电 子 科 技 大 学

实 验 报 告

学生姓名： 陈凯悦	学 号：2016060601020

一、实验课程名称：计算机操作系统

二、实验项目名称：系统化思维模式下计算机操作系统进程与资源管理设计

三、实验目的：

设计和实现进程与资源管理，具体地，编写 test shell，建立系统的进程管理、调度、资源管理和分配的知识体系，进一步理解操作系统进程调度和资源管理的微观原理和宏观技术实现。

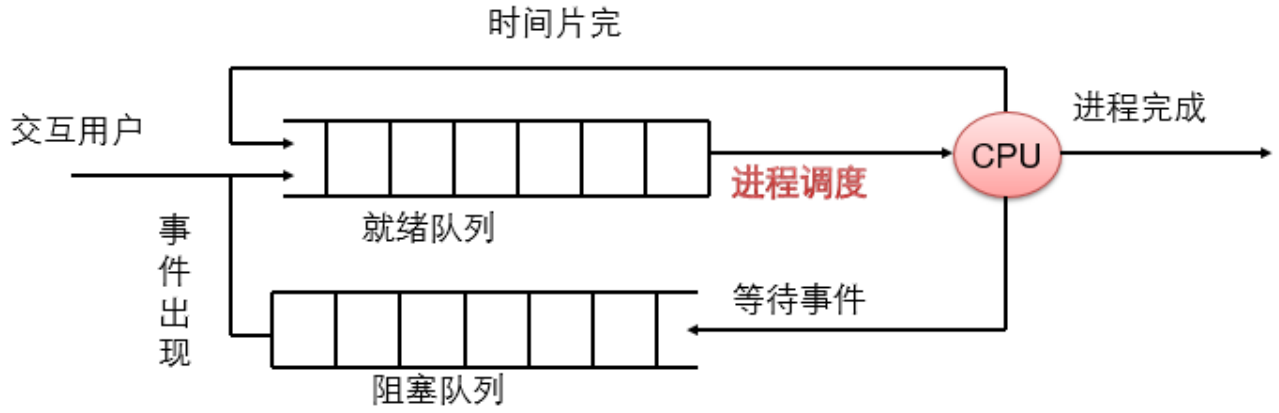
其中，进程和资源管理器的任务：

1. 能够完成进程的控制，包括进程的创建和撤销，进程的状态转换；
2. 能够基于优先级调度算法完成进程的调度，模拟时钟中断，在同优先级进程中采用时间片轮转调度进行调度；
3. 能够完成资源的分配和释放，并完成进程间的同步

Test shell 的任务：完成对用户命令的解释，将用户命令转化为对进程与资源控制的具体操作，（即驱动进程与资源管理器，将用户要求，并将执行结果输出到终端或指定文件中。

四、实验原理：

在仅有进程调度的情况下，系统中存在就绪队列和阻塞队列，其中，不同的优先级有不同的就绪队列，只有高优先级的就绪队列中没有进程时，当前就绪队列中的进程才可以按照 FCFS 原则进行调度。时钟中断时，进程也只能放入其优先级对应的就绪队列队尾。



五、实验内容：

5.1 系统功能需求分析

系统的架构总体架构分为三部分：

终端——>test shell——>进程和资源管理器

第一部分：通过终端（比如键盘输入）或者测试文件来给出相应的用户命令，以及模拟硬件引起的中断，本实验使用 process.txt 文件进行输入。

第二部分：test shell——作为驱动程序，连接终端和进程与资源管理器。

第三部分：进程与资源管理器——属于操作系统内核的功能，完成如下功能：进程创建、撤销和调度；完成多单位资源的管理，申请和释放；完成错误检测和定时器的中断功能。

5.2 总体框架设计

5.2.1 Test shell 的设计

1. Test shell 完成如下任务：

- 从终端或者测试文件读取命令；
- 将用户需求转换为调度内核函数（即进程和资源管理器）；
- 在终端或者输出文件中显示结果，比如当前运行的进程和错误信息等。

2. Test shell 应该读取以下命令，并根据命令调用进程和资源管理器。

- init ：初始化
- cr<process name><process's priority>：创建进程，参数包括进程名和进程的优先级
- de<process name>：撤销进程，参数为进程名
- req<resource name><# of units>：申请资源，参数为资源名和资源的数量
- to：time out，即一个时间片结束

本实验实现的额外功能：

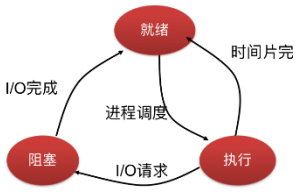
- lp：显示所有进程和它们的状态
- lr：显示所有资源和它们的状态
- pi：显示给定进程的信息

3. Test shell 的输出格式：给出每一时间片内在运行的进程

init x p q r x .....

5.2.2 进程管理的设计

1. 进程状态：三状态的进程状态



包括就绪 ready,阻塞 block，和执行 running

2.进程的操作：

创建 create：无—>ready

撤销 destory：running/ready/blocked—>无

请求资源 request：running—>blocked（当前资源不满足时，进程被阻塞）

释放资源 release：blocked—>ready（因申请资源而被阻塞的进程被唤醒）

时钟中断 time out：running—>ready
调度：ready—>running/running—>ready

3. 进程控制块的结构 PCB

利用 C 语言中的结构 struct 来定义 PCB，List 等。

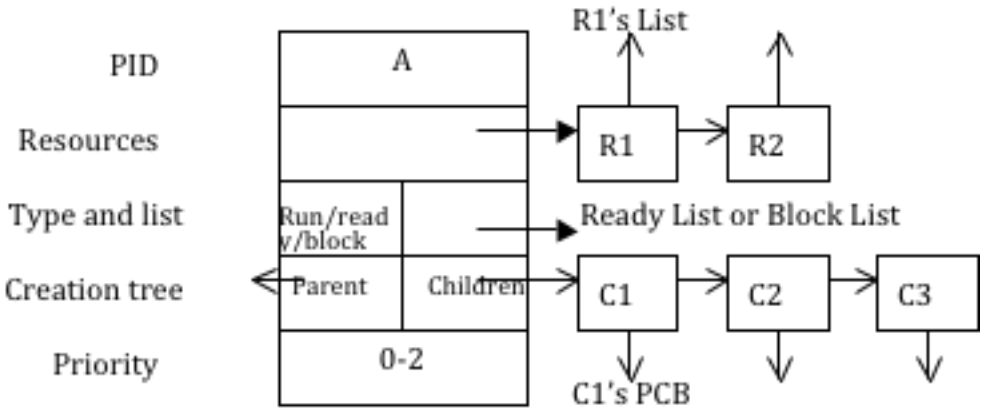
PID（name）

resources //: resource which is occupied

Status: Type & List// type: ready, block, running…., //List: RL(Ready list) or BL(block list)

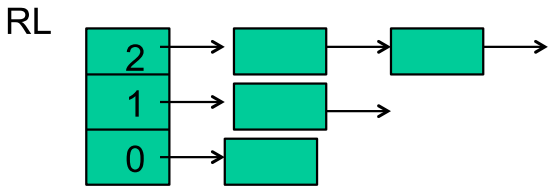
Creation_tree: Parent/Children

Priority: 0, 1, 2 (Init, User, System)



4. 就绪队列 Ready line

有三个具有不同优先级的就绪队列



3 个级别的优先级，且优先级固定无变化

2 = “system”

1 = “user”

0 = “init”

每个 PCB 要么在 RL 中，要么在 block list 中 。当前正在运行的进程，根据优先级，可以将其放在 RL 中相应优先级队列的首部。

5. 调度原则

- 基于 3 个优先级别的调度：2，1，0，必须等高优先级的进程运行结束或进入阻塞队列，低进程才可以运行
- 使用基于优先级的抢占式调度策略，在同一优先级内可以使用时间片轮转（RR）
- 基于函数调用来模拟时间共享
- 初始进程(Init process)具有双重作用：
  虚设的进程：具有最低的优先级，永远不会被阻塞
  进程树的根，第一个进程的父进程
- 进程的调度原则：
  1）如果新创建的进程优先级高于当前优先级，则当前优先级被放回就绪队列的队

