**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**

1. **实验室名称：计算机实验室**
2. **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**

1. **实验学时：4**
2. **实验原理：**
3. **进程的三种基本状态：**
4. 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。
5. 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。
6. 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。
7. **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)**

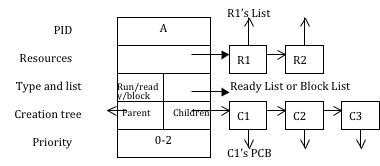


图2 PCB结构示意

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Ri有i个

资源控制块Resource control block (RCB) 如图5所示

* + RID: 资源的ID
  + Status: 空闲单元的数量
  + Waiting\_List: list of blocked process



图5 资源数据结构RCB

**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

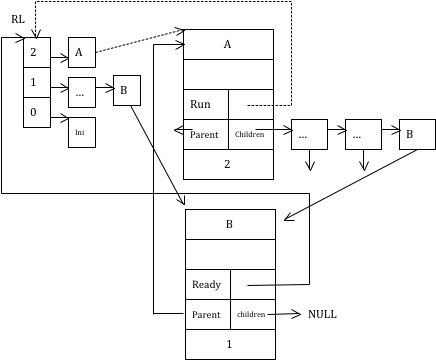


图4 进程数据结构间关系

(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

1. **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。

1. **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。

1. **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

1. **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 = []
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
16. self.R2 = RCB()
17. self.R2.Rid = 'R2'
18. self.R2.Initial = 2
19. self.R2.Remain = 2
21. self.R3 = RCB()
22. self.R3.Rid = 'R3'
23. self.R3.Initial = 3
24. self.R3.Remain = 3
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 = []
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].Pid
55. process\_sector.RL[0][0].type = 'running'
56. process\_sector.RL[0].pop(0)
57. **else**:
58. **pass**
59. **else**:
60. **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 RList1!')
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**
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 resource!')
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')
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)

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[0][0]
14. process\_sector.current\_process\_pid = process\_sector.RL[0][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\_sector.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\_r
18. process\_sector.current\_process.request\_resources\_num = request\_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 -= process\_sector.current\_process.request\_resources\_num
35. process\_sector.current\_process.resources.append('R2')
36. process\_sector.current\_process.resources\_num.append(process\_sector.current\_process.request\_resources\_num)
37. process\_sector.current\_process.request\_resources = None
38. process\_sector.current\_process.request\_resources\_num = -1
39. **else**:
40. #此时不能分配资源
41. process\_sector.current\_process.request\_resources = request\_r
42. process\_sector.current\_process.request\_resources\_num = request\_num
43. process\_sector.current\_process.type = 'blocked'
44. 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].Pid
48. process\_sector.RL[1].pop(0)
49. **elif** len(process\_sector.RL[0])==0 **and** len(process\_sector.RL[1])!=0:
50. process\_sector.current\_process = process\_sector.RL[1][0]
51. process\_sector.current\_process\_pid = process\_sector.RL[1][0].Pid
52. process\_sector.RL[1].pop(0)
53. **else**:
54. **pass**
55. **elif** process\_sector.current\_process.request\_resources == 'R3':
56. **if** process\_sector.current\_process.request\_resources\_num <= process\_sector.R3.Remain:
57. #此时可以给当前进程分配资源
58. process\_sector.R3.Remain -= process\_sector.current\_process.request\_resources\_num
59. process\_sector.current\_process.resources.append('R3')
60. process\_sector.current\_process.resources\_num.append(process\_sector.current\_process.request\_resources\_num)
61. process\_sector.current\_process.request\_resources = None
62. process\_sector.current\_process.request\_resources\_num = -1
63. **else**:
64. #此时不能分配资源
65. process\_sector.current\_process.request\_resources = request\_r
66. process\_sector.current\_process.request\_resources\_num = request\_num
67. process\_sector.current\_process.type = 'blocked'
68. process\_sector.resource\_list[2].append(process\_sector.current\_process)
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. **else**:
78. **pass**
79. **elif** process\_sector.current\_process.request\_resources == 'R4':
80. **if** process\_sector.current\_process.request\_resources\_num <= process\_sector.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\_sector.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\_num
91. process\_sector.current\_process.type = 'blocked'
92. process\_sector.resource\_list[3].append(process\_sector.current\_process)
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!检查输入文件是否错误！')
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
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**('\r')
46. **else**:
47. **print**('就绪队列0为空！\r')
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**('\r')
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',n.request\_resources,n.request\_resources\_num)
79. **print**('\r')
80. **else**:
81. **print**('资源3的阻塞队列为空！\r')
82. **if** len(process\_sector.R4.Waiting\_list)!=0:
83. **print**('在资源4的阻塞队列中的进程PID：', end=' ')
84. **for** q **in** process\_sector.R4.Waiting\_list:
85. **print**(q.Pid,end=' ')
86. **print**('\r')
87. **else**:
88. **print**('资源4的阻塞队列为空！\r')
89. **if** len(process\_sector.destroyed\_process)!=0:
90. **print**('已经被销毁的进程PID：',end=' ')
91. **for** s **in** process\_sector.destroyed\_process:
92. **print**(s.Pid,end=' ')
93. **print**('\r')
94. **else**:
95. **print**('销毁的进程队列为空！\r')
96. **print**('\r')
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.Remain,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. '''
104. **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

