电子科技大学

计算机科学与工程学院

标 准 实 验 报 告

（实验）课程名称 计算机操作系统

电子科技大学教务处制表

学生姓名：董文龙　　学　号：2018081309003

指导教师：薛瑞尼

实验时间：2020 年 11 月 21 日

一、实验室名称：电子科技大学清水河校区主楼 A2-412

二、实验项目名称：进程与资源管理综合实验

三、实验学时：4 学时

四、实验目的：

设计和实现进程与资源管理，并完成 Test shell 的编写，以建立系统的进程管理、调度、资源管理和分配的知识体系，从而加深对操作系统进程调度和资源管理功能的宏观理解和微观实现技术的掌握。

五、实验原理：

5.1 总体设计

系统总体架构如图 1 所示：

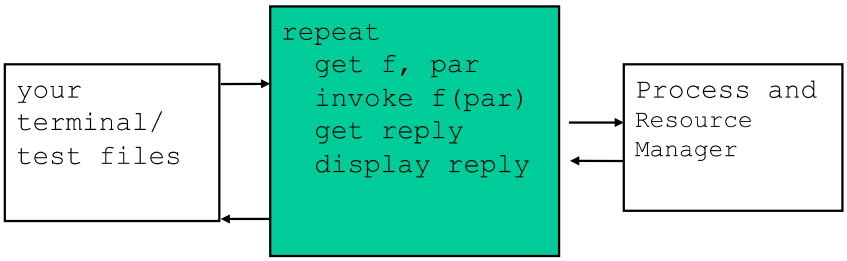


图 1 系统总体结构

图 1 右边部分为进程与资源管理器，是操作系统内核的功能。该管理器具有如下功能：

1. 完成进程创建、撤销和进程调度；
2. 完成多单元资源的管理；完成资源的申请和释放；
3. 完成错误检测和定时器中断功能。

图 1 中间绿色部分为驱动程序 test shell，设计与实现 test shell，该 test shell 将调度所设计的进程与资源管理器来完成测试。Test shell 的应具有的功能：

1. 从终端或者测试文件读取命令；
2. 将用户需求转换成调度内核函数（即调度进程和资源管理器）；
3. 在终端或输出文件中显示结果：如当前运行的进程、错误信息等。

图 1 最左端部分为：通过终端（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

5.2 Test shell 设计

Test_shell 的功能如 5.1 所述。

Test shell 要求完成的命令（Mandatory Commands）：

-init -cr <name> <priority>(=1 or 2) // create process -de <name> // delete process -req <resource name> <# of units> // request resource -rel <resource name> <# of units> // release resource -to // time out

可选实现的命令：

-lp: all processes and their status -lr: all resources and their status - provide information about a given process

5.3 进程管理设计

5.3.1 进程状态与操作

进程状态： ready/running/blocked

进程操作：

- 创建(create)： (none) -> ready
- 撤销(destroy)： running/ready/blocked -> (none)
- 请求资源(Request): running -> blocked (当资源没有时，进程阻塞)
- 释放资源(Release): blocked -> ready (因申请资源而阻塞的进程被唤醒)
- 时钟中断(Time_out): running -> ready
- 调度：ready -> running / running ->ready

5.3.2 进程控制块结构（PCB）

- 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)

