**电 子 科 技 大 学**

**实 验 报 告**

**实验一**

1. **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：**

进程与资源管理实验

**三、实验内容**

在实验室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。

**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，最右边部分为进程与资源管理器，属于操作系统内核的功能。该管理器具有如下功能：完成进程创建、撤销和进程调度；完成多单元 (multi\_unit)资源的管理；完成资源的申请和释放；完成错误检测和定时器中断功能。



图1 系统总体结构

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：

* + 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，代码示例如图1中绿色部分。

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 //list all processes and their status

-lr //list all resources and their status

-pi //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）
* CPU state — not used
* Memory — not used
* Open\_Files — not used
* Other\_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)

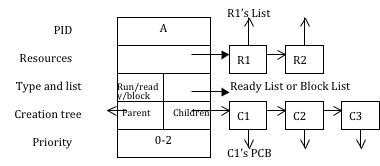


图2 PCB结构示意图

就绪进程队列Ready list (RL)如图3所示。



图3 Ready List数据结构示意图

有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：

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。

示例：

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。

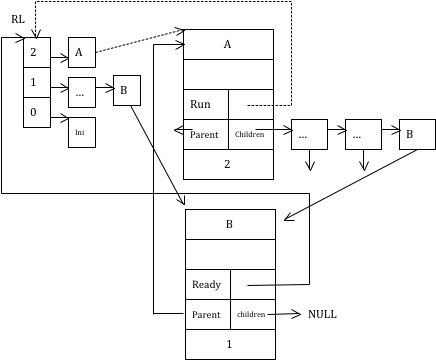


图4 进程数据结构间关系

(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：

进程可以由它的任何父进程或自己销毁(退出)。

**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示意图

**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
* 情况二：一类资源有多个的情况（multi\_unit）

**5.4.3释放资源**

* 情况一：一类资源只有1个的情况
* 情况二：一类资源有多个的情况

**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：具有最低的优先级，永远不会被阻塞；

进程树的根。

时钟中断（Time out）

* 模拟时间片到或者外部硬件中断。

**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。

**六、实验器材（设备、元器件）**

处理器：Intel® Core™ i5-8300H CPU @ 2.30GHz 2.30GHz

已安装的内存(RAM)：8GB

系统类型：64位操作系统，基于x64的处理器

IDE：JetBrains IntelliJ IDEA(Ultimate Version) 2020.1.3

JDK：1.8.0\_162

**七、实验步骤**

**7.1 框架设计**

* main类，是程序的Test Shell，主要实现了读取命令、语句转换和显示结果等功能。
* PCB（进程管理块）类，用于进程的管理, 主要实现了进程的创建、进程的切换与调度以及进程信息的输出等功能。
* Process类，定义了进程的数据结构，主要实现了删除进程、递归删除子进程树、销毁所有进程等功能。
* Resource类，定义了资源的数据结构，主要实现了请求资源、释放资源、输出资源信息等功能。另外，该类对每一个资源管理和维护了对应的阻塞队列，当进程请求资源不足时，将进程加入对应的阻塞队列中。
* Queue类，用于管理和维护进程队列，主要实现了3种优先级队列的入队、出队、删除等操作。

