

# 电子科技大学

# 实验报告

## 实验一

### 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。

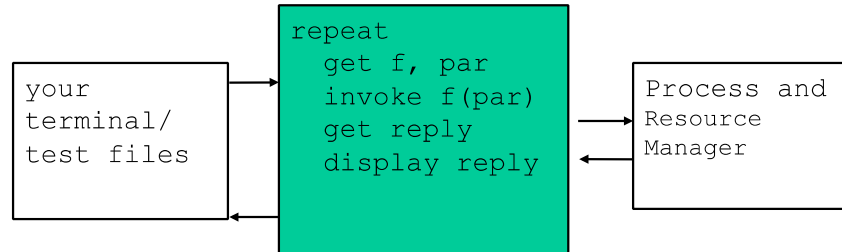


图 1 系统总体结构

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)

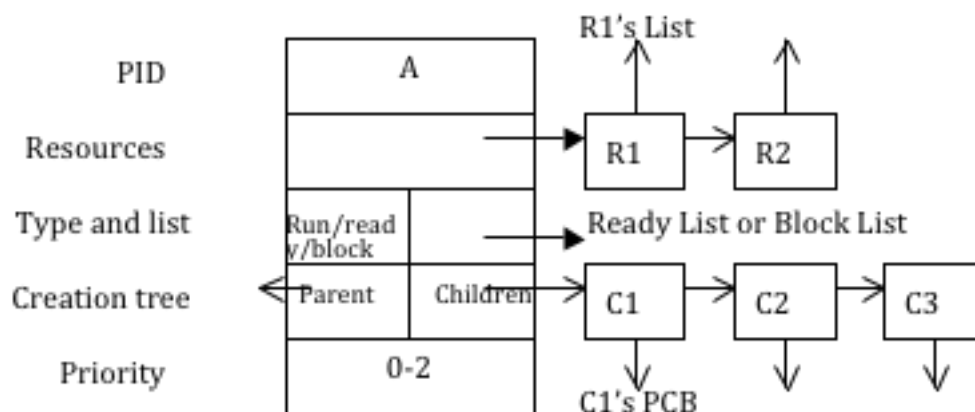


图 2 PCB 结构示意图

就绪进程队列 Ready list (RL)如图 3 所示。

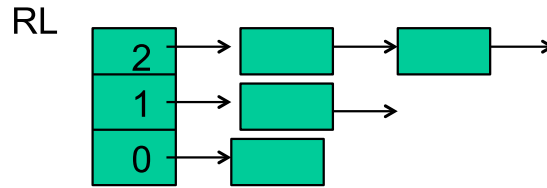


图 3 Ready List 数据结构示意图

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。

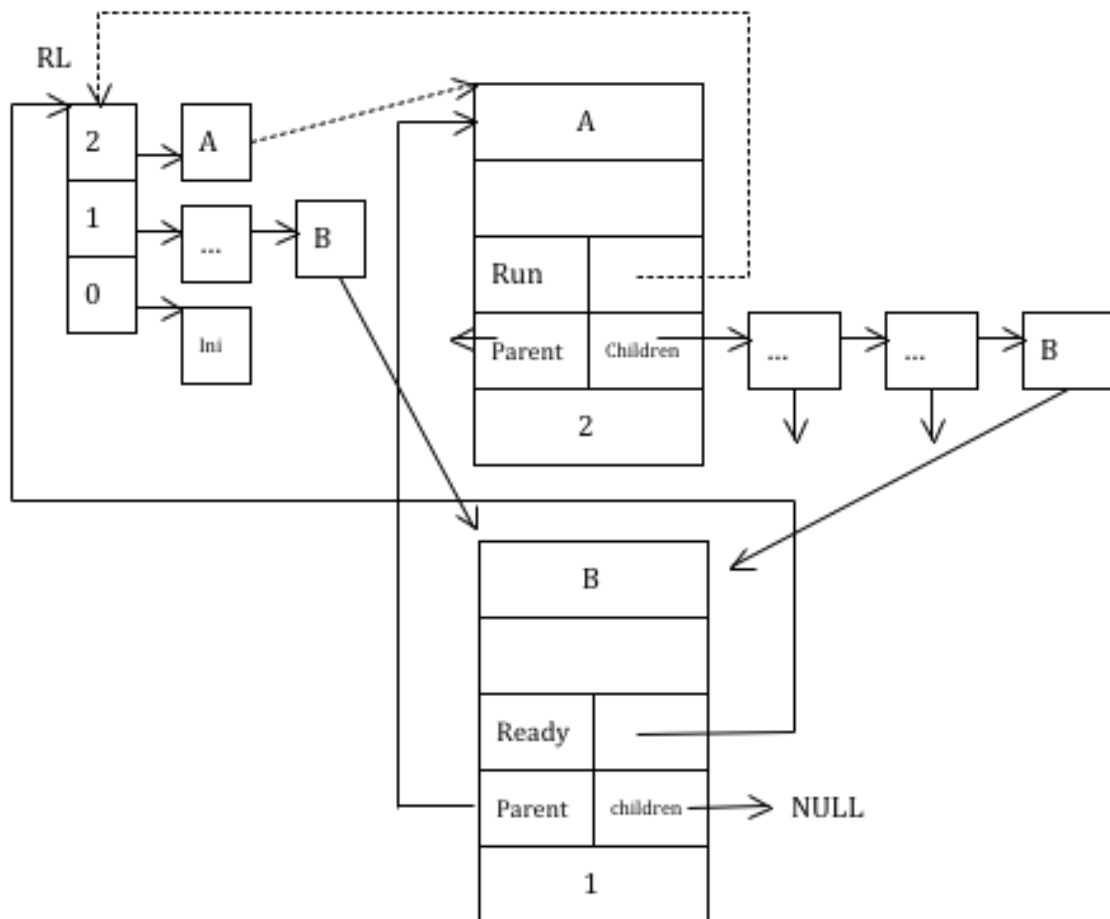


图 4 进程数据结构间关系

(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  $R_i$  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

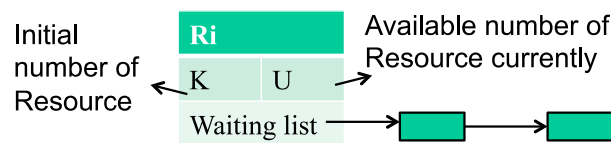


图 5 资源数据结构 RCB 示意图

### 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;
3.
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);
16.
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.     }
40.
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 par
ameters!");
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 le
gal parameters!");
65.                 continue;
66.             }
67.         } catch (Exception e) {System.out.println("Error!Plea
se 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 para
meters!");
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 permiss
           ion 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 par
           ameters!");
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!Ple
           ase 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 l
           egal parameters!");
129.            }
130.        }

```

```

131.             break;
132.         case "rel":
133.             if (cmds.length!=3)
134.             {
135.                 System.out.println("Error!Please enter the legal par
ameters!");
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!Ple
ase 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 l
egal 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 param
            eter 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. }

```

```

213.
214.     private static void loadFile(String filePath) throws IOException
215.     {
216.         InputStream inputStream=new FileInputStream(filePath);
217.         LineNumberReader lineNumberReader=new LineNumberReader(new FileRead
            r(filePath));
218.         String cmd=null;
219.         while((cmd=lineNumberReader.readLine())!=null)
220.         {
221.             if(!"".equals(cmd))
222.             {
223.                 testShell(cmd);
224.             }
225.         }
226.     }
227.
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;
7.
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;
28.
29.      private Pcb()
30.      {
31.          existProcess=new HashMap<>();
32.          pidGenerator=new AtomicInteger();
33.      }
34.
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.      }
51.
