

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 操作系统

电子科技大学教务处制表

电子科技大学

# 实 验 报 告

学生姓名：王朴真      学 号：2017060101015      指导教师：薛瑞尼

实验地点：主楼 A2-412

实验时间：12 月 05 日

一、实验室名称：      计算机学院实验中心

二、实验项目名称：进程管理器的实现

三、实验学时：4 学时

四、实验内容：

本实验需要实现一个进程管理器，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。

五、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。

六、实验步骤：

1、系统功能需求分析：

该进程管理器需要实现以下功能：

①创建进程(cr)：创建一个指定名称的新进程。

创建时需要比较它和当前正在执行进程的优先级，如果新进程的优先级大于执行态进程的优先级，那么新进程将抢占 CPU，并将原来的执行进程放入就绪队列，同时修改两个进程的状态，否则新进程直接放入就绪队列，而原来执行态进程不做改变。(注意：就绪队列中的进程顺序需要根据时间+优先级来排序。)

②申请资源(req): 处于执行态的进程在需要某种资源时需要申请该资源。

根据需求, 共有 4 种资源(R1, R2, R3, R4, 资源数量分别为 1, 2, 3, 4), 当执行态进程成功申请到资源时, 该进程继续执行, 如果所申请资源不够, 则申请失败, 需要将当前进程阻塞, 放入阻塞队列, 等待资源释放时被唤醒, 同时从就绪队列选择另一个合适的进程占用 CPU。(注意: 根据已给测试用例得知, 阻塞队列中的进程使用时间顺序排序, 而不关心优先级)

③时间片轮转(to): 模拟时间片轮转功能。

当接收到 to 指令时, 需要将当前执行态的进程换入就绪队列, 重新选择正确的就绪进程占用 CPU。

④删除进程(de): 删除指定名称的进程。

删除该进程时, 需要将其所有的孩子结点也同时删除, 如果被删除的进程中有执行态进程, 那么将重新选择进程占用 CPU。

⑤释放资源(rel): 释放某一资源, 并且要求能给定释放数目。

⑥列出系统当前所有进程及其状态信息(lp)。

⑦列出系统当前所有资源及其状态信息(lr)。

## 2、总体框架设计

各个功能实现的具体原理及思路在以上需求分析中已经给出。在这给出用到的方法或算法, 模块间的调用关系以及总体工作流程:

### 1) 核心思路:

根据需求分析, 可以发现, 对于本管理程序而言, 最核心的地方就是进程的状态转化, 以及伴随着进程状态转换而产生的一系列动作, 如资源的调用, 就绪队列和阻塞队列的使用等。所以, 我们创建一个进程控制块(PCB)来记录进程的信息, PCB 中就包括进程名称、状态、优先级、占用资源及其数量等信息, 然后根据 PCB 来对进程进行一系列操作。

### 2) 总体工作流程:

在 main 函数中使用 switch-case 结构来对不同的指令进行不同的操作。在本实验中, 将各种指令的功能封装到 instFuction 类中, Process 是进程的类, Process 包含 PCB, 另外 GlobalVariable 类是全局变量的集合, 该类中的成员使用 static 修饰。

3) 使用的方法及算法:

①类排序: 进程在就绪和阻塞队列中的顺序就基于 PCB 中的进程优先级来做类排序。

②动态数组: 用于阻塞队列和就绪队列, 进程状态转换时需要对动态数组中的进程进行修改操作。

③树: 用于描绘进程间的父子关系。

④递归: 用于寻找特定进程, 以及删除子树等操作。

⑤迭代器: 用于避免查找和删除并发执行而产生的数据不一致异常。

⑥其他基本程序方法, 如文件操作。

### 3、详细设计

#### 1) PCB 的设计

(注: 设计思路及理由已在代码注释中给出: )

```
public class Process implements Comparable<Process> {  
    /**  
     * PCB控制模块  
     */  
    String pName; //进程名称  
    int[] resources = {0, 0, 0, 0, 0, 0}; //共4种资源, 数组0位置为进程被阻塞时所需资源的类型, 数组5位置是唤醒该进程所需要的资源数目  
    int status; //ready:0, block:-1, running:1  
    int priority = -1; //进程优先级  
    Process parent = null; //一个进程最多有一个父进程  
    ArrayList<Process> childrenList = new ArrayList<>(); // 一个进程可能有多个子进程  
  
    @Override  
    public int compareTo(Process o) { return -(this.priority - o.priority); } //类排序, 从大到小排序  
}
```

#### 2) 系统资源的代码设计

设计思路: 创建一个全局变量集合, totalRe[]记录系统资源, flag 用法见注释。

```
public class GlobalVariable {  
    static int[] totalRe = {0, 1, 2, 3, 4}; //数组0位置保留不用, totalRe[i]的值表示Ri资源的数量  
    static int flag = -1; // 判断被删除的结点中是否有正在运行的结点, flag=1表示有, 则需要其他进程使用CPU  
}
```

#### 3) 创建进程

```

/**
 * 创建进程
 * @param name 进程名称
 * @param priority 进程优先级
 * @return
 */
Process create(String name, int priority) {
    Process p = new Process();
    p.pName = name;
    p.priority = priority;
    p.status = 0; // 新创建的进程初始化为就绪态
    return p;
}

```

#### 4) 申请资源

设计思路：使用之前创建的全局资源变量与被申请资源量作比较。

```

/**
 * 申请资源
 * @param type 资源类型
 * @param num 申请数量
 */
boolean request(int type, int num) {
    if (GlobalVariable.totalRe[type] >= num) {
        // 申请成功就将全局资源变量进行修改
        GlobalVariable.totalRe[type] = GlobalVariable.totalRe[type] - num;
        return true;
    } else {
        return false;
    }
}

```

在主程序调用 request()方法如下图。

```

if (instFunc.request(type, num)) {
    // 如果申请成功，则对当前运行进程做操作
    rp.resources[type] += num;
} else {
    rp.resources[0] = type; // 记录当前进程因哪种资源被阻塞
    rp.resources[5] = num; // 记录当前进程所需要改资源的数目
    rp.status = -1; // 阻塞态
    rpd = rp;
    blockList.add(rpd);
    // Collections.sort(blockList);
    // 下一个就绪进程使用CPU
    if (!readyList.isEmpty()) {
        rp = readyList.remove(index: 0);
        rp.status = 1;
    }
}
}

```

其中需要注意的是：使用进程资源数组 `resource[0]` 记录阻塞该进程的资源类型，用 `resource[5]` 记录要唤醒该进程需要多少 `resource[0]` 类型的资源。(关于 `resource[]` 详情，见第一张图 PCB 控制模块的注释)

### 5) 时间片轮转

设计思路：轮转时需要将执行态进程换下放入就绪队列，同时，涉及轮转操作的两个进程的状态需要修改，当进程加入就绪队列时就需要对就绪队列进行排序，由于已经实现了 `compareTo` 接口，故排序使用 `Collections.sort(readyList)` 即可。

```
/**
 * 时间片轮转
 * @param rp 当前执行进程
 * @param readyList 就绪队列
 * @return
 */
Process roundRobin(Process rp, ArrayList<Process> readyList) {
    //切换进程，将换下来的进程放到就绪队列末尾，然后按时间+优先级排序
    if (rp != null) { //如果不为空就轮转，如果为空，由于CPU资源不会空闲，除非已经没有进程需要使用，故不进行轮转
        rp.status = 0;
        readyList.add(rp);
        Collections.sort(readyList);

        rp = readyList.remove(index: 0); //轮转
        rp.status = 1;
    }

    return rp;
}
```