**7.2 具体实现**

main.java

|  |
| --- |
| 1. **import** java.io.\*; 2. **import** java.util.Scanner; 4. /\*\* 5. \* @BelongsProject:OS-process 6. \* @BelongsPackage:PACKAGE\_NAME 7. \* @Author:Uestc\_Xiye 8. \* @CreateTime:2020-12-01 16:53:00 9. \*/ 10. **public** **class** main { 11. **private** **static** **final** Pcb pcb=Pcb.getpcb(); 12. **private** **static** **final** Resource r1=**new** Resource(1,1); 13. **private** **static** **final** Resource r2=**new** Resource(2,2); 14. **private** **static** **final** Resource r3=**new** Resource(3,3); 15. **private** **static** **final** Resource r4=**new** Resource(4,4); 17. **public** **static** **void** main(String[] args) **throws** IOException 18. { 19. pcb.createProcess("init",0); 20. System.out.print("init"+" "); 21. **if**(args.length!=0) 22. { 23. loadFile(args[0]); 24. } 25. **else** 26. { 27. System.out.println(); 28. Scanner scanner=**new** Scanner(System.in); 29. **while**(scanner.hasNextLine()) 30. { 31. String input=scanner.nextLine(); 32. **if**(input.trim().equals("")) 33. { 34. **continue**; 35. } 36. testShell(input); 37. } 38. } 39. } 41. **public** **static** **void** testShell(String input) 42. { 43. String[] commands=**new** String[]{input}; 44. **for**(String command:commands) 45. { 46. String[] cmds=command.split("\\s+"); 47. String options=cmds[0]; 48. **switch**(options) 49. { 50. **case** "cr": 51. **if** (cmds.length!=3) 52. { 53. System.out.println("Error! Please enter the legal parameters!"); 54. } 55. **else** 56. { 57. String processName=cmds[1]; 58. **int** priority=0; 59. **try** 60. { 61. priority=Integer.parseInt(cmds[2]); 62. **if**(priority<=0 || priority>2) 63. { 64. System.out.println("Error!Please enter the legal parameters!"); 65. **continue**; 66. } 67. } **catch** (Exception e) {System.out.println("Error!Please enter the legal parameters!");} 68. **if**(pcb.isExistName(processName)) 69. { 70. System.out.println("Error!Process "+processName+"already exists!Please select another process name!"); 71. **break**; 72. } 73. pcb.createProcess(processName,priority); 74. } 75. **break**; 76. **case** "de": 77. **if**(cmds.length!=2) 78. { 79. System.out.println("Error!Please enter the legal parameters!"); 80. } 81. **else** 82. { 83. String processName=cmds[1]; 84. Process process = pcb.findProcess(processName); 85. **if**(process==**null**) 86. { 87. System.out.println("Error!Process " + processName + "does not exist!"); 88. } 89. **else** **if**(processName.equals("init")) 90. { 91. System.out.println("Error!You do not have permission to terminate the init process!"); 92. } 93. **else** 94. { 95. process.destroy(); 96. } 97. } 98. **break**; 99. **case** "req": 100. **if**(cmds.length!=3) 101. { 102. System.out.println("Error!Please enter the legal parameters!"); 103. } 104. **else** 105. { 106. String resourceName=cmds[1]; 107. **int** need=0; 108. **try** 109. { 110. need=Integer.parseInt(cmds[2]); 111. } **catch** (Exception e) {System.out.println("Error!Please enter the legal parameters!");} 112. Process currentProcess=pcb.getcurrentProcess(); 113. **switch**(resourceName) 114. { 115. **case** "R1": 116. r1.requestResource(currentProcess,need); 117. **break**; 118. **case** "R2": 119. r2.requestResource(currentProcess,need); 120. **break**; 121. **case** "R3": 122. r3.requestResource(currentProcess,need); 123. **break**; 124. **case** "R4": 125. r4.requestResource(currentProcess,need); 126. **break**; 127. **default**: 128. System.out.println("Error!Please enter the legal parameters!"); 129. } 130. } 131. **break**; 132. **case** "rel": 133. **if**(cmds.length!=3) 134. { 135. System.out.println("Error!Please enter the legal parameters!"); 136. } 137. **else** 138. { 139. String resourceName=cmds[1]; 140. **int** rel=0; 141. **try** 142. { 143. rel=Integer.parseInt(cmds[2]); 144. } **catch** (Exception e) {System.out.println("Error!Please enter the legal parameters!");} 145. Process currentProcess = pcb.getcurrentProcess(); 146. **switch**(resourceName) 147. { 148. **case** "R1": 149. r1.releaseResource(currentProcess,rel); 150. **break**; 151. **case** "R2": 152. r2.releaseResource(currentProcess,rel); 153. **break**; 154. **case** "R3": 155. r3.releaseResource(currentProcess,rel); 156. **break**; 157. **case** "R4": 158. r4.releaseResource(currentProcess,rel); 159. **break**; 160. **default**: 161. System.out.println("Error!Please enter the legal parameters!"); 162. } 163. } 164. **break**; 165. **case** "to": 166. pcb.timeout(); 167. **break**; 168. **case** "lp": 169. **if**(cmds.length==1) 170. { 171. pcb.printProcessTree(pcb.findProcess("init"),0); 172. } 173. **else** **if**(cmds.length<3 || !cmds[1].equals("-p")) 174. { 175. System.out.println("Error!Please enter a legal parameter or command!"); 176. } 177. **else** 178. { 179. String pName=cmds[2]; 180. Process process=pcb.findProcess(pName); 181. **if**(process==**null**) 182. { 183. System.out.println("Error!Process "+pName+"does not exist!"); 184. } 185. **else** 186. { 187. pcb.printProcessDetail(process); 188. } 189. } 190. **break**; 191. **case** "lr": 192. r1.printStatus(); 193. r2.printStatus(); 194. r3.printStatus(); 195. r4.printStatus(); 196. **break**; 197. **case** "exit": 198. System.out.println("Bye!"); 199. System.exit(0); 200. **case** "list": 201. pcb.printExistProcess(); 202. **break**; 203. **default**: 204. System.out.println("Error!Please enter the legal command!"); 205. **break**; 206. } 207. } 208. **if**(pcb.getcurrentProcess()!=**null**) 209. { 210. System.out.print(pcb.getcurrentProcess().getpName()+" "); 211. } 212. } 214. **private** **static** **void** loadFile(String filePath) **throws** IOException 215. { 216. InputStream inputStream=**new** FileInputStream(filePath); 217. LineNumberReader lineNumberReader=**new** LineNumberReader(**new** FileReader(filePath)); 218. String cmd=**null**; 219. **while**((cmd=lineNumberReader.readLine())!=**null**) 220. { 221. **if**(!"".equals(cmd)) 222. { 223. testShell(cmd); 224. } 225. } 226. } 228. } |