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 || currentPr
        ocess.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. }
94.
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. }

```

```

107.
108.     public static void timeout()
109.     {
110.         pcb.getCurrentProcess().setstate(Process.State.READY);
111.         scheduler();
112.     }
113.
114.     public void deleteexistProcess(Process process)
115.     {
116.         String name=process.getpName();
117.         existProcess.remove(name);
118.     }
119.
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.     }
134.
135.     public void printProcessDetail(Process process)
136.     {
137.         System.out.print(process.getpName()+" (PID:"+process.getpid()+", 进程状
138.         态: "+process.getstate()+", 优先级: "+process.getpriority() + ",");
139.         if(process.getresourceMap().isEmpty())
140.         {
141.             System.out.println(" (无资源占用) ");
142.         }
143.         else
144.         {
145.             StringBuilder stringBuilder=new StringBuilder();
146.             stringBuilder.append("(");
147.             for(Map.Entry<Resource,Integer> entry:process.getresourceMap().e
148.             ntrySet())
149.             {
150.                 Resource res=entry.getKey();

```



```

149.         int holdNum=entry.getValue();
150.         stringBuilder.append(",")
151.             .append("R")
152.             .append(res.gettrid())
153.             .append(":")
154.             .append(holdNum);
155.     }
156.     stringBuilder.append(" ");
157.     String result=stringBuilder.toString();
158.     System.out.println(result.replaceFirst(",",""));
159. }
160. }
161.
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. }
180.
181. public int createpid()
182. {
183.     return pidGenerator.getAndIncrement();
184. }
185.
186. public void addexistProcess(Process process)
187. {
188.     existProcess.put(process.getpName(),process);
189. }
190.
191. public static boolean isExistName(String name)
192. {

```

```
193.         return existProcess.containsKey(name);
194.     }
195.
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.     }
207.
208.     public void deleteProcess(Process process)
209.     {
210.     }
211.
212.     public void setCurrentProcess(Process currentProcess)
213.     {
214.         this.currentProcess=currentProcess;
215.     }
216.
217.     public void setPidGenerator(AtomicInteger pidGenerator)
218.     {
219.         this.pidGenerator=pidGenerator;
220.     }
221.
222.     public static Pcb getpcb()
223.     {
224.         return pcb;
225.     }
226.
227.     public Queue getreadyQueue()
228.     {
229.         return readyQueue;
230.     }
231.
232.     public Map<String,Process> getexistProcess()
233.     {
234.         return existProcess;
235.     }
236.
```