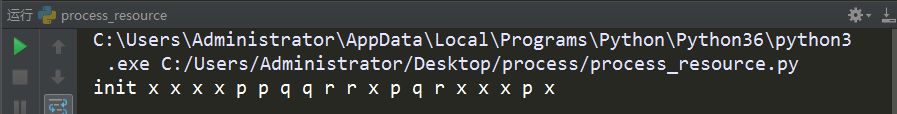
de q

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：



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

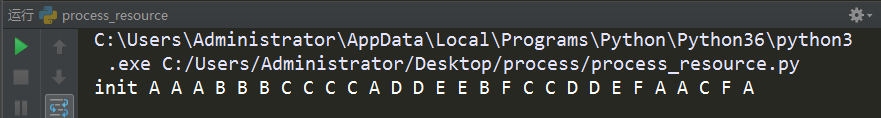
to

to

to

输出结果应为：init 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

测试结果如下：



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

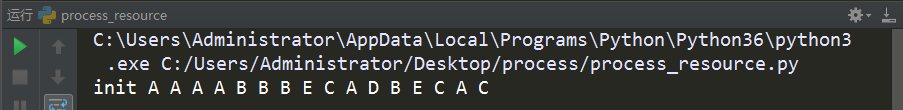
de B

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：



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

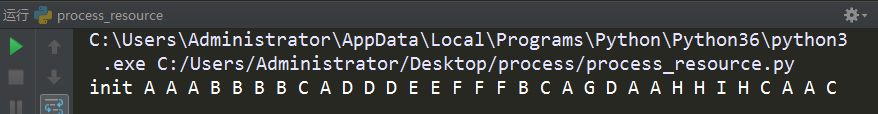
rel R3 1

to

输出结果应为：

init A A A B B B B C A D D D E E F F F B C A G D A A H H I H C A A C

测试结果如下：



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

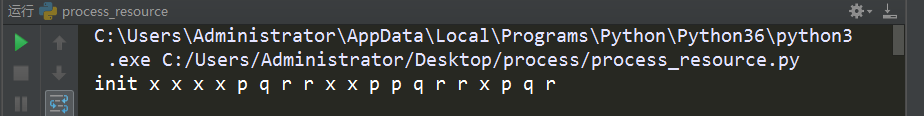
req R3 3

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：



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

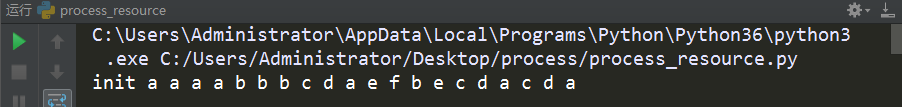
to

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：



代码可以通过所有的测试样例，实验成功！

1. **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。

1. **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**

1. **实验室名称：计算机实验室**
2. **实验项目名称：虚拟内存综合实验**
3. **实验学时：4**
4. **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。