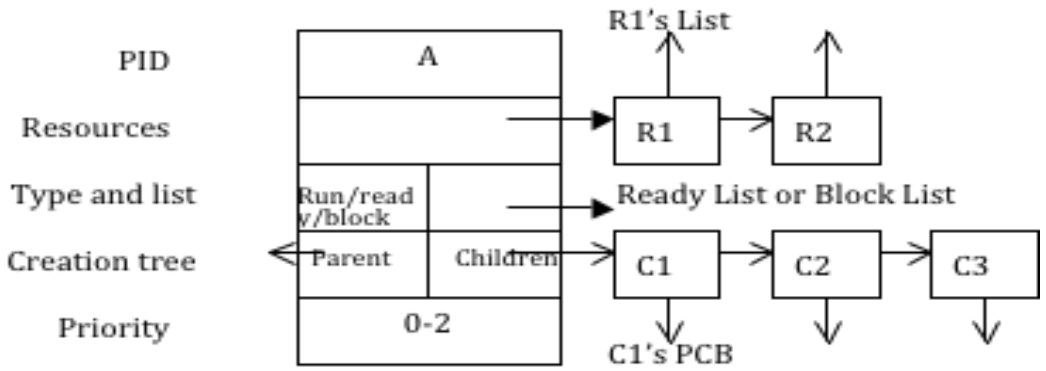


图 2 PCB 结构示意图

就绪进程队列（按优先级排列）：Ready list (RL)

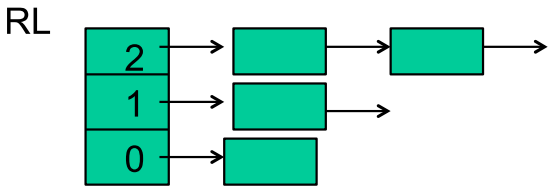


图 3 Ready list 数据结构

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。

5.3.3 主要函数

- **创建进程：**

Create(initialization parameters)// initialization parameters 可以为进程的 ID 和优先级，优先级：初始进程 0、用户进程 1 和系统进程 2。

```
{
    create PCB data structure
    initialize PCB using parameters //包括进程的 ID，优先级、状态等
    link PCB to creation tree /*连接父亲节点和兄弟节点，当前进程为 父亲节点，父亲节点中的子节点为兄弟节点*/
    insert(RL, PCB)//插入就绪相应优先级队列的尾部
    Scheduler()
}
```