Pcb.java

|  |
| --- |
| 1. **import** java.util.HashMap; 2. **import** java.util.LinkedList; 3. **import** java.util.List; 4. **import** java.util.Map; 5. **import** java.util.concurrent.ConcurrentHashMap; 6. **import** java.util.concurrent.atomic.AtomicInteger; 8. /\*\* 9. \* @BelongsProject:OS-process 10. \* @BelongsPackage:PACKAGE\_NAME 11. \* @Author:Uestc\_Xiye 12. \* @CreateTime:2020-12-01 16:52:29 13. \*/ 14. **public** **class** Pcb { 15. /\*\* 16. \* 变量说明 17. \* pcb: 进程控制块（Process Control Block） 18. \* readyQueue: 就绪队列 19. \* existProcess: 所有存活的进程，包括Running（运行状态）,Blocked（阻塞状态）,Ready（就绪状态） 20. \* currentProcess: 当前正在占用CPU的进程 21. \* pidGenerator: pid生成器，可以生成唯一的pid 22. \*/ 23. **private** **static** **final** Pcb pcb=**new** Pcb(); 24. **private** **static** **final** Queue readyQueue=Queue.getreadyQueue(); 25. **private** **static** Map<String,Process> existProcess; 26. **private** Process currentProcess; 27. **private** AtomicInteger pidGenerator; 29. **private** Pcb() 30. { 31. existProcess=**new** HashMap<>(); 32. pidGenerator=**new** AtomicInteger(); 33. } 35. **public** Process createProcess(String processName,**int** priority) 36. { 37. Process currentProcess=pcb.getcurrentProcess(); 38. // 为新建进程分配pid，进程名，优先级，进程状态，资源，父进程和子进程等信息 39. Process process=**new** Process(pcb.createpid(),processName,priority,Process.State.NEW,**new** ConcurrentHashMap<>(),currentProcess,**new** LinkedList<>()); 40. **if**(currentProcess!=**null**) 41. { 42. currentProcess.getchildren().add(process); 43. process.setparent(currentProcess); 44. } 45. pcb.addexistProcess(process); 46. readyQueue.addprocess(process); 47. process.setstate(Process.State.READY); 48. Pcb.scheduler(); 49. **return** process; 50. } 52. **public** **static** **void** scheduler() 53. { 54. Process currentProcess=pcb.getcurrentProcess(); 55. Process readyProcess=readyQueue.getprocess(); 56. **if**(readyProcess==**null**) 57. { 58. pcb.getcurrentProcess().setstate(Process.State.RUNNING); 59. **return**; 60. } 61. **else** **if**(currentProcess==**null**) 62. { 63. readyQueue.deleteProcess(readyProcess); 64. pcb.setcurrentProcess(readyProcess); 65. readyProcess.setstate(Process.State.RUNNING); 66. **return**; 67. } 68. **else** **if**(currentProcess.getstate()==Process.State.BLOCKED || currentProcess.getstate()==Process.State.TERMINATED) 69. { 70. readyQueue.deleteProcess(readyProcess); 71. pcb.setcurrentProcess(readyProcess); 72. readyProcess.setstate(Process.State.RUNNING); 73. } 74. **else** **if**(currentProcess.getstate()==Process.State.RUNNING) 75. { 76. **if**(currentProcess.getpriority()<readyProcess.getpriority()) 77. { 78. preempt(readyProcess,currentProcess); 79. } 80. } 81. **else** **if**(currentProcess.getstate()==Process.State.READY) 82. { 83. **if**(currentProcess.getpriority()<=readyProcess.getpriority()) 84. { 85. preempt(readyProcess,currentProcess); 86. } 87. **else** 88. { 89. currentProcess.setstate(Process.State.RUNNING); 90. } 91. } 92. **return**; 93. } 95. **public** **static** **void** preempt(Process readyProcess,Process currentProcess) 96. { 97. **if**(isExistName(currentProcess.getpName())) 98. { 99. readyQueue.addprocess(currentProcess); 100. currentProcess.setstate(Process.State.READY); 101. readyQueue.deleteProcess(readyProcess); 102. pcb.setcurrentProcess(readyProcess); 103. readyProcess.setstate(Process.State.RUNNING); 104. **return**; 105. } 106. } 108. **public** **static** **void** timeout() 