尾，当前进程更新为优先级高的进程

2）如果新创建的进程优先级低于或者和当前进程一样，那么当前进程不变，新进程放到其对应的就绪队列中。

3）时钟中断时，将当前进程从其就绪队列队首取出，放入队尾，然后再找优先级最高的就绪队列队首元素进行调度。

5.2.3 资源管理的设计

1. 资源控制块 RCB：利用 C 语言中的结构定义 struct 来定义。

资源的表示：设置固定的资源数量，4 类资源，R1，R2，R3，R4，每类资源 Ri 有 i 个资源控制块 Resource control block (RCB) 如图 5 所示

- RID: 资源的 ID
- Status: 空闲单元的数量
- Waiting_List: list of blocked process

2.资源申请原则

所有的资源申请请求按照 FIFO 的顺序进行，当前进程申请某一时，如果资源及其数量都满足，则更新资源的 RCB 内容和进程的 PCB 内容，当前进程不变，如果不满足，那么当前进程被阻塞，同时修改 PCB 和 RCB 的内容，调度最高优先级的就绪队列的队首进程。

3.资源的释放原则

释放某一资源的某一数量后，需要判断其阻塞队列队首进程是否可以被释放，如果可以，那么将其从阻塞队列中取出，放入就绪队列的队尾；如果不可以，那么只改变该资源的可用资源数，没有阻塞进程被释放。

5.3 系统的模块设计及其调用

本实验共设计了三个.h 文件和三个.cpp 文件，其内容和相互调用关系如下：

头文件

m.h：包含对main.cpp中所有函数的说明

Queue.h：包含PCB和就绪/阻塞队列Queue的定义，Queue.cpp中所有函数的说明

resQueue.h：包含RCB和资源队列resQueue的定义，resQueue.cpp中所有函数的说明

源文件

main.cpp：包含进程的初始化，创建，申请，时间片中断，调度等

Queue.cpp：包含对就绪/阻塞队列的所有操作

resQueue.cpp：包含对资源队列的所有操作

其中 main 函数还负责 test shell 的读文件和相关操作，通过 main 中的 test shell 模块从测试文件中读入各种命令。

然后，度读入的命令进行数据类型的装换，实现正确地传参，进而调用内核函数，执行进程管理和内存管理。

最后，在中断输出每一时刻正在运行的进程名称。

main.cpp 中的函数有：

```
void initrcb();//初始化 RCB
PCB initpcb();//初始化 PCB
//根据已知条件 创据已知 新的 进的进程，并将其 父进进程的孩子队列和 绪队列中
void createNewProcess(string name,int priority,PCB * parent);
//根据 进据进程名和优先 插销销该进程，并将其 有孩子 进孩子进程杀死 了递递
//由于 每于 PCB，要 么要么在就绪队列 么在阻塞队列中，所以 在这这两个队列中查询 程名 为name的PCB
void display_inf(string name);
int Sear_from_Ready(string name);
int Sear_from_Blocked(string name);
void C_Destory(string name);
void Destory(string name);
//在就 绪就绪队列中寻找 字的 进的 PCB，找到返回 1，将其从就 绪将其从就 除，否 则，否 0
int Destory_from_readyQ(string name);
//在阻塞 队阻塞队列中寻 字的 进的 PCB，找到返回 1，将其从阻塞 队将其从 除，否 则，否 0
int Destory_from_blockedQ(string name);
//释放给定资源队列的所 有资资
void release_res_list(resQueue* r);
//释放该进程的所有资源 ，并依次撤 销并依次撤
void Kill_Tree(PCB killed_pcb);
//根据申 请据申请的资源 量进进行分
bool RequestRes(string rid,int num);
//根据 资据 id和其数量 释其数量
int C_Release(string rid,int num);
void Release(string rid,int num);
//找到需要 调到需要调
void select_schedul_pro(Queue* q);
int search_maxprio();
void Scheduler();
//当申 请申请资源时，专 计的调度函数
void resScheduler();
//显示当前资源状态
void display_rcb();
//时间片轮转
void Time_out();
//主函数， 负函数，负责 函数
int main(int argc,char * argv[]);
```

Queue.cpp 中的函数有：

void QueueInit(Queue queue);//初始化*
void QueuePush(Queue queue,QDataTypeval);//插入，由于是队入，由能从队从队尾插*
void QueuePop(Queue queue);//删除，只能从队首删*
void QueueDelete(Queue queue,QDataTypeval);//从队队列中删除指定结*
QDataType QueueFront(const Queue queue);//返回队回队首结点*
int QueueEmpty(const Queue queue);//判断队断队列是否*
int QueueSize(const Queue queue);//返回队回队列结点的*
void QueuePrint(const Queue queue);//打印*

同样地，resQueue.cpp 中的函数有：

void resQueueInit(resQueue queue);//初始化*
void resQueuePush(resQueue queue,RCB val);//插入，由于是队入，由能从队从队尾插*
void resQueuePop(resQueue queue);//删除，只能从队首删*
void resQueueDelete(resQueue queue,RCB val);//从队队列中删除指定结*
RCB resQueueFront(const resQueue queue);//返回队回队首结点*
int resQueueEmpty(const resQueue queue);//判断队断队列是否*
int resQueueSize(const resQueue queue);//返回队回队列结点的*
void resQueuePrint(const resQueue queue);//打印*

六、实验器材（环境配置）：

编译软件：VS2013

七、详细实现——实验步骤及操作：

7.1 数据结构的设计

（1）进程状态：

```
typedef struct STATE {
    int     type;
    struct RCB  *RorB_List;
    int     BlockedNum;
}STATE;
```

Type：RUNNING（0），BLOCKED（1），READY（2），DESTORYED（3）；

（2）PCB 结构

```
typedef struct PCB {
    char        name[64];
    int     pid;//进程号
    int     priority;//优先级
    struct STATE    state;
    struct resQueue *res_list;//已拥有的资源队列
    struct PCB  *parent_pcb;    /* 父进程 */
    struct Queue    *child_list;    /* 子进程 */
}QDataType;
```

（3）进程队列的定义

```
typedef struct QNode {
    QDataType   val;
    struct QNode    *next;
} QNode;
typedef struct Queue {
    int size;   /* 队列元素个数 */
    QNode   *front; /* 指向队列第一个元素，如果队列为空，则等于 NULL */
    QNode   *rear;  /* 指向队列最后一个元素，如果队列为空，则等于 NULL */
} Queue;
```

（4）RCB 结构

```
typedef struct RCB {
    char        rid[64];
    int     init_amount;//初始数量
    int     avail_amount;//可用数量
    struct Queue    *waiting_queue; /* 资源的等候队列 */
```

```
}RCB;
```

（5）资源队列的定义

```
typedef struct resQNode {
    RCB     val;
    int     own_res_num;//已拥有该资源的数量
    struct resQNode *next;
} resQNode;
typedef struct resQueue {
    int     size;   /* 队列元素个数 */
    resQNode    *front; /* 指向队列第一个元素，如果队列为空，则等于 NULL */
    resQNode    *rear;  /* 指向队列最后一个元素，如果队列为空，则等于 NULL */
} resQueue;
```

7.2 进程管理的函数设计

进程管理部分包含进程的初始化，新进程的创建，时间片轮转，调度以及进程的撤销。

7.2.1 创建进程的函数设计

在创建新进程前，要创建一个优先级为 0 的初始化进程，首先要对进程的 PCB 进行初始化，初始化的函数为：

```
struct PCB *newpcb = (PCB*)malloc(sizeof(PCB));//全局变量
//初始化 PCB
PCB initpcb(){
    strcpy(newpcb->name, "init");
    newpcb->pid = pid;
    newpcb->priority = init;
    newpcb->state.type = READY;
    newpcb->state.RorB_List = (RCB*)malloc(sizeof(RCB));
    strcpy(newpcb->state.RorB_List->rid, "R0");
    newpcb->state.RorB_List->avail_amount = 0;
    newpcb->state.RorB_List->init_amount = 0;
    newpcb->state.RorB_List->waiting_queue = (Queue*)malloc(sizeof(Queue));
    QueueInit(newpcb->state.RorB_List->waiting_queue);
    //没有阻塞，所以阻塞队列为0号资源（空资源）
    newpcb->state.BlockedNum = 0;
    newpcb->res_list = (resQueue*)malloc(sizeof(resQueue));
    resQueueInit(newpcb->res_list);
    newpcb->parent_pcb = (PCB*)malloc(sizeof(PCB));
    newpcb->child_list = (Queue*)malloc(sizeof(Queue));
    QueueInit(newpcb->child_list);
    return *newpcb;
}
```