Init 进程在启动时创建，可以用来创建第一个系统进程或者用户进程。新创建的进程或者被唤醒的进程被插入到就绪队列（RL）的末尾。

例图 4 中，虚线表示进程 A 为运行进程，在进程 A 运行过程中，创建用户进程 B：cr B 1，数据结构间关系如图 4 所示

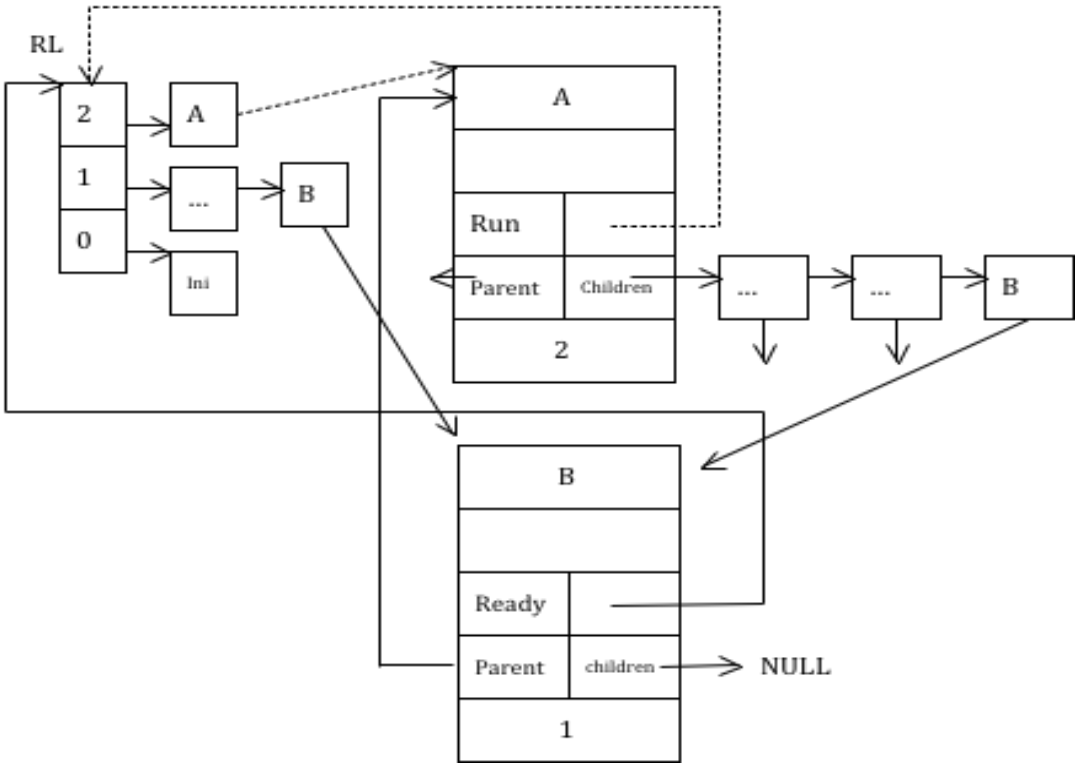


图 4 进程数据结构间关系

(为了简单起见，A 和 B 分别指向 RL 的链接可以不要)

- 撤销进程

```
Destroy (pid) {
    get pointer p to PCB using pid
    Kill_Tree(p)
    Scheduler() //调度其他进程执行
}
Kill_Tree(p) {
    for all child processes q Kill_Tree(q) //嵌套调用，撤销所有子孙进程
    free resources //和 release 调用类似的功能
    delete PCB and update all pointers
}
Process can be destroyed by any of its ancestors or by itself (exit)
```

5.4 资源管理设计

5.4.1 主要数据结构

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

图 5 资源数据结构 RCB

5.4.2 请求资源

所有的资源申请请求按照 FIFO 的顺序进行

- **情况一**：当一类资源数量本身只有一个的情况

```
Request(rid) {
   r = Get_RCB(rid);
   //只有一个资源时可以用 free 和 allocated 来表示资源状态
   if (r->Status == 'free') {
         r->Status = 'allocated‘;
         insert(self->Resources, r);//self 为当前请求资源的进程 PCB，insert 以后 r 为 self 进程占有的资源，参 PCB 结构图
}
else {
   self->Status.Type = 'blocked';
   self->Status.List = r;//point to block list, self is blocked by r
   remove(RL, self);// remove self from the ready list(self can be put at the head of RL when it is running).
    insert(r->Waiting_List, self);// 将进程 self 插入到资源 r 的等待队列尾部
    Scheduler();
}
}
```

- 情况二：一类资源有多个的情况（multi_unit）

```
// n 为请求资源数量
Request(rid, n) {
   r = Get_RCB(rid);
   // u 为 r->Status.u，即可用资源数量，参资源数据结构图
    if (u ≥ n){
         u-n;
         insert(self->Resources, r, n);//self 为当前请求资源的进程 PCB，insert 以后 n 个 r 为 self 进程占有的资源，参 PCB 结构图
} else {
   if (n>k) exit;//k 为资源 r 的总数，申请量超过总数时，将打印错误信息并退出
        self->Status.Type = 'blocked'; //只要 u<n，就不分配，进程阻塞
        self->Status.List = r;//point to block list 注意：此时 block list 是 n 个 r
        remove(RL, self);// remove self from the ready list，因为运行进程位于绪队列首部，所以此时将它从就绪队列移除
        insert(r->Waiting_List, self);// 将进程 self 插入到资源 r 的等待队列尾部
```

```
        Scheduler();
}
}
```

5.4.3 释放资源

- 情况一：一类资源只有 1 个的情况

```
Release(rid) {
   r = Get_RCB(rid);
   remove(self->Resources, r);//将 r 从进程 self 占用的资源中移走
//没有进程在等待资源 r
   if (r->Waiting_List == NIL){
        r->Status = 'free';
}
else {
   remove(r->Waiting_List, q);//q 为 waiting_list 中第一个阻塞进程
   q->Status.Type = 'ready';
   q->Status.List = RL;//就绪队列
   insert(q->Resources, r);
   insert(RL, q); //q 插入就绪队列中相应优先级队列的末尾
   Scheduler();
}
}
```

