

# 电子科技大学

## 实验报告

学生姓名：郭宇航 学号：2016100104014 指导教师：薛瑞尼

实验地点：主楼 A2-412

实验时间：2019.5.25

一、实验室名称：计算机实验室

二、实验项目名称：

系统化思维模式下计算机操作系统进程与资源管理设计

三、实验学时：4

四、实验原理：

1. 进程的三种基本状态：

- a. 就绪状态：当前进程已经分配得到除 CPU 外的所有必要的资源，只要再获得 CPU 即可运行，此时进程的状态被称为就绪状态。
- b. 执行状态：进程已经获得了 CPU，其程序正在执行，在单处理机的系统中只有一个进程处于执行状态。
- c. 阻塞状态：处于执行状态的进程由于请求资源失败或其他事情而暂停无法继续执行下去时，放弃处理机而暂时处于暂停状态，此时进程的状态被称为阻塞状态或者等待状态或者封锁状态。

2. 总体的设计：

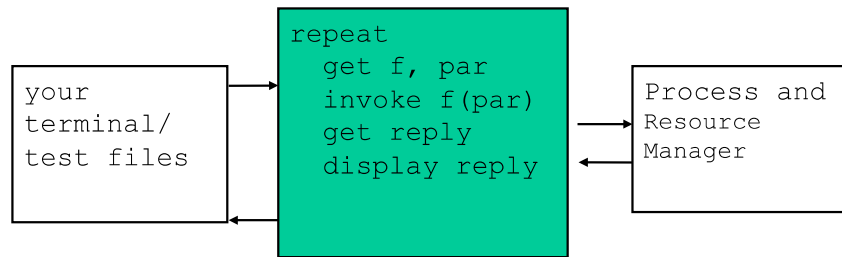


图 1 系统总体结构

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

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

图 1 最左端部分为：通过终端（如键盘输入）或者测试文件来给出相应的用户命令，以及模拟硬件引起的中断

Test\_shell 的功能如 4.1 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

### 3. 进程和资源管理设计

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

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)**

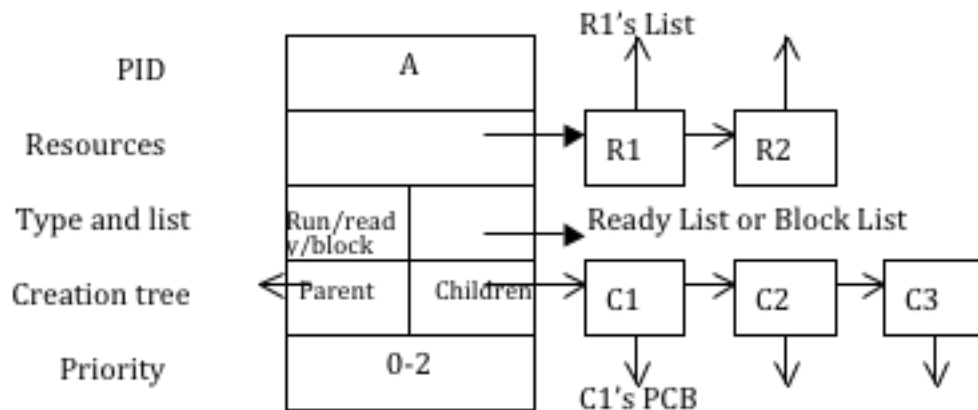


图 2 PCB 结构示意图

就绪进程队列：Ready list (RL)

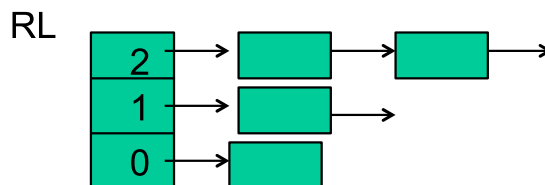


图 3 Ready list 数据结构

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。

资源数据结构：

**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

#### 4. 主要函数

- 创建进程:

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

{

create PCB data structure

initialize PCB using parameters //包括进程的 ID，优先级、状态等

link PCB to creation tree /\*连接父亲节点和兄弟节点，当前进程为父亲节点，父亲节点中的子节点为兄弟节点\*/

insert(RL, PCB)//插入就绪相应优先级队列的尾部

Scheduler()

}

Init 进程在启动时创建,可以用来创建第一个系统进程或者用户进程。

新创建的进程或者被唤醒的进程被插入到就绪队列（RL）的末尾。

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

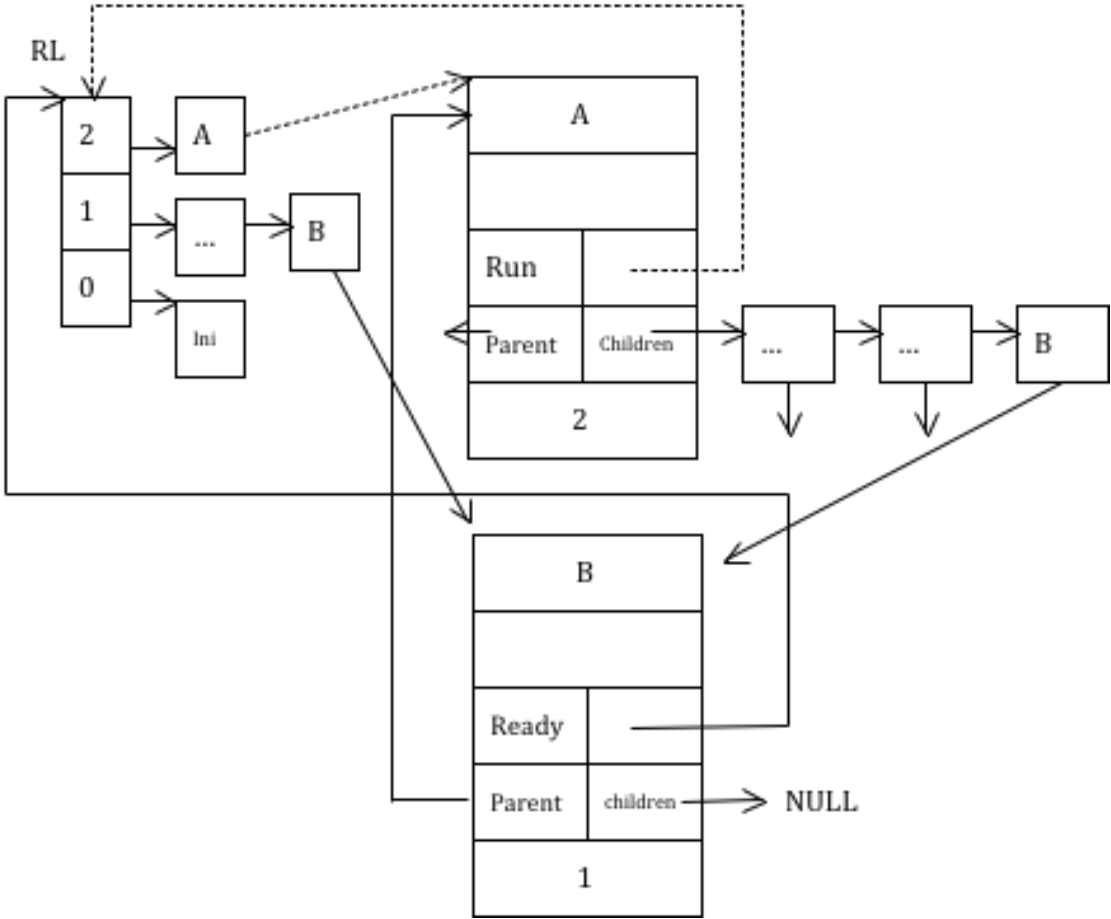


图 4 进程数据结构间关系

(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

Destroy (pid)

```
{
    get pointer p to PCB using pid
    Kill_Tree(p)
    Scheduler() //调度其他进程执行
}
```

Kill\_Tree(p)

```
{  
    for all child processes q Kill_Tree(q) //嵌套调用，撤销所有子孙进程  
    free resources //和 release 调用类似的功能  
    delete PCB and update all pointers  
}
```

Process can be destroyed by any of its ancestors or by itself (exit)

### ● 进程请求资源

所有的资源申请请求按照 FIFO 的顺序进行

### ● 情况一：当一类资源数量本身只有一个的情况

Request(rid)

```
{  
    r = Get_RCB(rid);  
    if (r->Status == 'free') //只有一个资源时可以用 free 和 allocated 来  
表示资源状态  
    {  
        r->Status = 'allocated';  
        insert(self->Resources, r); //self 为当前请求资源的进程 PCB,  
insert 以后 r 为 self 进程占有的资源, 参 PCB 结构图  
    }  
    else  
    {
```

```

self->Status.Type = 'blocked';

self->Status.List = r;//point to block list, self is blocked by r

remove(RL, self);// remove self from the ready list(self can be put at
the head of RL when it is running).