初始化新进程之后，根据从 txt 中读入的信息，更改初始化的 PCB 的信息，得到新的 PCB，代码如下：

```
1.  //根据已知条件创建一个新的进程，并将其加入父进程的孩子队列和就绪队列中
2.  void createNewProcess(char name[64], int priority, PCB *parent){
3.      //parent 为创建新进程时的当前进程
4.      pid++;
5.      PCB temp_pcb = initpcb();
6.      //初始化 pcb 的参数
7.      strcpy(temp_pcb.name, name);
8.      temp_pcb.pid = pid;
9.      temp_pcb.priority = priority;
10.     temp_pcb.parent_pcb = parent;
11.     //将产生的进程加入当前进程的子进程队列中
12.     QueuePush(parent->child_list, temp_pcb);
13.     //将新产生的进程放入就绪队列中
14.     QueuePush(Ready_queue[priority], temp_pcb);
15.     Scheduler();
16. }
```

7.2.2 时间片轮转的函数设计

```
1.  void Time_out(){
2.      int i = currentProcess.priority;
3.      PCB p = currentProcess;
4.      QueuePop(Ready_queue[i]);
5.      p.state.type = 2;
6.      QueuePush(Ready_queue[i], p);
7.      Ready_queue[i]->front->val.state.type = 0;
8.      currentProcess = Ready_queue[i]->front->val;
9.      Scheduler();
10. }
```

此函数实现了，将当前进程从就绪队列队首取出，放入就绪队列的队尾的操作，并找到当前进程，进行调度。

7.2.3 进程调度的函数设计

此模块包含三个函数：select_schedul_pro 函数负责判断当前进程是否需要被更换，并输出当前进程的进程名；search_maxprio 函数负责找到当前优先级最高的就绪队列;Schedul 函数负责将上述两个函数包装起来进行调用。

```
1.  //找到需要调度的进程
2.  void select_schedul_pro(Queue *q){
3.      QNode *p = q->front;
4.      if (currentProcess.priority < p->val.priority || currentProcess.state.type != 0 || cur
    rentProcess.state.type == 3 || currentProcess.pid == 0)
```

```
    {
        if (currentProcess.name == "init")
        {
            QueuePop(q);
        }
        int j = currentProcess.priority;
        PCB temp = Ready_queue[j]->front->val;
        QueuePop(Ready_queue[j]);
        temp.state.type = 2;
        QueuePush(Ready_queue[j], temp);
        //printf("%d", currentProcess.state.type);
        p->val.state.type = 0;//RUNNING=0
        currentProcess = p->val;
    }
    printf("%s ", currentProcess.name);
}
int search_maxprio(){
    if (Ready_queue[2]->size != 0 && Ready_queue[2]->front->val.priority == 2 && Ready_que
ue[2]->rear->val.priority == 2 && Ready_queue[2]->front->val.state.type != 1)
    {
        return 2;//currentProcess = Ready_queue[2]->front->val;
    }
    else if (Ready_queue[1]->size != 0)
    {
        //currentProcess = Ready_queue[1]->front->val;
        return 1;
    }
    return 0;
}
void Scheduler(){
    int i = search_maxprio();
    select_schedul_pro(Ready_queue[i]);
}
```

7.2.4 撤销进程的函数设计

主函数为 C_Destory()，它调用了 Destory 函数，由于进程要么在就绪队列中要么在阻塞队列中，Destory 函数负责从就绪队列或者阻塞队列中寻找要撤销的进程，找到之后将其从所在队列中删除，释放其占有的资源，释放因为这些资源而阻塞的进程，同时用递归的算法将其子进程同样撤销。

```
//根据进程名和优先级，插销该进程，并将其所有孩子进程杀死，用到了递归
//由于每个 PCB，要么在就绪队列，要么在阻塞队列中，所以在这两个队列中查询进程名为 name 的 PCB
void C_Destory(string name){
    Destory(name);
```

```
    int i = search_maxprio();
    Ready_queue[i]->front->val.state.type = 0;
    currentProcess = Ready_queue[i]->front->val;
    Scheduler();
}
void Destory(string name){
    int i = Destory_from_readyQ(name);
    if (i == 0){
        int j = Destory_from_blockedQ(name);
    }
}
//以下为 Destory 的子函数
//在就绪队列中寻找该名字的进程 PCB，找到返回 1，将其从就绪队列中移除，否则返回 0
int Destory_from_readyQ(string name){
    QNode *temp_node;
    for (int prio = 0; prio < 3; prio++)
    {
        temp_node = Ready_queue[prio]->front;
        while (temp_node != NULL)
        {
            if (temp_node->val.name == name)
            {
                temp_node->val.state.type = 3;
                Kill_Tree(temp_node->val);
                QueueDelete(Ready_queue[prio], temp_node->val);
                return 1;
            }
            else
            {
                temp_node = temp_node->next;
            }
        }
    }
    return 0;
}
//在阻塞队列中寻找该名字的进程 PCB，找到返回 1，将其从阻塞队列中移除，否则返回 0
int Destory_from_blockedQ(string name){
    QNode *temp_node;
    temp_node = Blocked_queue->front;
    while (temp_node != NULL)
    {
        if (temp_node->val.name == name)
        {
            for (int i = 1; i < 5; i++)
```

```
            {
                if (myrcb[i]->rid == temp_node->val.state.RorB_List->rid)
                    QueueDelete(myrcb[i]->waiting_queue, temp_node->val);
            }
            temp_node->val.state.type = 3;
            Kill_Tree(temp_node->val);
            QueueDelete(Blocked_queue, temp_node->val);
            return 1;
        }
        else
        {
            temp_node = temp_node->next;
        }
    }
    return 0;
}
//释放该进程的所有资源，并依次撤销其子进程
void Kill_Tree(PCB killed_pcb){
    release_res_list(killed_pcb.res_list);//释放该进行的所有资源
    Queue *temp_childQ = killed_pcb.child_list;//该进程的孩子进程队列
    while (temp_childQ->front != NULL)
    {
        Destory(temp_childQ->front->val.name);
        QueuePop(temp_childQ);
    }
}
```

7.3 资源管理的函数设计

此模块包含资源的申请，释放函数以及适用于资源管理的调度函数。

7.3.1 资源初始化

本实验设置了一个 0 号资源，其初始数量为 0，每次初始化 PCB 时都会给其分配一个 0 号资源，因此需要初始化 5 个 RCB，如下：

```
RCB *myrcb[5];//全局变量

//初始化 RCB
void initrcb(){
    myrcb[0] = (RCB*)malloc(sizeof(RCB));
    strcpy(myrcb[0]->rid, "R0");
    myrcb[0]->init_amount = 0;
    myrcb[0]->avail_amount = 0;
    myrcb[0]->waiting_queue = (Queue*)malloc(sizeof(Queue));
    QueueInit(myrcb[0]->waiting_queue);
```

```
11.
12.     myrcb[1] = (RCB*)malloc(sizeof(RCB));
13.     strcpy(myrcb[1]->rid, "R1");
14.     myrcb[1]->init_amount = 1;
15.     myrcb[1]->avail_amount = 1;
16.     myrcb[1]->waiting_queue = (Queue*)malloc(sizeof(Queue));
17.     QueueInit(myrcb[1]->waiting_queue);
18.
19.     myrcb[2] = (RCB*)malloc(sizeof(RCB));
20.     strcpy(myrcb[2]->rid, "R2");
21.     myrcb[2]->init_amount = 2;
22.     myrcb[2]->avail_amount = 2;
23.     myrcb[2]->waiting_queue = (Queue*)malloc(sizeof(Queue));
24.     QueueInit(myrcb[2]->waiting_queue);
25.
26.     myrcb[3] = (RCB*)malloc(sizeof(RCB));
27.     strcpy(myrcb[3]->rid, "R3");
28.     myrcb[3]->init_amount = 3;
29.     myrcb[3]->avail_amount = 3;
30.     myrcb[3]->waiting_queue = (Queue*)malloc(sizeof(Queue));
31.     QueueInit(myrcb[3]->waiting_queue);
32.
33.     myrcb[4] = (RCB*)malloc(sizeof(RCB));
34.     strcpy(myrcb[4]->rid, "R4");
35.     myrcb[4]->init_amount = 4;
36.     myrcb[4]->avail_amount = 4;
37.     myrcb[4]->waiting_queue = (Queue*)malloc(sizeof(Queue));
38.     QueueInit(myrcb[4]->waiting_queue);
39.     //printf("RCB 初始化完成\n");
40. }
```