- 情况二：一类资源有多个的情况

```
//rid 为资源 ID，n 为释放的资源数量
Release(rid,n) {
   r = Get_RCB(rid);/*remove r from self->resources, and u= u + n，将资源 r 从当前进程占用的资源列表里移除，并且资源 r 的可用数量从 u 变为 u+n*/
   remove(self->Resources, r, n);/*如果阻塞队列不为空，且阻塞队列首部进程需求的资源数 req 小于等于可用资源数量 u，则唤醒这个阻塞进程，放入就绪队列*/
   while (r->Waiting_List != NIL && u>=req_num) {
   u=u- req_num; //可用资源数量减少
   remove(r->Waiting_List, q); //  从资源 r 的阻塞队列中移除
   q->Status.Type = 'ready';
   q->Status.List = RL;
   insert(q->Resources, r); //插入 r 到 q 所占用的资源中
   insert(RL, q); //  插入 q 到就绪队列
}
 Scheduler(); //基于优先级的抢占式调度策略，因此当有进程获得资源时，需要查看当前的优先级情况并进行调度
}
```

5.5 进程调度与时钟中断设计

调度策:略

- 基于 3 个优先级别的调度：2，1，0
- 使用基于优先级的抢占式调度策略，在同一优先级内使用时间片轮转（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：永远不会被阻塞，进程树的根。

- Scheduler:

Called at the end of every kernel call

```
Scheduler() {
   find highest priority process p
   if (self->priority < p->priority ||self->Status.Type != 'running' ||self == NIL)
   reempt(p, self)//在条件(3)(4)(5)下抢占当前进程
}
```

Condition (3): called from create or release, 即新创建进程的优先级或资源释放后唤醒进程的优先级高于当前进程优先级

Condition (4): called from request or time-out， 即请求资源使得当前运行进程阻塞或者时钟中断使得当前运行进程变成就绪

Condition (5): called from destroy，进程销毁

Preemption: //抢占，将 P 变为执行，输出当前运行进程的名称

- Change status of p to running (status of self already changed to ready/blocked)
- Context switch—output name of running process

- 时钟中断（Time out）：模拟时间片到或者外部硬件中断

```
Time_out() {
   find running process q; //当前运行进程 q
   remove(RL, q);// remove from head? yes
   q->Status.Type = 'ready';
   insert(RL, q);// insert into tail? yes
   Scheduler();
}
```

5.6 系统初始化设计

启动时初始化管理器：

具有 3 个优先级的就绪队列 RL 初始化；

Init 进程；

4 类资源，R1，R2，R3，R4，每类资源 Ri 有 i 个

异常处理说明：总体原则，对无法满足的需求如果不能阻塞则直接丢弃，如：

- 申请不存在的资源
- 申请的资源数超过其总数
- 删除不存在的进程

六、实验内容：

在提供的软硬件环境中，设计并实现一个基本的进程与资源管理器。该管理器能够完成进程的控制，如进程创建与撤销、进程的状态转换；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算法进行调度；能够完成资源的分配与释放，并完成进程之间的同步。该管理器同时也能完成从用户终端或者指定文件读取用户命令，通过 Test shell 模块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，并将执行结果输出到终端或指定文件中。