```

```

insert(r->Waiting_List, self);// 将进程 self 插入到资源 r 的等待队
列尾部

```

```

Scheduler();

}

}

```

## ● 情况二：一类资源有多个的情况（multi\_unit）

Request(rid, n) // n 为请求资源数量

```

{

r = Get_RCB(rid);

if (u ≥ n) // u 为 r->Status.u, 即可用资源数量, 参资源数据结构
图

```

```

{

u-n;

insert(self->Resources, r, n);//self 为当前请求资源的进程 PCB,

```

insert 以后 n 个 r 为 self 进程占有的资源, 参 PCB 结构图

```

}

else

{

```



if (n>k) exit;//k 为资源 r 的总数，申请量超过总数时，将打印错误信息并退出

self->Status.Type = 'blocked'; //只要  $u < n$ ，就不分配，进程阻塞

self->Status.List = r;//point to block list 注意：此时 block list 是 n 个 r

remove(RL, self);// remove self from the ready list，因为运行进程位于就绪队列首部，所以此时将它从就绪队列移除

insert(r->Waiting\_List, self);// 将进程 self 插入到资源 r 的等待队列尾部

Scheduler();

}

}

## ● 进程释放资源

### ● 情况一：一类资源只有 1 个的情况

Release(rid)

{

r = Get\_RCB(rid);

remove(self->Resources, r);//将 r 从进程 self 占用的资源中移走

if (r->Waiting\_List == NIL) //没有进程在等待资源 r

{

r->Status = 'free';

}

```

else
{
    remove(r->Waiting_List, q); //q 为 waiting_list 中第一个阻塞进程
    q->Status.Type = 'ready';
    q->Status.List = RL; //就绪队列
    insert(q->Resources, r);
    insert(RL, q); //q 插入就绪队列中相应优先级队列的末尾
    Scheduler();
}
}

```

● 情况二：一类资源有多个的情况

Release(rid,n) //rid 为资源 ID, n 为释放的资源数量

```

{
    r = Get_RCB(rid);

    /*remove r from self->resources, and u= u + n, 将资源 r 从当前进程占
    用的资源列表里移除, 并且资源 r 的可用数量从定义结构体: PCB 和
    RCB。PCB: pid + type + resource + resource number + 优先级; RCB:
    四类资源: R1, R2, R3, R4 每类资源的结构: u 变为 u+n*/

```

```

    remove(self->Resources, r, n);

```

```

/*如果阻塞队列不为空, 且阻塞队列首部进程需求的资源数 req 小于
等于可用资源数量 u, 则唤醒这个阻塞进程, 放入就绪队列*/

```

```

while (r->Waiting_List != NIL && u>=req_num)
{
    u=u- req_num; //可用资源数量减少
    remove(r->Waiting_List, q); // 从资源 r 的阻塞队列中移除
    q->Status.Type = 'ready';
    q->Status.List = RL;
    insert(q->Resources, r); //插入 r 到 q 所占用的资源中
    insert(RL, q); // 插入 q 到就绪队列
}

Scheduler(); //基于优先级的抢占式调度策略，因此当有进
程获得资源时，需要查看当前的优先级情况并进行调度
}

```

## 5. 进程调度与时钟中断的设计

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

● Scheduler:

Called at the end of every kernel call

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

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

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

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

Preemption: //抢占, 将 P 变为执行, 输出当前运行进程的名称

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

- 时钟中断 (Time out): 模拟时间片到或者外部硬件中断

Time\_out()

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

## 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

## 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）：

硬件条件：华硕飞行堡垒 fx50j i5-4200HQ 4G

软件平台：Windows10 专业版 python3.6.0 Pycharm 2017.2.3

## 八、实验步骤：

首先定义进程控制块 PCB 与资源控制块 RCB 以及整个系统的进程处理模块：process\_handle\_sector

```
1. class RCB:
2.     def __init__(self):
3.         self.Rid = None
4.         self.Initial = 0
5.         self.Remain = 0
6.         self.Waiting_list = []
7.
8. class process_handle_sector:
9.     def __init__(self):
10.         #定义资源 1 的阻塞队列
11.         self.R1 = RCB()
12.         self.R1.Rid = 'R1'
13.         self.R1.Initial = 1
14.         self.R1.Remain = 1
15.
16.         self.R2 = RCB()
17.         self.R2.Rid = 'R2'
18.         self.R2.Initial = 2
19.         self.R2.Remain = 2
20.
21.         self.R3 = RCB()
22.         self.R3.Rid = 'R3'
```

```

23.         self.R3.Initial = 3
24.         self.R3.Remain = 3
25.
26.         self.R4 = RCB()
27.         self.R4.Rid = 'R4'
28.         self.R4.Initial = 4
29.         self.R4.Remain = 4
30.         #初始化三个不同优先级的就绪队列
31.         self.RL = [[],[],[ ]]
32.         self.current_process = None
33.         self.current_process_pid = -1
34.         self.RCB_list = [self.R1,self.R2,self.R3,self.R4]
35.         self.resource_list = [self.R1.Waiting_list,
36.                                self.R2.Waiting_list,
37.                                self.R3.Waiting_list,
38.                                self.R4.Waiting_list]
39.         self.destroyed_process = []
40.
41. class PCB:
42.     def __init__(self):
43.         self.Pid = None
44.         self.type = 'ready' #对于一个进程来说: ready/running/blocked
45.         self.resources = []
46.         self.resources_num = []
47.         self.priority = None
48.         self.request_resources = None
49.         self.request_resources_num = -1
50.         self.parent = None
51.         self.children = []

```

下面给出一些主要函数的代码部分：

首先是创建进程函数：

```

1. #创建一个进程
2. def create_process(command,process_sector):
3.     #create PCB data structure
4.     process = PCB()
5.     #initialize PCB using parameters 包括进程的 ID, 优先级、状态等
6.     process.Pid = command[1]
7.     process.priority = int(command[2])
8.     if process_sector.current_process != None:
9.         process.parent = process_sector.current_process
10.        process_sector.current_process.children.append(process)

```

```

11.     #寻找当前执行的进程，并将原本在就绪队列中的进程 pop 出来然后改为正在运行的程序
12.     if process.priority == 2:
13.         process_sector.RL[0].append(process)
14.         process.type = 'ready'
15.         #优先级为 2 的进程，直接放入就绪队列 1，并且判断是否是就绪队列 1 中的唯一元
            素，如果是直接执行？是否有问题？
16.     elif process.priority == 1:
17.         process_sector.RL[1].append(process)
18.         process.type = 'ready'
19.     else:
20.         #此时为优先级为 0 的进程，虚拟进程
21.         process_sector.RL[2].append(process)
22.         process.type = 'Virtual'
23.         ....
24.     上面的思路很简单，创建进程后先将创建的进程放入就绪队列中，其他先不动，下面再考
        虑当前进程的问题
25.     ...
26.     if process_sector.current_process == None:
27.         if len(process_sector.RL[0]) != 0:
28.             # 如果 system 级别的进程就绪队列不为空的话，从就绪队列 2 中取出 system
                级别的队首就绪进程作为当前进程
29.             process_sector.current_process = process_sector.RL[0][0]
30.             process_sector.current_process.Pid = process_sector.RL[0][0].Pid
31.             process_sector.RL[0][0].type = 'running'
32.             process_sector.RL[0][0].parent = 'root'
33.             process_sector.RL[0].pop(0)
34.         elif len(process_sector.RL[0]) == 0 and len(process_sector.RL[1]) !=
            0:
35.             process_sector.current_process = process_sector.RL[1][0]
36.             process_sector.current_process.Pid = process_sector.RL[1][0].Pid
37.             process_sector.RL[1][0].type = 'running'
38.             process_sector.RL[1][0].parent = 'root'
39.             process_sector.RL[1].pop(0)
40.         else:
41.             print('ERROR,process is a virtual process')
42.     else:
43.         if process_sector.current_process.priority == 2:
44.             pass
45.         elif process_sector.current_process.priority == 1:
46.             ....
47.         这个地方不存在当前进程优先级为 1 同时优先级为 2 的就绪队列中还有很多进
            程！

```



```

48.         ...
49.         if len(process_sector.RL[0])!=0:
50.             #这种情况只存在一种可能性就是刚刚创建的进程优先级为 2
51.             process_sector.current_process.type = 'ready'
52.             process_sector.RL[1].append(process_sector.current_process)
53.             process_sector.current_process = process_sector.RL[0][0]
54.             process_sector.current_process_pid = process_sector.RL[0][0]
55.             .Pid
56.             process_sector.RL[0][0].type = 'running'
57.             process_sector.RL[0].pop(0)
58.         else:
59.             pass
60.     else:
61.         pass

```

销毁进程函数:

```

1. def block2ready_function(waiting_list,RCB_use,ready_list):
2.     count = 0
3.     RCB_use_cal_remain = RCB_use.Remain
4.     #print('RCB_use',RCB_use.Rid)
5.     if len(waiting_list) != 0:
6.         #计算在阻塞队列中有多少进程在释放资源后可以进入就绪队列
7.         for i in waiting_list:
8.             if i.request_resources_num <= RCB_use_cal_remain:
9.                 count +=1
10.                RCB_use_cal_remain -= i.request_resources_num
11.            else:
12.                RCB_use_cal_remain -= i.request_resources_num
13.        #print('可以从阻塞队列出来的进程数为: ',count)
14.        #count 计算了所有可以进入就绪队列的阻塞进程数量，遍历这些进程，拿走资源且进入就绪状态
15.        if count <= 0:
16.            #表示释放的资源连阻塞队列的第一个进程都无法满足
17.            pass
18.        else:
19.            #这种情况下释放的资源可以满足一些阻塞的进程使用
20.            for j in range(count):
21.                #print(waiting_list[j].priority)
22.                waiting_list[j].type = 'ready'
23.                waiting_list[j].resources.append(RCB_use.Rid)
24.                #print('此处的资源为: ',waiting_list[j].resources)

