

# 电子科技大学

## 实验报告

学生姓名：关文聪 学号：2016060601008 指导教师：薛瑞尼

实验地点：主楼 A2-412

实验时间：2019 年 5 月

一、实验室名称：计算机实验室

二、实验项目名称：进程与资源管理实验

三、实验学时：4

四、实验原理：

### 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。

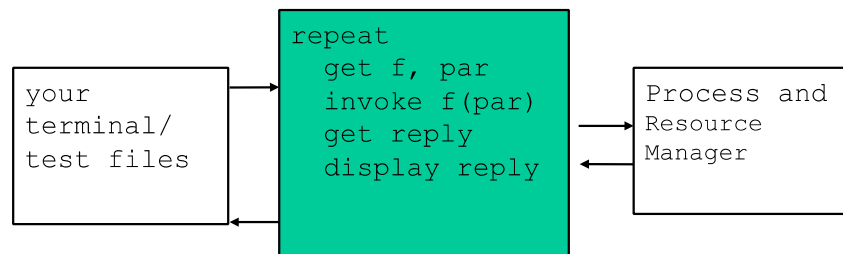


图 1 系统总体结构

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。

## 2. Test Shell 设计

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
```

(注: 具体的功能实现此处略去, 详见第八部分: 实验步骤)

## 3. 进程管理设计

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

## 4. 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)

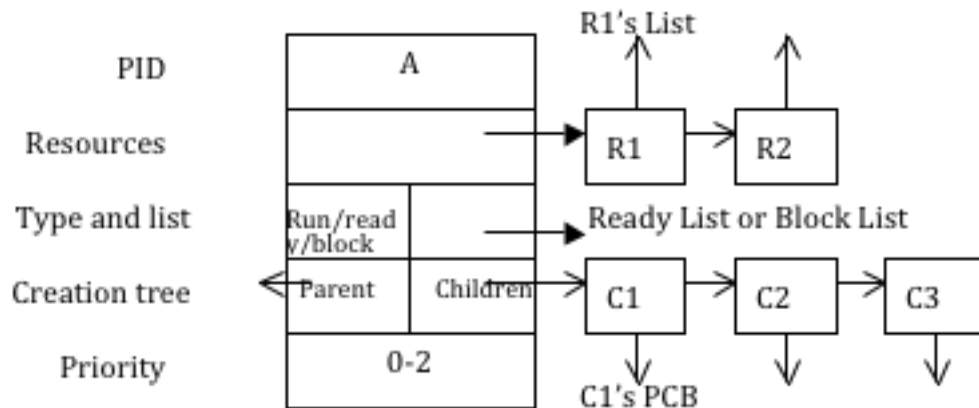


图 2 PCB 结构示意图

就绪进程队列: Ready list (RL)

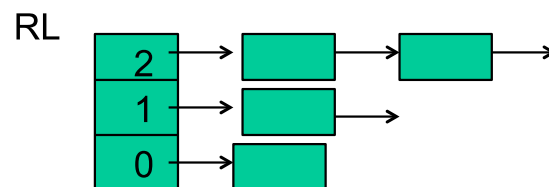


图 3 Ready list 数据结构

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。

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：

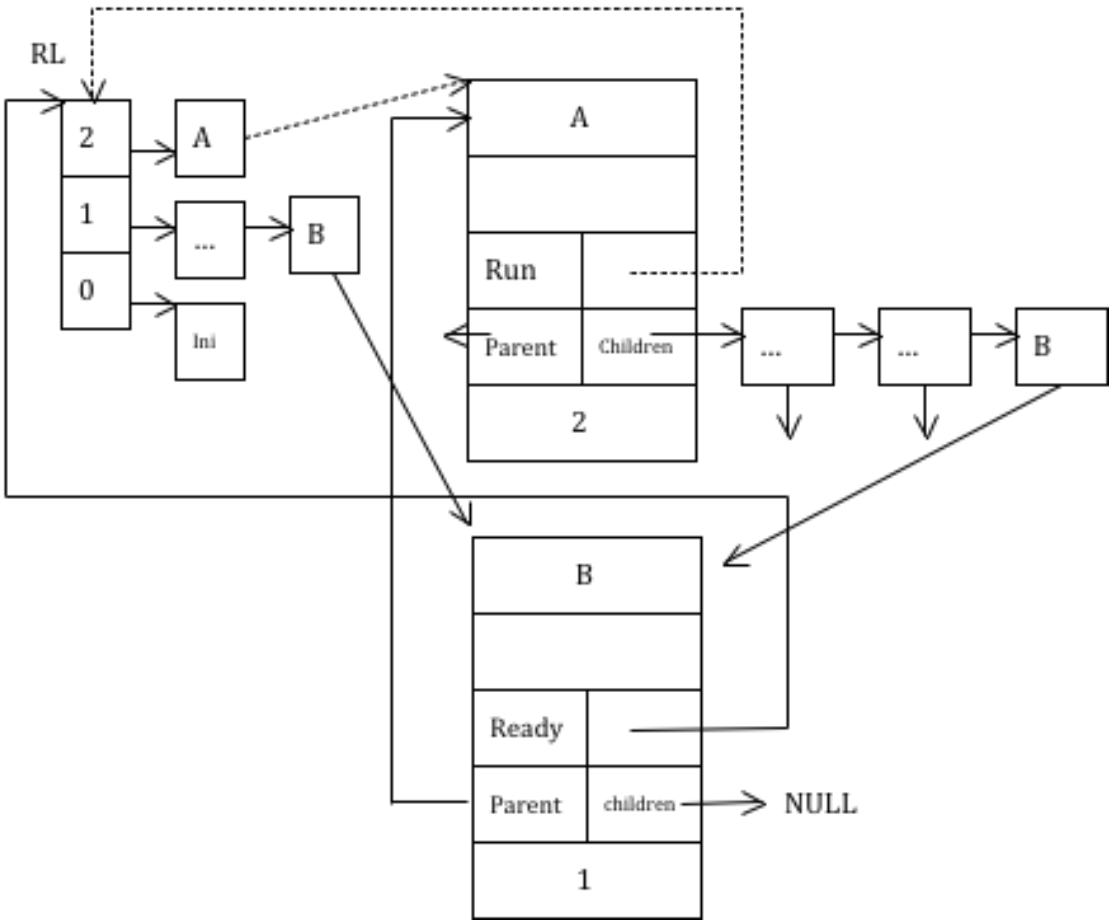


图 4 进程数据结构间关系

(为了简单起见，A 和 B 分别指向 RL 的链接可以不要)

5. 资源管理设计

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

## 6. 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：
  - (1) 虚设的进程：具有最低的优先级，永远不会被阻塞
  - (2) 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

## 7. 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

## 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。

## 七、实验器材（设备、元器件）：

个人计算机、java version "1.8.0\_162"、JetBrains IntelliJ IDEA (Ultimate Version) 2019

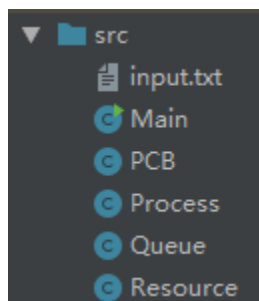
## 八、实验步骤：

### a) 系统功能需求分析：

与第四部分（实验原理）相同，系统总体上由 Test Shell、进程管理模块部分、进程控制块结构（PCB）、资源管理模块部分、进程调度与时钟中断模块部分等组成。其中，各部分的原理以及具体结构在第四部分（实验原理）已经描述过，此处不再赘述，具体分析与代码详见第四部分（实验原理）以及详细设计部分。

### b) 总体框架设计：

项目的总体框架与结构如图所示：



其中 Main 类是系统的主函数入口部分，主要对应实现的是 Test Shell 部分，根据输入的命令进行解析，调用系统的不同功能模块，并输出相应结果。

PCB 类是整个系统的主要核心功能实现类，用于进程的管理，主要对应的是

进程管理、进程控制块结构以及进程调度与时钟中断部分。该类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等功能。

Process 类是进程的实体类,该类定义了进程的数据结构,实现了对应的 get、set 方法,并且主要实现了删除进程和子进程树的功能。

Queue 类是队列类, 该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。

Resource 类是资源的实体类, 该类定义了资源的数据结构, 实现了对应的 get、set 方法, 并且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。

#### c) 详细设计:

Test Shell 部分: 是系统的中心部分, 起着读取命令、连接核心部件、输出结果的作用。首先, 通过 Test Shell 读入各种命令, 在系统的设计中, 同时设计了从命令行输入命令以及从文件读取两种输入方式。然后, 对命令进行分析, 将用户的需求转换成调度内核函数, 也就是说, 通过调度进程和资源管理器, 实现创建进程、撤销进程、进程调度、对资源进行管理、申请和释放资源、检测错误和定时器中断等功能, 从而模拟一个操作系统对进程进行调度和对资源进行管理的过程。最后, 在终端或者输出文件中, 把一系列操作后的结果显示出来, 包括当前运行的进程、错误信息等。

Test Shell 部分核心代码 (注: 因篇幅所限仅展现核心代码, 完整代码另附):