109. { 110. pcb.getcurrentProcess().setstate(Process.State.READY); 111. scheduler(); 112. } 114. **public** **void** deleteexistProcess(Process process) 115. { 116. String name=process.getpName(); 117. existProcess.remove(name); 118. } 120. **public** **void** printProcessTree(Process process,**int** retract) 121. { 122. **for** (**int** i=0;i<retract;i++) 123. { 124. System.out.print("  "); 125. } 126. System.out.print("|-"); 127. printProcessDetail(process); 128. List<Process> children=process.getchildren(); 129. **for**(Process child:children) 130. { 131. printProcessTree(child,retract+1); 132. } 133. } 135. **public** **void** printProcessDetail(Process process) 136. { 137. System.out.print(process.getpName()+"(PID:"+process.getpid()+",进程状态："+process.getstate()+",优先级："+process.getpriority() + ","); 138. **if**(process.getresourceMap().isEmpty()) 139. { 140. System.out.println("(无资源占用))"); 141. } 142. **else** 143. { 144. StringBuilder stringBuilder=**new** StringBuilder(); 145. stringBuilder.append("("); 146. **for**(Map.Entry<Resource,Integer> entry:process.getresourceMap().entrySet()) 147. { 148. Resource res=entry.getKey(); 149. **int** holdNum=entry.getValue(); 150. stringBuilder.append(",") 151. .append("R") 152. .append(res.getrid()) 153. .append(":") 154. .append(holdNum); 155. } 156. stringBuilder.append(")"); 157. String result=stringBuilder.toString(); 158. System.out.println(result.replaceFirst(",","")); 159. } 160. } 162. **public** **void** printExistProcess() 163. { 164. StringBuilder stringBuilder=**new** StringBuilder(); 165. stringBuilder.append("existList:["); 166. **for**(Map.Entry<String,Process> entry:existProcess.entrySet()) 167. { 168. String name=entry.getKey(); 169. String state=entry.getValue().getstate().toString(); 170. stringBuilder.append(",") 171. .append(name) 172. .append("(") 173. .append(state) 174. .append(")"); 175. } 176. stringBuilder.append("]"); 177. String result=stringBuilder.toString(); 178. System.out.println(result.replaceFirst(",", "")); 179. } 181. **public** **int** createpid() 182. { 183. **return** pidGenerator.getAndIncrement(); 184. } 186. **public** **void** addexistProcess(Process process) 187. { 188. existProcess.put(process.getpName(),process); 189. } 191. **public** **static** **boolean** isExistName(String name) 192. { 193. **return** existProcess.containsKey(name); 194. } 196. **public** Process findProcess(String processName) 197. { 198. **for**(Map.Entry<String, Process> entry:existProcess.entrySet()) 199. { 200. **if**(processName.equals(entry.getKey())) 201. { 202. **return** entry.getValue(); 203. } 204. } 205. **return** **null**; 206. } 208. **public** **void** deleteProcess(Process process) 209. { 210. } 212. **public** **void** setcurrentProcess(Process currentProcess) 213. { 214. **this**.currentProcess=currentProcess; 215. } 217. **public** **void** setPidGenerator(AtomicInteger pidGenerator) 218. { 219. **this**.pidGenerator=pidGenerator; 220. } 222. **public** **static** Pcb getpcb() 223. { 224. **return** pcb; 225. } 227. **public** Queue getreadyQueue() 228. { 229. **return** readyQueue; 230. } 232. **public** Map<String,Process> getexistProcess() 233. { 234. **return** existProcess; 235. } 237. **public** Process getcurrentProcess() 238. { 239. **return** currentProcess; 240. } 242. **public** AtomicInteger getPidGenerator() 243. { 244. **return** pidGenerator; 245. } 246. } |