七、实验器材：

PC 一台：

处理器：Intel(R)Core(TM) i7-8550U CPU @ 1.80GHz　2.00GHz

已安装的内存(RAM)：20.0GB

系统类型：64 位操作系统，基于 x64 的处理器

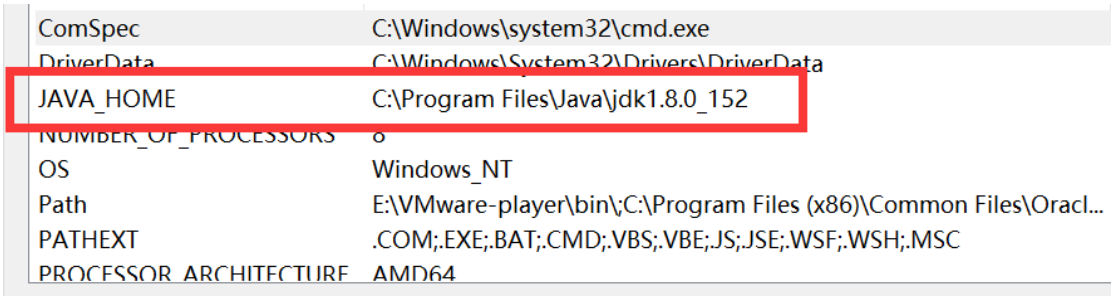
IDE：IntelliJ IDEA 2020.2 x64

JDK：1.8

八、实验步骤及结果：

8.1 开发环境搭建

1. windows 安装 IntelliJ IDEA，下载并安装 JDK 1.8，安装之后进行系统环境变量的配置，如图 6 所示：



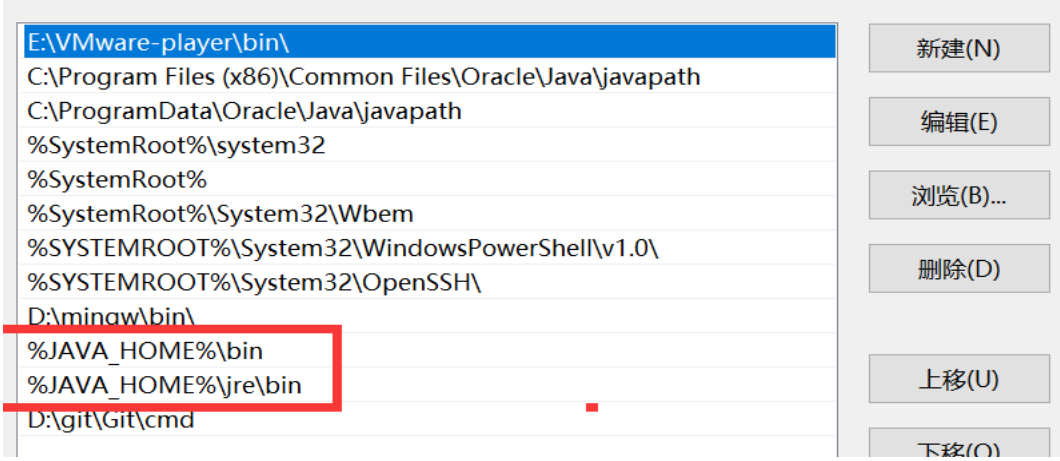


图 6 环境变量添加

8.2 系统功能需求分析

该管理器的功能主要分为：人机交互部分、进程管理部分和资源管理三部分。其中，人机交互部分用于生成命令行窗口，读取用户命令，以作出相应反应，并将结果显示至终端；进程管理部分用于完成进程创建、撤销和调度；资源管理部分则用于完成资源的申请和释放。Shell 应该具有的功能：

- 从终端或者测试文件读取命令；
- 将用户需求转换成调度内核函数（即调度进程和资源管理器）；
- 在终端或输出显示文件中显示结果：如当前运行的进程、错误信息等。