```
1. // 对输入的命令进行处理, 适用于键盘录入或者文件读入
2. public static void exec(String input) {
3.     String[] commands = new String[]{input};
4.     for (String command : commands) { //对不同的输入命令进行处理
5.         String[] cmds = command.split("\\s+");
6.         String options = cmds[0];
7.         switch (options) {
8. //             case "init":
9. //                 if (pcb.findProcess("init") != null) { // 已经完成了初始化, 系统中已经有 init 进程
10. //                     System.out.println("错误! 已完成过初始化! init 进程已存在!");
11. //                 }
12. //             default:
13. //                 // 处理其他命令
14. //             }
15.         }
16.     }
17. }
```

```

11. //                      } else {
12. //                      pcb.createProcess("init", 0); // 创建 init 进
    程, 优先级设定为 0
13. //                      System.out.println("初始化成功! init 进程已创建!
    ");
14. //                      }
15. //                      break;
16.                      case "cr":
17. //                      if (pcb.findProcess("init") == null) { // 检查是否完成
    了初始化
18. //                      System.out.println("错误! 系统未初始化! 请先执行 init
    命令初始化! ");
19. //                      } else
20.                      if (cmds.length != 3) { // 检查输入格式是否正确
21.                      System.out.println("错误! 请输入合法的参数! ");
22.                      } else {
23.                      String processName = cmds[1];
24.                      int priority = 0;
25.                      try { // 检查优先级的输入是否正确
26.                      priority = Integer.parseInt(cmds[2]);
27.                      if (priority <= 0 || priority > 2) {
28.                      System.out.println("错误! 请输入合法的参数!
    ");
29.                      continue;
30.                      }
31.                      } catch (Exception e) {
32.                      System.out.println("错误! 请输入合法的参数! ");
33.                      }
34.                      if (pcb.exsitName(processName)) { // 检查用户输入的进
    程名是否已经存在
35.                      System.out.println("错误! 进程名
    " + processName + "已经存在! 请选择其它的进程名! ");
36.                      break;
37.                      }
38.                      pcb.createProcess(processName, priority);
39.                      }
40.                      break;
41.                      case "de":
42. //                      if (pcb.findProcess("init") == null) { // 检查是否完成
    了初始化
43. //                      System.out.println("错误! 系统未初始化! 请先执行 init
    命令初始化! ");
44. //                      } else
45.                      if (cmds.length != 2) { // 检查输入格式是否正确

```



```

46.         System.out.println("错误！请输入合法的参数！");
47.     } else {
48.         String processName = cmds[1];
49.         Process process = pcb.findProcess(processName);
50.         if (process == null) { // 检查用户输入的进程名是否已经
            存在
51.             System.out.println("错误！没有名为
                " + processName + "的进程！");
52.         } else if (processName.equals("init")) { // 设定不允
            许用户删除系统 init 进程
53.             System.out.println("错误！您没有权限终止 init 进程！
                ");
54.         } else {
55.             process.destroy();
56.             // process.killSubTree(); // 将进程自身包括其所有子
            进程终止
57.             // PCB.scheduler();
58.             // System.out.println("终止进程成功！");
59.         }
60.     }
61.     break;
62.     case "req":
63.         // if (pcb.findProcess("init") == null) { // 检查是否完成
            了初始化
64.             // System.out.println("错误！系统未初始化！请先执行 init
            命令初始化！");
65.         } else
66.         if (cmds.length != 3) { // 检查输入格式是否正确
67.             System.out.println("错误！请输入合法的参数！");
68.         } else {
69.             String resourceName = cmds[1];
70.             int needNum = 0;
71.             try {
72.                 needNum = Integer.parseInt(cmds[2]);
73.             } catch (Exception e) {
74.                 System.out.println("错误！请输入合法的参数！");
75.             }
76.             Process currentProcess = pcb.getCurrentProcess(); //
            获取当前进程
77.             switch (resourceName) { // 检查资源名称，请求对应资源
78.                 case "R1":
79.                     R1.request(currentProcess, needNum);
80.                     break;
81.                 case "R2":

```

```

82.                R2.request(currentProcess, needNum);
83.                break;
84.            case "R3":
85.                R3.request(currentProcess, needNum);
86.                break;
87.            case "R4":
88.                R4.request(currentProcess, needNum);
89.                break;
90.            default:
91.                System.out.println("错误！请输入合法的参数！
    ");
92.        }
93.    }
94.    break;
95.    case "rel":
96.        //            if (pcb.findProcess("init") == null) { // 检查是否完成
    了初始化
97.        //                System.out.println("错误！系统未初始化！请先执行 init
    命令初始化！");
98.        //            } else
99.            if (cmds.length != 3) { // 检查输入格式是否正确
100.                System.out.println("错误！请输入合法的参数！");
101.            } else {
102.                String resourceName = cmds[1];
103.                int relNum = 0;
104.                try {
105.                    relNum = Integer.parseInt(cmds[2]);
106.                } catch (Exception e) {
107.                    System.out.println("错误！请输入合法的参数！");
108.                }
109.                Process currentProcess = pcb.getCurrentProcess(); /
    / 获取当前进程
110.                switch (resourceName) { // 检查资源名称，释放对应资
    源
111.                    case "R1":
112.                        R1.release(currentProcess, relNum);
113.                        break;
114.                    case "R2":
115.                        R2.release(currentProcess, relNum);
116.                        break;
117.                    case "R3":
118.                        R3.release(currentProcess, relNum);
119.                        break;
120.                    case "R4":

```

```

121.                R4.release(currentProcess, relNum);
122.                break;
123.            default:
124.                System.out.println("错误！请输入合法的参数！
");
125.            }
126.        }
127.        break;
128.        case "to":
129.            pcb.timeout();
130.            break;
131.        case "lp":
132.            //            if (pcb.findProcess("init") == null) { // 检查是否完成
                了初始化
133.            //            System.out.println("错误！系统未初始化！请先执行
                init 命令初始化！");
134.            //            }
135.            if (cmds.length == 1) { // lp 命令打印所有进程树和信息
136.                pcb.printProcessTree(pcb.findProcess("init"), 0);
137.            } else if (cmds.length < 3 || !cmds[1].equals("-
                p")) { // lp -p pname 命令打印某具体进程的信息
138.                System.out.println("错误！请输入合法的参数或命令！
");
139.            } else {
140.                String pname = cmds[2];
141.                Process process = pcb.findProcess(pname);
142.                if (process == null) {
143.                    System.out.println("错误！没有名为" + pname + "
                的进程！");
144.                } else {
145.                    pcb.printProcessDetail(process);
146.                }
147.            }
148.            break;
149.        case "lr":
150.            R1.printCurrentStatus();
151.            R2.printCurrentStatus();
152.            R3.printCurrentStatus();
153.            R4.printCurrentStatus();
154.            break;
155.        case "help":
156.            printHelp();
157.            break;
158.        case "exit":

```

```

159.             System.out.println("Good Bye! ");
160.             System.exit(0);
161. //             case "list":
162. //                 pcb.printExistProcess();
163. //                 break;
164.             default:
165.                 System.out.println("错误！请输入合法的命令！");
166.                 break;
167.         }
168.     }
169.     if (pcb.getCurrentProcess() != null) {
170.         System.out.print(pcb.getCurrentProcess().getProcessName() + "
");
171.     }
172. }

```

进程管理部分：是系统的核心部分，具体实现进程的创建、撤销、请求资源、释放资源、时钟中断、调度等核心功能。均采用面向对象的编程方法实现。

进程的定义：

```

1. private int PID; // 进程 ID
2. private String processName; // 进程名
3. private int priority; // 进程优先级
4. private State state; // 进程状态
5. private ConcurrentHashMap<Resource, Integer> resourceMap; // 进程持有的资源和相应数量
6. private Resource blockResource; // 如果进程状态为阻塞的话，这个属性就指向被阻塞的资源，否则应该为 null
7. private Process parent; // 进程的父进程
8. private List<Process> children; // 进程的子进程
9.
10. // 进程的五状态：NEW（新建），READY（就绪），RUNNING（运行），BLOCKED（阻塞），TERMINATED（终止）
11. public enum State {
12.     NEW, READY, RUNNING, BLOCKED, TERMINATED
13. }
14.
15. public Process(int PID, String processName, int priority, State state, ConcurrentHashMap<Resource, Integer> resourceMap, Process parent, List<Process> children) {
16.     this.PID = PID;
17.     this.processName = processName;

```

```

18.     this.priority = priority;
19.     this.state = state;
20.     this.resourceMap = resourceMap;
21.     this.parent = parent;
22.     this.children = children;
23. }

```

## 进程的创建：