```
237.     public Process getcurrentProcess()
238.     {
239.         return currentProcess;
240.     }
241.
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;
4.
5. /**
6.  * @BelongsProject:OS-process
7.  * @BelongsPackage:PACKAGE_NAME
8.  * @Author:Uestc_Xiye
9.  * @CreateTime:2020-12-01 16:52:02
10. */
11.
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;
32.
33.     private static final Pcb pcb=Pcb.getpcb();
34.     private static final Queue readyQueue=Queue.getreadyQueue();
35.
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.     }
48.
49.     public Process(int pid,String pName,int priority,State state,ConcurrentHa
        shMap<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.     }
59.
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.     }
71.

```

```

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
83.
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.     }
121.
122.     public void destroy()
123.     {
124.         deleteProcessTree();
125.         Pcb.scheduler();
126.         return;
127.     }
128.
129.     public void setpid(int pid)
130.     {
131.         this.pid=pid;
132.     }
133.
134.     public void setpName(String pName)
135.     {
136.         this.pName=pName;
137.     }
138.
139.     public void setpriority(int priority)
140.     {
141.         this.priority=priority;
142.     }
143.
144.     public void setstate(State state)
145.     {
146.         this.state=state;
147.     }
148.
149.     public void setblockedResource(Resource blockedResource)
150.     {
151.         this.blockedResource=blockedResource;
152.     }
153.
154.     public void setrMap(ConcurrentHashMap<Resource,Integer> resourceMap)
155.     {
156.         this.resourceMap=resourceMap;
157.     }
158.
159.     public void setparent(Process parent)

```

```
160.     {
161.         this.parent=parent;
162.     }
163.
164.     public void setchildren(List<Process> children)
165.     {
166.         this.children=children;
167.     }
168.
169.     public int getpid()
170.     {
171.         return pid;
172.     }
173.
174.     public String getpName()
175.     {
176.         return pName;
177.     }
178.
179.     public int getpriority()
180.     {
181.         return priority;
182.     }
183.
184.     public State getstate()
185.     {
186.         return state;
187.     }
188.
189.     public Resource getblockedResource()
190.     {
191.         return blockedResource;
192.     }
193.
194.     public ConcurrentHashMap<Resource,Integer> getresourceMap()
195.     {
196.         return resourceMap;
197.     }
198.
199.     public Process getparent()
200.     {
201.         return parent;
202.     }
203.
```

```

204.     public List<Process> getchildren()
205.     {
206.         return children;
207.     }
208.
209. }

```

## Resource.java

```

1. import java.util.Deque;
2. import java.util.LinkedList;
3. import java.util.Map;
4.
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();
25.
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;
35.
36.     public BlockedProcess(Process process,int need)
37.     {
38.         this.process=process;
39.         this.need=need;
40.     }
41.
42.     public void setprocess(Process process)
43.     {
44.         this.process=process;
45.     }
46.
47.     public void setneed(int need)
48.     {
49.         this.need=need;
50.     }
51.
52.     public Process getprocess()
53.     {
54.         return process;
55.     }
56.
57.     public int getneed()
58.     {
59.         return need;
60.     }
61. }
62.
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. }
70.
71. public void addremainingResource(int num)
72. {
73.     this.remainingResource+=num;
74. }
75.
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.     }
86.
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. }
125.
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. }
160.
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().getpri
                    ority())
183.                 {
184.                     pcb.preempt(readyProcess, pcb.getcurrentProcess());
185.                 }
186.             }
187.             else
188.             {
189.                 break;
190.             }
191.         }
192.     }
193.
194.     public void printStatus()
195.     {
196.         StringBuilder stringBuilder=new StringBuilder();
197.         stringBuilder.append("resource-").append(rid)
198.             .append("{maxResource=").append(maxResource)
199.             .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. }
214.
215.     public void setrid(int rid)
216.     {
217.         this.rid=rid;
218.     }
219.
220.     public void setmaxResource(int maxResource)
221.     {
222.         this.maxResource=maxResource;
223.     }
224.
225.     public void setremainingResource(int remainingResource)
226.     {
227.         this.remainingResource=remainingResource;
228.     }
229.
230.     public void setblockedDeque (Deque<BlockedProcess> blockedDeque)
231.     {
232.         this.blockedDeque=blockedDeque;
233.     }
234.
235.     public int getrid()
236.     {
237.         return rid;
238.     }
239.
240.     public int getmaxResource()
241.     {
242.         return maxResource;
243.     }
244.
245.     public int getremainingResource()
246.     {
247.         return remainingResource;

```

```
248.     }
249.
250.     public Deque<BlockedProcess> getblockedDeque()
251.     {
252.         return blockedDeque;
253.     }
254.
255. }
```

## Queue.java

```
1. import java.util.Deque;
2. import java.util.LinkedList;
3.
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();
17.
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.     }
27.
28.     public void addprocess(Process process)
29.     {
30.         int priority=process.getpriority();
```

```
31.         Deque<Process> d=deque[priority];
32.         d.addLast(process);
33.     }
34.
35.     public boolean deleteProcess(Process process)
36.     {
37.         int priority = process.getpriority();
38.         Deque<Process> d=deque[priority];
39.         return d.remove(process);
40.     }
41.
42.     public static Queue getreadyQueue()
43.     {
44.         return readyQueue;
45.     }
46.
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：