段描述符

索引号

GDT或LDT

图2 GDT或LDT示例

每一个段描述符由8个字节组成，如图3：

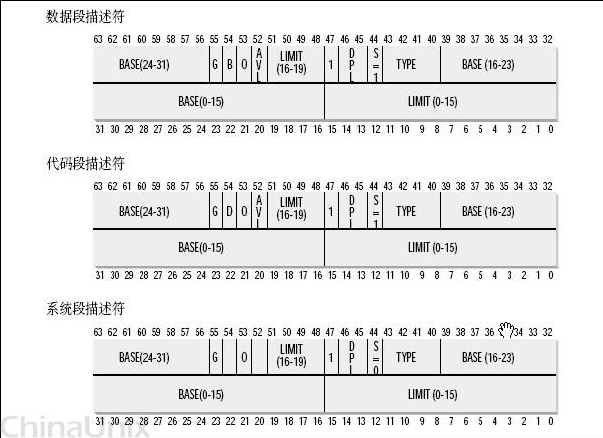


图3 段描述符

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。段基地址＋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段的基地址；

（3）根据DS段的基地址＋段内偏移量，获取所需单元的线性地址。

GDT

LDT

GDTR

LDTR

DS

段基址

段内偏移量

线性地址

(1)

(2)

(3)