```

1. public Process createProcess(String processName, int priority) {
2.     Process currentProcess = pcb.getCurrentProcess();
3.     // 为新建进程分配 PID, 进程名, 优先级, 进程状态, 资源, 父进程和子进程信息
   等
4.     Process process = new Process(pcb.generatePID(), processName, priority, Process.State.NEW, new ConcurrentHashMap<>(), currentProcess, new LinkedList<>());
5.     if (currentProcess != null) { // 排除创建的进程为第一个进程的特殊情况
6.         currentProcess.getChildren().add(process); // 新创建进程作为当前进程的子进程
7.         process.setParent(currentProcess); // 旧进程作为新创建进程的父进程
8.     }
9.     pcb.addExistList(process); // 将新创建的进程放在 ExistList 中
10.    readyQueue.addProcess(process); // 将新创建的进程放入就绪队列中
11.    process.setState(Process.State.READY); // 成功进入就绪队列的进程, 其状态将置为就绪状态
12.    PCB.scheduler(); // 调度
13.    return process;
14. }

```

进程的撤销（即进程的删除，此处涉及到的操作比较复杂，首先需要断开进程与父进程的连接，并对以该进程为根的进程子树递归删除，修改进程状态为终止状态并完成释放资源等相关操作）：

```

1. // 删除进程时调用
2. public void destroy() {
3.     killSubTree();
4.     PCB.scheduler();
5.     return;
6. }
7.

```

```

8. // 删除子进程
9. public void removeChild(Process process) {
10.     for (Process child : children) {
11.         if (child == process) {
12.             children.remove(child);
13.             return;
14.         }
15.     }
16. }
17.
18. // 删除进程以及以其为根的进程树
19. public void killSubTree() {
20.     if (!children.isEmpty()) { // 当进程子树不为空
21.         int childNum = children.size();
22.         for (int i = 0; i < childNum; i++) {
23.             Process child = children.get(0);
24.             child.killSubTree(); // 递归删除子树
25.         }
26.     }
27.     // 对不同状态的进程处理
28.     if (this.getState() == State.TERMINATED) { // 进程状态已为终止状态，说明删除成功
29.         pcb.killProcess(this);
30.         return;
31.     } else if (this.getState() == State.READY) { // 进程状态为就绪状态，从就绪队列删除，修改其状态为终止状态
32.         readyQueue.removeProcess(this);
33.         pcb.killProcess(this);
34.         this.setState(State.TERMINATED);
35.     } else if (this.getState() == State.BLOCKED) { // 进程状态为阻塞状态，从阻塞队列删除，修改其状态为终止状态
36.         Resource blockResource = this.getBlockResource();
37.         blockResource.removeBlockProcess(this);
38.         pcb.killProcess(this);
39.         this.setState(State.TERMINATED);
40.     } else if (this.getState() == State.RUNNING) { // 进程状态为运行状态时直接终止，修改其状态为终止状态
41.         this.setState(State.TERMINATED);
42.         pcb.killProcess(this);
43.     }
44.     // 清除进程的 parent 和 child 指针
45.     parent.removeChild(this);
46.     parent = null;
47.     // 释放资源

```

```

48.     for (Resource resource : resourceMap.keySet()) {
49.         resource.release(this);
50.     }
51.     return;
52. }

```

进程申请资源（需要判断能否成功申请，若失败，进程将阻塞）：

```

1.     // 进程请求资源
2.     public void request(Process process, int need) {
3.         if (need > max) { // 请求数量大于最大数量时申请失败
4.             System.out.println("请求资源失败！请求资源大于最大数量！");
5.             return;
6.         } else if (need > remaining && !"init".equals(process.getProcessName(
7.             ))) { // 对于非 init 进程需要阻塞
8.             blockDeque.addLast(new BlockProcess(process, need)); // 加入阻塞
9.             队列
10.            process.setState(Process.State.BLOCKED); // 设置进程为阻塞状态
11.            process.setBlockResource(this);
12.            PCB.scheduler(); //调度
13.            // System.out.println("资源申请失败，进程阻塞");
14.            return;
15.        } else if (need > remaining && "init".equals(process.getProcessName(
16.            ))) { //init 进程不阻塞
17.            // System.out.println("资源申请失败，进程阻塞");
18.            return;
19.        } else { // 可正常分配资源
20.            remaining = remaining - need; // 剩余资源数量减少
21.            Map<Resource, Integer> resourceMap = process.getResourceMap();
22.            if (resourceMap.containsKey(this)) {
23.                Integer alreadyNum = resourceMap.get(this);
24.                resourceMap.put(this, alreadyNum + need); // 已分配资源增加
25.            } else {
26.                resourceMap.put(this, need);
27.            }
28.        }
29.    }
30. }

```

进程释放资源（释放后若有阻塞进程能唤醒则将其唤醒）：

注意：此处设计了两个同名函数实现重载，只有一个参数的函数实现的是释放当前持有的所有资源，带有两个参数的函数实现的是释放指定数量的资源。

```

1.      // 进程释放资源并唤醒阻塞进程
2.      public void release(Process process) {
3.          int num = 0;
4.          num = process.getResourceMap().remove(this);
5.          if (num == 0) {
6.              return;
7.          }
8.          remaining = remaining + num; // 释放资源
9.          while (!blockDeque.isEmpty()) {
10.             BlockProcess blockProcess = blockDeque.peekFirst();
11.             int need = blockProcess.getNeed();
12.             if (remaining >= need) { // 若剩余资源数量大于 need，则可以唤醒阻塞
                队列队头的一个进程
13.                 Process readyProcess = blockProcess.getProcess(); // 从阻塞队
                列取出进程
14.                 request(readyProcess, need); // 进程请求资源
15.                 blockDeque.removeFirst(); // 从阻塞队列移除该进程
16.                 readyQueue.addProcess(readyProcess); // 加入就绪队列
17.                 readyProcess.setState(Process.State.READY); // 进程设为就绪状
                态
18.                 readyProcess.setBlockResource(null); // 此时已没有被阻塞资源
19.                 if (readyProcess.getPriority() > pcb.getCurrentProcess().get
                Priority()) { // 如果唤醒的进程优先级高于当前进程优先级则抢占执行
20.                     pcb.preempt(readyProcess, pcb.getCurrentProcess());
21.                 }
22.             } else {
23.                 break;
24.             }
25.         }
26.     }
27.
28.
29.      // 进程释放资源并唤醒阻塞进程
30.      public void release(Process process, int num) {
31.          //      if (num > process.getResourceMap().get(this)) {
32.          //          System.out.println("错误！请输入合法的参数");
33.          //          return;
34.          //      }
35.          if (num == 0) {
36.              return;
37.          }
38.          //      if (num == process.getResourceMap().get(this)) {

```



```

39. //          num = process.getResourceMap().remove(this);
40. //      }
41.     remaining = remaining + num; // 释放资源
42.     while (!blockDeque.isEmpty()) {
43.         BlockProcess blockProcess = blockDeque.peekFirst();
44.         int need = blockProcess.getNeed();
45.         if (remaining >= need) { // 若剩余资源数量大于 need，则可以唤醒阻塞
            队列队头的一个进程
46.             Process readyProcess = blockProcess.getProcess();// 从阻塞队
            列取出进程
47.             request(readyProcess, need); // 进程请求资源
48.             blockDeque.removeFirst(); // 从阻塞队列移除该进程
49.             readyQueue.addProcess(readyProcess); // 加入就绪队列
50.             readyProcess.setState(Process.State.READY); // 进程设为就绪状
            态
51.             readyProcess.setBlockResource(null); // 此时已没有被阻塞资源
52.             if (readyProcess.getPriority() > pcb.getCurrentProcess().get
                Priority()) { // 如果唤醒的进程优先级高于当前进程优先级则抢占执行
53.                 pcb.preempt(readyProcess, pcb.getCurrentProcess());
54.             }
55.         } else {
56.             break;
57.         }
58.     }
59. }

```

## 时钟中断：

```

1. // 时间片轮转（RR），时间片完后切换进程
2. public static void timeout() {
3.     pcb.getCurrentProcess().setState(Process.State.READY); // 时间片完直接将当
        前运行进程置为就绪状态
4.     scheduler(); // 调度
5. }

```

## 进程切换：

```

1. // 进程切换
2. public static void preempt(Process readyProcess, Process currentProcess) {
3.     readyQueue.addProcess(currentProcess); // 将当前进程加入就绪队列中
4.     currentProcess.setState(Process.State.READY); // 将进程状态置为就绪状态
5.     readyQueue.removeProcess(readyProcess); // 从就绪队列取出一个就绪进程
6.     pcb.setCurrentProcess(readyProcess); // 将该进程设为当前运行的进程
7.     readyProcess.setState(Process.State.RUNNING); // 该进程状态设为运行状态

```