### 6) 释放资源

```
/**
 * 释放资源
 *
 * @param type 释放资源类型
 * @param num 释放数量
 * @param rp 当前运行的进程
 */
void release(int type, int num, Process rp, ArrayList<Process> readyList, ArrayList<Process> blockList) {
    //释放一个进程需要对全局变量resource修改，并且修改等待该资源的进程PCB，修改他们的状态以及所在的链表
    rp.resources[type] -= num;
    GlobalVariable.totalRe[type] += num;

    wakeup(readyList, blockList); //释放资源可能会唤醒被阻塞的进程
}
```

其中 `wakeup()` 方法如下图：

```

/**
 * 从阻塞队列中唤醒进程到就绪队列
 *
 * @param readyList
 * @param blockList
 */
void wakeUp(ArrayList<Process> readyList, ArrayList<Process> blockList) {
    Iterator<Process> iterator = blockList.iterator();
    while (iterator.hasNext()) {
        Process bp = iterator.next();
        // 判断资源是否足够 (注: resource[0] 记录缺少哪一类型的资源, resource[5] 记录需要多少该资源才能唤醒)
        if (GlobalVariable.totalRe[bp.resources[0]] >= bp.resources[5]) {
            GlobalVariable.totalRe[bp.resources[0]] -= bp.resources[5]; // 唤醒前对全局资源做修改
            bp.resources[bp.resources[0]] += bp.resources[5]; // 被唤醒进程所占有的bp.resource[0]类型的资源数目
            readyList.add(bp);
            bp.status = 0;
            Collections.sort(readyList);
            iterator.remove(); // 唤醒到就绪列表后从阻塞列表中删除
        } else {
            break;
        }
    }
}

```

注意：如上所说，由于每一个进程 PCB 中的 resource[0] 和 resource[5] 分别记录了因哪类资源被阻塞以及需要多少 resource[0] 类型的资源才能唤醒该进程，所以在判断的时候，只需要判断 `GlobalVariable.totalRe[p.resource[0]] >= p.resource[5]` 是否为真，就可以判断进程 p 是否可以被唤醒。

## 7) 销毁进程

设计思路：给定一个进程名称来删除该进程及其所有子进程，那么就需要先找到该进程，使用 findP() 方法找到要删除的进程 x，找到后才进行删除操作。

但是如果找到该进程就直接删除，这时并没有同步删除其孩子结点，并且由于父节点的删除，孩子结点将很难再定位，导致删除不完全，程序达不到要求。故在此给出两种解决方案：

① 使用递归的方式，找到给定删除的进程，然后以此进程为根节点，进行遍历，从叶节点开始删除，直至删除到根节点。

② 将需要被删除的结点先全部记录下来，最后统一删除。

综合分析，使用第二种方案，在程序中使用 recoList 来记录需要被删除的结点，记录使用递归的方式，将父进程、子进程、子进程的子进程……全部记录下来，代码如下图：

```

/**
 * 将需要删除的结点存入recoList中
 *
 * @param p
 * @param recoList
 */
void recoDel(Process p, ArrayList<Process> recoList) {
    recoList.add(p);
    for (Process px : p.childrenList) {
        recoDel(px, recoList);
    }
}

```

从而，销毁一个进程的代码如下：

```

* 销毁进程
* @param rp 当前运行进程
* @param name 被删除进程名称
* @param readyList 就绪队列
* @param blockList 阻塞队列
*/
void destroy(Process rp, String name, ArrayList<Process> readyList, ArrayList<Process> blockList) {
    //销毁进程会释放资源，释放资源会导致阻塞进程被唤醒
    Process x = null;
    ArrayList<Process> recoList = new ArrayList<>(); //记录需要的删除结点，最后统一删除

    x = findP(name, rp, readyList, blockList); //找到给定名称的进程，用临时变量x记录
    recoDel(x, recoList); //将要被删除的所有进程记录到recoList
    for (Process p : recoList) {
        if (p.status == -1) {
            blockList.remove(p);
        } else if (p.status == 0) {
            readyList.remove(p);
        } else if (p.status == 1) {
            GlobalVariable.flag = 1; //如果执行态结点也在删除队列，那么先将全局变量flag置1，作为标记
        }
        p.status = -2; //删除后将进程状态置为-2
        if (p.parent != null) {
            p.parent.childrenList.remove(p); //删除被删除进程的孩子结点
        }
        totalRel(p); //删除会释放所有的资源，此方法为释放所有资源
    }
}