8.3 总体框架设计

设计的总体框架示意图如图 7 所示

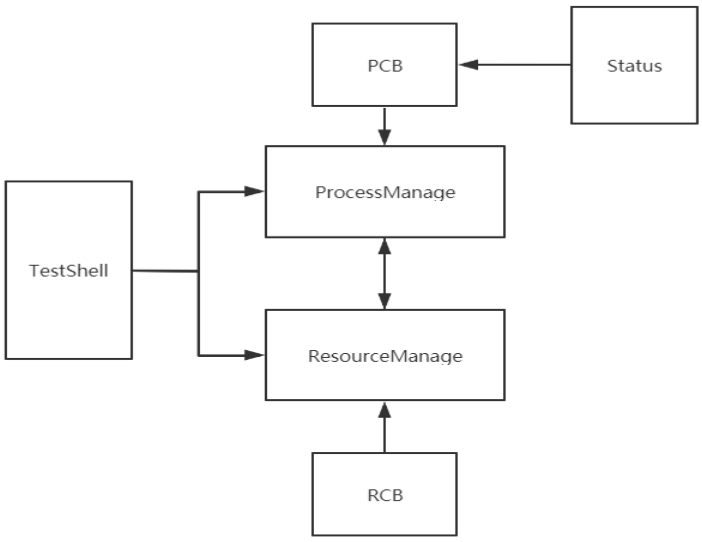


图 7 进程与资源管理示意图

TestShell 模块负责接受用户的输入命令或者读入文件内容，之后交付给 ProcessManage 和 ResourceManage 模块进行处理，对 PCB 和 RCB 模块进行管理。其中 PCB 的进程状态由 Status 模块进行定义。

8.4 详细设计

8.4.1 TestShell 模块

一、读取文件方法：

```
private static void loadFile(String filePath) throws IOException {
    new FileInputStream(filePath);
    LineNumberReader reader = new LineNumberReader(new FileReader(filePath));
    String cmd = null;

    while((cmd = reader.readLine()) != null) {
        if (!"".equals(cmd)) {
            exec(cmd);
        }
    }

}
```

二、用户输入方法：

```
        Scanner scanner = new Scanner(System.in); //scanner接收输入
        while (scanner.hasNextLine()) {
            String input = scanner.nextLine();
            if (input.trim().equals("")) { //空命令时继续接收
                continue;
            }
            exec(input);
        }
    }
```

执行语句方法：

```
String[] commands = new String[]{input};
for (String command : commands) { //对不同的输入命令进行处理
    String[] cmds = command.split( regex: "\\s+");
    switch (cmds.length) {
        case 1:
            if ("init".equals(cmds[0])) {
                ProcessManage.Init();
            }else if ("to".equals(cmds[0])) {
                ProcessManage.timeout();
            }else {
                System.out.println("please input the correct command");
            }
            break;
```

8.4.2 ProcessManage 和 ResourceManage 模块

初始化函数：

```
public static void Init() {
    PCB initProcess = new PCB( PID: "init", new Status(Status.RUNNING, ready_list[0]), pcb_parent: null, priority: 0);
    currentProcess = initProcess;
    ready_list[0].add(currentProcess);
}
```

创建进程函数：

```
public static void CreateProcess(String PID, Integer priority) {
    PCB createProcess = new PCB(PID, new Status(Status.READY, ready_list[priority]), currentProcess, priority);
    currentProcess.pcb_child.add(createProcess);
    ready_list[priority].add(createProcess);
    schedule();
}
```

时间片轮转函数：

```
public static void timeout() {
    int flag = -1;
    for (int i = 2; i >= 0; i--) {
        if (ready_list[i].size() > 0) {
            for (int j = 0; j < ready_list[i].size(); j++) {
                PCB nextProcess = ready_list[i].get(j);
                if (nextProcess.status.state != Status.RUNNING && nextProcess.status.state != Status.BLOCK) {
                    if (nextProcess.priority >= currentProcess.priority || currentProcess.status.state != Status.RUN
                        flag = 1;
                        preempt(nextProcess);//抢占进程
                        break;
                    }
                }
            }
        }
    }
    if (flag == -1) {
        /*System.out.println("process "+currentProcess.PID+" is running.");*/
    }
}
```

删除进程：

首先在就绪队列和阻塞队列进行查找，这里就不截图了（略长，可看源码）

```
    if (deleteProcess != null) {
        deleteProcess.status.state = Status.DEAD;
        List<String> children = new ArrayList<>();
        for (int i = 0; i < deleteProcess.pcb_child.size(); i++) {
            children.add(deleteProcess.pcb_child.get(i).PID);
        }
        ResourceManage.clearDestroyProcess(deleteProcess.PID);
        for (int i = 0; i < children.size(); i++) {
            deleteProcess(children.get(i));                          //递归删除
        }
        deleteProcess.pcb_parent.pcb_child.remove(deleteProcess);
        for (int i = 0; i < deleteProcess.request_resource.size(); i++) {   //资源释放
            Integer releaseResourceIndex = deleteProcess.request_resource.get(i);
            ResourceManage.releaseResources(deleteProcess,  resourceName: "R"+releaseResourceIndex,
                    deleteProcess.other_resource[releaseResourceIndex]);
        }
    }else {
        System.out.println("no this process");
    }
```