```

```

25.         waiting_list[j].resources_num.append(waiting_list[j].request
           _resources_num)
26.         RCB_use.Remain -= waiting_list[j].request_resources_num
27.         waiting_list[j].request_resources = None
28.         waiting_list[j].request_resources_num = -1
29.         if waiting_list[j].priority == 2:
30.             ready_list[0].append(waiting_list[j])
31.             waiting_list.pop(j)
32.         elif waiting_list[j].priority == 1:
33.             ready_list[1].append(waiting_list[j])
34.             #print('The process has been insert to RList!')
35.             waiting_list.pop(j)
36.         else:
37.             ready_list[2].append(waiting_list[j])
38.             waiting_list.pop(j)
39.     else:
40.         #此时阻塞队列为空，不进行任何操作
41.         pass
42.
43. #对于单个进程进行销毁
44. def destroyed_single_process(process,process_sector):
45.     process.type = 'destroyed'
46.     process_sector.destroyed_process.append(process)
47.     #print('正在销毁的进程的 PID 以及占用资源的数量:
           ',process.Pid,process.resources)
48.     #if process.Pid == 'B':
49.         #print('当前 R3 的资源数量: ',process_sector.R3.Remain)
50.         #print('进程 B 的资源:
           ',process.resources[0],process.resources_num[0])
51.     if process.resources == []:
52.         pass
53.     else:
54.         for i in range(len(process.resources)):
55.             if process.resources[i] == 'R1':
56.                 process_sector.R1.Remain += process.resources_num[i]
57.             elif process.resources[i] == 'R2':
58.                 process_sector.R2.Remain += process.resources_num[i]
59.             elif process.resources[i] == 'R3':
60.                 process_sector.R3.Remain += process.resources_num[i]
61.             elif process.resources[i] == 'R4':
62.                 process_sector.R4.Remain += process.resources_num[i]
63.             else:
64.                 print('error,this process does not have this kind of resourc
           e!')

```

```

65.     #释放资源后判断是否存在在阻塞队列中的进程可以进入就绪队列
66.     for j in range(len(process_sector.resource_list)):
67.         block2ready_function(process_sector.resource_list[j],process_sector.
            RCB_list[j],process_sector.RL)
68.     if process.Pid == process_sector.current_process_pid:
69.         if len(process_sector.RL[0]) != 0:
70.             process_sector.current_process = process_sector.RL[0][0]
71.             process_sector.current_process_pid = process_sector.RL[0][0].Pid
72.             process_sector.RL[0].pop(0)
73.         elif len(process_sector.RL[0]) == 0 and len(process_sector.RL[1]) !=
            0:
74.             process_sector.current_process = process_sector.RL[1][0]
75.             process_sector.current_process_pid = process_sector.RL[1][0].Pid
76.             process_sector.RL[1].pop(0)
77.         elif process in process_sector.RL[0]:
78.             process_sector.RL[0].remove(process)
79.         elif process in process_sector.RL[1]:
80.             process_sector.RL[1].remove(process)
81.         elif process in process_sector.R1.Waiting_list:
82.             process_sector.R1.Waiting_list.remove(process)
83.         elif process in process_sector.R2.Waiting_list:
84.             process_sector.R2.Waiting_list.remove(process)
85.         elif process in process_sector.R3.Waiting_list:
86.             process_sector.R3.Waiting_list.remove(process)
87.         elif process in process_sector.R4.Waiting_list:
88.             process_sector.R4.Waiting_list.remove(process)
89.         else:
90.             print('Error,can not find where is the process')
91.
92. #递归删除进程及其子进程（树状结构）
93. def destroy_recursion(process,process_sector):
94.     destroyed_single_process(process,process_sector)
95.     for j in range(len(process_sector.resource_list)):
96.         block2ready_function(process_sector.resource_list[j],process_sector.
            RCB_list[j],process_sector.RL)
97.     if process.children != []:
98.         for i in process.children:
99.             destroy_recursion(i,process_sector)
100.
101.
102. def destroy_process(command,process_sector):
103.     destroy_pid = command[1]

```

```

104.     ....
105.     需要销毁的进程一共有三大类，第一类是当前进程，第二类是在就绪队列中，第三类是在
        阻塞队列中
106.     '''
107.     if destroy_pid == process_sector.current_process_pid:
108.         #print('需要销毁的进程是当前执行的进程! ')
109.         destroy_recursion(process_sector.current_process,process_sector)
110.     if len(process_sector.RL[0]) != 0:
111.         for i in process_sector.RL[0]:
112.             if destroy_pid == i.Pid:
113.                 #print('需要销毁的进程在就绪队列 0')
114.                 destroy_recursion(i,process_sector)
115.     if len(process_sector.RL[1]) !=0:
116.         for j in process_sector.RL[1]:
117.             if destroy_pid == j.Pid:
118.                 #print('需要销毁的进程在就绪队列 1')
119.                 destroy_recursion(j,process_sector)
120.     if len(process_sector.R1.Waiting_list)!=0:
121.         for k in process_sector.R1.Waiting_list:
122.             if destroy_pid == k.Pid:
123.                 #print('需要销毁的进程在资源 1 的阻塞队列中')
124.                 destroy_recursion(k,process_sector)
125.     if len(process_sector.R2.Waiting_list)!=0:
126.         for m in process_sector.R2.Waiting_list:
127.             if destroy_pid == m.Pid:
128.                 #print('需要销毁的进程在资源 2 的阻塞队列中')
129.                 destroy_recursion(m,process_sector)
130.     if len(process_sector.R3.Waiting_list) != 0:
131.         for n in process_sector.R3.Waiting_list:
132.             if destroy_pid == n.Pid:
133.                 #print('需要销毁的进程在资源 3 的阻塞队列中')
134.                 destroy_recursion(n,process_sector)
135.     if len(process_sector.R4.Waiting_list) != 0:
136.         for q in process_sector.R4.Waiting_list:
137.             if destroy_pid == q.Pid:
138.                 #print('需要销毁的进程在资源 4 的阻塞队列中')
139.                 destroy_recursion(q,process_sector)

```

时间片周期轮转函数：

```

1.  #时间周期转完更换当前进程
2.  def change_current_process(process_sector):
3.      if len(process_sector.RL[0]) != 0:
4.          if process_sector.current_process.priority == 2:

```

```

5.         process_sector.current_process.type = 'ready'
6.         process_sector.RL[0].append(process_sector.current_process)
7.         process_sector.current_process = process_sector.RL[0][0]
8.         process_sector.current_process_pid = process_sector.RL[0][0].Pid

9.         process_sector.RL[0].pop(0)
10.        elif process_sector.current_process.priority == 1:
11.            process_sector.current_process.type = 'ready'
12.            process_sector.RL[1].append(process_sector.current_process)
13.            process_sector.current_process = process_sector.RL[1][0]
14.            process_sector.current_process_pid = process_sector.RL[1][0].Pid

15.            process_sector.RL[1].pop(0)
16.        elif len(process_sector.RL[0]) == 0 and len(process_sector.RL[1])!=0:
17.            #这里表示就绪队列 1 为空，没有优先级为 2 的进程在等待，直接考虑就绪队列 2
18.            if process_sector.current_process.priority == 1:
19.                process_sector.current_process.type = 'ready'
20.                process_sector.RL[1].append(process_sector.current_process)
21.                process_sector.current_process = process_sector.RL[1][0]
22.                process_sector.current_process_pid = process_sector.RL[1][0].Pid

23.                process_sector.RL[1].pop(0)
24.            else:
25.                pass
26.        else:
27.            #这里表示两个就绪队列都为空，则当前没有其他的进程，这里不考虑第三优先级的进
            程
28.            pass

```

请求资源函数：

```

1. #请求资源操作
2. def request_resources(command, process_sector):
3.     request_r = command[1]
4.     request_num = int(command[2])
5.     process_sector.current_process.request_resources = request_r
6.     process_sector.current_process.request_resources_num = request_num
7.     if process_sector.current_process.request_resources == 'R1':
8.         if process_sector.current_process.request_resources_num <= process_s
            ector.R1.Remain:
9.             #此时可以给当前进程分配资源
10.            process_sector.R1.Remain -
            = process_sector.current_process.request_resources_num
11.            process_sector.current_process.resources.append('R1')

```

```

12.         process_sector.current_process.resources_num.append(process_sector.current_process.request_resources_num)
13.         process_sector.current_process.request_resources = None
14.         process_sector.current_process.request_resources_num = -1
15.     else:
16.         #此时不能分配资源
17.         process_sector.current_process.request_resources = request_resource
18.         process_sector.current_process.request_resources_num = request_resource_num
19.         process_sector.current_process.type = 'blocked'
20.         process_sector.resource_list[0].append(process_sector.current_process)
21.         if len(process_sector.RL[0])!=0:
22.             process_sector.current_process = process_sector.RL[0][0]
23.             process_sector.current_process_pid = process_sector.RL[0][0].Pid
24.             process_sector.RL[1].pop(0)
25.         elif len(process_sector.RL[0])==0 and len(process_sector.RL[1])!=0:
26.             process_sector.current_process = process_sector.RL[1][0]
27.             process_sector.current_process_pid = process_sector.RL[1][0].Pid
28.             process_sector.RL[1].pop(0)
29.         else:
30.             pass
31.     elif process_sector.current_process.request_resources == 'R2':
32.         if process_sector.current_process.request_resources_num <= process_sector.R2.Remain:
33.             #此时可以给当前进程分配资源
34.             process_sector.R2.Remain -
35.             = process_sector.current_process.request_resources_num
36.             process_sector.current_process.resources.append('R2')
37.             process_sector.current_process.resources_num.append(process_sector.current_process.request_resources_num)
38.             process_sector.current_process.request_resources = None
39.             process_sector.current_process.request_resources_num = -1
40.         else:
41.             #此时不能分配资源
42.             process_sector.current_process.request_resources = request_resource
43.             process_sector.current_process.request_resources_num = request_resource_num
44.             process_sector.current_process.type = 'blocked'
45.             process_sector.resource_list[1].append(process_sector.current_process)