```

注意：上面代码中单独记录了 `GlobalVariable.flag = 1`，即如果被删除结点中有运行态的进程，那么就需要对运行态进程 `rp` 做修改，重新选择新的进程使用 CPU，代码如下：

```

if (GlobalVariable.flag == 1) { //如果返回值是1，则说明是当前运行结点也被删除，需要重新挑选进程占用CPU
    rp.status = -2;
    rp = null;
    if (!readyList.isEmpty()) {
        rp = readyList.remove(index: 0);
        rp.status = 1;
    }
}

```



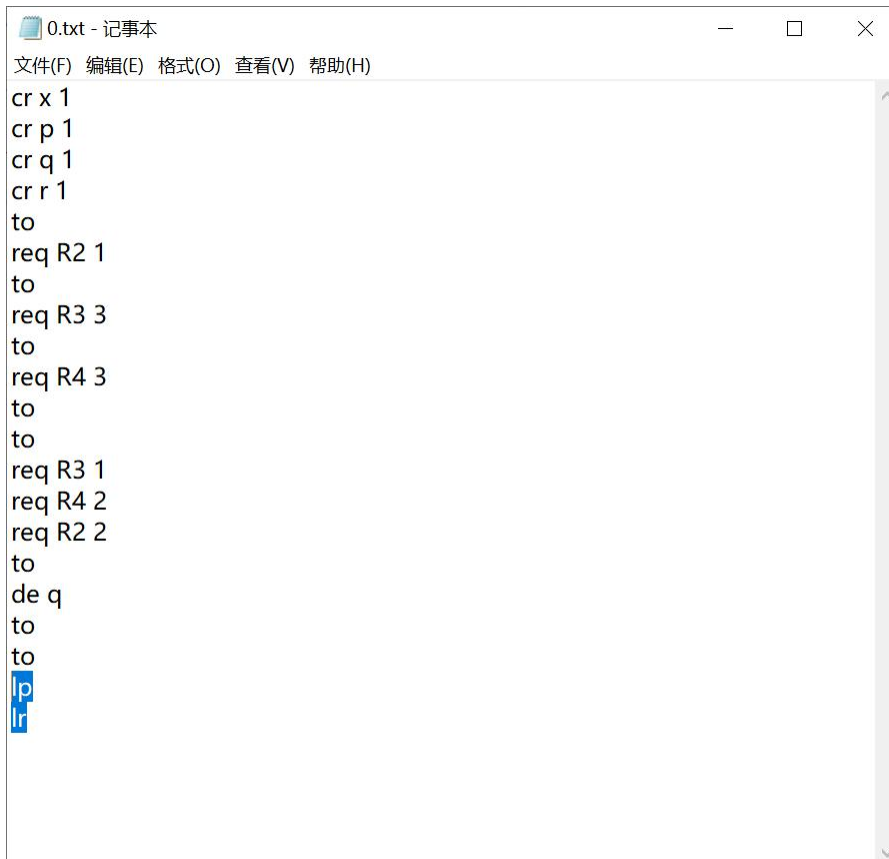
## 8) 显示系统所有进程和资源的信息 (lp、lr)

```
System.out.println("所有进程及其状态如下: (1表示运行态, 0表示就绪态, -1表示阻塞态)");
System.out.println("pName" + "\t" + "status");
if(rp != null){
    System.out.println(rp.pName + "\t\t" + rp.status);
}
for(Process pro : readyList){
    System.out.println(pro.pName + "\t\t" + pro.status);
}
for(Process pro : blockList){
    System.out.println(pro.pName + "\t\t" + pro.status);
}

System.out.println("所有资源及其状态如下: ");
System.out.println("type" + "\t" + "total" + "\t" + "available");
for(int i = 1; i < 5; i++){
    System.out.println("R" + i + "\t\t" + i + "\t\t" + GlobalVariable.totalRe[i]);
}
break;
```