Process.java

|  |
| --- |
| 1. **import** java.util.List; 2. **import** java.util.Map; 3. **import** java.util.concurrent.ConcurrentHashMap; 5. /\*\* 6. \* @BelongsProject:OS-process 7. \* @BelongsPackage:PACKAGE\_NAME 8. \* @Author:Uestc\_Xiye 9. \* @CreateTime:2020-12-01 16:52:02 10. \*/ 12. **public** **class** Process { 13. /\*\* 14. \* 变量说明 15. \* pid: 进程的id，唯一 16. \* pName: 进程的名字 17. \* priority: 进程的优先级 18. \* state: 进程的状态，有五种，具体见于State中 19. \* blockedResource: 如果进程状态为阻塞的话，这个属性指向被阻塞的资源，否则为NULL 20. \* resourceMap: 进程持有的资源和相应数量 21. \* parent: 进程的父进程 22. \* children: 进程的子进程（们） 23. \*/ 24. **private** **int** pid; 25. **private** String pName; 26. **private** **int** priority; 27. **private** State state; 28. **private** Resource blockedResource; 29. **private** ConcurrentHashMap<Resource,Integer> resourceMap; 30. **private** Process parent; 31. **private** List<Process> children; 33. **private** **static** **final** Pcb pcb=Pcb.getpcb(); 34. **private** **static** **final** Queue readyQueue=Queue.getreadyQueue(); 36. /\*\* 37. \* 进程的五种状态 38. \* NEW: 新建状态 39. \* READY: 就绪状态 40. \* RUNNING: 执行状态 41. \* BLOCKED: 阻塞状态 42. \* TERMINATED: 终止状态 43. \*/ 44. **public** **enum** State 45. { 46. NEW,READY,RUNNING,BLOCKED,TERMINATED 47. } 49. **public** Process(**int** pid,String pName,**int** priority,State state,ConcurrentHashMap<Resource,Integer> resourceMap,Process parent,List<Process> children) 50. { 51. **this**.pid=pid; 52. **this**.pName=pName; 53. **this**.priority=priority; 54. **this**.state=state; 55. **this**.resourceMap=resourceMap; 56. **this**.parent=parent; 57. **this**.children=children; 58. } 60. **public** **void** deleteChild(Process process) 61. { 62. **for**(Process child:children) 63. { 64. **if**(child==process) 65. { 66. children.remove(child); 67. **return**; 68. } 69. } 70. } 72. **public** **void** deleteProcessTree() 73. { 74. **if**(!children.isEmpty()) 75. { 76. **for**(**int** i=0;i<children.size();i++) 77. { 78. Process child=children.get(0); 79. child.deleteProcessTree();// 递归删除子树 80. } 81. } 82. // 对不同状态的进程处理 84. // 若进程状态为终止状态，说明删除成功 85. **if**(**this**.getstate()==State.TERMINATED) 86. { 87. pcb.deleteProcess(**this**); 88. **return**; 89. } 90. // 若进程状态为就绪状态，则从就绪队列删除，修改其状态为终止状态 91. **else** **if**(**this**.getstate()==State.READY) 92. { 93. readyQueue.deleteProcess(**this**); 94. pcb.deleteProcess(**this**); 95. **this**.setstate(State.TERMINATED); 96. } 97. // 若进程状态为阻塞状态，则从阻塞队列删除，修改其状态为终止状态 98. **else** **if**(**this**.getstate()==State.BLOCKED) 99. { 100. Resource blockedResource=**this**.getblockedResource(); 101. blockedResource.deleteblockedProcess(**this**); 102. pcb.deleteProcess(**this**); 103. **this**.setstate(State.TERMINATED); 104. } 105. // 若进程状态为运行状态时直接终止，则修改其状态为终止状态 106. **else** **if**(**this**.getstate()==State.RUNNING) 107. { 108. pcb.deleteProcess(**this**); 109. **this**.setstate(State.TERMINATED); 110. } 111. // 清除进程的parent和child指针 112. parent.deleteChild(**this**); 113. parent=**null**; 114. // 释放资源 115. **for**(Resource resource:resourceMap.keySet()) 116. { 117. resource.releaseResource(**this**); 118. } 119. **return**; 120. } 122. **public** **void** destroy() 123. { 124. deleteProcessTree(); 125. Pcb.scheduler(); 126. **return**; 127. } 129. **public** **void** setpid(**int** pid) 130. { 131. **this**.pid=pid; 132. } 134. **public** **void** setpName(String pName) 135. { 136. **this**.pName=pName; 137. } 139. **public** **void** setpriority(**int** priority) 140. { 141. **this**.priority=priority; 142. } 144. **public** **void** setstate(State state) 145. { 146. **this**.state=state; 147. } 149. **public** **void** setblockedResource(Resource blockedResource) 150. { 151. **this**.blockedResource=blockedResource; 152. } 154. **public** **void** setrMap(ConcurrentHashMap<Resource,Integer> resourceMap) 155. { 156. **this**.resourceMap=resourceMap; 157. } 159. **public** **void** setparent(Process parent) 160. { 161. **this**.parent=parent; 162. } 164. **public** **void** setchildren(List<Process> children) 165. { 166. **this**.children=children; 167. } 169. **public** **int** getpid() 170. { 171. **return** pid; 172. } 174. **public** String getpName() 175. { 176. **return** pName; 177. } 179. **public** **int** getpriority() 180. { 181. **return** priority; 182. } 184. **public** State getstate() 185. { 186. **return** state; 187. } 189. **public** Resource getblockedResource() 190. { 191. **return** blockedResource; 192. } 194. **public** ConcurrentHashMap<Resource,Integer> getresourceMap() 195. { 196. **return** resourceMap; 197. } 199. **public** Process getparent() 200. { 201. **return** parent; 202. } 204. **public** List<Process> getchildren() 205. { 206. **return** children; 207. } 209. } |