调度函数：

```java
public static void schedule() {
    for (int i = 2; i >= 0; i--) {
        if (ready_list[i].size() > 0) {
            PCB nextProcess = ready_list[i].get(0);
            if (currentProcess.status.state == Status.DEAD) {
                ready_list[currentProcess.priority].remove(currentProcess);
                nextProcess.status.state = Status.RUNNING;
                currentProcess = nextProcess;
            }else if (nextProcess.priority > currentProcess.priority || currentProcess.status.state != Status.RUNNIN
                preempt(nextProcess);
            }
        }
    }
}
```

请求资源：

依次遍历 R1,R2,R3,R4，这里就只截了 R1 的图（可看源码）

```java
if ("R1".equals(resourceName)) {
    if (R1.RID >= count) {
        if (R1.number >= count){
            R1.number -= count;
            return true;
        }else {
            PCB nowProcess = ProcessManage.changeToBlock(process, count);
            R1.list.add(nowProcess);
        }
    }else {
        System.out.println("The number of resources applied exceeds the total number of resources");
    }
```

释放资源：

依次遍历 R1,R2,R3,R4，这里就只截了 R1 的图（可看源码）

```java
if ("R1".equals(resourceName)) {
    /*System.out.println(Process.PID + " release R1 " + count);*/
    R1.number += count;
    if (R1.list.size() > 0) {
        int req_num = R1.list.get(0).other_resource[0];
        int index = 0;
        while ( index < R1.list.size() && R1.number >= req_num ) {
            req_num = R1.list.get(index).other_resource[0];
            /*System.out.print("wake up process "+R1.list.get(index).PID + ". ");*/
            R1.number -= req_num;
            ProcessManage.changeToReady(R1.list.get(index),  resourceIndex: 1);
            R1.list.get(index).other_resource[1] = req_num;
            R1.list.remove(index);
            index++;
        }
        /* System.out.println();*/
    }
```

打印当前进程名字：

```java
public static void showCurrentProcess() { System.out.print(currentProcess.PID+" "); }
```

打印就绪队列进程：

```
public static void listReady() {
    for (int i = 2; i >= 0; i--) {
        System.out.print(i + ":");
        for (int j = 0; j < ready_list[i].size(); j++) {
            if (!(ready_list[i].get(j).status.state == Status.RUNNING)) {
                if (j == (ready_list[i].size()-1)) {
                    System.out.print(ready_list[i].get(j).PID );
                }else {
                    System.out.print(ready_list[i].get(j).PID+"-");
                }
            }
        }
        System.out.print(" ");
    }
}
```

打印阻塞队列进程：

依次遍历 R1,R2,R3,R4,这里的截图就只截了 R1 的遍历（可看源码）

```
public static void listBlock() {
    System.out.print("R1 ");
    for (int i = 0; i < R1.list.size(); i++) {
        System.out.print(R1.list.get(i).PID+" ");
    }
    System.out.println();
```

打印当前资源情况：

```
public static void listResources() {
    System.out.println("R1 "+R1.number);
    System.out.println("R2 "+R2.number);
    System.out.println("R3 "+R3.number);
    System.out.println("R4 "+R4.number);
}
```