## 七、实验数据及结果分析:

1、本程序实现了 lp 和 lr 两个附加指令, 为了保证输出格式的规范, 现将 lp 和 lr 加入到 0.txt 末尾, 用于展示 lp 和 lr 的正确性:



```
0.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
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
lp
lr
```

```

"C:\Program Files (x86)\Java\jdk1.8.0_102\bin\java.exe" ...
请输入指令文件路径:
C:\Users\tzbx_\Documents\课程课件\操作系统\实验\input0.txt
init x x x x p p q q r r x p q r x x x p x

所有进程及其状态如下: (1表示运行态, 0表示就绪态, -1表示阻塞态)
pName    status
x         1
p         0
r         -1

所有资源及其状态如下:
type     total    available
R1        1        1
R2        2        1
R3        3        2
R4        4        1

```

## 2、现测试给定的十个用例:

```

C:\WINDOWS\system32\cmd.exe
Environment: windows
*197;42n./0.txt passed*[0m
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
init x x x x p p q q r r x p q r x x x p x
init x x x x p p q q r r x p q r x x x p x
*197;42n./1.txt passed*[0m
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
init A A A B B B C C C C A D D E E B F C C D D E F A A C F A
init A A A B B B C C C C A D D E E B F C C D D E F A A C F A
*197;42n./2.txt passed*[0m
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
init A A A A B B B B E C A D B E C A C
init A A A A B B B B E C A D B E C A C
*197;42n./3.txt passed*[0m
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
init A A A B B B B B C A D D D E E F F F B C A G D A A H H I H C A A C
init A A A B B B B B C A D D D E E F F F B C A G D A A H H I H C A A C
*197;42n./4.txt passed*[0m
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
init x x x x p q r r x x p p q r r x p q r
init x x x x p q r r x x p p q r r x p q r
*197;42n./5.txt passed*[0m
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
init a a a a b b b c d a e f b e c d a c d a
init a a a a b b b c d a e f b e c d a c d a
*197;42n./6.txt passed*[0m
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
init a a b b b a a c d d b a a a
init a a b b b a a c d d b a a a
*197;42n./7.txt passed*[0m
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
init z z z x x x c z v x x c v z c
init z z z x x x c z v x x c v z c
*197;42n./8.txt passed*[0m
0 1 2 3 4 5 6 7 8 9 10 11 12 13
init a a s s s s a d f s a a a
init a a s s s s a d f s a a a
*197;42n./9.txt passed*[0m
0 1 2 3 4 5 6 7 8 9 10 11
init x x x y y y y x z initinit
init x x x y y y y x z initinit

```

测试程序显示每一个测试样例都为 passed，程序输出了正确的期望序列，进程管理器功能实现正确。

## **八、总结及心得体会：**

通过本次实验，我实现了一个简单的进程管理器，在实现过程中，遇到了很多问题，比如：在资源释放时，进程的唤醒顺序该如何确定？删除一个进程时如何删除其所有子孙节点等等，通过一步步分析这些问题，查阅资料，调试程序，最后一一解决。这让我对进程的管理有了更加深刻的理解，及时巩固了在课堂上学习的理论知识，而且通过编程实践，极大地锻炼了我的代码的逻辑组织能力和代码设计的全局观，提升了自己的设计能力，为以后的程序、算法等的设计打下了良好的基础，受益良多。

**报告评分：**

**指导教师签字：**