7.3.2 资源申请的函数设计

由于申请资源的函数 RequestRes 中已经对当前进程进行了更新，因此最大优先级就绪队列就是当前进程所在的队列，因此这里对调度函数进行了改进，如下为专门为资源申请设计的调度函数：

```
1.  void resScheduler(){
2.      select_schedul_pro(Ready_queue[currentProcess.priority]);
3.  }
```

以下函数为申请资源的函数：包括对资源可用数量的判断和更新，对当前进程状态的更新，以及对就绪队列和阻塞队列内容的更新：

```
1.  //根据申请的资源及其数量进行分配
2.  bool RequestRes(string rid, int num){
```

```
    for (int i = 1; i < 5; i++)
    {
        if (rid == myrcb[i]->rid)
        {
            int availNum = myrcb[i]->avail_amount - num;
            if (availNum < 0){//资源不够，则不分配
                currentProcess.state.type = BLOCKED;
                //当前进程阻塞的资源为 myrcb[i],阻塞的数量为 num
                currentProcess.state.RorB_List = myrcb[i];
                currentProcess.state.BlockedNum = num;
                QueuePop(Ready_queue[currentProcess.priority]);//当前进程从就绪队列中移除
                QueuePush(Blocked_queue, currentProcess);//将进程放入阻塞队列中
                QueuePush(myrcb[i]->waiting_queue, currentProcess);
                int currID = search_maxprio();
                Ready_queue[currID]->front->val.state.type = 0;
                currentProcess = Ready_queue[currID]->front->val;
                resScheduler();
                return false;//可用资源不够，阻塞
            }
            else
            {
                myrcb[i]->avail_amount = availNum;
                resQueuePush(currentProcess.res_list, *myrcb[i]);//当前进程所拥有的资源队列增加一项
                currentProcess.res_list->rear->own_res_num = num;//当前进程拥有此类新增资源的数量
                resScheduler();
                return true;
            }
        }
    }
    return false;
}
```

7.3.3 资源释放的函数设计

释放资源的同时要释放由于该资源而阻塞的进程，C_Release 函数如下，它包含对资源数量和状态的更新，对相关进程的状态的更新，以及当前进程的调度，调用了 Request 函数：

```
int C_Release(string rid, int num){
    resQNode *res=currentProcess.res_list->front;
    while (res!=NULL)
    {
        if (res->val.rid==rid)
        {
```

```
            res->own_res_num = res->own_res_num - num;
            if (res->own_res_num==0)
            {
                resQueueDelete(currentProcess.res_list, res->val);
            }
            break;
        }
        res = res->next;
    }
    Release(rid, num);
    Scheduler();
    return 1;
}
```

Request 函数用来根据输入的资源名和资源数量进行释放，被 C_Request 函数调用，实现的具体的操作如下：

```
//根据资源 id 和其数量释放资源
void Release(string rid, int num){
    for (int i = 1; i < 5; i++)
    {
        if (myrcb[i]->rid == rid)
        {
            //resQueueDelete(currentProcess.res_list, *myrcb[i]);
            myrcb[i]->avail_amount = myrcb[i]->avail_amount + num;
            while (myrcb[i]->waiting_queue->size != 0 && myrcb[i]->waiting_queue->front->val.state.BlockedNum <= myrcb[i]->avail_amount)
            {
                //如果该资源的等待队列中，有进程不再被阻塞
                //资源数量减少
                myrcb[i]->avail_amount = myrcb[i]->avail_amount - myrcb[i]->waiting_queue->front->val.state.BlockedNum;
                //进程需要的资源数
                int m = myrcb[i]->waiting_queue->front->val.state.BlockedNum;
                //找到等待队列中的进程 PCB
                PCB q = QueueFront(myrcb[i]->waiting_queue);
                //将该进程从资源的等待队列中移除
                QueuePop(myrcb[i]->waiting_queue);
                //将该进程从阻塞队列中移除
                if (q.pid == Blocked_queue->front->val.pid&&q.state.type == 1)
                {
                    QueuePop(Blocked_queue);
                }
                else
```

```
                {
                    QueueDelete(Blocked_queue, q);
                }
                //改变该进程的状态
                q.state.type = READY;
                q.state.RorB_List = myrcb[0];
                q.state.BlockedNum = 0;
                //将获得的资源放入该进程的资源列表中
                resQueuePush(q.res_list, *myrcb[i]);
                q.res_list->rear->own_res_num = m;
                //将该进程放入就绪队列中
                QueuePush(Ready_queue[q.priority], q);
            }
        }
    }
}
```

7.4 test shell 的设计

主函数为：

int main(int argc, char **argv)；即从命令行传递参数，需要将 txt 文件的路径传入。

格式为：process.exe 文件路径

从 txt 文件中读取命令并解析，将正确的参数传给上述的进程和资源管理函数，按照命令的读取顺序依次执行，test shell 的代码如下：

```
FILE *fp;
    if ((fp = fopen(argv[1], "rt")) == NULL){
        printf("Cannot open file, press any key to exit!\n");
        getchar();
        //return 0;
        exit(1);}
    char str[N+1];
    printf("init ");
    while (!feof(fp))//如果文件没有到达末尾则循环
        {fgets(str, N, fp);//读取一行
        if (&str[6] == "\n" || str[2] == '\n' || str[8] == '\n' || str[4] == '\n')
             str[strlen(str) - 1] = 0;  //除了最后一行，都去掉回车符
        char *ptr = NULL;
        char *op = NULL;
        char *n = NULL;
        char *a = NULL;
        ptr = strtok(str, " ");
        int f = 0;
        while (ptr != NULL) {
```

```
            if (f == 0) op = ptr;
            if (f == 1) n = ptr;
            if (f == 2) a = ptr;
            ptr = strtok(NULL, " ");
            f++;}
        if (strcmp(op, "cr") == 0)
            createNewProcess(n, atoi(a), ¤tProcess);
        if (strcmp(op, "req") == 0)
            RequestRes(n, atoi(a));
        if (strcmp(op, "rel") == 0)
            C_Release(n, atoi(a));
        if (strcmp(op, "de") == 0)
            C_Destory(n);
        if (strcmp(op, "to") == 0)
            Time_out();
        if (strcmp(op, "lr") == 0)
        {
            printf("\n\n 当前所有资源的状态为\n");
            display_rcb();
        }
        if (strcmp(op, "lp") == 0)
        {
            printf("\n\n 当前所有进程名和进程的状态为\n");
            int curp = currentProcess.priority;
            printf("就绪队列为(优先级分别为 0，1，2)\n");
            for (int i = 0; i <= curp; i++){
            QueuePrint(Ready_queue[i]);
            }
            printf("阻塞队列为\n");
            QueuePrint(Blocked_queue);
        }
        if (strcmp(op, "pi") == 0)
            display_inf(n);
    }
    fclose(fp);
```

7.5 额外功能的实现

功能一：显示所有资源及其状态【lr】

本实验中资源较少，通过依次输出实现即可：

```
void display_rcb(){
    printf("资源 ID:%s,初始数量:%d,可用数
量:%d\n", myrcb[1]->rid, myrcb[1]->init_amount, myrcb[1]->avail_amount);
```

```
3.      printf("资源 ID:%s,初始数量:%d,可用数
    量:%d\n", myrcb[2]->rid, myrcb[2]->init_amount, myrcb[2]->avail_amount);
4.      printf("资源 ID:%s,初始数量:%d,可用数
    量:%d\n", myrcb[3]->rid, myrcb[3]->init_amount, myrcb[3]->avail_amount);
5.      printf("资源 ID:%s,初始数量:%d,可用数
    量:%d\n", myrcb[4]->rid, myrcb[4]->init_amount, myrcb[4]->avail_amount);}
```