8.5 测试

一、对于脚本读取 txt 文件的测试，如图 8 所示：

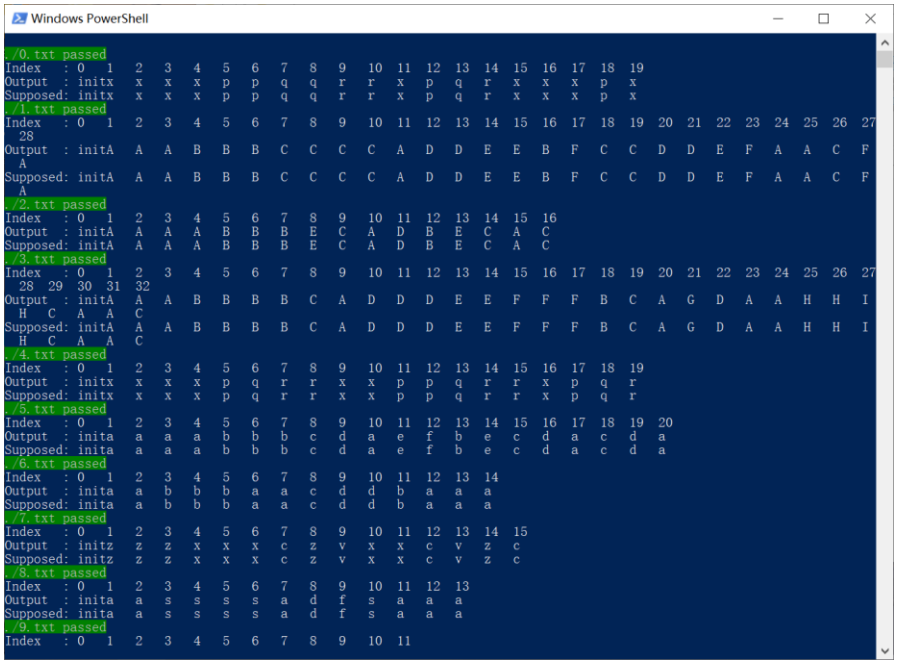


图 8 测试结果

二、对于用户命令行的输入测试(同时进行理论分析)：

（以 2.txt 为例，内容如图 9 所示）注：控制台当前输出，始终为正在运行的进程。

```
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
```

图 9

1. 输入 cr　A　1，输入 cr　B　1，输入 cr　C　1 结果如图 10 所示；

图 10

2. 输入 req R1，正在运行的这个进程 A 会申请 1 个 R1 资源。系统的 R1 资源数量会变为 1，结果如图 11 所示：

图 11

3. 输入 to,队列中 B 将会运行,结果如图 12 所示：

图 12

4. 输入 cr D 1,B 将继续运行。结果如图 13 所示：

图 13

5. 输入 req R2 2，正在运行的这个进程 B 会申请 2 个 R2 资源。系统的 R2 资源数量会变为 2，结果如图 14 所示：

图 14

6. 输入 cr E 2,由于 E 的优先级会比 B 高，会抢占掉 B，所以输出的进程应该为 E。结果如图 15 所示：

图 15

7. 输入 req R2 1,由于 R2 的资源已经被分配掉，所以 E 进程会进入阻塞队列，就绪队列中的 C 会执行，结果如图 16 所示：

图 16

8. 连续三次输入 to 命令，应该会输出就绪队列中的 A -D-B，结果如图 17 所示：

图 17

9. 输入 rel R2 1，此时 B 进程中的 R2 资源将会被释放掉，这时后 E 进程就会被唤醒，结果如图 18 所示：

图 18

10. 输入 de B,B 的儿子 D 和 E 都会被删除掉，此时将会执行进程 C，如图 19 所示：

图 19

11. 输入 to，进程 A 将会执行。如图 20 所示：

图 20

12. 输入 to，进程 C 将会执行。如图 21 所示：

图 21

与 Result.txt 文件一致，实验成功。

九、总结及心得体会：

通过本次实验我加深了对于进程与资源管理的理解，对于系统的进程管理、调度和资源管理的有了更深层次的理解，并且提高了自己对操作系统进程调度和资源管理的掌握。

十、对本实验过程及方法、手段的改进建议：

1.可以试着改进成反馈的调度算法，更加灵活，不过同时也会增加复杂度。

2.java 的包的命名格式一般会写为 com.**.**而不会在 java 目录下直接创建文件，希望测试脚本的 Readme 文档可以更加优化一些，或者写的更清晰一些。

报告评分：

指导教师签字：