Resource.java

|  |
| --- |
| 1. **import** java.util.Deque; 2. **import** java.util.LinkedList; 3. **import** java.util.Map; 5. /\*\* 6. \* @BelongsProject:OS-process 7. \* @BelongsPackage:PACKAGE\_NAME 8. \* @Author:Uestc\_Xiye 9. \* @CreateTime:2020-12-01 16:51:48 10. \*/ 11. **public** **class** Resource { 12. /\*\* 13. \* 变量说明 14. \* rid: 资源的id，唯一 15. \* maxResource: 资源的最大数量 16. \* remainingResource: 剩余的资源的数量 17. \* blockedDeque: 在一个资源上阻塞的进程队列 18. \*/ 19. **private** **int** rid; 20. **private** **int** maxResource; 21. **private** **int** remainingResource; 22. **private** Deque<BlockedProcess> blockedDeque; 23. **private** **static** **final** Pcb pcb=Pcb.getpcb(); 24. **private** **static** **final** Queue readyQueue=Queue.getreadyQueue(); 26. **public** **class** BlockedProcess 27. { 28. /\*\* 29. \* 变量说明 30. \* process: 进程 31. \* need: 需要请求的资源数量 32. \*/ 33. **private** Process process; 34. **private** **int** need; 36. **public** BlockedProcess(Process process,**int** need) 37. { 38. **this**.process=process; 39. **this**.need=need; 40. } 42. **public** **void** setprocess(Process process) 43. { 44. **this**.process=process; 45. } 47. **public** **void** setneed(**int** need) 48. { 49. **this**.need=need; 50. } 52. **public** Process getprocess() 53. { 54. **return** process; 55. } 57. **public** **int** getneed() 58. { 59. **return** need; 60. } 61. } 63. **public** Resource(**int** rid,**int** maxResource) 64. { 65. **this**.rid=rid; 66. **this**.maxResource=maxResource; 67. **this**.remainingResource=maxResource; 68. blockedDeque=**new** LinkedList<>(); 69. } 71. **public** **void** addremainingResource(**int** num) 72. { 73. **this**.remainingResource+=num; 74. } 76. **public** **void** deleteblockedProcess(Process process) 77. { 78. **for**(BlockedProcess blockedProcess:blockedDeque) 79. { 80. **if**(blockedProcess.getprocess()==process) 81. { 82. blockedDeque.remove(blockedProcess); 83. } 84. } 85. } 87. **public** **void** requestResource(Process process,**int** need) 88. { 89. // 若请求数量大于最大数量,则请求失败 90. **if**(need>maxResource) 91. { 92. System.out.println("Request Resource Failed!"); 93. **return**; 94. } 95. // 对于非init进程,将该进程加入阻塞队列，并设置进程为阻塞状态 96. **else** **if**(need>remainingResource && !"init".equals(process.getpName())) 97. { 98. blockedDeque.addLast(**new** BlockedProcess(process,need)); 99. process.setstate(Process.State.BLOCKED); 100. process.setblockedResource(**this**); 101. Pcb.scheduler(); 102. **return**; 103. } 104. // 对于init进程,不阻塞 105. **else** **if**(need>remainingResource && "init".equals(process.getpName())) 106. { 107. **return**; 108. } 109. // 若可以正常分配资源，则剩余资源的数量减少，已分配资源的数量增加 110. **else** 111. { 112. remainingResource-=need; 113. Map<Resource,Integer> resourceMap=process.getresourceMap(); 114. **if**(resourceMap.containsKey(**this**)) 115. { 116. Integer alreadyNum=resourceMap.get(**this**); 117. resourceMap.put(**this**,alreadyNum+need); 118. } 119. **else** 120. { 121. resourceMap.put(**this**,need); 122. } 123. } 124. } 126. **public** **void** releaseResource(Process process) 127. { 128. **int** num=0; 129. num=process.getresourceMap().remove(**this**); 130. **if**(num==0) 131. { 132. **return**; 