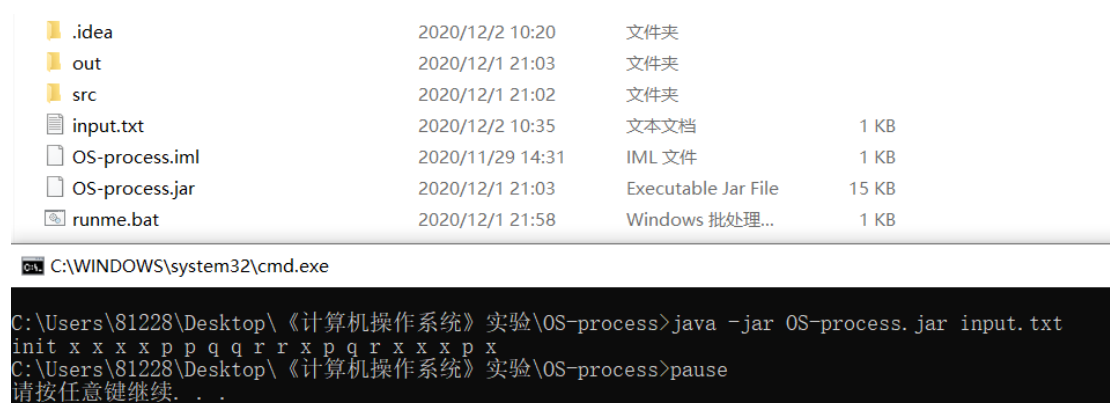


图 6 程序执行结果图

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。

# 电子科技大学

# 实验报告

## 实验二

### 二、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：

虚拟内存综合实验

### 三、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。

### 四、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。

### 五、实验原理

#### 5.1 物理地址，线性地址，逻辑地址，虚拟地址

##### 物理地址：

物理地址最好理解，我们可以简单的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，相信并不是一个所谓的数组，但是做出这样的比拟，有利于更好的理解。

还依稀记得这张图：



图 7 计算机组成局部图

### 逻辑地址：

与物理地址比较相对的是逻辑地址，一种理解是，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]。比如我们在程序中定义一个变量 `int g=3`；相应的汇编代码应该是 `mov [g],3`；那么这个 `g` 应该放在哪儿呢？实际上我们可以看到，这个 `g` 的地址总在在编译，链接之后就会一个确定的地址；而这个确定的地址我们叫做逻辑地址。

### 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。

## 5.2 CPU 段式内存管理：逻辑地址转换为线性地址

逻辑地址示意图如图 8 所示。一个逻辑地址由两部分组成，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。



图 8 逻辑地址示意图

最后两位涉及权限检查。

索引号，或者直接理解成数组下标——那它总要对应一个数组，它应该是指向一个东西的？而这个东西就是“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，如图 9 所示。