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。

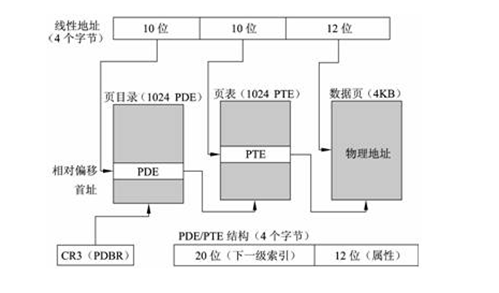


图5 线性地址结构

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。

1. **实验目的：**

通过实验，掌握段页式内存管理机制，理解地址转换的过程。

1. **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。**

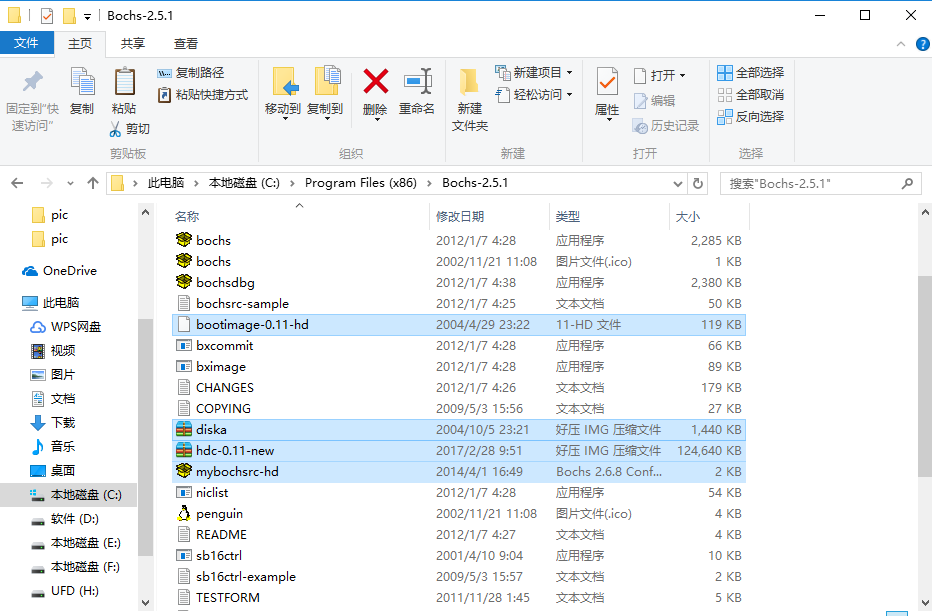
1. **实验器材（设备、元器件）：**

**Linux内核（0.11版）+ Boches虚拟机**

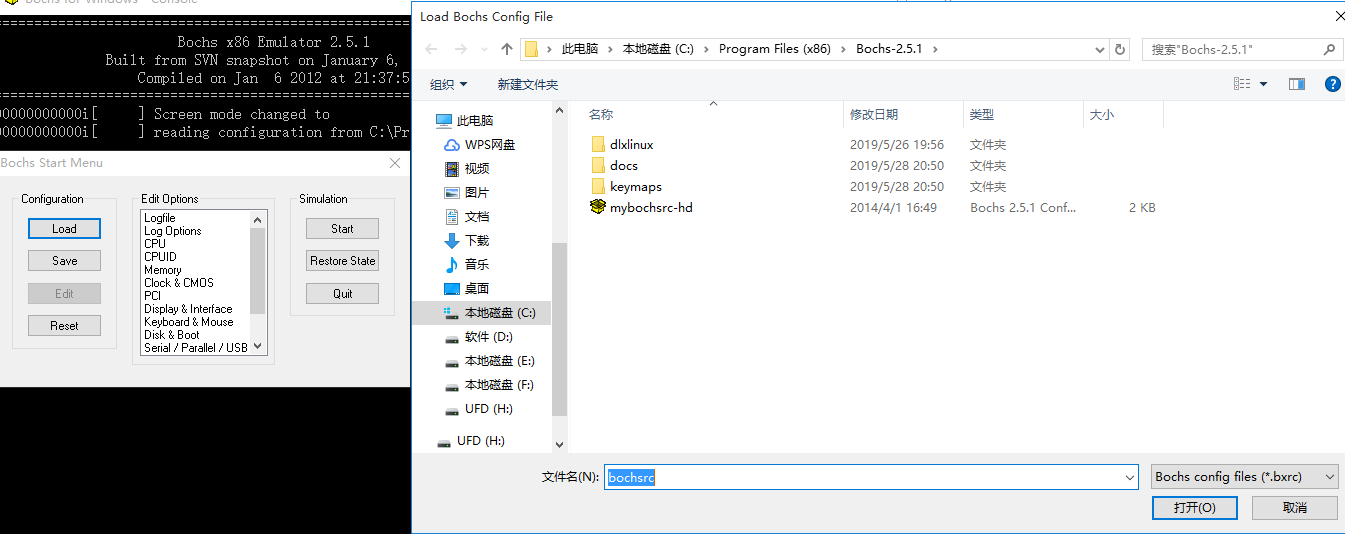
1. **实验步骤：**

**搭建实验环境：**

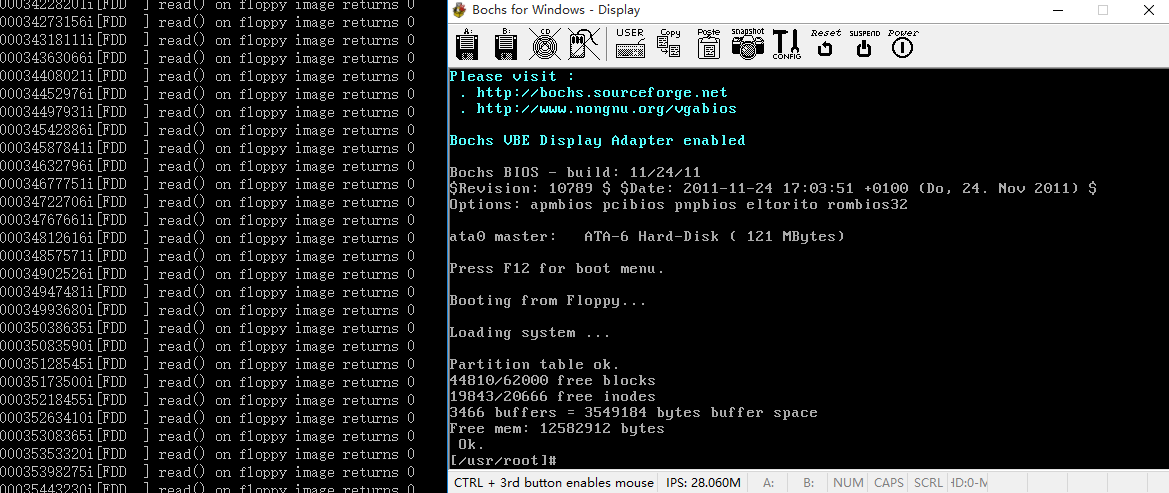
这里使用的是boches2.5.1的版本，安装过程如下：首先下载安装包成功安装boches2.5.1，然后将linux0.11的内核文件拷贝入安装boches2.5.1的文件夹下：