133. } 134. remainingResource+=num; 135. **while**(!blockedDeque.isEmpty()) 136. { 137. BlockedProcess blockedProcess = blockedDeque.peekFirst(); 138. **int** need=blockedProcess.getneed(); 139. // 若剩余资源数量大于need，则唤醒阻塞队列队头的一个进程 140. **if**(remainingResource>= need) 141. { 142. Process readyProcess=blockedProcess.getprocess(); 143. requestResource(readyProcess,need); 144. blockedDeque.removeFirst(); 145. readyQueue.addprocess(readyProcess); 146. readyProcess.setstate(Process.State.READY); 147. readyProcess.setblockedResource(**null**); 148. // 若唤醒的进程优先级高于当前进程优先级,则抢占执行 149. **if**(readyProcess.getpriority()>pcb.getcurrentProcess().getpriority()) 150. { 151. pcb.preempt(readyProcess,pcb.getcurrentProcess()); 152. } 153. } 154. **else** 155. { 156. **break**; 157. } 158. } 159. } 161. **public** **void** releaseResource(Process process,**int** num) 162. { 163. **if**(num==0) 164. { 165. **return**; 166. } 167. remainingResource+=num; 168. **while**(!blockedDeque.isEmpty()) 169. { 170. BlockedProcess blockedProcess = blockedDeque.peekFirst(); 171. **int** need=blockedProcess.getneed(); 172. // 若剩余资源数量大于need，则唤醒阻塞队列队头的一个进程 173. **if**(remainingResource>= need) 174. { 175. Process readyProcess=blockedProcess.getprocess(); 176. requestResource(readyProcess,need); 177. blockedDeque.removeFirst(); 178. readyQueue.addprocess(readyProcess); 179. readyProcess.setstate(Process.State.READY); 180. readyProcess.setblockedResource(**null**); 181. // 若唤醒的进程优先级高于当前进程优先级,则抢占执行 182. **if**(readyProcess.getpriority()>pcb.getcurrentProcess().getpriority()) 183. { 184. pcb.preempt(readyProcess,pcb.getcurrentProcess()); 185. } 186. } 187. **else** 188. { 189. **break**; 190. } 191. } 192. } 194. **public** **void** printStatus() 195. { 196. StringBuilder stringBuilder=**new** StringBuilder(); 197. stringBuilder.append("resource-").append(rid) 198. .append("{maxResource=").append(maxResource) 199. .append(",remainingResource:").append(remainingResource) 200. .append(",") 201. .append("blockedDeque["); 202. **for**(BlockedProcess bProcess:blockedDeque) 203. { 204. stringBuilder.append(",{") 205. .append(bProcess.getprocess().getpName()) 206. .append(":") 207. .append(bProcess.getneed()) 208. .append("}"); 209. } 210. stringBuilder.append("]}"); 211. String result=stringBuilder.toString(); 212. System.out.println(result.replace("[,", "[")); 213. } 215. **public** **void** setrid(**int** rid) 216. { 217. **this**.rid=rid; 218. } 220. **public** **void** setmaxResource(**int** maxResource) 221. { 222. **this**.maxResource=maxResource; 223. } 225. **public** **void** setremainingResource(**int** remainingResource) 226. { 227. **this**.remainingResource=remainingResource; 228. } 230. **public** **void** setblockedDeque(Deque<BlockedProcess> blockedDeque) 231. { 232. **this**.blockedDeque=blockedDeque; 233. } 235. **public** **int** getrid() 236. { 237. **return** rid; 238. } 240. **public** **int** getmaxResource() 241. { 242. **return** maxResource; 243. } 245. **public** **int** getremainingResource() 246. { 247. **return** remainingResource; 248. } 250. **public** Deque<BlockedProcess> getblockedDeque() 251. { 252. **return** blockedDeque; 253. } 255. } |