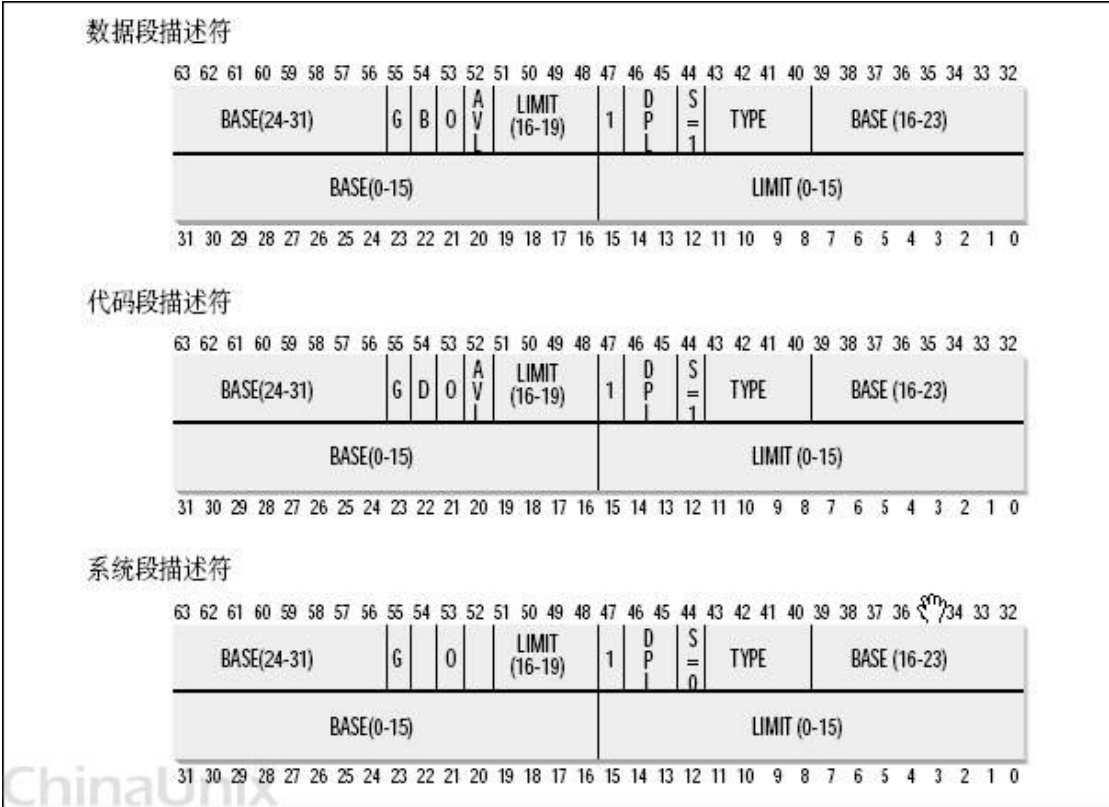


图 9 数据段描述符、代码段描述符、系统段描述符示意图

而在汇编里面我们用一个数据结构定义，如图 10 所示。

```
[SECTION .gdt]
; GDT
;
; 段基址, 段界限, 属性
LABEL_GDT: Descriptor 0, 0, 0 ; 空描述符
LABEL_DESC_CODE32: Descriptor 0, SegCode32Len - 1, DA_C + DA_32; 非一致代码段
LABEL_DESC_VIDEO: Descriptor 0B8000h, 0ffffh, DA_DRW ; 显存首地址
; GDT 结束
```

图 10 段描述符的数据结构

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。那究竟什么时候该用 GDT，什么时候该用 LDT 呢？这是由段选择符中的 T1 字段表示的，=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。具体如图 11 所示。

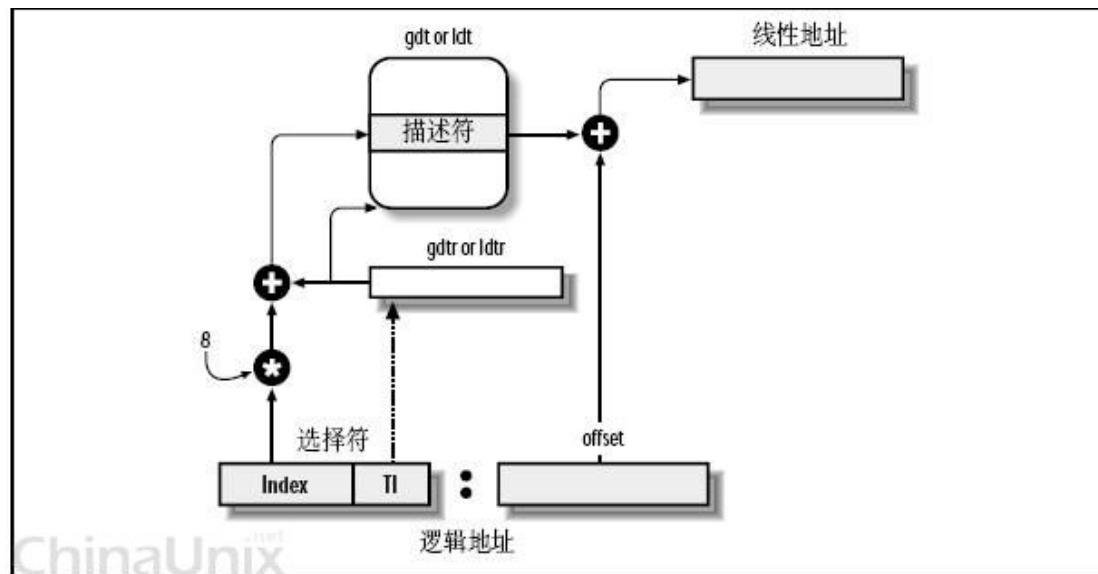


图 11 逻辑地址与线性地址的转换示意图

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。

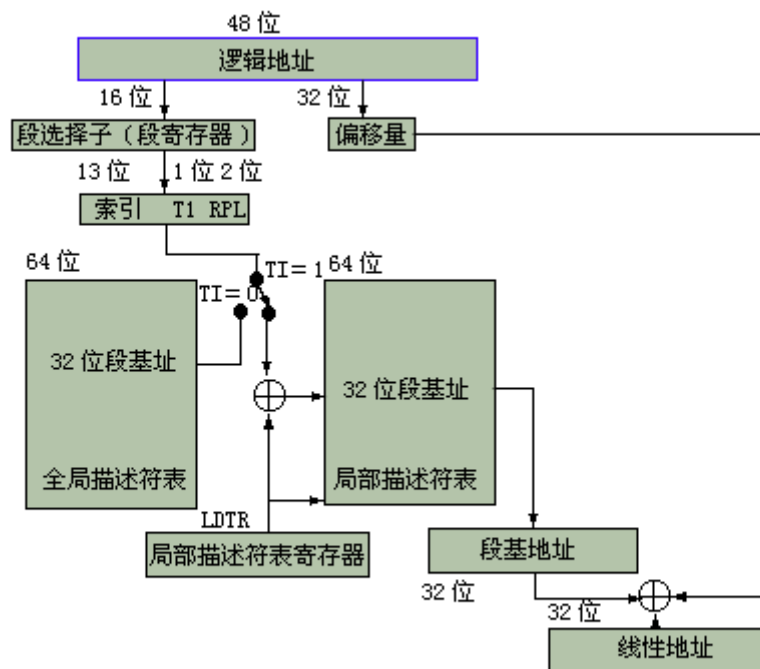


图 12 逻辑地址与线性地址的转换完整示意图

但是实际的情况并不是这么简单：linux 和 windows 的做法貌似是不同的。

### 5.3 CPU 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  $\text{total\_page}[2^{20}]$  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  $\text{total\_page}$  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。