```

```

45.         if len(process_sector.RL[0])!=0:
46.             process_sector.current_process = process_sector.RL[0][0]
47.             process_sector.current_process_pid = process_sector.RL[0][0]
48.             .Pid
49.             process_sector.RL[1].pop(0)
50.         elif len(process_sector.RL[0])==0 and len(process_sector.RL[1])!=
51.             =0:
52.             process_sector.current_process = process_sector.RL[1][0]
53.             process_sector.current_process_pid = process_sector.RL[1][0]
54.             .Pid
55.             process_sector.RL[1].pop(0)
56.         else:
57.             pass
58.         elif process_sector.current_process.request_resources == 'R3':
59.             if process_sector.current_process.request_resources_num <= process_s
60.                 ector.R3.Remain:
61.                 #此时可以给当前进程分配资源
62.                 process_sector.R3.Remain -
63.                 = process_sector.current_process.request_resources_num
64.                 process_sector.current_process.resources.append('R3')
65.                 process_sector.current_process.resources_num.append(process_sect
66.                     or.current_process.request_resources_num)
67.                 process_sector.current_process.request_resources = None
68.                 process_sector.current_process.request_resources_num = -1
69.             else:
70.                 #此时不能分配资源
71.                 process_sector.current_process.request_resources = request_r
72.                 process_sector.current_process.request_resources_num = request_n
73.                 um
74.                 process_sector.current_process.type = 'blocked'
75.                 process_sector.resource_list[2].append(process_sector.current_pr
76.                     ocess)
77.                 if len(process_sector.RL[0])!=0:
78.                     process_sector.current_process = process_sector.RL[0][0]
79.                     process_sector.current_process_pid = process_sector.RL[0][0]
80.                     .Pid
81.                     process_sector.RL[0].pop(0)
82.                 elif len(process_sector.RL[0])==0 and len(process_sector.RL[1])!=
83.                     =0:
84.                     process_sector.current_process = process_sector.RL[1][0]
85.                     process_sector.current_process_pid = process_sector.RL[1][0]
86.                     .Pid
87.                     process_sector.RL[1].pop(0)
88.                 else:

```

```

78.         pass
79.     elif process_sector.current_process.request_resources == 'R4':
80.         if process_sector.current_process.request_resources_num <= process_s
            ector.R4.Remain:
81.             #此时可以给当前进程分配资源
82.             process_sector.R4.Remain -
            = process_sector.current_process.request_resources_num
83.             process_sector.current_process.resources.append('R4')
84.             process_sector.current_process.resources_num.append(process_sect
            or.current_process.request_resources_num)
85.             process_sector.current_process.request_resources = None
86.             process_sector.current_process.request_resources_num = -1
87.         else:
88.             #此时不能分配资源
89.             process_sector.current_process.request_resources = request_r
90.             process_sector.current_process.request_resources_num = request_n
            um
91.             process_sector.current_process.type = 'blocked'
92.             process_sector.resource_list[3].append(process_sector.current_pr
            ocess)
93.             if len(process_sector.RL[0])!=0:
94.                 process_sector.current_process = process_sector.RL[0][0]
95.                 process_sector.current_process_pid = process_sector.RL[0][0]
            .Pid
96.                 process_sector.RL[1].pop(0)
97.             elif len(process_sector.RL[0])==0 and len(process_sector.RL[1])!
            =0:
98.                 process_sector.current_process = process_sector.RL[1][0]
99.                 process_sector.current_process_pid = process_sector.RL[1][0]
            .Pid
100.                 process_sector.RL[1].pop(0)
101.             else:
102.                 pass
103.         else:
104.             print('Error,没有这种类型的资源可以申请! ')

```

释放资源函数:

```

1. #释放资源
2. def release_resources(command,process_sector):
3.     release_r = command[1]
4.     release_num = int(command[2])
5.     #print('应该释放的资源为: ',release_r,release_num)
6.     if release_r == 'R1':

```



```
7.         if 'R1' in process_sector.current_process.resources:
8.             release_index_r1 = process_sector.current_process.resources.index('R1')
9.             process_sector.current_process.resources_num[release_index_r1] -
= release_num
10.            process_sector.R1.Remain += release_num
11.            block2ready_function(process_sector.R1.Waiting_list,process_sector.R1,process_sector.RL)
12.        else:
13.            print('ERROR,CurrentProcess does not own this resource!')
14.        elif release_r == 'R2':
15.            if 'R2' in process_sector.current_process.resources:
16.                release_index_r2 = process_sector.current_process.resources.index('R2')
17.                process_sector.current_process.resources_num[release_index_r2] -
= release_num
18.                process_sector.R2.Remain += release_num
19.                block2ready_function(process_sector.R2.Waiting_list,process_sector.R2,process_sector.RL)
20.            else:
21.                print('ERROR,CurrentProcess does not own this resource!')
22.        elif release_r == 'R3':
23.            if 'R3' in process_sector.current_process.resources:
24.                release_index_r3 = process_sector.current_process.resources.index('R3')
25.                #print('在资源中的索引: ',release_index_r3)
26.                process_sector.current_process.resources_num[release_index_r3] -
= release_num
27.                #print('增加之前 R3 的数量',process_sector.R3.Remain)
28.                process_sector.R3.Remain += release_num
29.                #print('增加之后 R3 的数量',process_sector.R3.Remain)
30.                block2ready_function(process_sector.R3.Waiting_list,process_sector.R3,process_sector.RL)
31.            else:
32.                print('ERROR,CurrentProcess does not own this resource!')
33.        elif release_r == 'R4':
34.            if 'R4' in process_sector.current_process.resources:
35.                release_index_r4 = process_sector.current_process.resources.index('R4')
36.                process_sector.current_process.resources_num[release_index_r4] -
= release_num
37.                process_sector.R4.Remain += release_num
38.                block2ready_function(process_sector.R4.Waiting_list,process_sector.R4,process_sector.RL)
```

```

39.         else:
40.             print('ERROR,CurrentProcess does not own this resource!')
41.     else:
42.         print('ERROR!检查输入文件是否正确! ')
43.     if process_sector.current_process.priority == 2:
44.         pass
45.     elif process_sector.current_process.priority == 1:
46.         if len(process_sector.RL[0])!=0:
47.             #在释放了进程之后，存在优先级更高的进程从阻塞队列中出来，需要抢占运行
48.             process_sector.current_process.type = 'ready'
49.             process_sector.RL[1].append(process_sector.current_process)
50.             process_sector.current_process = process_sector.RL[0][0]
51.             process_sector.current_process_pid = process_sector.RL[0][0].Pid
52.             process_sector.RL[0][0].type = 'running'
53.             process_sector.RL[0].pop(0)

```

读取 txt 文件的代码：

```

1. #读取 txt 文件
2. def load_test_shell(filepath):
3.     shell_txt = []
4.     with open(filepath,'r') as file_load:
5.         while True:
6.             lines = file_load.readline()
7.             if not lines:
8.                 break
9.             pass
10.            shell_txt.append(lines.split())
11.    return shell_txt

```

汇总上述的所有函数的代码以及主函数：

```

1. #主函数
2. def main_function(shell_text,process_sector):
3.     init_process = init_function('init')
4.     result = []
5.     result.append(init_process)
6.     for i in shell_text:
7.         if i[0] == 'cr':
8.             create_process(i,process_sector)
9.             result.append(scheduler(process_sector))
10.        elif i[0] == 'to':

```

```

11.         change_current_process(process_sector)
12.         result.append(scheduler(process_sector))
13.         elif i[0] == 'req':
14.             request_resources(i,process_sector)
15.             result.append(scheduler(process_sector))
16.         elif i[0] == 'de':
17.             destroy_process(i,process_sector)
18.             result.append(scheduler(process_sector))
19.         elif i[0] == 'rel':
20.             release_resources(i,process_sector)
21.             result.append(scheduler(process_sector))
22.         else:
23.             print('Error!检查输入文件! ')
24.     return result
25.
26. if __name__ == '__main__':
27.     shell_txt = load_test_shell('5.txt')
28.     #print(shell_txt)
29.     process_sector = process_handle_sector()
30.     process_result = main_function(shell_txt,process_sector)
31.     for i in process_result:
32.         print(i,end=' ')
33.     print('\n')
34.     print('*****this is for debug*****')
35.     #print('the length of ready_List1',len(process_sector.RL[1]))
36.     #print('destroyed process list',process_sector.destroyed_process[0].Pid)
37.     #print('当前进程的 PID',process_sector.current_process_pid)
38.     #print('当前进程的资源占用: ',process_sector.current_process.resources)
39.     #print('当前进程的资源占用数量:
    ',process_sector.current_process.resources_num)
40.     #print('当前 R3 的资源数量',process_sector.R3.Remain)
41.     if len(process_sector.RL[0])!=0:
42.         print('在就绪队列 0 中（system 级别的进程）进程 PID: ',end=' ')
43.         for j in process_sector.RL[0]:
44.             print(j.Pid,end=' ')
45.         print('\n')
46.     else:
47.         print('就绪队列 0 为空! \n')
48.     if len(process_sector.RL[1])!=0:
49.         print('在就绪队列 1 中（优先级为 1 级别的进程）进程 PID: ',end=' ')
50.         for i in process_sector.RL[1]:
51.             print(i.Pid,end=' ')
52.         print('\n')