功能二：显示所有进程及其状态【lp】

任何一个未被销毁的进行，要么在就绪队列中，要么在阻塞队列中，因此显示所有进程可以通过打印每一个就绪队列和阻塞队列来实现。

```
1.  if (strcmp(op, "lp") == 0)
2.          {
3.              printf("\n\n 当前所有进程名和进程的状态为\n");
4.              int curp = currentProcess.priority;
5.              printf("就绪队列为(优先级分别为 0，1，2)\n");
6.              for (int i = 0; i <= curp; i++){
7.              QueuePrint(Ready_queue[i]);
8.              }
9.              printf("阻塞队列为\n");
10.             QueuePrint(Blocked_queue);}
```

功能三：显示指定进程的信息和状态【pi】:

同样地，进程在就绪队列或者阻塞队列中，可以先在就绪队列中查找，若没有查找到，则在阻塞队列中找，若找到了，则输出其进程 ID，状态，优先级和子进程等信息：

```
1.  void display_inf(string name){
2.      int i = Sear_from_Ready(name);
3.      if ( i==0)
4.      {//如果不再就绪队列中，一定在阻塞队列中
5.          int j = Sear_from_Blocked(name);
6.      }
7.  }
8.  int Sear_from_Ready(string name){//在就绪队列中寻找
9.      QNode *temp_node;
10.     for (int prio = 0; prio < 3; prio++)
11.     {
12.         temp_node = Ready_queue[prio]->front;
13.         while (temp_node != NULL)
14.         {
15.             if (temp_node->val.name == name)
16.             {
17.                 printf("\n 进程%s 的信息如下： \n", temp_node->val.name);
18.                 printf("进程 ID： %d\n 状态： %d\n 优先
    级： %d\n", temp_node->val.pid, temp_node->val.state.type, temp_node->val.priority);
```

```
                printf("状态的含义运行为 0;阻塞为 1;就绪为 2，销毁为 3\n");
                printf("当前进程的子进程为：\n");
                QueuePrint(currentProcess.child_list);
                return 1;
            }
            else
            {
                temp_node = temp_node->next;
            }
        }
    }
    return 0;
}
int Sear_from_Blocked(string name){//在阻塞队列中寻找
    QNode *temp_node;
    temp_node = Blocked_queue->front;
    while (temp_node != NULL)
    {
        if (temp_node->val.name == name)
        {
            printf("\n 进程%s 的信息如下：/n", name);
            printf("进程 ID：%d\t 状态：%d\t 优先
级：%d\t", temp_node->val.pid, temp_node->val.state.type, temp_node->val.priority);
            printf("当前进程的子进程为：\n");
            QueuePrint(currentProcess.child_list);
            return 1;
        }
        else
        {
            temp_node = temp_node->next;
        }
    }
    return 0;
}
```

7.5 txt 文件的内容格式要求

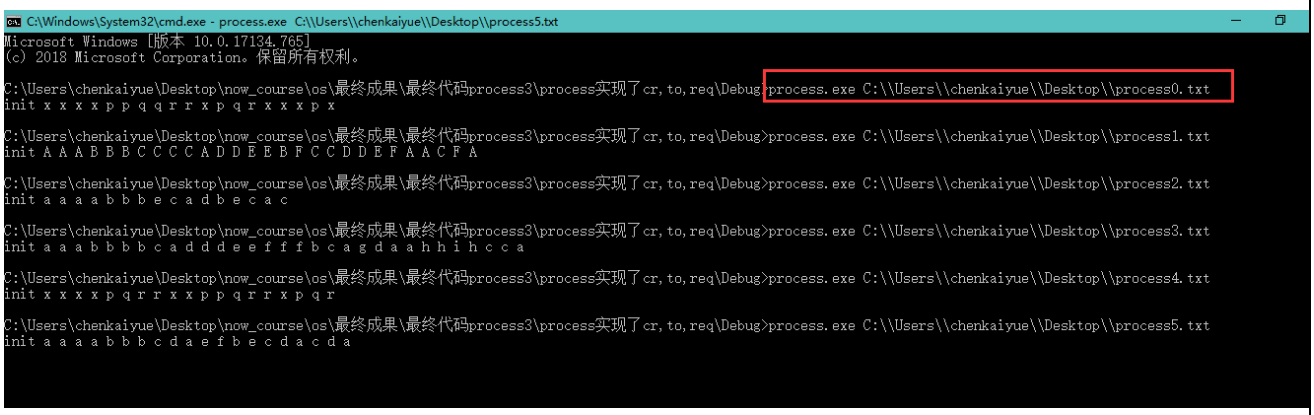
如下图，txt 文本文件要求每一条命令占一行，命令内部的操作和对象都要用空格隔开，最后一条语句不加回车符，如图所示：

process0.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(

```
cr x 1
cr p 1
cr q 1
cr r 1
to
req R2 1
to
req R3 3
to
req R4 3
to
to
req R3 1
req R4 2
req R2 2
to
de q
to
to
```

从命令行执行 exe 文件的格式：在 exe 文件所在目录下，输入 exe 文件名+空格+txt 文件路径。

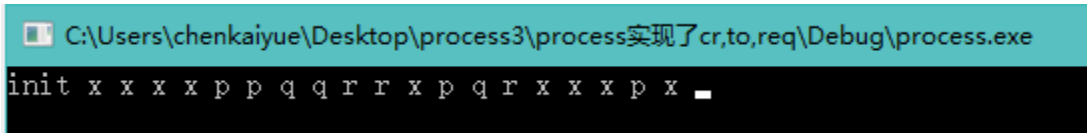


八、测试用例及结果分析：

用 5 个测试用例进行测试，得到的结果与理论结果一致，证明程序正确。

8.1 测试用例 1

第一个测试用例同 7.5 中的用例，输出的结果为：



与理论输出一致。

8.2 测试用例 2

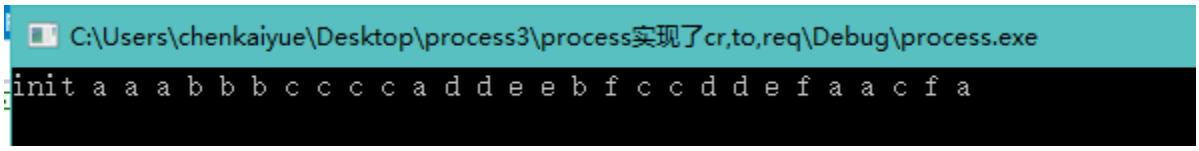
第二个测试用例为：

process1.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看

```
cr a 1
cr b 1
cr c 1
to
cr d 1
cr e 1
to
cr f 1
req R1 1
req R2 2
to
req R2 1
req R3 3
to
req R4 4
to
req R3 2
to
rel R2 1
to
rel R3 2
to
to
req R3 3
de b
to
to
to
```

输出结果为：



与理论输出一致。

8.3 测试用例 3

输入的用例为：

process2.txt - 记事本

文件(F) 编辑(E) 格式(O)

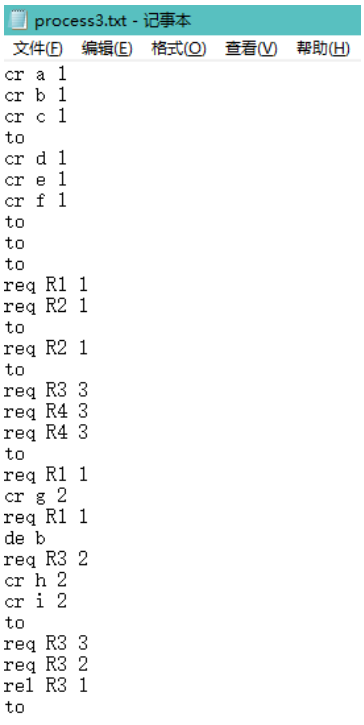
```
cr a 1
cr b 1
cr c 1
req R1 1
to
cr d 1
req R2 2
cr e 2
req R2 1
to
to
to
rel R2 1
de b
to
to
```