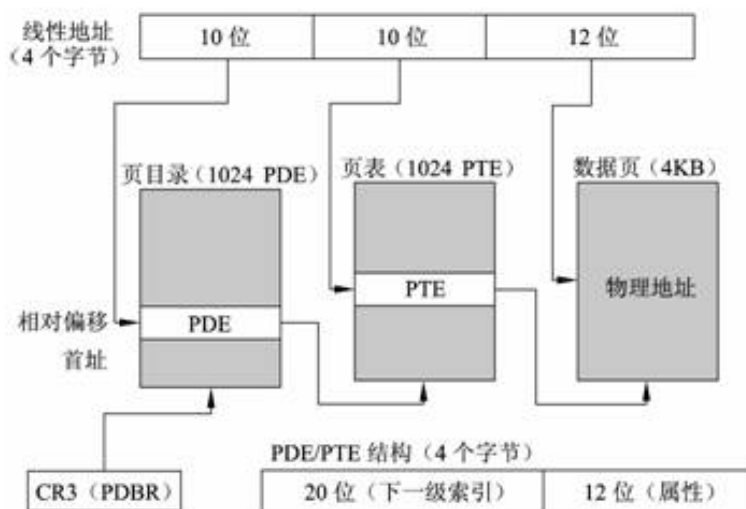


图 13 页式内存管理示意图

如图 13 所示。描述：

1. cr0 最高位确定是否采用分页机制，在分页单元中，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，面目录索引(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。

## 5.4 常用命令

- c, 启动 linux;
- Ctrl + c, 切换控制台;
- sreg, 查看段寄存器值;
- creg, 查看控制寄存器值;
- xp /2W 0xabcd, 显示地址 abcd 之后的 2 个字的内容;
- setpmem 0xabcd 4 0, 将地址 abcd 后 4 字节改为 0。

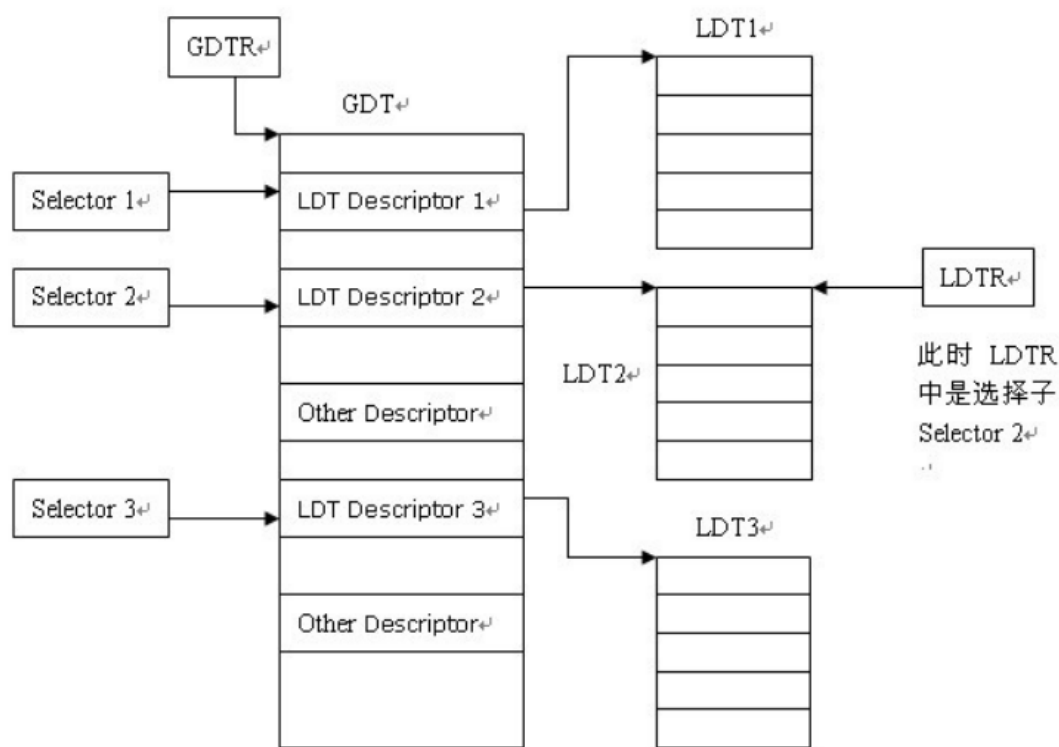


图 14 GDTR、GDT、LDTR、LDT 示意图

## 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

Linux 内核版本：0.11

Bochs 虚拟机版本：Bochs-2.5.1

## 七、实验步骤



1. 运行 Bochs-2.5.1.exe，安装 Bochs 虚拟机，默认安装路径为 C:\Program Files (x86)\Bochs-2.5.1；
2. 安装完毕后，复制如图 15 所示实验所需文件至 Bochs 根目录下；





名称	修改日期
 bootimage-0.11-hd	2020/11/21 14:43
 diska.img	2020/11/21 14:43
 hdc-0.11-new.img	2020/11/21 14:45
 mybochsrc-hd.bxrc	2020/11/21 14:43