拷贝之后打开bochsdbg可执行文件，加载mybochsrc-hd文件：

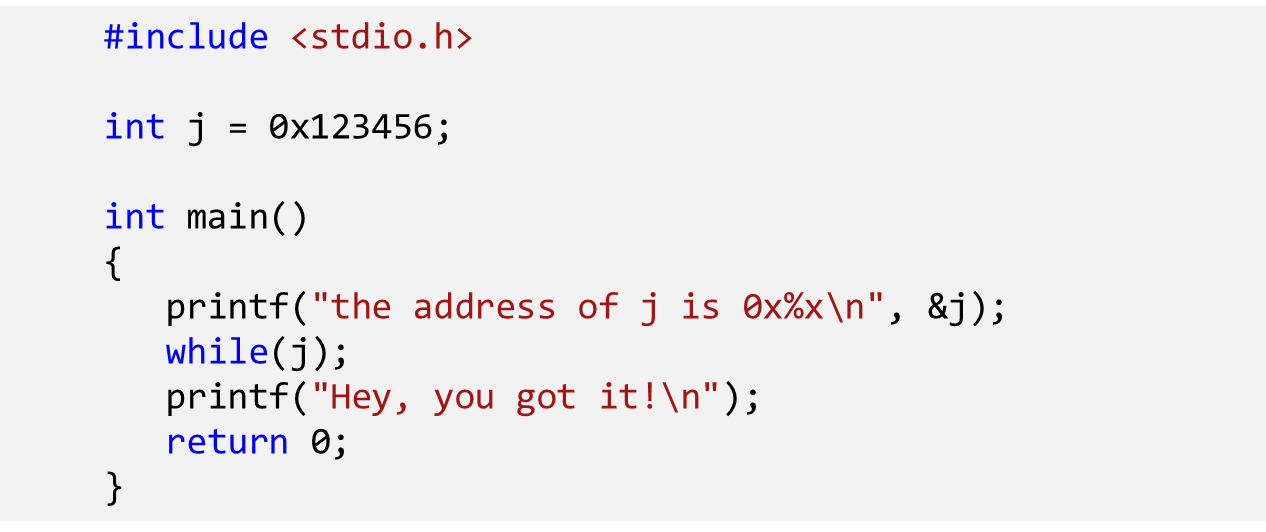


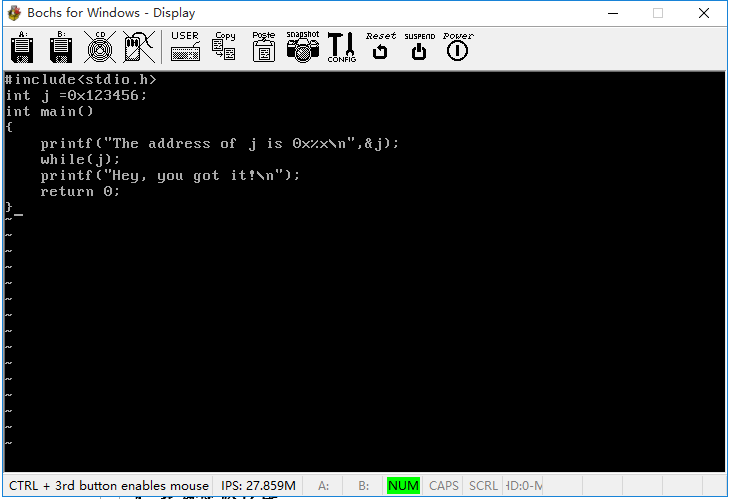
加载成功后再控制窗口输入c启动boches下虚拟的Linux运行窗口：



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：

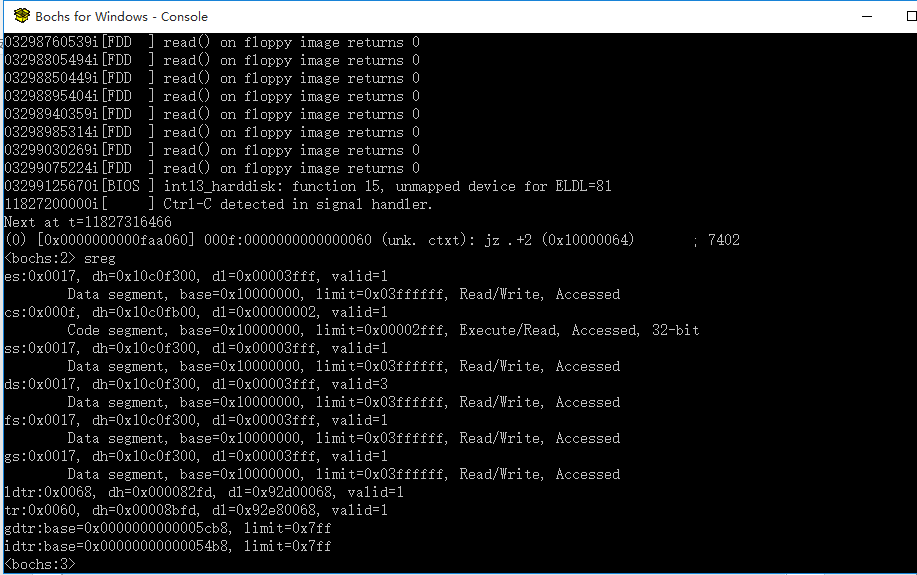
首先启动vi编辑，新建一个名为experiment2.c的文件，输入以下C语言代码：  




