两种状态下的进程，其中*TASK_STOPPED*又专门用于*SIGSTP*等*IPC*信号的响应，而*TASK_ZOMBIE*指的是已退出而暂时没有被父进程收回资源的"僵死"进程。

2. counter

long counter;

该属性记录的是当前时间片内该进程还允许运行的时间。

事实上，schedule() 函数只是个外层的封装，实际调用的还是__schedule() 函数， __schedule() 接受一个参数，该参数为 bool 型，false 表示非抢占，自愿调度，而 true 则相反。

*schedule()*函数将遍历就绪队列中的所有进程，调用*goodness()*函数计算每一个进程的权值*weight*，从中选择权值最大的进程投入运行。

__schedule 的实现大概可以分为四个部分：

- 针对当前进程的处理
- 选择下一个需要执行的进程
- 执行切换工作
- 收尾工作

针对当前进程的处理

```
static void __schedule(bool preempt)
{
    ...
    prev = rq->curr;
    schedule_debug(prev);
.................................1
    // 禁止本地中断，防止与中断的竞争行为
    local_irq_disable();
    ...
    // 更新本地 runqueue 的 clock 和 clock_task 变量，这两个变量代表 runqueue 的时间.
    update_rq_clock(rq);

    switch_count = &prev->nivcsw;
.................................2
    if (!preempt && prev->state) {
        if (unlikely(signal_pending_state(prev->state, prev))) {
            prev->state = TASK_RUNNING;
        } else {
            deactivate_task(rq, prev, DEQUEUE_SLEEP | DEQUEUE_NOCLOCK);
            prev->on_rq = 0;
            ...
            if (prev->flags & PF_WQ_WORKER) {
                struct task_struct *to_wakeup;

                to_wakeup = wq_worker_sleeping(prev);
                if (to_wakeup)
                    try_to_wake_up_local(to_wakeup, &rf);
            }
        }
        switch_count = &prev->nvcsw;
    }
```

```
}
```

选择下一个需要执行的进程

```
schedule->__schedule->pick_next_task

static inline struct task_struct *
pick_next_task(struct rq *rq,  struct task_struct *prev,  struct rq_flags *rf)
{
    const struct sched_class *class;
    struct task_struct *p;

    if (likely((prev->sched_class == &idle_sched_class ||
            prev->sched_class == &fair_sched_class) &&
           rq->nr_running == rq->cfs.h_nr_running)) {     ........1

        p = fair_sched_class.pick_next_task(rq,  prev,  rf);  .....2
        if (unlikely(p == RETRY_TASK))
            goto again;

        /* Assumes fair_sched_class->next == idle_sched_class */
        if (unlikely(!p))
            p = idle_sched_class.pick_next_task(rq,  prev,  rf);

        return p;
    }

again:
    for_each_class(class) {            ........................3
        p = class->pick_next_task(rq,  prev,  rf);
        if (p) {
            if (unlikely(p == RETRY_TASK))
                goto again;
            return p;
        }
    }
}
```

执行进程切换

```
rq->nr_switches++;
rq->curr = next;
++*switch_count;   // 这个变量是 curr->nivcsw 或者 curr->nvcsw
schedule->__schedule->context_switch:

static __always_inline struct rq *
context_switch(struct rq *rq,  struct task_struct *prev,
           struct task_struct *next,  struct rq_flags *rf)
{
    struct mm_struct *mm,  *oldmm;

    // 切换的准备工作，包括更新统计信息、设置 next->on_cpu 为 1 等。
    prepare_task_switch(rq,  prev,  next);

    // 获取即将切换到的进程的 mm。
```

```
    mm = next->mm;
    // 当前的 active_mm
    oldmm = prev->active_mm;

    // mm = NULL 表示下一个进程还是一个内核线程
    if (!mm) {
        // 如果是内核线程，依旧不需要切换 mm，依旧保存 active_mm
        next->active_mm = oldmm;
        // 相当于添加引用计数：atomic_inc(&mm->mm_count);
        mmgrab(oldmm);
        enter_lazy_tlb(oldmm, next);
    } else
        switch_mm_irqs_off(oldmm, mm, next);

    if (!prev->mm) {
        prev->active_mm = NULL;
        rq->prev_mm = oldmm;
    }

    ...

    rq_unpin_lock(rq, rf);
    spin_release(&rq->lock.dep_map, 1, _THIS_IP_);

    switch_to(prev, next, prev);
    barrier();

    return finish_task_switch(prev);
}
```

收尾工作

```
barrier();
    return finish_task_switch(prev);
```

四、实习总结

本次实习收获颇丰，这也是我第一次亲自模拟写一个操作系统的进程调度的程序。

本次实习中我先设计了“最高优先数优先”调度算法，深入理解了动态优先级和静态优先级的区别，深入理解了进程控制块的定义和作用。在我的模拟程序中进行测试，最终实现了在单处理机环境下对多道程序处理机调度的方法。加深了解处理器调度的工作原理，进一步理解了理解进程控制块PCB的原理及数据结构。然后，根据所学知识，我实现了“简单轮转法调度算法”。最后再阅读了Linux的kernel中的schedule()函数，进一步了解了Linux的进程调度算法。本次实习成功。