图 15 实验所需文件

3. 如图 16 所示，运行 bochsdbg.exe，点击 Load 加载配置文件，选择 mybochsrc-hd.bxrc 文件，再点击 Start 启动 Bochs 虚拟机；

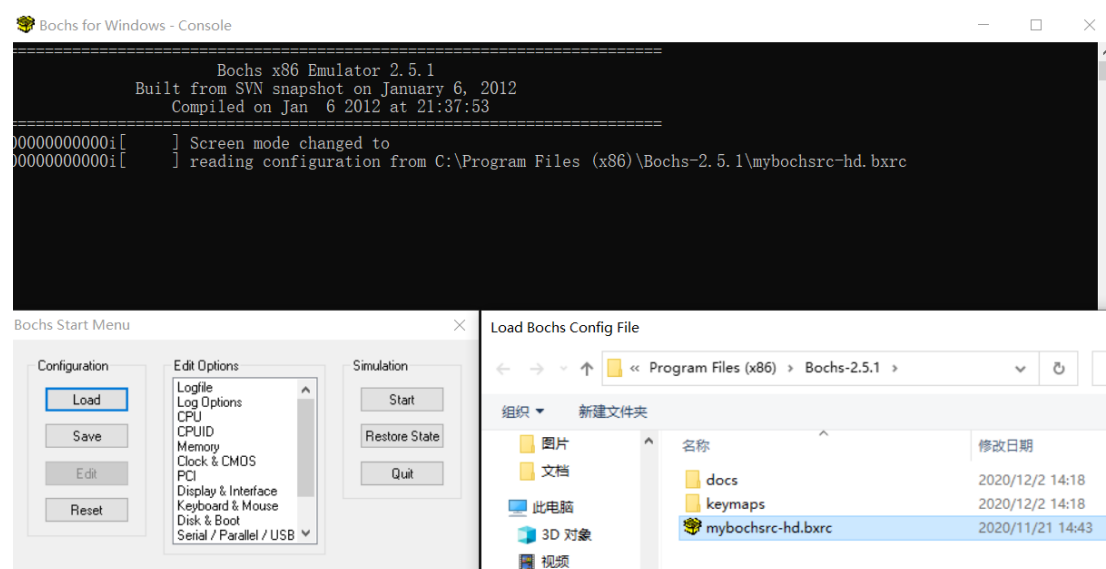


图 16 加载实验配置文件并启动 Bochs 虚拟机

4. Bochs 虚拟机启动后，会出现两个窗口：Console 窗口和 Display 窗口，如图 17 所示。Console 窗口为 Bochs 命令输入窗口，Display 窗口为 Linux 操作系统窗口；

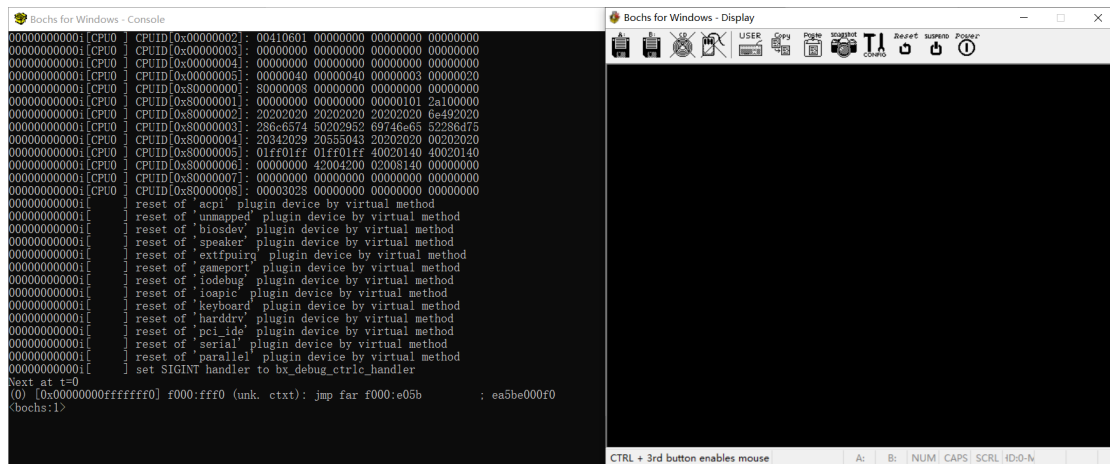


图 17 Console 窗口和 Display 窗口

## 5. 在 Console 窗口输入命令：c，加载 Linux 操作系统；

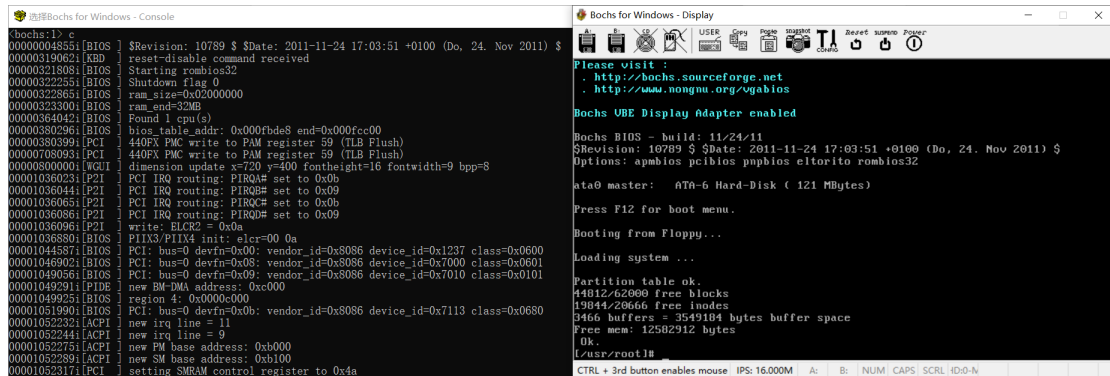


图 18 加载 Linux 操作系统

## 6. 在 Display 窗口输入命令：vi test.c，输入实验程序，输入命令：:wq，保存程序并退出；



9. 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 2。索引号后一位（即低位起倒数第三位，用高亮标识的位置）为 T1 位，可以读出 T1=1，所以段描述符存放在局部段描述符表(ADT)中，且应在局部段描述符表的第 3 项（起始号为 0）；
10. 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）；
11. 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，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。
12. 输入命令：xp /2w 0x5d20，查看 GDT 中对应的表项，如图 22 所示。

```
<bochs:3> xp /2w 0x5d20
[bochs]:
0x00000000000005d20 <bogus+      0>:      0x92d00068      0x000082fd
```

图 22 GDT 中对应的表项

接下来进行计算与地址拼接转化：注意到 0x92d00068 为低位（0-15 位），0x000082fd 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 1101 1001 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 1101 1001 0010 1101 0000（0x00fd92d0）。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 0x00fd92d0+2\*8，得到的结果如图 23 所示。

```

<bochs:5> 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, 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=0x92d00068, valid=1
tr:0x0060, dh=0x00008bfd, dl=0x92e80068, valid=1
gdtr:base=0x000000000000005cb8, limit=0x7fff
idtr:base=0x0000000000000054b8, limit=0x7fff
<bochs:6> xp /2w 0x00fd92d0+2*8
[bochs]:
0x0000000000fd92e0 <bogus+      0>:  0x00003fff  0x10c0f300

```

图 23 xp /2w 0x00fd92d0+2\*8 的结果图

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。

13. 由图 21 可知：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=0x00003004=0001 0000  