得到的输出为：

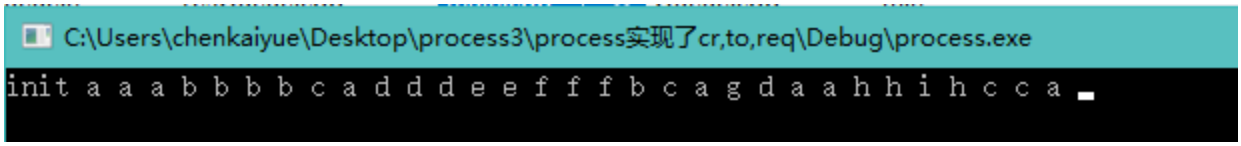
与理论输出一致。

8.4 测试用例 4

输入的用例为：

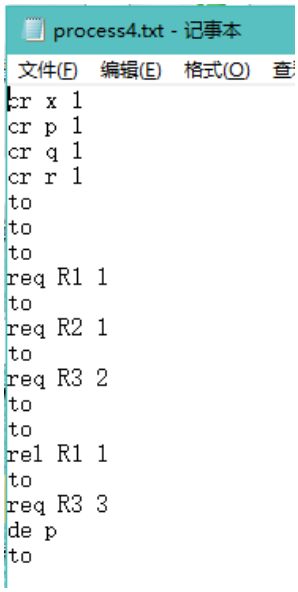


输出的结果为：

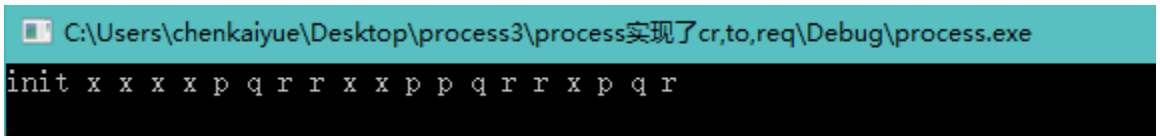


8.5 测试用例 5

测试用例为：

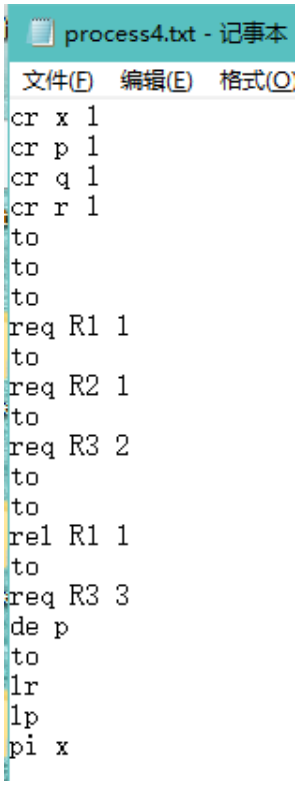


输出结果为：

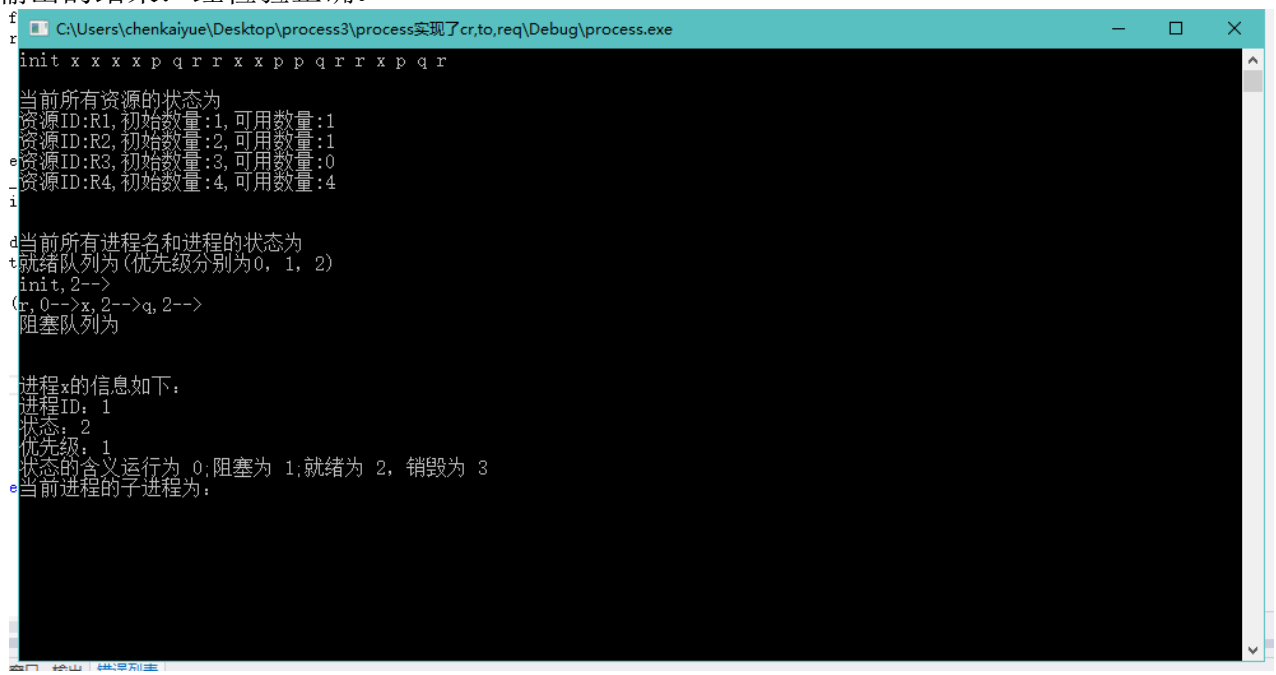


与理论输出一致

当增加额外的功能 lr,lp 和 pi 时，例如下面的命令，可以实时看到当前进程和资源的状态：



输出的结果：经检验正确。



8.6 数据分析

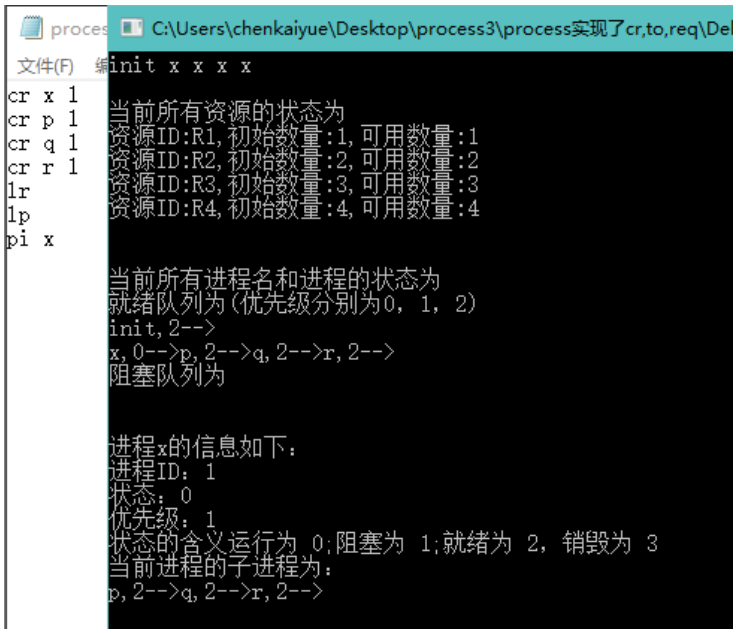
针对测试用例 5，分析在进程和资源管理过程中具体的执行，逐步输入执行命令，以此来