```
8.     return;
9. }
```

## 进程调度（系统核心功能）：

```
1. // 进程调度
2. public static void scheduler() {
3.     Process currentProcess = pcb.getCurrentProcess();
4.     Process readyProcess = readyQueue.getProcess();
5.     if (readyProcess == null) { // 就绪队列为空时，CPU 正在运行的只有 init 进程
6.         pcb.getCurrentProcess().setState(Process.State.RUNNING); // 状态设为运行状态
7.         return;
8.     } else if (currentProcess == null) { // 实际上，此处只有在刚初始化系统时才可能发生
9.         readyQueue.removeProcess(readyProcess);
10.        pcb.setCurrentProcess(readyProcess);
11.        readyProcess.setState(Process.State.RUNNING);
12.        return;
13.    } else if (currentProcess.getState() == Process.State.BLOCKED || currentProcess.getState() == Process.State.TERMINATED) { // 当前进程被阻塞或者已经被终止
14.        readyQueue.removeProcess(readyProcess); // 从就绪队列取出一个就绪进程
15.        pcb.setCurrentProcess(readyProcess); // 将该进程设为当前运行的进程
16.        readyProcess.setState(Process.State.RUNNING); // 该进程状态设为运行状态
17.    } else if (currentProcess.getState() == Process.State.RUNNING) { // 新创建了进程，或者阻塞队列中进程转移到 readyList
18.        if (currentProcess.getPriority() < readyProcess.getPriority()) { // 若就绪进程优先级更高，则切换进程
19.            preempt(readyProcess, currentProcess);
20.        }
21.    } else if (currentProcess.getState() == Process.State.READY) { // 时间片完的情况
22.        if (currentProcess.getPriority() <= readyProcess.getPriority()) { // 若有优先级大于或等于当前进程的就绪进程，则切换进程
23.            preempt(readyProcess, currentProcess);
24.        } else { // 如果没有高优先级的就绪进程，则当前进程依然继续运行
25.            currentProcess.setState(Process.State.RUNNING);
26.        }
27.    }
28.    return;
29. }
```

资源管理（包括了资源的定义以及阻塞进程、阻塞队列的定义和实现）：

```
1. private int RID; //资源 ID
2. private int max; //分配的资源最大数量
3. private int remaining; //剩余的资源数量
4. private Deque<BlockProcess> blockDeque; //在该资源上阻塞的进程队列
5.
6. // 阻塞进程
7. class BlockProcess {
8.     private Process process;
9.     private int need; //需要请求的资源数量
10.
11.     public BlockProcess(Process process, int need) {
12.         this.process = process;
13.         this.need = need;
14.     }
15.
16.     public Process getProcess() {
17.         return process;
18.     }
19.
20.
21.     public int getNeed() {
22.         return need;
23.     }
24.
25. }
26.
27. public Resource(int RID, int max) {
28.     this.RID = RID;
29.     this.max = max;
30.     this.remaining = max;
31.     blockDeque = new LinkedList<>();
32. }
33.
34. // 在阻塞队列中直接删除指定进程，在终止进程时调用
35. public boolean removeBlockProcess(Process process) {
36.     for (BlockProcess bProcess : blockDeque) {
37.         if (bProcess.getProcess() == process) {
38.             blockDeque.remove(bProcess);
39.             return true;
40.         }
41.     }
42.     return false;
```

```
43. }
```

进程队列的实现与管理、维护（由一个专门的 Queue 类完成）：

```
1. import java.util.Deque;
2. import java.util.LinkedList;
3.
4. /**
5.  * 队列类，用于管理与维护进程队列
6.  */
7. public class Queue {
8.     private Deque<Process>[] deque; // 不同优先级就绪队列组成数组
9.     private static final Queue readyQueue = new Queue(); // 单例设计模式-饿汉
        式
10.
11.     private Queue() {
12.         deque = new LinkedList[3];
13.         for (int i = 0; i < 3; i++) { // 进程有 3 种不同优先级，因此构造 3 个就绪队
            列
14.             deque[i] = new LinkedList<>();
15.         }
16.     }
17.
18.     public static Queue getReadyQueue() {
19.         return readyQueue;
20.     }
21.
22.     // 将进程加入到其对应优先级的就绪队列中
23.     public void addProcess(Process process) {
24.         int priority = process.getPriority();
25.         Deque<Process> deque = deque[priority];
26.         deque.addLast(process);
27.     }
28.
29.     // 获得就绪队列里面优先级最高的进程。若队列为空，则返回 null
30.     public Process getProcess() {
31.         for (int i = 2; i >= 0; i--) { // 因为是获取优先级最高的进程，因此应该按
            优先级从高到低遍历
32.             Deque<Process> deque = deque[i];
33.             if (!deque.isEmpty()) {
34.                 return deque.peekFirst(); // 当队列不空时返回队列中第一个进程
35.             }
36.         }
37.         return null; // 若队列为空，则返回 null
```

```

38.     }
39.
40.     // 删除就绪队列里面指定的进程。删除成功返回 true，若进程不存在就返回 false。
41.     public boolean removeProcess(Process process) {
42.         int priority = process.getPriority();
43.         Deque<Process> deque = deques[priority];
44.         return deque.remove(process);
45.     }
46. }

```

d) 测试：

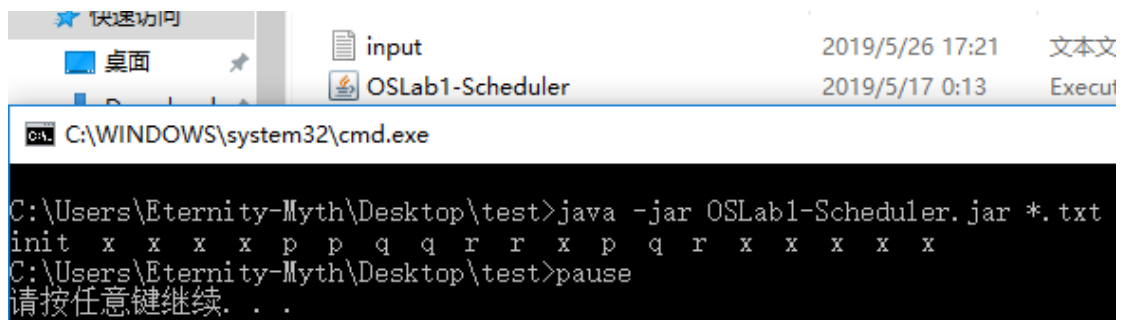
输入测试命令或将测试命令放在测试文件 input.txt 中，内容为：

```

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

```

将源代码打包成 Jar 文件，放到与 input.txt 相同的目录下执行，结果如下图所示：