```

```

53.     else:
54.         print('就绪队列 1 为空! \r')
55.         ....
56.     for i in process_sector.RL[1]:
57.         if i.Pid == 'B':
58.             print('进程 B 的资源以及数量',i.resources,i.resources_num)
59.             ...
60.     if len(process_sector.R1.Waiting_list)!=0:
61.         print('在资源 1 的阻塞队列中的进程 PID: ',end=' ')
62.         for k in process_sector.R1.Waiting_list:
63.             print(k.Pid,end=' ')
64.         print('\r')
65.     else:
66.         print('资源 1 的阻塞队列为空! \r')
67.     if len(process_sector.R2.Waiting_list)!=0:
68.         print('在资源 2 的阻塞队列中的进程 PID: ',end=' ')
69.         for m in process_sector.R2.Waiting_list:
70.             print(m.Pid,end=' ')
71.         print('\r')
72.     else:
73.         print('资源 2 的阻塞队列为空! \r')
74.     if len(process_sector.R3.Waiting_list)!=0:
75.         print('在资源 3 的阻塞队列中的进程 PID: ', end=' ')
76.         for n in process_sector.R3.Waiting_list:
77.             print(n.Pid,end=' ')
78.             #print('进程 B 的资源申请情况
79.             ',n.request_resources,n.request_resources_num)
80.             print('\r')
81.     else:
82.         print('资源 3 的阻塞队列为空! \r')
83.     if len(process_sector.R4.Waiting_list)!=0:
84.         print('在资源 4 的阻塞队列中的进程 PID: ', end=' ')
85.         for q in process_sector.R4.Waiting_list:
86.             print(q.Pid,end=' ')
87.         print('\r')
88.     else:
89.         print('资源 4 的阻塞队列为空! \r')
90.     if len(process_sector.destroyed_process)!=0:
91.         print('已经被销毁的进程 PID: ',end=' ')
92.         for s in process_sector.destroyed_process:
93.             print(s.Pid,end=' ')
94.         print('\r')
95.     else:
96.         print('销毁的进程队列为空! \r')

```

```

96.     print('\n')
97.     print('*****Resources Situation*****')
98.     print('Resources Remains:R1:%d 个,R2:%d 个,R3:%d 个,R4:%d 个
          '%(process_sector.R1.Remain,process_sector.R2.Remain,process_sector.R3.Remai
          n,process_sector.R4.Remain))
99.     ....
100.     print('the length of R3 Waiting list',process_sector.R3.Waiting_list[0]
          .Pid)
101.     print('R3 remains',process_sector.R3.Remain)
102.     print('destroyed process list',process_sector.destroyed_process[0].Pid)
103.     '''

```

## 九、实验结论：

下面给出一些测试样例的测试情况：

测试样例:0.txt 如下：

输入测试命令或将测试命令放在测试文件 0.txt 中，内容为：

```

cr x 1
cr p 1
cr q 1
cr r 1
to
req R2 1
to
req R3 3
to
req R4 3
to
to
req R3 1
req R4 2
req R2 2
to
de q
to
to

```

输出结果应为：init x x x x p p q q r r x p q r x x x p x

测试结果如下：

```
运行 process_resource
C:\Users\Administrator\AppData\Local\Programs\Python\Python36\python3
.exe C:/Users/Administrator/Desktop/process/process_resource.py
init x x x x p p q q r r x p q r x x x p x
```

测试样例:1.txt 如下:

输入测试命令或将测试命令放在测试文件 1.txt 中, 内容为:

```
cr A 1
cr B 1
cr C 1
to
cr D 1
cr E 1
to
cr F 1
req R1 1
req R2 2
to
req R2 1
req R3 3
to
req R4 4
to
req R3 2
to
rel R2 1
to
rel R3 2
to
to
req R3 3
de B
to
to
to
```

输出结果应为: init AAABBBCCCCADDEEBFCCDDEFAACFA

测试结果如下:

```
运行 process_resource
C:\Users\Administrator\AppData\Local\Programs\Python\Python36\python3
.exe C:/Users/Administrator/Desktop/process/process_resource.py
init A A A B B B C C C C A D D E E B F C C D D E F A A C F A
```

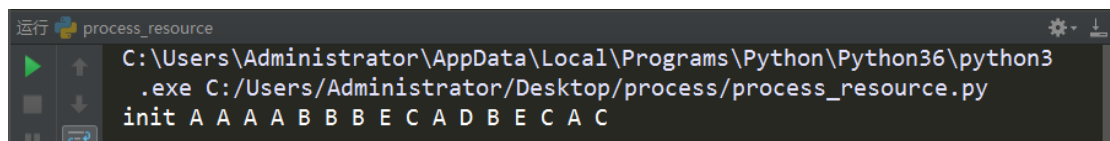
测试样例:2.txt 如下:

输入测试命令或将测试命令放在测试文件 2.txt 中，内容为：

```
cr A 1
cr B 1
cr C 1
req R1 1
to
cr D 1
req R2 2
cr E 2
req R2 1
to
to
to
rel R2 1
de B
to
to
```

输出结果应为：init A A A A B B B E C A D B E C A C

测试结果如下：



```
运行 process_resource
C:\Users\Administrator\AppData\Local\Programs\Python\Python36\python3
.exe C:/Users/Administrator/Desktop/process/process_resource.py
init A A A A B B B E C A D B E C A C
```

测试样例:3.txt 如下：

输入测试命令或将测试命令放在测试文件 3.txt 中，内容为：

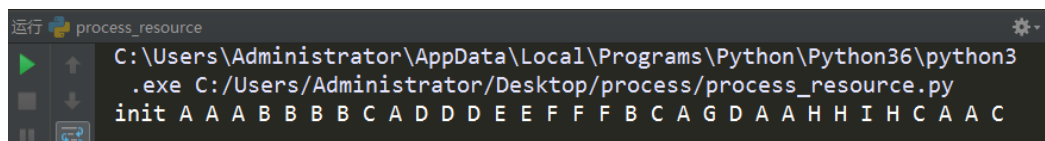
```
cr A 1
cr B 1
cr C 1
to
cr D 1
cr E 1
cr F 1
to
to
to
req R1 1
req R2 1
to
req R2 1
to
req R3 3
```

```
req R4 3
req R4 3
to
req R1 1
cr G 2
req R1 1
de B
req R3 2
cr H 2
cr I 2
to
req R3 3
req R3 2
to
rel R3 1
to
```

输出结果应为：

initAAABBBBCADDDEEFFFB CAGDAAHHIHCAAC

测试结果如下：



```
运行 process_resource
C:\Users\Administrator\AppData\Local\Programs\Python\Python36\python3
.exe C:/Users/Administrator/Desktop/process/process_resource.py
initAAABBBBCADDDEEFFFB CAGDAAHHIHCAAC
```

测试样例:4.txt 如下：

输入测试命令或将测试命令放在测试文件 4.txt 中，内容为：

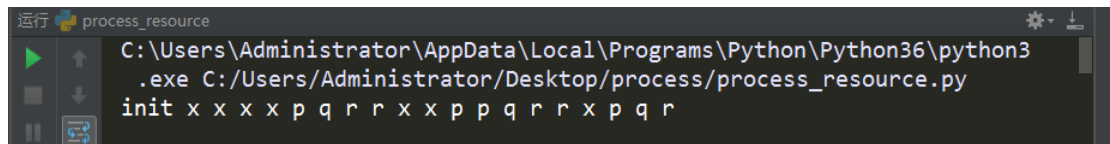
```
cr x 1
cr p 1
cr q 1
cr r 1
to
to
to
req R1 1
to
req R2 1
to
req R3 2
to
to
rel R1 1
to
```



```
req R3 3
de p
to
```

输出结果应为: `init x x x x p q r r x x p p q r r x p q r`

测试结果如下:



```
运行 process_resource
C:\Users\Administrator\AppData\Local\Programs\Python\Python36\python3
.exe C:/Users/Administrator/Desktop/process/process_resource.py
init x x x x p q r r x x p p q r r x p q r
```

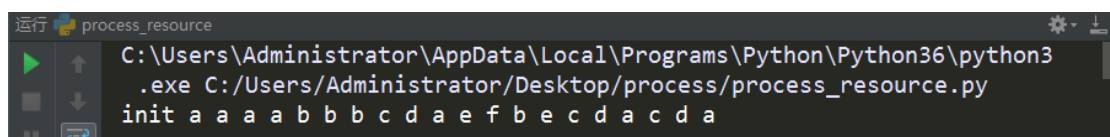
测试样例:5.txt 如下:

输入测试命令或将测试命令放在测试文件 5.txt 中, 内容为:

```
cr a 1
cr b 1
cr c 1
cr d 1
to
cr f 1
req R1 1
to
to
to
cr e 2
req R1 1
to
de b
req R1 1
to
to
to
to
to
```

输出结果应为: `init a a a a b b b c d a e f b e c d a c d a`

测试结果如下:



```
运行 process_resource
C:\Users\Administrator\AppData\Local\Programs\Python\Python36\python3
.exe C:/Users/Administrator/Desktop/process/process_resource.py
init a a a a b b b c d a e f b e c d a c d a
```

代码可以通过所有的测试样例, 实验成功!