Queue.java

|  |
| --- |
| 1. **import** java.util.Deque; 2. **import** java.util.LinkedList; 4. /\*\* 5. \* @BelongsProject:OS-process 6. \* @BelongsPackage:PACKAGE\_NAME 7. \* @Author:Uestc\_Xiye 8. \* @CreateTime:2020-12-01 16:51:38 9. \*/ 10. **public** **class** Queue { 11. /\*\* 12. \* deque: 不同优先级就绪队列组成数组 13. \* readyQueue: 就绪队列 14. \*/ 15. **private** Deque<Process>[] deque; 16. **private** **static** **final** Queue readyQueue=**new** Queue(); 18. **private** Queue() 19. { 20. //因为进程有3种不同优先级，所以构造3个就绪队列 21. deque=**new** LinkedList[3]; 22. **for**(**int** i=0;i<3;i++) 23. { 24. deque[i]=**new** LinkedList<>(); 25. } 26. } 28. **public** **void** addprocess(Process process) 29. { 30. **int** priority=process.getpriority(); 31. Deque<Process> d=deque[priority]; 32. d.addLast(process); 33. } 35. **public** **boolean** deleteProcess(Process process) 36. { 37. **int** priority = process.getpriority(); 38. Deque<Process> d=deque[priority]; 39. **return** d.remove(process); 40. } 42. **public** **static** Queue getreadyQueue() 43. { 44. **return** readyQueue; 45. } 47. **public** Process getprocess() 48. { 49. **for** (**int** i=2;i>=0;i--) 50. { 51. Deque<Process> d=deque[i]; 52. **if**(!d.isEmpty()) 53. { 54. **return** d.peekFirst(); 55. } 56. } 57. **return** **null**; 58. } 59. } |

**八、实验数据及结果分析**

将测试命令放在测试文件input.txt中，内容为：

1. cr x 1
2. cr p 1
3. cr q 1
4. cr r 1
5. to
6. req R2 1
7. to
8. req R3 3
9. to
10. req R4 3
11. to
12. to
13. req R3 1
14. req R4 2
15. req R2 2
16. to
17. de q
18. to
19. to

将程序打包成Jar文件，放到与input.txt相同的目录下执行，结果如图6所示：



图6 程序执行结果图

经过对比与验证，该输出结果与实验指导书中给出的预期输出结果一致，实验成功。

**九、总结及心得体会：**

本次实验完成了main类、Pcb类、Process类、Resource类、Queue类的设计和代码实现，加深了对于进程与资源管理的理解，建立了系统的进程管理、调度、资源管理和分配的知识体系，并且加深了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。在设计并实现一个完整的进程与资源管理系统的过程中，采用了面向对象的编程思想以及模块化编程的实现方式，提高了编程能力。

**十、对本实验过程及方法、手段的改进建议：**

1. 多测试几组命令，验证代码的可靠性；

2. 更加完善代码注释，提高代码的可读性。