```
C:\Users\Eternity-Myth\Desktop\test>java -jar OSLab1-Scheduler.jar *.txt
init x x x x p p q q r r x p q r x x x x x
C:\Users\Eternity-Myth\Desktop\test>pause
请按任意键继续. . .
```

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是一致的，说明实验成功。

### 结果分析：

首先，在系统初始启动时，创建 `init` 进程，其优先级为 0，输出当前正在执行的进程为 `init`。初始化资源 `R1`、`R2`、`R3`、`R4`，数量分别为 1、2、3、4。随后，输入命令“`cr x 1`”创建进程 `x`，其优先级为 1，由于 `x` 的优先级高于 `init` 进程，因此，创建完成后即调度进程 `x` 执行，输出当前正在执行的进程为 `x`。接下来，输入命令“`cr p 1`”“`cr q 1`”“`cr r 1`”依次创建进程 `p`、`q`、`r`，由于这三个进程的优先级都是 1，不高于当前执行的进程 `x`，因此，创建完成后系统当前执行进程依然为 `x`，分别输出当前执行进程为 `x`，此时优先级为 1 的就绪队列中依次为进程 `p`、`q`、`r`。

输入命令“`to`”后，进程 `x` 时间片完，回到就绪队列。系统调度执行进程 `p`，输出当前执行进程为 `p`，此时优先级为 1 的就绪队列中依次为进程 `q`、`r`、`x`。

输入命令“`req R2 1`”，为进程 `p` 申请 1 个 `R2` 资源。由于申请数量少于 `R2` 资源剩余数量，因此可以申请，不会发生阻塞，输出当前执行进程为 `p`，`R2` 资源剩余量为  $2-1=1$ ，优先级为 1 的就绪队列中依次为进程 `q`、`r`、`x`。

输入命令“`to`”后，进程 `p` 时间片完，回到就绪队列。系统调度执行进程 `q`，输出当前执行进程为 `q`，此时优先级为 1 的就绪队列中依次为进程 `r`、`x`、`p`。

输入命令“`req R3 3`”，为进程 `q` 申请 3 个 `R3` 资源。由于申请数量少于 `R3` 资源剩余数量，因此可以申请，不会发生阻塞，输出当前执行进程为 `q`，`R3` 资源剩余量为  $3-3=0$ ，优先级为 1 的就绪队列中依次为进程 `r`、`x`、`p`。

输入命令“`to`”后，进程 `q` 时间片完，回到就绪队列。系统调度执行进程 `r`，输出当前执行进程为 `r`，此时优先级为 1 的就绪队列中依次为进程 `x`、`p`、`q`。

输入命令“req R4 3”，为进程 r 申请 3 个 R4 资源。由于申请数量少于 R4 资源剩余数量，因此可以申请，不会发生阻塞，输出当前执行进程为 r，R4 资源剩余量为  $4-3=1$ ，优先级为 1 的就绪队列中依次为进程 x、p、q。

输入命令“to”后，进程 r 时间片完，回到就绪队列。系统调度执行进程 x，输出当前执行进程为 x，此时优先级为 1 的就绪队列中依次为进程 p、q、r。

输入命令“to”后，进程 x 时间片完，回到就绪队列。系统调度执行进程 p，输出当前执行进程为 p，此时优先级为 1 的就绪队列中依次为进程 q、r、x。

输入命令“req R3 1”，为进程 p 申请 1 个 R3 资源。由于 R3 资源剩余数量为 0，小于申请数量，因此失败，进程 p 阻塞，系统调度执行进程 q，输出当前执行进程为 q，R3 资源剩余量为 0，优先级为 1 的就绪队列中依次为进程 r、x，R3 资源的阻塞队列为 p。

输入命令“req R4 2”，为进程 q 申请 2 个 R4 资源。由于 R4 资源剩余数量为 1，小于申请数量，因此失败，进程 q 阻塞，系统调度执行进程 r，输出当前执行进程为 r，R4 资源剩余量为 1，优先级为 1 的就绪队列中依次为进程 x，R4 资源的阻塞队列为 q。

输入命令“req R2 2”，为进程 r 申请 2 个 R2 资源。由于 R2 资源剩余数量为 1，小于申请数量，因此失败，进程 r 阻塞，系统调度执行进程 x，输出当前执行进程为 x，R2 资源剩余量为 1，此时优先级为 1 的就绪队列为空，R2 资源的阻塞队列为 r。

输入命令“to”后，进程 x 时间片完，回到就绪队列，但此时优先级为 1 的就绪队列为空，因此 x 是该就绪队列唯一的进程，系统依然调度执行进程 x，输出当前执行进程为 x，此时优先级为 1 的就绪队列依然为空。

输入命令“de q”后，删除进程 q 以及其子进程，由于进程 q 没有子进程，因此只删除了进程 q 自身，并释放其持有的 3 个 R3 资源。资源释放后，资源 R3 阻塞队列的进程 p 可以成功申请到其需要申请的 1 个 R3 资源，因此 p 回到优先级为 1 的就绪队列中。R3 资源剩余量为  $3-1=2$ ，此时优先级为 1 的就绪队列只有进程 p。但由于进程 p 的优先级与当前正在执行的进程 x 优先级相同，不能抢占，因此系统依然执行进程 x，输出当前执行进程为 x。

输入命令“to”后，进程 x 时间片完，回到就绪队列。系统调度执行进程 p，

输出当前执行进程为 p，此时优先级为 1 的就绪队列中只有进程 x。

输入命令“to”后，进程 p 时间片完，回到就绪队列。系统调度执行进程 x，输出当前执行进程为 x，此时优先级为 1 的就绪队列中只有进程 p。

综上所述，输出结果为：init x x x x p p q q r r x p q r x x x p x，与实验指导书中给出的预期输出结果一致。

## 九、实验结论：

通过本次实验，进一步巩固了课堂所学的进程管理和资源管理的相关知识，并在熟练掌握课堂所学内容的基础上，使用 Java 语言简单模拟了计算机操作系统对进程和资源的管理和调度，成功实现了时间片轮转（RR）调度算法，并且系统能顺利通过测试用例，取得与预期一致的结果，说明实现取得成功。

## 十、总结及心得体会：

通过本次实验，加深了对课堂所学的进程管理和资源管理相关知识的理解，对进一步理解计算机系统与底层实现原理是一次非常好的指引。并且在设计实现一个完整系统的过程中，采用了面向对象的编程思想以及模块化编程的实现方式，是一次获益匪浅的尝试，进一步提高了动手能力。

## 十一、对本实验过程及方法、手段的改进建议：

可以在原系统的基础上，进一步实现更多种类的进程调度算法，如 SJF、SRT、HRRN、多级反馈队列等等，有利于进一步理解、比较这些调度算法。

报告评分：

指导教师签字：



# 电子科技大学

## 实验报告

学生姓名：关文聪 学号：2016060601008 指导教师：薛瑞尼

实验地点：主楼 A2-412

实验时间：2019 年 5 月

一、实验室名称：计算机实验室

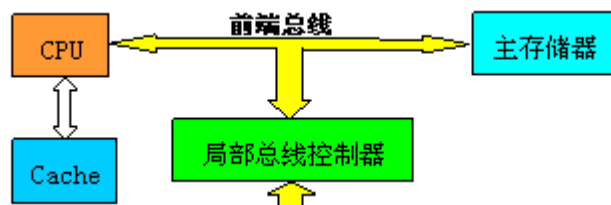
二、实验项目名称：虚拟内存综合实验

三、实验学时：4

四、实验原理：

物理地址：

把内存比作一个大的数组（为了分析方便），每个数组都有其下标，这个下标标识了内存中的地址，这个实实在在的在内存中的地址，我们称之为物理地址。但是在用于内存芯片级的单元寻址，与处理器和 CPU 连接的地址总线相对应。



逻辑地址：

与物理地址比较相对的是逻辑地址，这个地址就是在程序中我们把它放到的位置；而这个位置通常是由编译器给出的。另外的一种理解是：逻辑地址指的是机器语言指令中，用来指定一个操作数或者是一条指令的地址。Intel 段式管理中：，“一个逻辑地址，是由一个段标识符加上一个指定段内相对地址的偏移量，表示为 [段标识符：段内偏移量]。”

虚拟地址：

Virtual Address, 简称 VA, 由于 Windows 程序时运行在 386 保护模式下，

这样程序访问存储器所使用的逻辑地址称为虚拟地址。实际上因为我们现代程序中地址都是虚拟的，所以这里的虚拟地址和线性地址是等价了的。

线性地址：

线性地址（Linear Address）也叫虚拟地址(virtual address)是逻辑地址到物理地址变换之间的中间层。在分段部件中逻辑地址是段中的偏移地址，然后加上基地址就是线性地址。

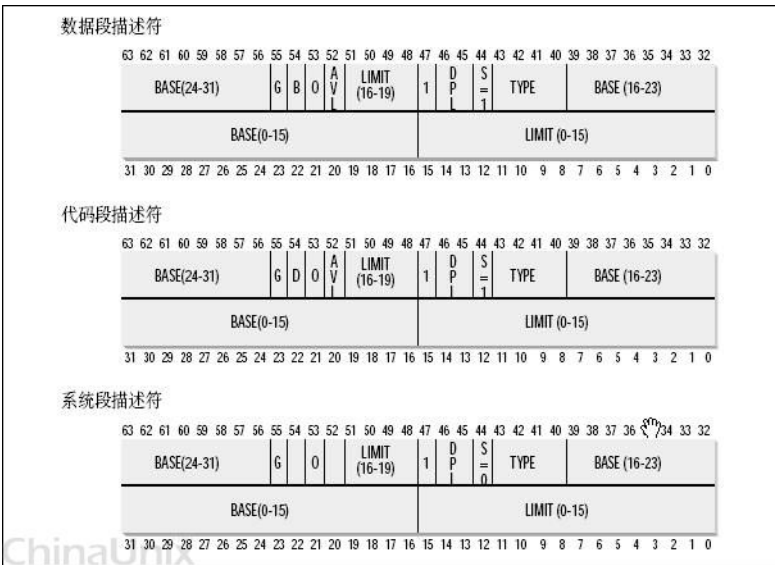
（二）CPU 段式内存管理：逻辑地址转换为线性地址；

一个逻辑地址由两部份组成，段标识符：段内偏移量。段标识符是由一个 16 位长的字段组成，称为段选择符。其中前 13 位是一个索引号。后面 3 位包含一些硬件细节：



最后两位涉及权限检查。

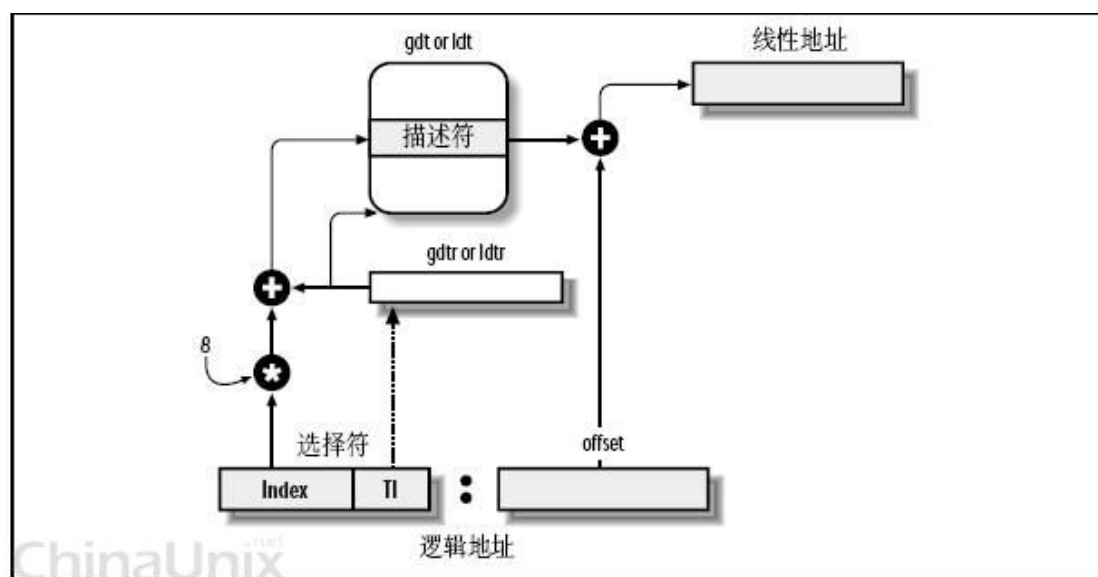
索引号，或者直接理解成数组下标——指向“段描述符(segment descriptor)”，段描述符具体地址描述了一个段。这样，很多个段描述符，就组了一个数组，叫“段描述符表”，这样，可以通过段标识符的前 13 位，直接在段描述符表中找到一个具体的段描述符，这个描述符就描述了一个段，每一个段描述符由 8 个字节组成，如下图：



Base 字段，它描述了一个段的开始位置的线性地址。

Intel 设计是，一些全局的段描述符，就放在“全局段描述符表(GDT)”中，一些局部的，例如每个进程自己的，就放在所谓的“局部段描述符表(LDT)”中。段选择符中的 T1 字段=0，表示用 GDT，=1 表示用 LDT。

GDT 在内存中的地址和大小存放在 CPU 的 gdtr 控制寄存器中，而 LDT 则在 ldtr 寄存器中。具体如下图：

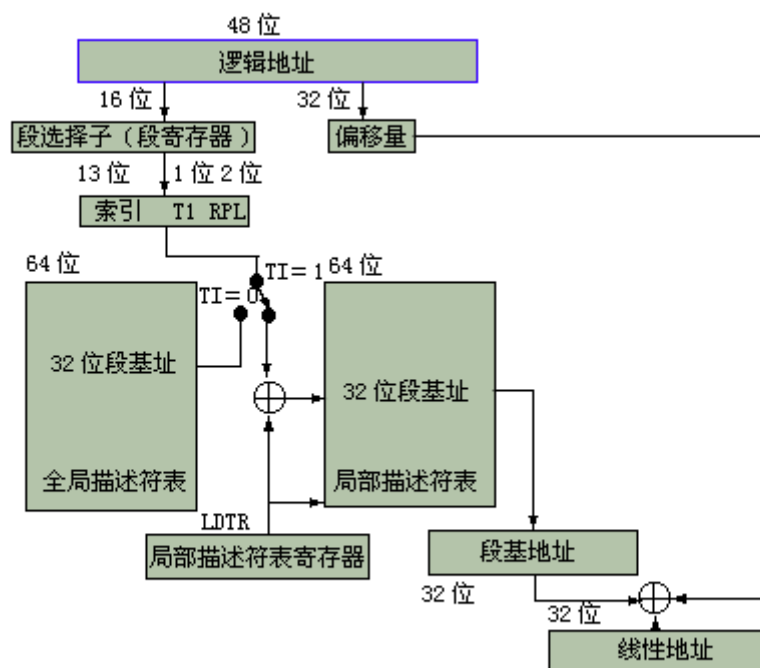


首先，给定一个完整的逻辑地址[段选择符：段内偏移地址]

1、看段选择符的 T1=0 还是 1，知道当前要转换是 GDT 中的段，还是 LDT 中的段，再根据相应寄存器，得到其地址和大小。我们就有了一个数组了。

2、拿出段选择符中前 13 位，可以在这个数组中，查找到对应的段描述符，这样，它了 Base，即基地址就知道了。

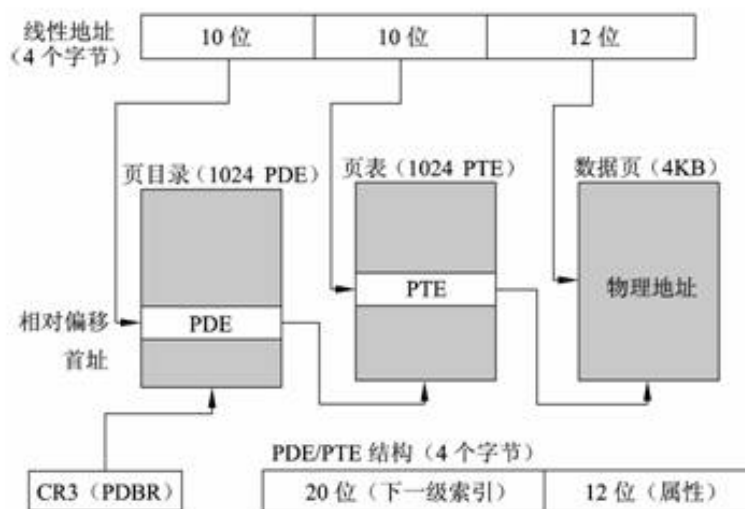
3、把 Base + offset，就是要转换的线性地址了。对于软件来讲，原则上就需要把硬件转换所需的信息准备好，就可以让硬件来完成这个转换了。



CPU 的页式内存管理：

CPU 的页式内存管理单元，负责把一个线性地址，最终翻译为一个物理地址。从管理和效率的角度出发，线性地址被分为以固定长度为单位的组，称为页，例一个 32 位的机器，线性地址最大可为 4G，可以用 4KB 为一个页来划分，这页，整个线性地址就被划分为一个  $total\_page[2^{20}]$  的大数组，共有 2 的 20 个次方个页。这个大数组我们称之为页目录。目录中的每一个目录项，就是一个地址——对应的页的地址。

另一类“页”，我们称之为物理页，或者是页框、页帧的。是分页单元把所有的物理内存也划分为固定长度的管理单位，它的长度一般与内存页是一一对应的。这里注意到，这个  $total\_page$  数组有  $2^{20}$  个成员，每个成员是一个地址（32 位机，一个地址也就是 4 字节），那么要单单要表示这么一个数组，就要占去 4MB 的内存空间。为了节省空间，引入了一个二级管理模式的机器来组织分页单元。如图：



描述：

1、分页单元中，页目录是唯一的，它的地址放在 CPU 的 cr3 寄存器中，是进行地址转换的开始点。

2、每一个活动的进程，因为都有其独立的对应的虚拟内存（页目录也是唯一的），那么它也就对应了一个独立的页目录地址。——运行一个进程，需要将它的页目录地址放到 cr3 寄存器中，将别的保存下来。

3、每一个 32 位的线性地址被划分为三部份，页目录索引 (10 位)：页表索引 (10 位)：偏移 (12 位)

转换步骤：

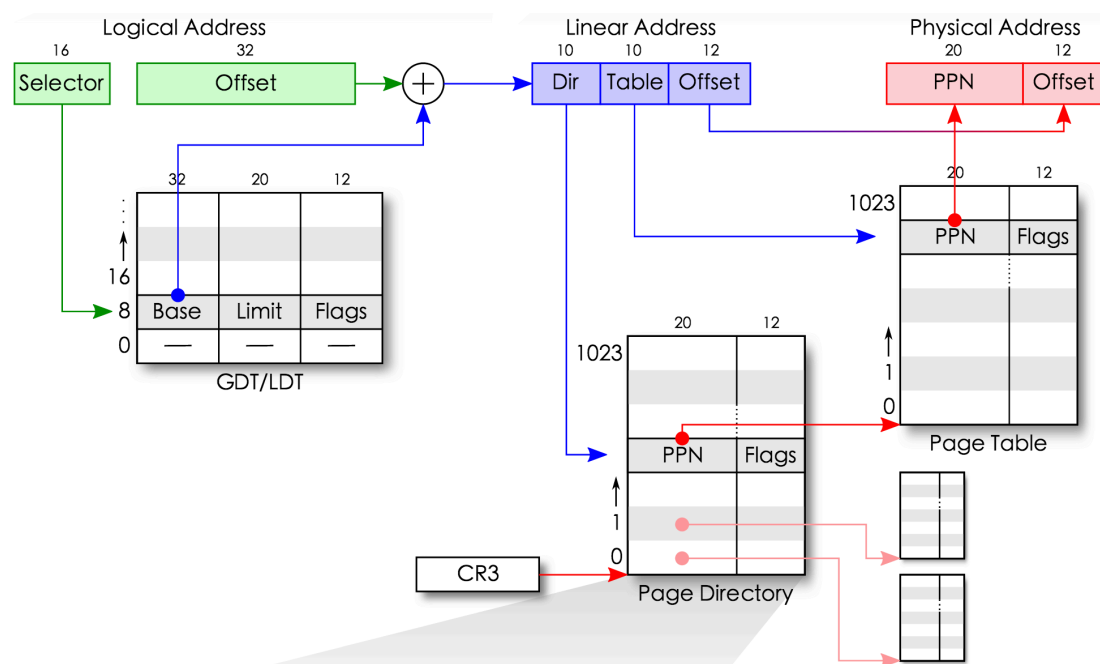
1、从 cr3 中取出进程的页目录地址（操作系统负责在调度进程的时候，把这个地址装入对应寄存器）；

2、根据线性地址前十位，在数组中，找到对应的索引项，因为引入了二级管理模式，页目录中的项，不再是页的地址，而是一个页表的地址。（又引入了一个数组），页的地址被放到页表中去了。

3、根据线性地址的中间十位，在页表（也是数组）中找到页的起始地址；

4、将页的起始地址与线性地址中最后 12 位相加，得到最终我们想要的物理地址；

完整的地址转化过程：



## 五、实验目的：

通过实验，掌握段页式内存管理机制，理解地址转换的过程

- (1) 掌握计算机的寻址过程
- (2) 掌握页式地址地址转换过程
- (3) 掌握计算机各种寄存器的用法

## 六、实验内容：

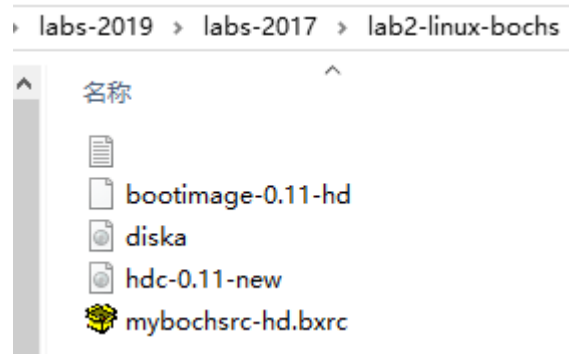
通过手工查看系统内存，并修改特定物理内存的值，实现控制程序运行的目的。

## 七、实验器材（设备、元器件）：

个人计算机、Linux 内核（0.11）+ Bochs 虚拟机

## 八、实验步骤：

1. 进入 bochs 官方网站 <http://bochs.sourceforge.net/>，下载 bochs 并安装（本次实验所使用的 bochs 版本为 bochs-2.6.9）
2. 安装完毕后，复制实验所需文件（如下图所示）至 bochs 安装根目录下。



3. 打开 mybochsrc-hd.bxrc 配置文件，修改配置。修改 romimage 路径与 vagromimage 路径（与本机安装 bochs 的路径一致）