## **十、总结及心得体会：**

这次实验花了比较长的时间去做，主要原因一开始写的不是很规范，导致大量的逻辑判断之后代码的可读性下降，因此后面在增加了一些测试样例后重新撰写了代码，将各个部分的功能拆分再合并。确定巩固了自己对于课堂上所学到的一系列的进程的调度，管理以及资源调度管理的相关知识点，本次使用 python 实现，因此很多时候调用了一些方便的 api 函数，优点在于可以将重点放在关于进程资源调度的问题上而不纠结于一些基础判断修改功能的实现。最终的代码成功模拟关于进程的创建，时间片轮转的调度以及进程的销毁，资源的释放等功能，在所有的测试样例中都测试成功，说明最终代码功能实现的还是非常完整的！自己在其中确实收获了很多的知识，进一步强化了自己的动手能力。

## **十一、对本实验过程及方法、手段的改进建议：**

可以考虑加入更多的功能，包括引入调度的其他算法以及资源分配的其他算法。

**报告评分：**

**指导教师签字：**

# 电子科技大学

## 实验报告

学生姓名：郭宇航 学号：2016100104014 指导教师：薛瑞尼

实验地点：主楼 A2-412

实验时间：2019.5.25

十二、实验室名称：计算机实验室

十三、实验项目名称：虚拟内存综合实验

十四、实验学时：4

十五、实验原理：

### 1. 逻辑地址到线性地址的转换

逻辑地址：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]。”

段标识符：也称为段选择符，段标识符是由一个 16 位长的字段组成，其中前 13 位是一个索引号。后面 3 位包含一些硬件细节：

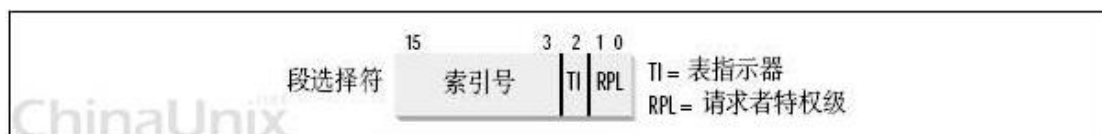


图 1 段选择符

索引号： 可以看作是段的编号，也可以看做是相关段描述符在段表中的索引位置。系统中的段表有两类：GDT 和 LDT。

GDT：全局段描述符表，整个系统一个，GDT 表中存放了共享段的描述符，以及 LDT 的描述符

（每个 LDT 本身被看作一个段）

LDT: 局部段描述符表, 每个进程一个, 进程内部的各个段的描述符, 就放在 LDT 中。

T1 字段: Intel 设计思想是: 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, T1=0, 表示相应的段描述符在 GDT 中, T1=1 表示表示相应的段描述符在 LDT 中。

段描述符: 具体描述了一个段。在段表中, 存放了很多段描述符。我们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, 也就是说, 段标识符的前 13 位是相关段描述符在段表中的索引位置。

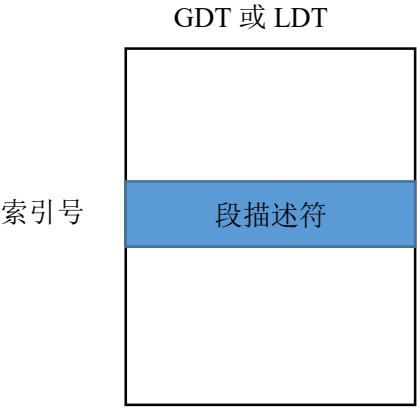


图 2 GDT 或 LDT 示例

每一个段描述符由 8 个字节组成, 如图 3:

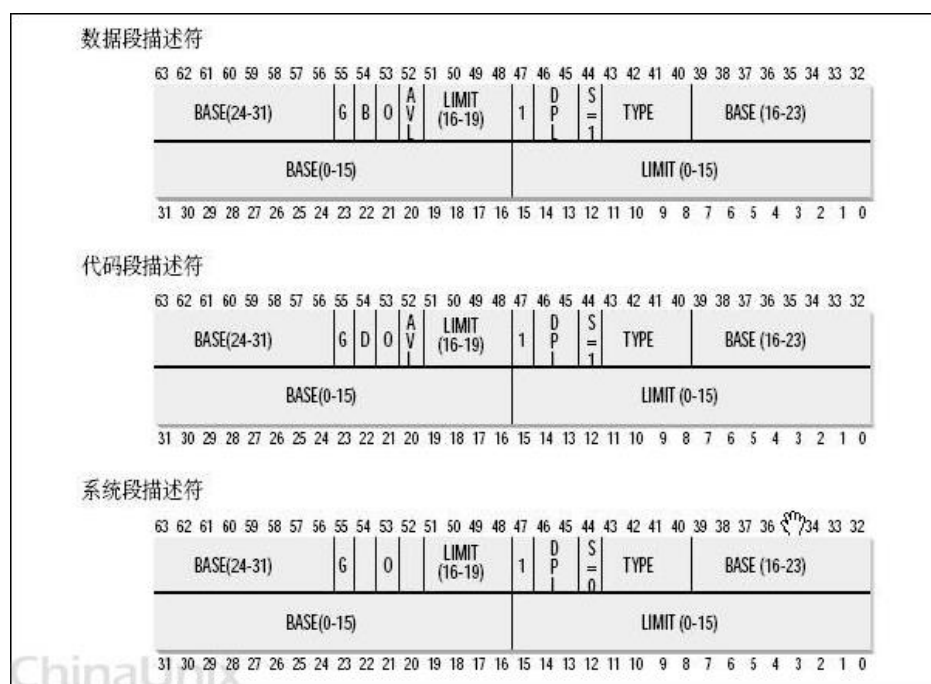


图 3 段描述符

Base 字段：它描述了一个段的开始位置：段基址。

相关寄存器：

GDTR：存放 GDT 在内存中的起始地址和大小

LDTR：分两种情况：

(1) 当段选择符中的 T1=1 时，表示段描述符存放在 LDT 中，如何找到 LDT 呢，LDT 本身也被看作一个段，LDT 的起始地址存放在 GDT 中，此时 LDTR 存放的就是 LDT 在 GDT 中的索引。这也是本实验关注的情况。

(2) 当段选择符中的 T1=0 时，表示段描述符存放在 GDT 中，通过 GDTR 找到 GDT，此时 LDTR 存放的是 LDT 的起始地址，当 T1=0 时，不涉及对 LDT 和 LDTR 的使用。

段选择符：如在 DS，SS 等寄存器内存储，取高 13 位作为在相应段表（如上例中的 DS 的高 13 位为对应段在 LDT）中的索引。

线性地址： 段标识符用来标明一个段的编号，具体的，我们需要通过段的编号，查找段表，来获得这个段的起始地址，即段基址。段基址+段内偏移量，就得到线性地址。

从逻辑地址到线性地址的转换过程,如图 4 所示(以 T1=1 为例,此时从段选择符中分离出段描述符和 T1 字段, T1=1, 表明段描述符存放在 LDT 中);

(1) 从 GDTR 中获得 GDT 的地址,从 LDTR 中获得 LDT 在 GDT 中的偏移量,查找 GDT,从中获取 LDT 的起始地址;

(2) 从 DS 中的高 13 位获取 DS 段在 LDT 中索引位置,查找 LDT,获取 DS 段的段描述符,从而获取 DS 段的基址;

(3) 根据 DS 段的基址+段内偏移量,获取所需单元的线性地址。

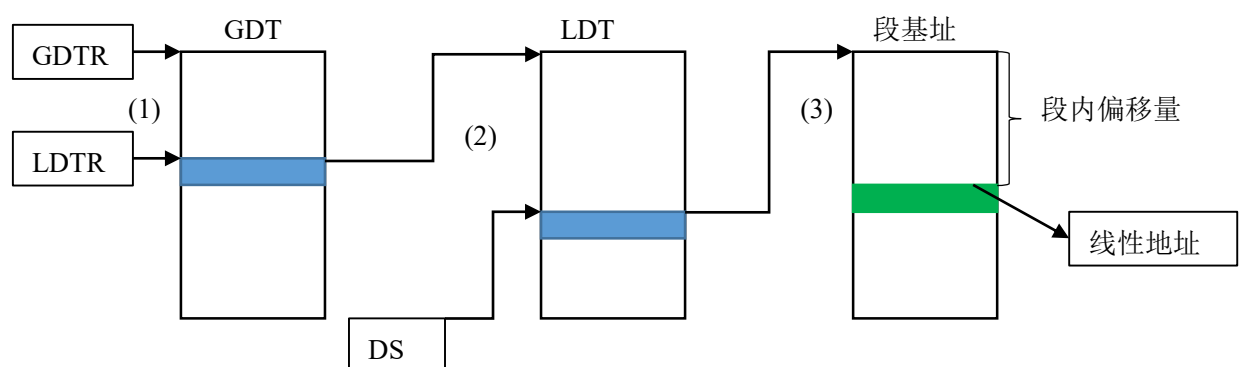


图 4 逻辑地址到线性地址的转换

## 2. 线性地址到物理地址的转换

物理地址：分段是面向用户，而分页则是面向系统，以提高内存的利用率，简言之，内存空间是按照分页来管理的。一个 32 位的机器，支持的内存空间是 4G，在页面大小为 4KB 的情况下，如果采用二级分页管理方式，线性地址结构如图 5 所示。

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，因为页面大小为 4K)。最终，我们需要根据线性地址，来获得物理地址。