0000 0000 0011 0000 0000 0100。线性地址被划分为三部分，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；

14. 输入命令：creg，如图 24 所示。

```

<bochs:28> creg
CR0=0x80000001b: PG cd nw ac wp ne ET TS em MP PE
CR2=page fault laddr=0x000000000804253c
CR3=0x0000000000000000
    PCD=page-level cache disable=0
    PWT=page-level write-through=0
CR4=0x00000000: smep osxsave pcid fsgsbase smx vmx osxmmexcpt osfxsr pce pge mce pae pse de tsd pvi vme
EFER=0x00000000: ffxsr nxe lma lme sce

```

图 24 CR 寄存器的值

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=0=0x100。输入命令：xp/w 0x100，查看页目录(PDE)的值，结果如图 25 所示；

```

<bochs:8> xp /w 0x100
[bochs]:
0x00000000000000100 <bogus+      0>:  0x00fa6027

```

图 25 PDE 的值

PTE 的值为  $0x00fa6027=0000\ 0000\ 1111\ 1010\ 0110\ 0000\ 0010\ 0111$ 。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  $0x00fa6000$ 。同理，对应页表（PTE）地址为  $0x00fa6000+3*4=0x00fa600c$ 。输入命令：xp /w  $0x00fa600c$ ，查看页表（PTE）的值，结果如图 26 所示。

```
<bochs:9> xp /w 0x00fa600c
[bochs]:
0x0000000000fa600c <bogus+      0>:    0x00fa3067
```

图 26 PTE 的值

页表（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  $0x00fa3000+4=0x00fa3004$ ；

15. 输入命令：xp /w  $0x00fa3004$ ，结果如图 27 所示。

```
<bochs:10> xp /w 0x00fa3004
[bochs]:
0x0000000000fa3004 <bogus+      0>:    0x00123456
```

图 27 地址  $0x00fa3004$  的值

将结果与 j 的值对比，显示的结果正确，即已经找到了 j 所在的正确的物理地址  $0x00003004$ ；

16. 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。Display 窗口如图 28 所示。

```
<bochs:11> setpmem 0x00fa3004 4 0
<bochs:12> c
the address of j is 0x3004
the terminated normally!
[/usr/root]#
```

图 28 Console 窗口输入的命令和 Display 窗口显示的结果

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，实验顺利完成。

## 八、总结及心得体会：

通过本次实验，我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。在实验过程中，我错误地在 Display 窗口按下 Ctrl+C，使得 Linux 操作系统中断运行，导致后面实验的错误，在纠正错误后，程序能正常运

行退出，实验取得成功。

## 九、对本实验过程及方法、手段的改进建议：

完善地址映射实验指导书，使同学能够更好地书写实验内容、实验目的、实验原理等部分，也使得同学能够更好地完成实验。