```
6  romimage: file="C:\Users\Eternity-Myth\Desktop\Bochs-2.6.9\BIOS-bochs-latest"
7  vagromimage: file="C:\Users\Eternity-Myth\Desktop\Bochs-2.6.9\VGABIOS-lgpl-latest"
```

随后，直接删除如下图所示的配置：

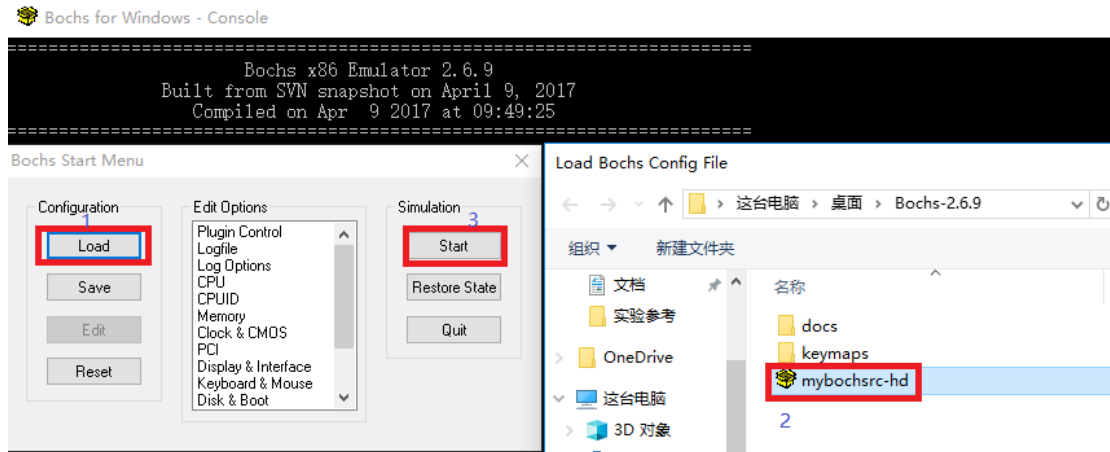
```
29  cpuid: family=6, model=0x03, stepping=3, mmx=1, apic=xapic, sse=sse2, sse4a=0, sep=1, aes=0, xsave=0, xsaveopt=0, movbe=0,
    smep=0, x86_64=1, lg_pages=0, pcid=0, fsgsbase=0, mwait=1, mwait_is_nop=0
30  cpuid: vendor_string="GenuineIntel"
31  cpuid: brand_string="Intel(R) Pentium(R) 4 CPU"
```