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.

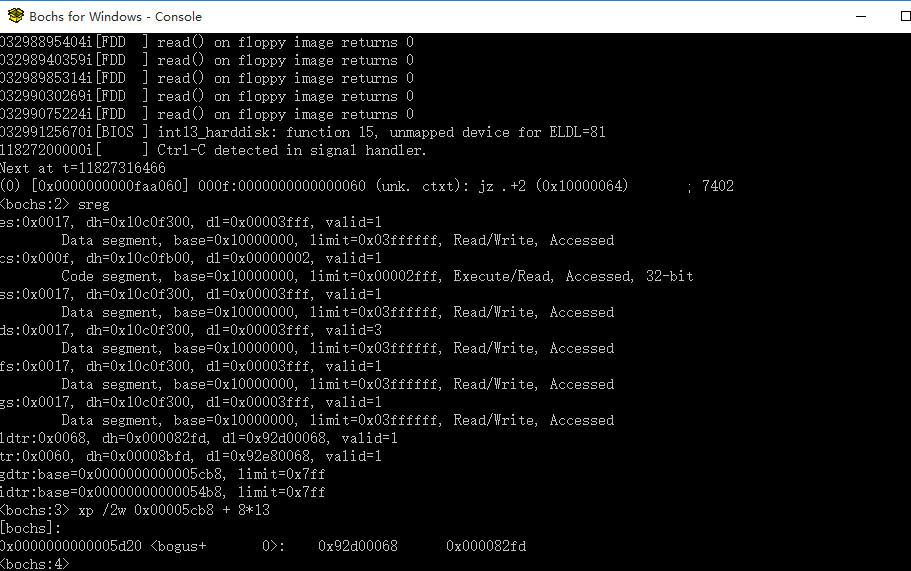
由于逻辑地址是由一个段标识符+offset组成，而段标识符的存储位置在由段标识符中的T1字段决定，首先我们需要寻找段标识符的信息，因此在控制窗口中输入sreg查看寄存器的信息：



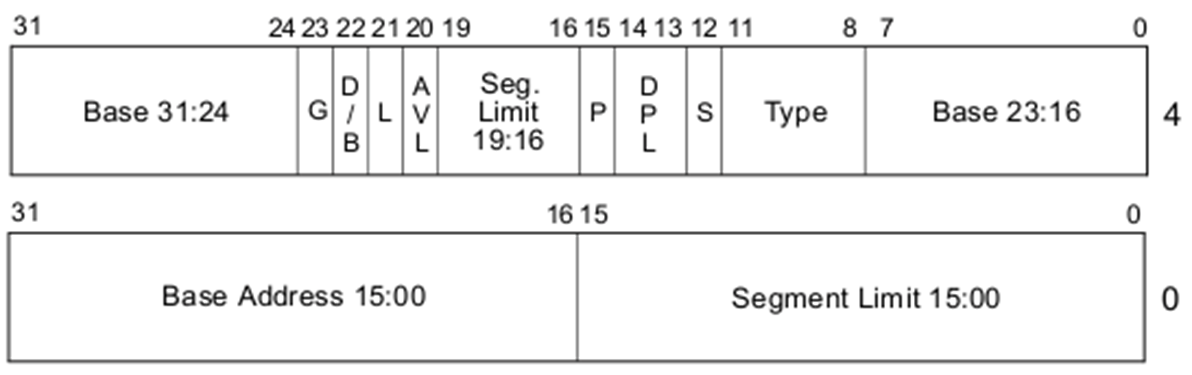
我们实验数据存储在ds数据寄存器中，而我们看到ds寄存器的段信息是：0x0017，换算为二进制后我们发现对应的T1字段为1，索引号为2，因此我们知道此段标识符存储在LDT段表中的第三项。