监视每个时刻各个资源，阻塞队列和各个就绪队列的内容和状态，用到了本实验设计的额外功能 lp，lr 和 pi，可以在任意时刻显示进程，资源的相关信息。

当输入前 4 条语句时，由于优先级都为 1，所以运行的进程一直为 x，并且 p,q,r 均为 x 的子进程：

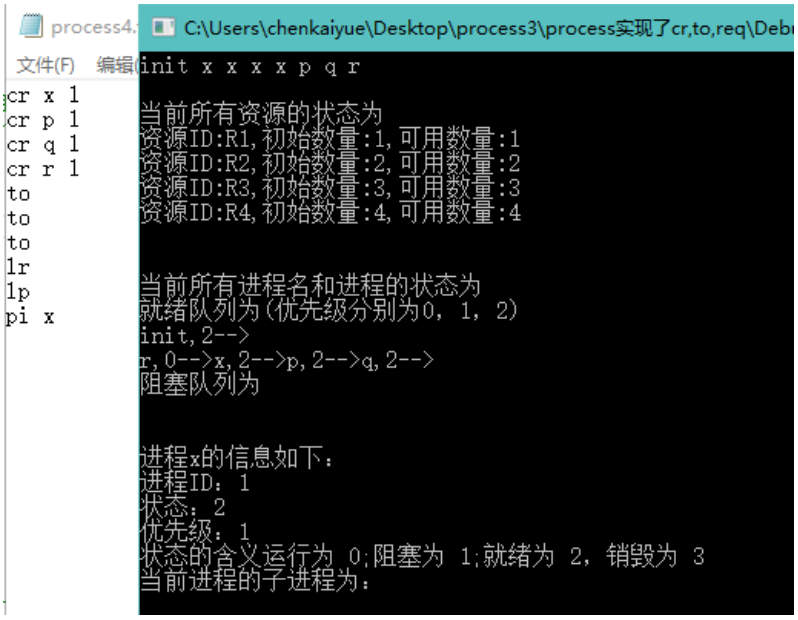
命令----------当前进程
cr x 1----------x
cr p 1----------p
cr q 1----------q
cr r 1----------r

此时，优先级为 1 的就绪队列中有 x-p-q-r，默认最高优先级的队首为正在运行的进程。阻塞队列中无进程。



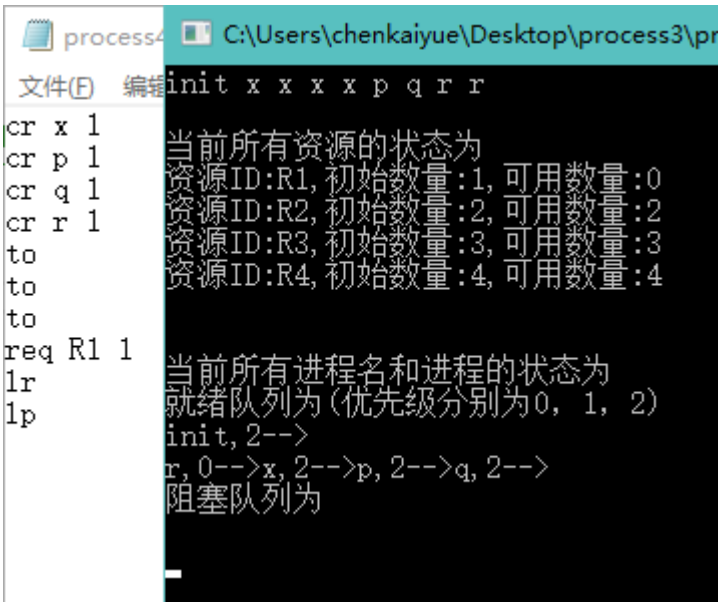
接着输三个 to 命令

命令----------当前进程
to----------p
时钟中断，将 x 从就绪队列的队首取出，放到队尾，队列为 p-q-r-x，因此当前进程为 p
to----------q
时钟中断，将 p 从就绪队列的队首取出，放到队尾，队列为 q-r-x-p，因此当前进程为 q
to----------r
时钟中断，将 q 从就绪队列的队首取出，放到队尾，队列为 r-x-p-q，因此当前进程为 r



Req R1 1----------x

申请资源：req R1 1，即 r 进程申请 1 个 r1 资源，可以满足，因此当前进程依然是 r，且 R1 的可用资源减少为 0。



命令----------当前进程

to----------x

时钟中断，进程 r 从就绪队列的队首取出放到队尾，就绪队列为 x-p-q-r，此时运行的进程为 x

Req R2 1----------x

申请 1 个资源 R2，可以被满足，因此 x 不会被阻塞，当前进程仍为 x，R2 资源的可用数量变为 1。

to----------p

时钟中断，进程 x 从就绪队列的队首取出，放到就绪队列的队尾，就绪队列为 p-q-r-x，当前运行的进程为 p

Req R3 2----------p

申请 2 个资源 R3，可以被满足，因此 p 不会被阻塞，当前进程仍为 p，R3 资源的可用数量变为 1。

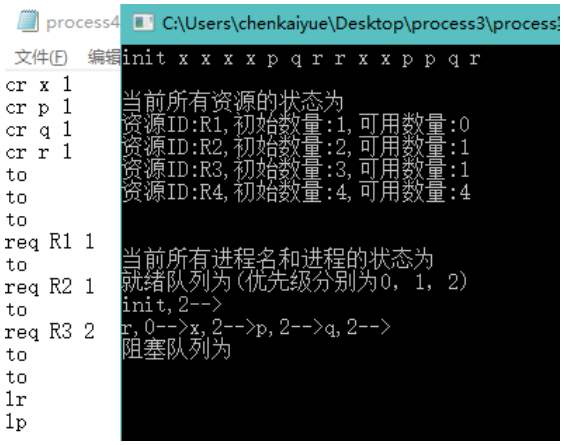
to----------q

时钟中断，进程 p 从就绪队列的队首取出，放到就绪队列的队尾，就绪队列为 q-r-x-p，当前运行的进程为 q

to----------r

时钟中断，进程 q 从就绪队列的队首取出，放到就绪队列的队尾，就绪队列为 r-x-p-q，当前运行的进程为 r

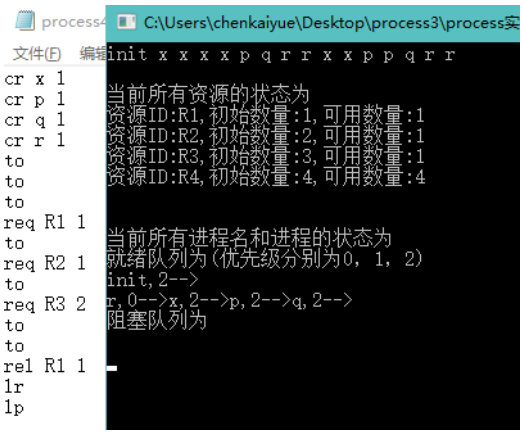
运行结果如下：



命令----------当前进程

rel R1 1 ----------r

释放 1 个资源 R1，由于没有阻塞队列为空，因此没有进程被阻塞，当前进程仍然为 r，R1 的数量变为 1。



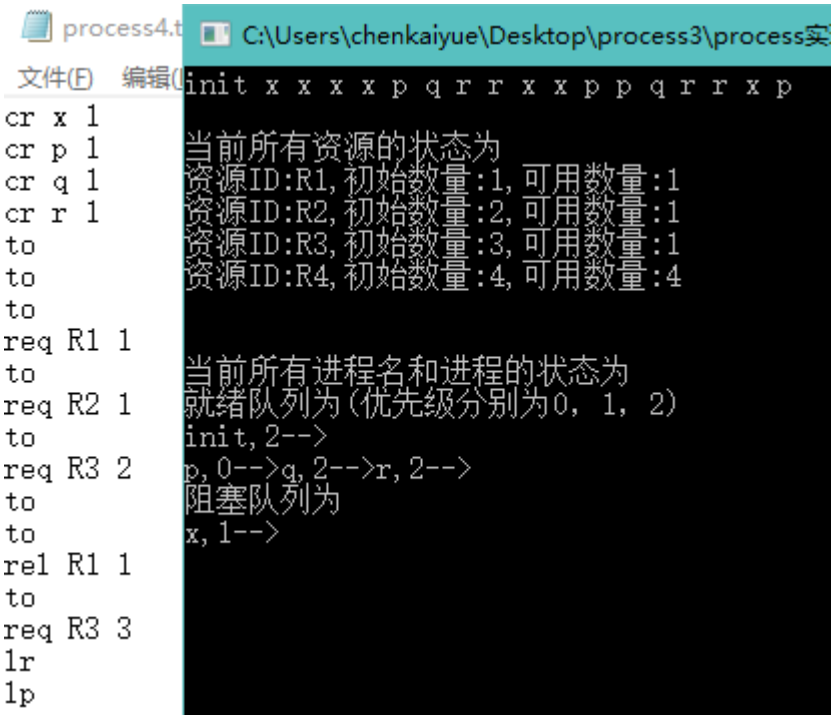
命令----------当前进程

to ----------x

时钟中断，将 r 从就绪队列队首取出放入就绪队列队尾，就绪队列为 x-p-q-r，当前进程为 x

Req R3 3----------p

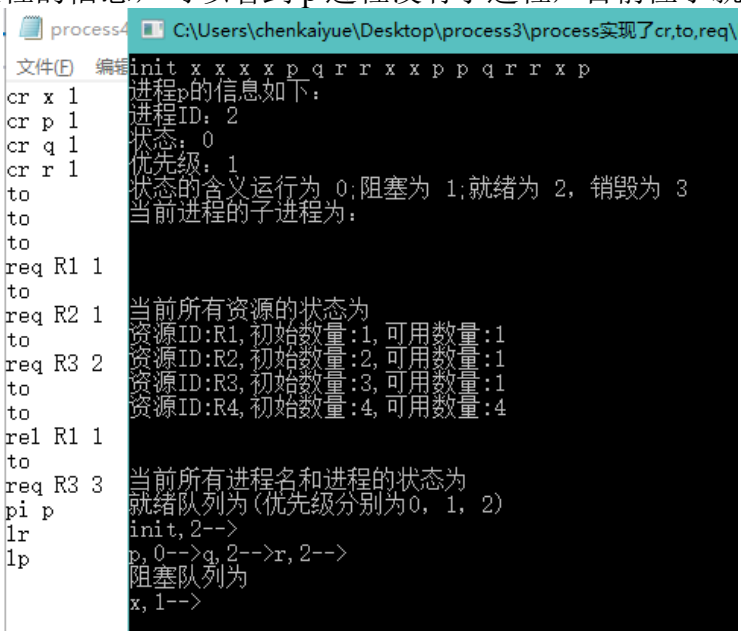
申请 3 个资源 R3，由于 R3 的可用数量只有 1，因此此申请不能被满足，x 被阻塞，当前进程为就绪队列的队首元素 p，R3 的可用资源仍然为 1.，就绪队列为 p-q-r，阻塞队列为 x



命令----------当前进程

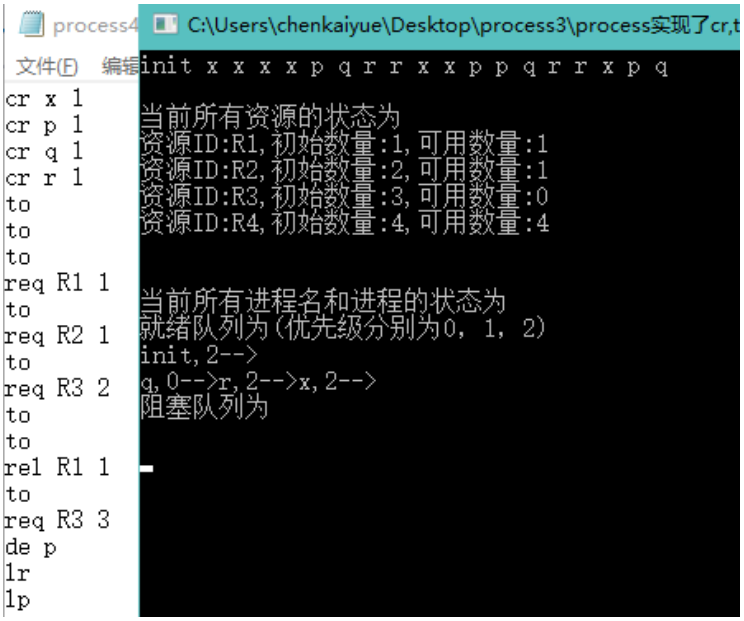
pi p----------

pi 命令显示 p 进程的信息，可以看到 p 进程没有子进程，目前位于就绪队列的队首。



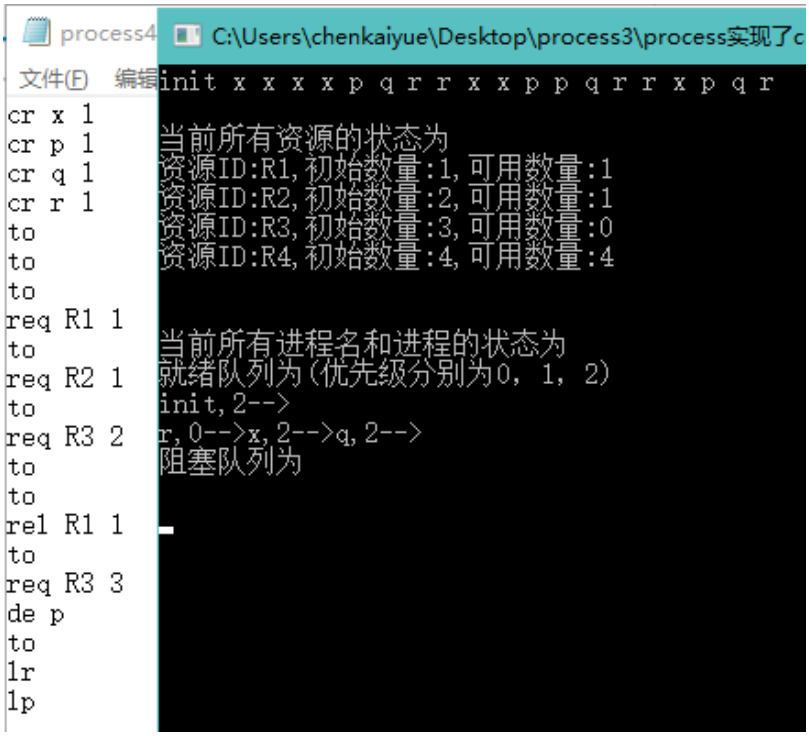
de p ----------q

撤销 p 进程，由于 p 进程没有子进程，因此只释放 p 进程所占有的 2 个 R3 资源，此时 R3 资源有 3 个，x 进程得以释放，进入就绪队列的队尾，同时拥有了它所申请的 3 个 R3 资源，R3 资源的可用数量变为 0，p 进程从就绪队列队首移除，此时就绪队列为 q-r-x，因此当前进程为 q.



to----------r

时钟中断，将 q 进程从就绪队列的队首取出放到就绪队列的队尾，就绪队列为 r-x-q，当前运行的进程为 r



最后，以上步骤已经详细地分析了每一步的运行原理和实验结果，实验成功。

九、实验结论：

本次实验过程，写代码加上 debug，用时大约两周左右，在熟练掌握进程管理原理和调度和对 C 语言数据结构的基础上，在 visual stdio 平台上使用 C 语言实现了操作系统对进程和资源的管理，成功地实现了基于优先级和时间片轮转的抢占式调度算法，通过 5 个不同的测试用例，一一验证并确认正确，实验成功。

十、总结及心得体会：

此次实验耗时较长，尤其是 debug 的过程，主要原因是设计的数据结构略复杂，对指针的用法没有理解透彻，出现了很多诸如指针越界，缓冲区溢出的错误，因此刚开始走了一些弯路，但是总的来说，还是受益匪浅，尤其对 C 语言的指针，数组，结构体，队列，链表的理解更进一步；对于模块化编程的熟练程度也进一步提高。也进一步认识到了在程序中添加错误提示和设置断点进行调试的重要性。

实验报告写到这里，心情不免有些激动，在自己的努力下成功写出程序并执行成功的成就感，通过实践进一步对理论知识加深了巩固和理解的满足感，以及终于可以全身心地投入期末复习的开心一起交织，同时还有对于操作系统这门课的进一步认识，基于最近的国际形势，我进一步认识到操作系统的重要性，当学会了原理之后，可以在代码世界里实现从无到有的突破，可以结合不同的软硬件知识，实现一些很有意义的创造，这也更加坚定了我进一步学习研究操作系统的决心。

十一、对本实验过程及方法、手段的改进建议：

本实验只使用了基于优先级和时间片轮转的策略进行调度，但是我们在理论课中还学习了很多其他的调度算法，如果能够在实验中针对不同的状况采用不同的调度方式，会进一步加深对各种调度策略的理解。

报告评分：

指导教师签字：