```
39  pn timer: enabled=0
```

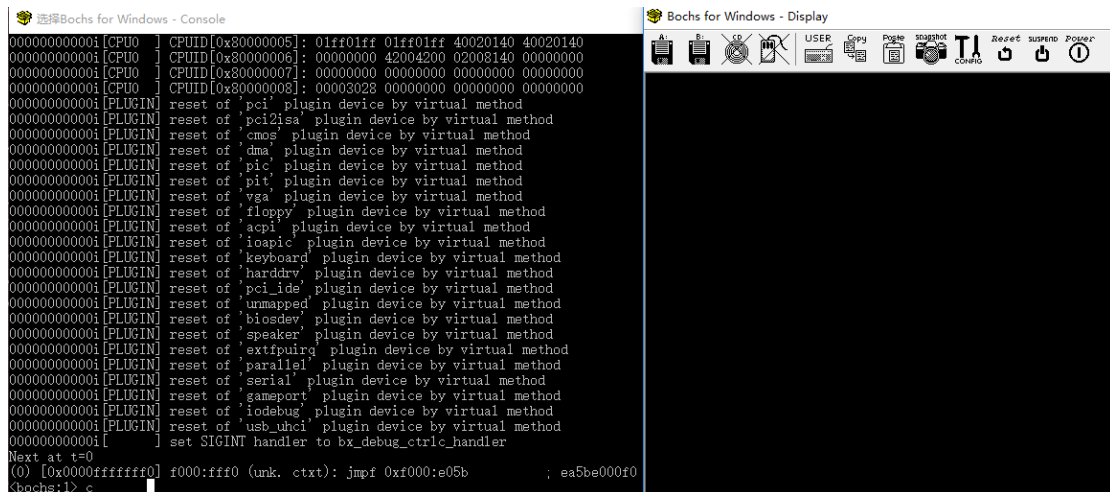
```
49  keyboard_type: mf
50  keyboard_serial_delay: 250
51  keyboard_paste_delay: 100000
52  keyboard_mapping: enabled=0, map=
53  user_shortcut: keys=none
```

完成后，保存配置文件并退出。

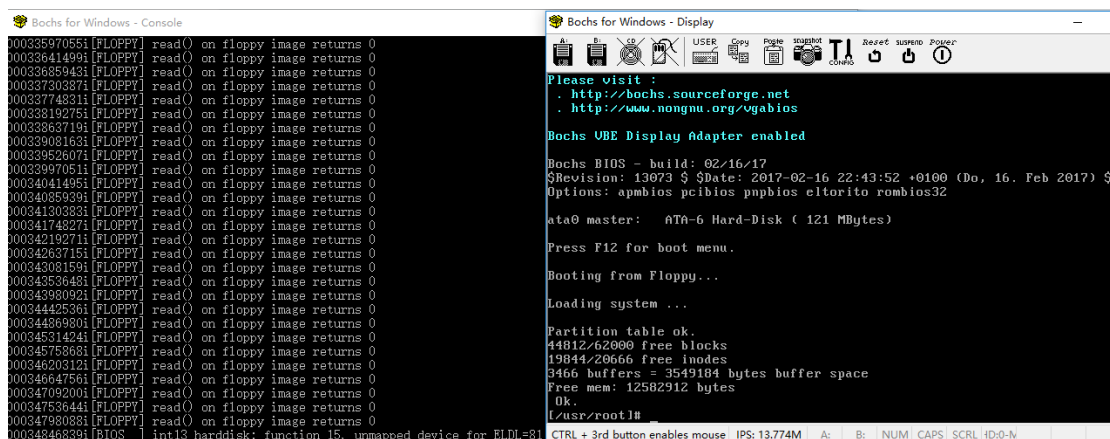
4. 打开安装目录下的 bochsdbg.exe 文件，选择“Load”加载配置文件，选择刚刚修改的 mybochsrc-hd 文件加载。无错误出现，随后点击“Start”启动 bochs 虚拟机。



5. bochs 虚拟机成功启动后，会出现两个窗口。一个窗口为 bochs 命令控制窗口 Console，另一个为启动的 Linux 操作系统窗口 Display。如下图所示。在 Console 窗口输入“c”，继续加载 Linux 操作系统。



6. 加载成功后，在 Display 中成功出现了 Linux 操作系统界面：



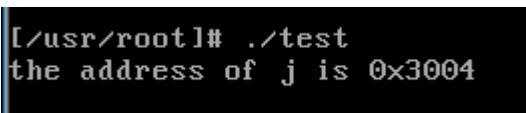


7. 在 Display 命令行输入“vi test.c”编辑输入实验程序，保存退出。



```
#include<stdio.h>
int j=0x123456;
int main(){
    printf("the address of j is 0x%x\n",&j);
    while(j);
    printf("program terminated normally!\n");
    return 0;
}
```

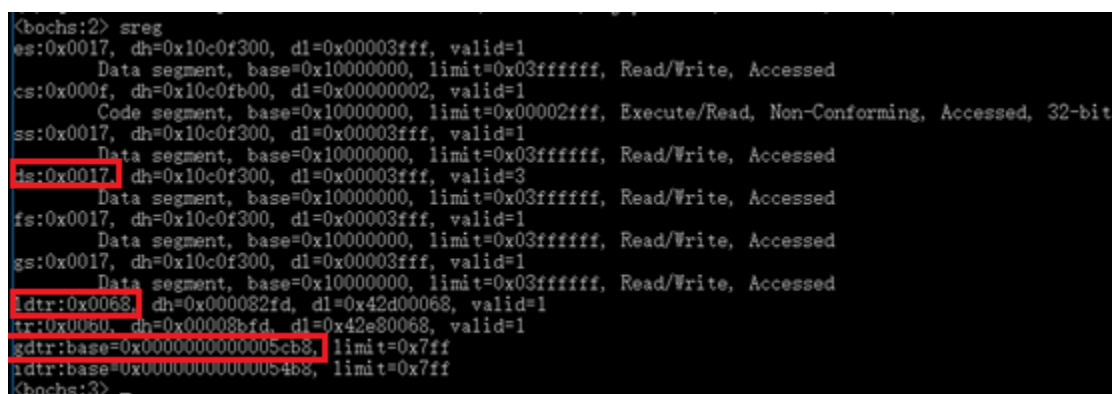
8. 使用命令“gcc -o test test.c”编译 C 程序，编译成功后，使用命令“./test”运行程序，结果如图所示：



```
[/usr/root]# ./test
the address of j is 0x3004
```

可以看出，程序运行至 while(j) 语句时进入死循环，未运行第二条输出语句。

9. 回到 Console 窗口，按下 Ctrl+C 中断当前运行。输入命令“sreg”查看段的具体信息，如图所示：



```
<bochs:2> sreg
es:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
  Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
cs:0x000f, dh=0x10c0fb00, dl=0x00000002, valid=1
  Code segment, base=0x10000000, limit=0x00002fff, Execute/Read, Non-Conforming, Accessed, 32-bit
ss:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
  Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
ds:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=3
  Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
fs:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
  Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
gs:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
  Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
ldtr:0x0068, dh=0x000082fd, dl=0x42d00068, valid=1
tr:0x0060, dh=0x00008bfd, dl=0x42e80068, valid=1
gdtr:base=0x00000000000005cb8, limit=0x7ff
ldtr:base=0x000000000000054b3, limit=0x7ff
<bochs:3>
```

10. 计算：读 ds 段信息，根据 ds 段为 0x0017=0000 0000 0001 0111。其中，高 13 位代表索引号（用红色标注出），可以读出索引号为 02。索引号后一位（即低位起倒数第三位，用高亮标识的位置）为 TI 位，可以读出 TI=1，所以段描述符存放在“局部段描述符表(LDT)”中，且应在 LDT 表的第 3 项。（起始号为 0）

11. 计算：读 LDTR 寄存器信息，其存放了 LDTX 描述符在 GDT 中的位置。LDTR 为 0x0068=0000 0000 0110 1000。其中，高 13 位代表索引号（用红色标注出），可以读出索引号为 13，则表示 LDT 起始地址存放在 GDT 表的第 14 项。

(起始号为 0)

12. 计算：读 GDTR 寄存器信息，GDTR 为 0x5CB8，即 GDT 在内存中的起始地址为 0x5CB8。注意到每个段描述符由 8 个字节组成，可以计算 LDT 的首地址为：0x5CB8（起始地址）+8\*13（偏移）=0x5D20。

13. 输入命令 “xp /2w 0x5D20” 查看 GDT 中对应的表项如图所示：

```
<bochs:33> xp /2w 0x5D20
[bochs]:
0x00000000000005d20 <bogus+ 0>: 0xa2d00068 0x000082fa
```

接下来进行计算与地址拼接转化：注意到 0xa2d00068 为低位（0-15 位），0x000082fa 为高位（16-31 位），即 LDT 的段描述符用二进制位表示应为：

0000 0000 0000 0000 1000 0010 1111 1010 1010 0010 1101 0000 0000

0000 0110 1000，根据段描述符结构，第 16-31 位对应基址的第 0-15 位（用红色标注出），高 32 位的第 0-7 位对应基址的第 16-23 位（用蓝色标注出），高 32 位的第 24-31 位对应基址的第 24-31 位（用绿色标注出）。故按此原理拼接地址，可以得到 LDT 的基址为：0000 0000 1111 1010 1010 0010 1101 0000

(0x00FAA2D0)。可以作如下验证：由之前的结果可得：因为段描述符在 LDT 表中的偏移为 2，输入命令 “xp /2w 0x00FAA2D0+2\*8”，得到的结果如图所示：

```
<bochs:16> sreg
es:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
cs:0x000f, dh=0x10c0fb00, dl=0x00000002, valid=1
Code segment, base=0x10000000, limit=0x00002fff, Execute/Read, Non-Conforming, Accessed, 32-bit
ss:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
ds:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=3
Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
fs:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
gs:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
ldtr:0x0068, dh=0x000082fa, dl=0xa2d00068, valid=1
tr:0x0060, dh=0x00008bfa, dl=0xa2e80068, valid=1
gdtr:base=0x00000000000005cb8, limit=0x7ff
idtr:base=0x000000000000054b8, limit=0x7ff
<bochs:17> xp /2w 0x00FAA2D0+2*8
[bochs]:
0x000000000000faa2e0 <bogus+ 0>: 0x00003fff 0x10c0f300
```

查看 ds 段的段描述符信息，与 sreg 显示的 ds 段的 dl、dh 寄存器的值相同。

14. 计算线性地址：由之前的结果可得：ds 段的基址为 0x10000000，程序运行显示 j 的段内偏移地址为 0x3004，所以线性地址为：

0x10000000+0x3004=0x10003004=0001 0000 0000 0000 0011 0000 0000 0100

线性地址被划分为三部份，前 10 位为页目录索引(用红色标注出)，接下来的 10 位为页表索引(用蓝色标注出)，最低 12 位为偏移(用绿色标注出)。可得页

目录索引为 64，页表索引为 3，偏移量为 4。

15. 输入命令“creg”查看寄存器 CR3 的值，如图所示：

```
<bochs:18> creg
CR0=0x8000001b: PG cd nw ac wp ne ET TS em MP PE
CR2=page_fault_laddr=0x0000000010002fa8
CR3=0x0000000000000000
PCU=page-level cache disable=0
PWT=page-level write-through=0
CR4=0x00000000: pke smap smep osxsave pcid fsgsbase smx vmx osxmmexcpt umip osfxsr pce pge mce pae pse de tsd pvi vme
CR8: 0x0
EFER=0x00000000: ffxsr nxe lma lme sce
```

寄存器 CR3 的值为 0，即页目录表的起始地址为 0。因此，对应页目录（PDE）地址为  $0+64*4=256=0x100$ 。输入命令“xp /w 0x100”查看页目录（PDE）的值：

```
<bochs:22> xp /w 0x100
[bochs]:
0x0000000000000100 <bogus+ 0>: 0x00fa5027
```

PDE 的值为  $0x00fa5027=0000\ 0000\ 1111\ 1010\ 0101\ 0000\ 0010\ 0111$ 。只取其前 20 位（用红色标注出）作为下一级的索引，即下一级的索引为  $0x00fa5000$ 。同理，对应页表（PTE）地址为  $0x00fa5000+3*4=0x00fa500c$ 。输入命令“xp /w 0x00fa500c”查看页表（PTE）的值：

```
<bochs:24> xp /w 0x00fa500c
[bochs]:
0x000000000000fa500c <bogus+ 0>: 0x00fa3067
```

PTE 的值为  $0x00fa3067=0000\ 0000\ 1111\ 1010\ 0011\ 0000\ 0110\ 0111$ 。同理只取其前 20 位（用红色标注出）作为下一级的索引，即下一级的索引为  $0x00fa3000$ 。因此，得到物理地址为  $0x00fa3000+4=0x00fa3004$ 。

16. 输入命令“xp /w 0x00fa3004”查看该地址的值：

```
<bochs:25> xp /w 0x00fa3004
[bochs]:
0x000000000000fa3004 <bogus+ 0>: 0x00123456
```

显示的结果正确，即已经找到了 j 所在的正确的物理地址  $0x00fa3004$ 。输入命令“setpmem 0x00fa3004 4 0”将物理地址  $0x00fa3004$  的开始 4 个字节的值设置为 0，随后输入命令“c”继续运行 Linux 系统：

```
0x000000000000fa3004 <bogus+ 0>: 0x00123456
<bochs:26> setpmem 0x00fa3004 4 0
<bochs:27> c
the address of j is 0x3004
program terminated normally
!t/usr/root!#
```

注意到程序成功执行了第二条输出语句并返回退出，该结果证明了以上实验步骤是正确的。至此，已顺利完成实验，取得成功。

## **九、 实验结论：**

通过本次实验，详细学习了计算机的段页式内存管理机制，掌握了地址转换的过程，并且在实际操作中，能成功寻找到变量存储的具体位置，并对变量的数值成功进行修改，程序能正常运行退出，实验取得成功。

## **十、 总结及心得体会：**

通过本次实验，接触到了 Bochs 虚拟机与 Linux 内核，开拓了眼界。实验过程中详细学习了计算机的段页式内存管理机制，掌握了地址转换的过程，巩固加深了对课程知识的理解。通过实践操作，加强了动手能力，获益良多。

## **十一、 对本实验过程及方法、手段的改进建议：**

暂无

报告评分：

指导教师签字：