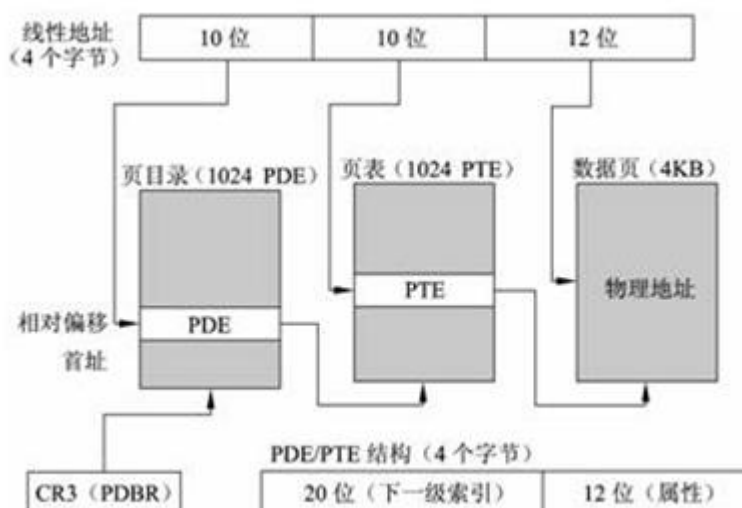


图 5 线性地址结构

将线性地址转换成物理地址的步骤：

- (1)、因为页目录表的地址放在 CPU 的 cr3 寄存器中，因此首先从 cr3 中取出进程的页目录表地址（操作系统负责在调度进程的时候，已经把这个地址装入对应寄存器）；
- (2)、根据线性地址前十位，在页目录表中，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，不是页的地址，而是一个页表的起始地址。
- (3)、查找页表，根据线性地址的中间十位，在页表中找到数据页的起始地址；
- (4)、将页的起始地址与页内偏移量（即线性地址中最后 12 位）相加，得到最终我们想要的物理地址。

## 十六、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。

## 十七、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。

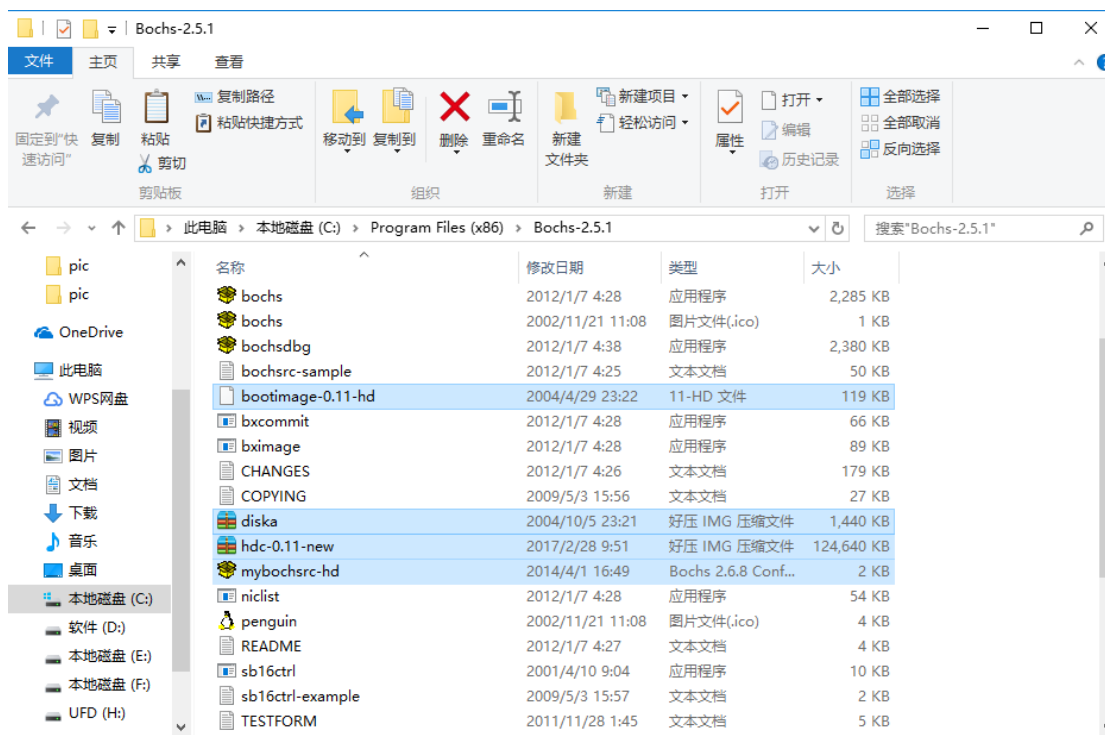
## 十八、实验器材（设备、元器件）：

Linux 内核（0.11 版）+ Boches 虚拟机

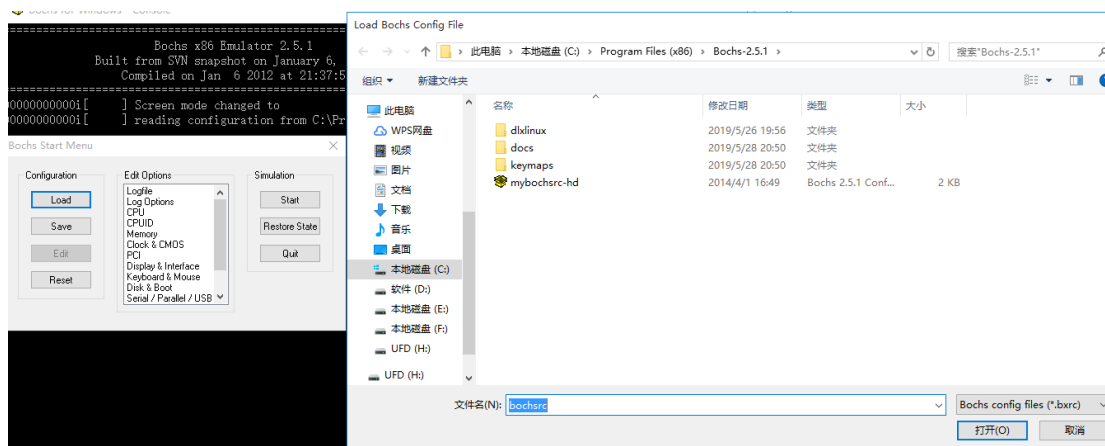
## 十九、实验步骤：

### 搭建实验环境：

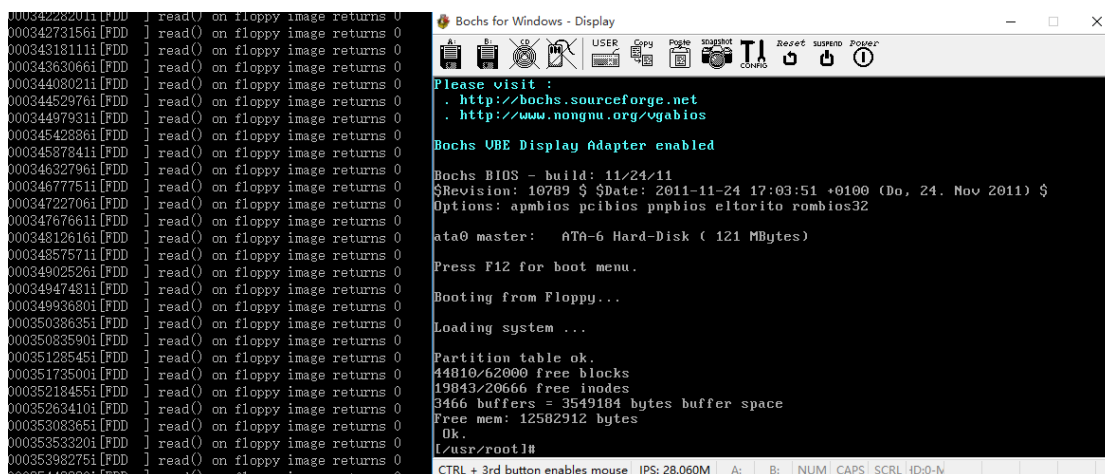
这里使用的是 boches2.5.1 的版本，安装过程如下：首先下载安装包成功安装 boches2.5.1，然后将 linux0.11 的内核文件拷贝入安装 boches2.5.1 的文件夹下：