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。

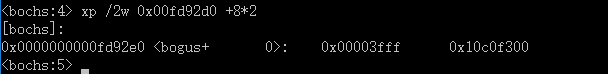
由于GDTR的基地址为：0x5cb8，而GDT中存储的段的属性是一个8字节的数据结构，我们使用xp /2w指令查看LDT在GDT中的索引信息：xp /2w 0x5cb8+13\*8：



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



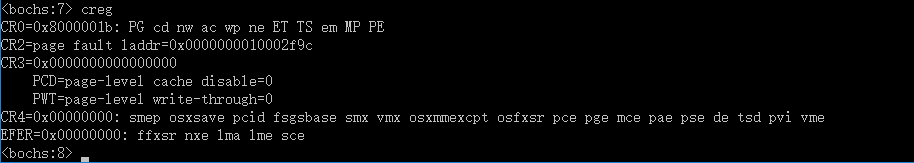
根据ds段标识符信息，ds段存储在LDT中的第3项，使用xp /2w 0x00fd92d0+2\*8查看ds寄存器的段描述符：



基于ds的寄存器的段描述符可以得到其基地址：0x10000000

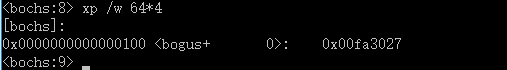
进而很容易计算出j的线性地址为：0x10000000+0x3004，换算为二进制可以得到其页目录号为64，页表号为3，页内偏移为4

使用creg查看寄存器CR3的信息：



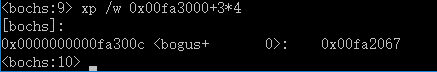
不难发现寄存器CR3的值为0，表明页目录表的起始地址为0

计算一级页表：xp /w 64\*4:



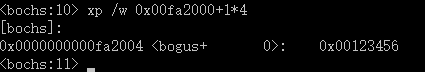
得到地址为：0x00fa3027，则可以得到下一级页表索引为：0x00fa3000

计算二级页表：xp /w 0x00fa3000+3\*4:



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

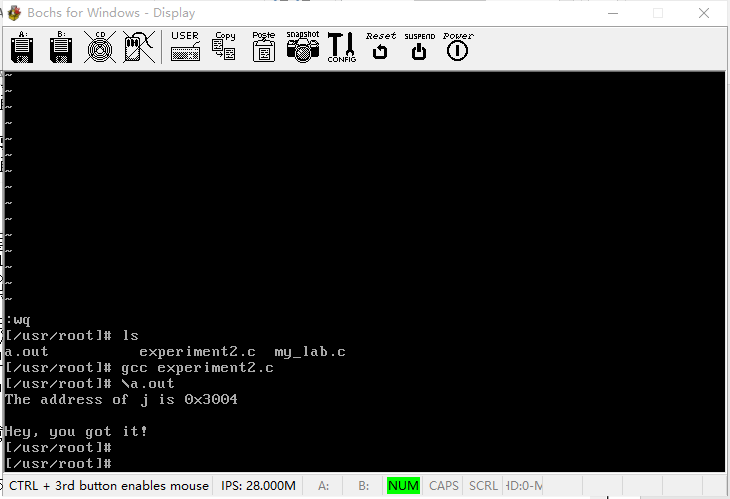
计算三级页表：xp /w 0x00fa2000+4



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



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：



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。

1. **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。

1. **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：**