拷贝之后打开 bochsdbg 可执行文件，加载 mybochsrc-hd 文件：



加载成功后再控制窗口输入 c 启动 boches 下虚拟的 Linux 运行窗口：



由此 boches2. 5. 1+Linux0. 11 的内核实验环境已经配置完成。



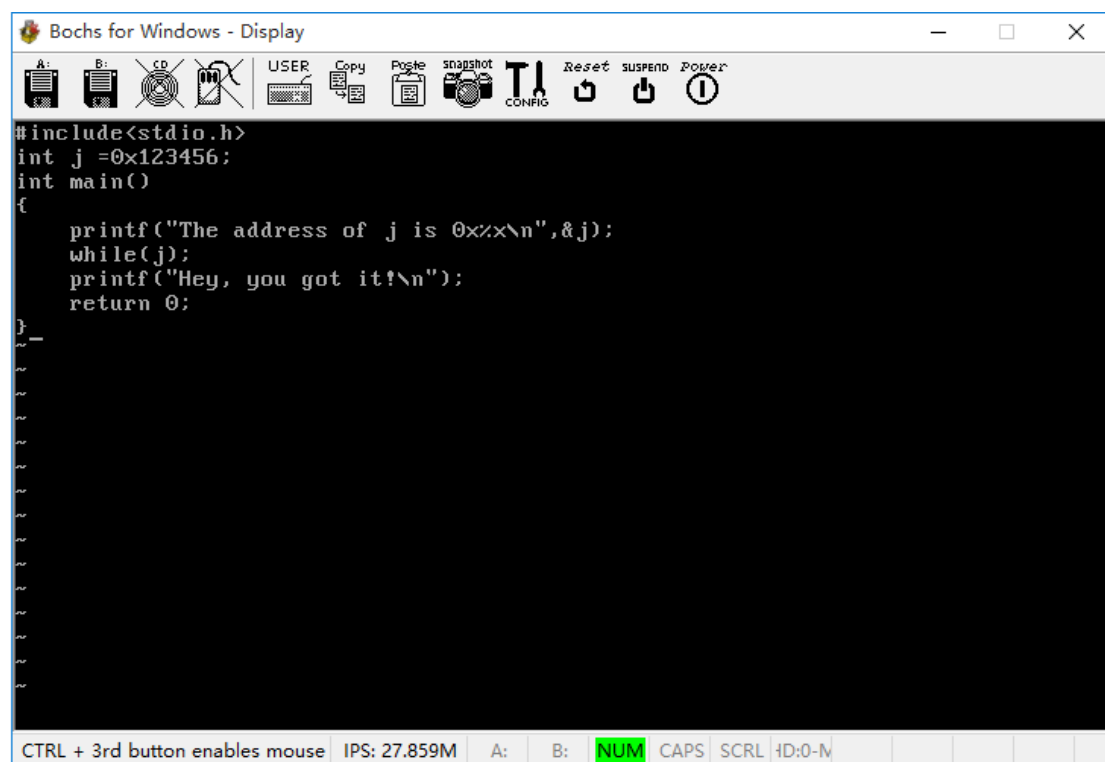
## 2. 实验主体部分:

首先启动 vi 编辑, 新建一个名为 experiment2.c 的文件, 输入以下 C 语言代码:

```
#include <stdio.h>

int j = 0x123456;

int main()
{
    printf("the address of j is 0x%x\n", &j);
    while(j);
    printf("Hey, you got it!\n");
    return 0;
}
```



输入成功后 esc 保存并退出:wq

在控制窗口按住 ctrl+c 启动断点调试模式.

由于逻辑地址是由一个段标识符+offset 组成, 而段标识符的存储位置在由段标识符中的 T1 字段决定, 首先我们需要寻找段标识符的信息, 因此在控制窗口中输入 sreg 查看寄存器的信息:

```
Bochs for Windows - Console
03298760539i[FDD ] read() on floppy image returns 0
03298805494i[FDD ] read() on floppy image returns 0
03298850449i[FDD ] read() on floppy image returns 0
03298895404i[FDD ] read() on floppy image returns 0
03298940359i[FDD ] read() on floppy image returns 0
03298985314i[FDD ] read() on floppy image returns 0
03299030269i[FDD ] read() on floppy image returns 0
03299075224i[FDD ] read() on floppy image returns 0
03299125670i[BIOS] int13_harddisk: function 15, unmapped device for ELDL=81
11827200000i[ ] Ctrl-C detected in signal handler.
Next at t=11827316466
(0) [0x0000000000faa060] 000f:0000000000000060 (unk. ctxt): jz .+2 (0x10000064) ; 7402
<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, 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=0x0000000000005cb8, limit=0x7ff
idtr:base=0x00000000000054b8, limit=0x7ff
<bochs:3>
```

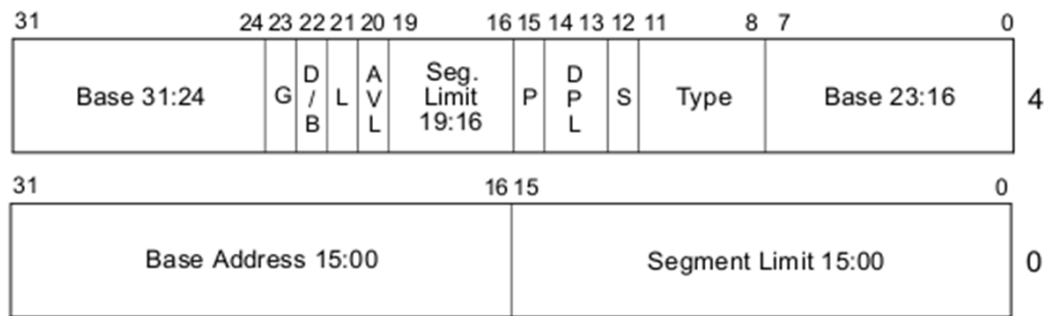
我们实验数据存储在 ds 数据寄存器中,而我们看到 ds 寄存器的段信息是:0x0017, 换算为二进制后我们发现对应的 T1 字段为 1, 索引号为 2, 因此我们知道此段标识符存储在 LDT 段表中的第三项。

由于我们知道 LDT 是存储在 GDT 中的, 需要通过 LDTR 进行索引, 因此再查看 LDTR 寄存器的段信息: 0x0068, 换算后可以发现 LDT 存储在 GDT 的索引为 13, 表示为 GDT 中的第 14 项。

由于 GDTR 的基地址为: 0x5cb8, 而 GDT 中存储的段的属性是一个 8 字节的数据结构, 我们使用 xp /2w 指令查看 LDT 在 GDT 中的索引信息: xp /2w 0x5cb8+13\*8:

```
Bochs for Windows - Console
03298895404i[FDD ] read() on floppy image returns 0
03298940359i[FDD ] read() on floppy image returns 0
03298985314i[FDD ] read() on floppy image returns 0
03299030269i[FDD ] read() on floppy image returns 0
03299075224i[FDD ] read() on floppy image returns 0
03299125670i[BIOS] int13_harddisk: function 15, unmapped device for ELDL=81
11827200000i[ ] Ctrl-C detected in signal handler.
Next at t=11827316466
(0) [0x0000000000faa060] 000f:0000000000000060 (unk. ctxt): jz .+2 (0x10000064) ; 7402
<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, 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=0x0000000000005cb8, limit=0x7ff
idtr:base=0x00000000000054b8, limit=0x7ff
<bochs:3> xp /2w 0x00005cb8 + 8*13
[bochs]:
0x0000000000005d20 <bogus+ 0>: 0x92d00068 0x000082fd
<bochs:4>
```

得到 LDT 的段描述符的信息: 0x92d00068 和 0x000082fd, 将其第 16-31, 0-7, 24-31 位进行拼接得到 LDT 的基地址为: 0x00fd92d0



根据 ds 段标识符信息, ds 段存储在 LDT 中的第 3 项, 使用 `xp /2w 0x00fd92d0+2*8` 查看 ds 寄存器的段描述符:

```
<bochs:4> xp /2w 0x00fd92d0 +8*2
[bochs]:
0x0000000000fd92e0 <bogus+      0>:   0x00003fff   0x10c0f300
<bochs:5> _
```

基于 ds 的寄存器的段描述符可以得到其基地址: 0x10000000

进而很容易计算出 j 的线性地址为: 0x10000000+0x3004, 换算为二进制可以得到其页目录号为 64, 页表号为 3, 页内偏移为 4

使用 `creg` 查看寄存器 CR3 的信息:

```
<bochs:7> creg
CR0=0x8000001b: PG cd nw ac wp ne ET TS em MP PE
CR2=page fault laddr=0x0000000010002f9c
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
<bochs:8> _
```

不难发现寄存器 CR3 的值为 0, 表明页目录表的起始地址为 0

计算一级页表: `xp /w 64*4`:

```
<bochs:8> xp /w 64*4
[bochs]:
0x0000000000000100 <bogus+      0>:   0x00fa3027
<bochs:9> _
```

得到地址为: 0x00fa3027, 则可以得到下一级页表索引为: 0x00fa3000

计算二级页表: `xp /w 0x00fa3000+3*4`:

```
<bochs:9> xp /w 0x00fa3000+3*4
[bochs]:
0x0000000000fa300c <bogus+      0>:   0x00fa2067
<bochs:10> _
```

得到地址 0x00fa2067, 则可以得到下一级页表的索引为: 0x00fa2000

计算三级页表: `xp /w 0x00fa2000+4`

```
<bochs:10> xp /w 0x00fa2000+1*4
[bochs]:
0x0000000000fa2004 <bogus+      0>:   0x00123456
<bochs:11> _
```

得到物理地址为：0x00123456，表示我们寻址成功，下面直接将此地址的值修改为 0，修改指令为：setpmem 0x00fa2004 4 0

```
[bochs]: 0x0000000000fa2004 <bogus+ 0>: 0x00123456
<bochs:11> setpmem 0x00fa2004 4 0
```

修改之后在控制窗口键入 `c` 重新运行 `experiment2.c` 代码，得到如下结果：

Bochs for Windows - Display

```

:wg
[/usr/root]# ls
a.out          experiment2.c  my_lab.c
[/usr/root]# gcc experiment2.c
[/usr/root]# \a.out
The address of j is 0x3004

Hey, you got it!
[/usr/root]#
[/usr/root]#

```

CTRL + 3rd button enables mouse IPS: 28.000M A: B: NUM CAPS SCRL HD:0-M

系统提示“Hey, you got it!”成功跳出死循环，表明 j 的值修改成功，即物理寻址成功。

## 二十、总结及心得体会:

通过本次实验,我对操作系统内存中的地址转换原理和机制有了更加深刻的认识,学会了计算机的寻址过程、地址转换过程以及各类寄存器的使用方法,同时拓展了 **linux** 虚拟机下的相关操作知识。

二十一、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: