

电子科技大学 计算机 学院

实 验 报 告

(实验) 课程名称 计算机操作系统

电子科技大学

实 验 报 告

学生姓名： 郭志猛 学 号： 2017080201005 指导教师： 刘杰彦

实验地点： 家中

实验时间： 2020 年 5 月

一、实验室名称： 计算机实验室

二、实验项目名称： 进程与资源管理器设计

三、实验学时： 6 学时

四、实验原理：

本次实验需要我们根据计算机操作系统进程管理和资源管理的基本原理和关键技术，完成总体设计、Test shell 设计、进程管理设计、资源管理设计等，下面我们依次介绍原理。

1. 总体设计

系统总体架构如图 1 所示，最右边部分为进程与资源管理器，属于操作系统内核的功能。该管理器具有如下功能：完成进程创建、撤销和进程调度；完成多单元 (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。

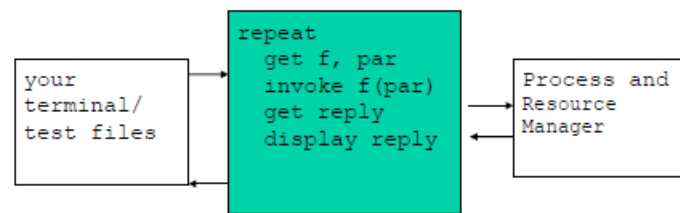


图1 系统总体结构

中间绿色部分为驱动程序 Test shell，该 Test shell 可以调度所设计的进程与资源管理器来完成测试。基本功能有：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。

最左端部分代表通过终端或者测试文件来给出相应的用户命令，以及模拟硬件引起的中断。

2. Test shell 设计

Test shell 可以完成读取命令、将命令转换为调用函数、输出结果等功能。具体为以下命令。

- -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
- -list ready // list all processes in the ready queue
- -list block // list all processes in the block queue

- -list res //list all available resources
- -pr <name> //print pcb information about a given process.

3. 进程管理设计

本实验中进程状态有 ready/running/blocked 三种。且我们需要针对进程定义各种操作，如下。

- 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

其中设计进程控制块数据结构的编写，如下。

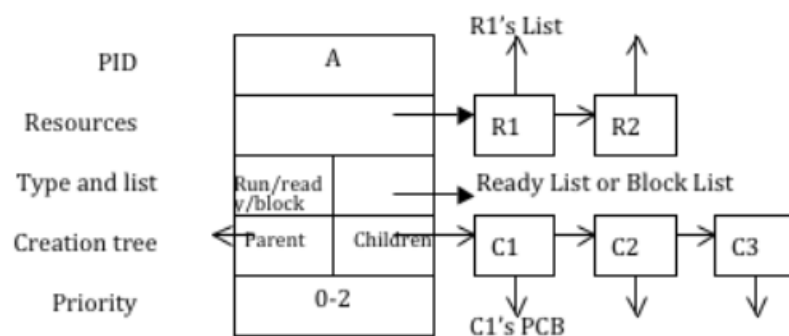


图 2 PCB 结构示意图

同时，还要完成进程队列的编写。就绪进程队列为 Ready list(TL),结构如下。

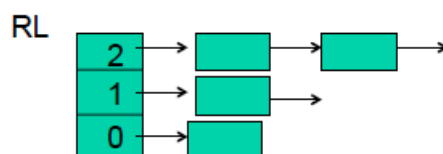


图 3 Ready list 数据结构

4. 资源管理设计

首先编写资源控制块来表示资源，设置固定的资源数量，4 类资源，， R1， R2， R3， R4， 简单定义每类资源 Ri 有 i 个资源控制块 Resource control block (RCB),每类资源一个。如图所示。

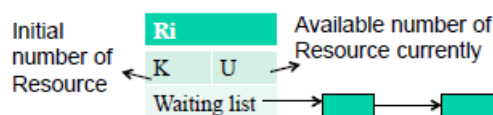


图 5 资源数据结构 RCB

然后编写函数实现对资源的请求和释放。

5. 进程调度与时钟中断设计

调度策略如下。

- 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。

时钟中断时模拟时间片结束或者外部硬件中断。

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 R_i 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，完成对时钟中断的处理，在同优先级进程中采用时间片轮转调度算法进行调度；能够完成资源的分配与释放，并完成进程之间的同步。该管理器同时也能完成从用户终端或者指定文件读取用户命令，通过 Test shell 模块完成对用户命令的解释，将用户命令转化为对进程与资源控制的具体操作，并将执行结果输出到终端或指定文件中。

七、实验器材（设备、元器件）：

个人主机，操作系统为 Windows 10，语言为 Python，使用 Vscode 编写。

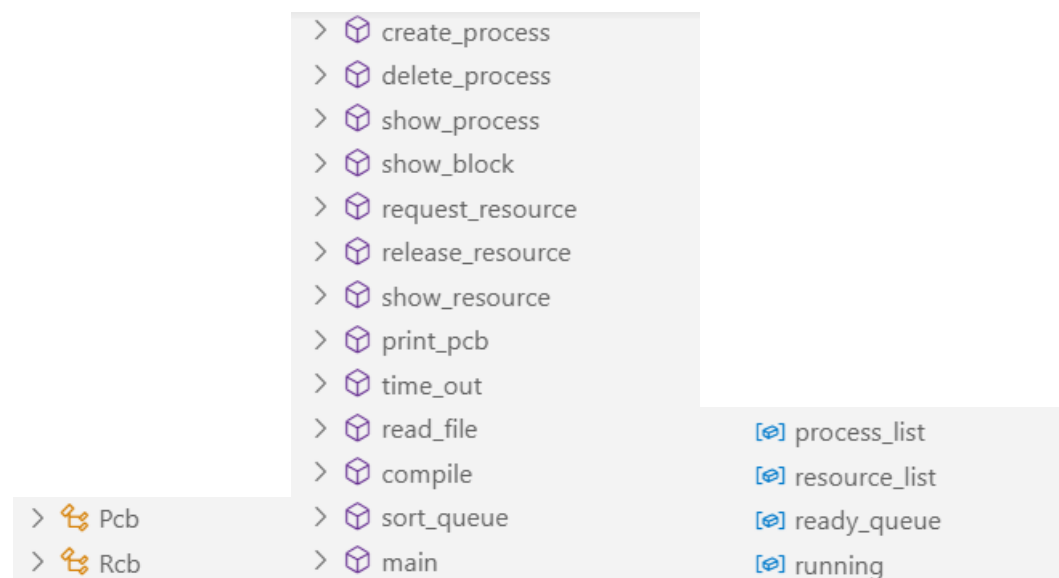
八、实验步骤：

1. 系统功能需求分析；

首先需要编写Test shell，将我们的输入命令转到对应的函数；然后编写进程控制块和资源控制块的结构，我们可以用Python类来编写；接着需要编写各项操作实现对进程和资源控制，如创建进程、撤销进程、请求资源、释放资源、时钟中断等函数，我们可以直接用Python函数实现。

2. 总体框架设计:

本次实验代码用Python编写，整体框架如下。



最左侧是定义的Pcb类和Rcb类，用来创建PCB和RCB的数据结构。

最右侧定义了4个全局变量，分别代表所有进程的列表，资源的列表，就绪队列，正在运行的进程。这4个全局变量便于各函数编写时输入数据和修改数据。

中间一列是编写的13个函数，main是主函数入口，这里完成了最开始的初始化，并且定义了循环用于不断输入指令；输入指令后进行判断，如果从文件读取命令就转到read_file，如果从键盘读入就运行compile；从compile识别出指令类型后，转到具体的函数，创建进程对应create_process函数，删除进程对应delete_process函数，展示ready队列对应show_process函数，展示block队列对应show_block函数，请求资源对应request_resource函数，释放资源对应release_resource函数，展示资源对应show_resource函数，打印PCB对应print_pcb函数，时钟中断对应time_out函数。另外，在对队列调整后，可能需要重新调整排序，所以创建进程、删除进程、请求资源、释放资源、时钟中断后都会调用sort_queue函数。

3. 具体模块的设计

A. 全局变量和类的定义


```

2  # 进程列表
3  process_list = []
4  # 资源列表
5  resource_list = []
6  # 进程队列
7  ready_queue = []
8  # 正在运行的进程
9  running = ''
10
11 # 进程控制块
    You, 4 minutes ago | 1 author (You)
12 class Pcb:
13     def __init__(self, name = '', priority = 0):
14         self.name = name
15         self.CPU_state = False
16         self.Memory = False
17         self.Open_files = False
18         self.resources = {'R1':0, 'R2':0, 'R3':0, 'R4':0}
19         self.status = ''
20         self.list = []
21         self.parent = ''
22         self.children = []
23         self.priority = priority
24
25 # 资源控制块
    You, 2 days ago | 1 author (You)
26 class Rcb:
27     def __init__(self, name = '', status = 0):
28         self.name = name
29         self.status = status
30         self.list = []

```

此次实验中我定义了四个全局变量和两个类，其中 `process_list` 用来记录所有存在的进程的信息，便于我们在函数编写时查找进程信息。`resource_list` 存储了所有资源的信息，我们在打印阻塞队列时，可以直接利用 `resource_list` 遍历。`ready_queue` 存储了 `ready` 状态进程的队列。`running` 则具体指当前正在运行的进程。

B. 主要控制逻辑

```
363 def main():
364     global process_list
365     global resource_list
366     global ready_queue
367     global running
368     # 初始化资源和PCB
369     pcb = create_process('init', 0)
370     r1 = Rcb('R1', 1)
371     r2 = Rcb('R2', 2)
372     r3 = Rcb('R3', 3)
373     r4 = Rcb('R4', 4)
374     # 将进程和资源加入全局变量
375     resource_list.append(r1)
376     resource_list.append(r2)
377     resource_list.append(r3)
378     resource_list.append(r4)
379     # 读入命令
380     while(1):
381         print("shell>", end='')
382         command = []
383         command = input()
384         if command.split()[0] == 'read':
385             read_file()
386         else:
387             compile(command)
```

```

315 # 读取文件
316 def read_file():
317     with open('input.txt', 'r') as f:
318         for line in f:
319             compile(line)
320
321 # 对命令处理
322 def compile(command):
323     command = command.split()
324     if command[0] == 'read':
325         read_file()
326     elif command[0] == 'cr':
327         create_process(command[1], int(command[2]))
328     elif command[0] == 'de':
329         delete_process(command[1])
330     elif command[0] == 'list' and command[1] == 'ready':
331         show_process()
332     elif command[0] == 'list' and command[1] == 'block':
333         show_block()
334     elif command[0] == 'list' and command[1] == 'res':
335         show_resource()
336     elif command[0] == 'to':
337         time_out()
338     elif command[0] == 'req':
339         request_resource(command[1], int(command[2]))
340     elif command[0] == 'rel':
341         release_resource(command[1])
342     elif command[0] == 'pr':
343         print_pcb(command[1])

```

主要控制逻辑部分包括三个函数，main 函数，read_file 函数，compile 函数。

main 函数是程序的入口，它会完成创建 init 进程，初始化资源的工作，然后通过循环不断读入指令。对于读进来的指令，它进行判断，要么读取固定文件，就转到了 read_file 函数；要么直接对读进来的指令进行解析，调用 compile 函数。

compile 函数将输入的命令分割之后，根据内容转到不同的函数，并将参数传入。

read_file 函数读取文件每一行，并调用 compile 函数进行解析。

C. 进程的控制

```
32  # 创建进程          You, 2 days ago • basic function
33  def create_process(name, priority):
34      global process_list
35      global ready_queue
36      global running
37      # 名称是否重复
38      for pro in process_list:
39          if pro.name == name:
40              print("名字重复了")
41              return
42      # 进程初始化
43      process = Pcb(name, priority)
44      # 判断进程状态
45      if running == '':
46          process.parent = 'null'
47          if ready_queue == []:
48              running = process
49              process.status = 'running'
50              ready_queue.append(process)
51          else:
52              ready_queue.append(process)
53              process.status = 'ready'
54      else:
55          process.parent = running.name
56          running.children.append(name)
57          if running.priority >= process.priority:
58              ready_queue.append(process)
59              process.status = 'ready'
60          else:
61              ready_queue.append(process)
62              running.status = 'ready'
63              running = process
64              process.status = 'running'
65      # 将进程加入队列并妥善排列
66      process_list.append(process)
67      sort_queue()
68      # 输出
69      print('process ' + running.name + ' is running')
70
```

```

71 # 删除进程
72 def delete_process(name):
73     global running
74     global process_list
75     global resource_list
76     global ready_queue
77     # 停止运行
78     if running.name == name:
79         running = ''
80     # 返还资源
81     index = []
82     for pro in process_list:
83         if pro.name == name:
84             for resource in resource_list:
85                 if pro.resources[resource.name] != 0:
86                     index = resource.name
87                     resource.status += pro.resources[resource.name]
88                     pro.resources[resource.name] = 0
89     # 从队列中删除
90     for pro in ready_queue:
91         if pro.name == name:
92             ready_queue.remove(pro)
93     # 从各资源队列中删除
94     for pro in process_list:
95         if pro.name == name:
96             for resource in resource_list:
97                 if pro in resource.list:
98                     resource.list.remove(pro)
99     # 从列表中删除
100     for pro in process_list:
101         if pro.name == name:
102             process_list.remove(pro)
103     # 看看能不能唤醒某进程

104 process = ''
105 for resource in resource_list:
106     if resource.name == index:
107         if resource.list[0].resources[index] <= resource.status:
108             process = resource.list[0]
109             # 判断进程状态
110             if running == '':
111                 if ready_queue == []:
112                     running = process
113                     process.status = 'running'
114                     ready_queue.append(process)
115             else:
116                 ready_queue.append(process)
117                 process.status = 'ready'
118         else:
119             if running.priority >= process.priority:
120                 ready_queue.append(process)
121                 process.status = 'ready'
122             else:
123                 ready_queue.append(process)
124                 running.status = 'ready'
125                 running = process
126                 process.status = 'running'
127     # 对队列进行排序
128     sort_queue()
129     # 输出
130     if process == '':
131         print('release ' + index)
132     else:
133         print('release ' + index + '. wake up process ' + process.name)
134

```

```

291 # 模拟时钟中断
292 def time_out():
293     global ready_queue
294     global running
295     # 运行中的加入ready队列
296     for process in process_list:
297         if process.name == running.name:
298             ready_queue.append(process)
299             ready_name = process.name
300     # ready队列头进入running
301     if ready_queue[1].name == 'init':
302         running = ready_queue[2]
303     else:
304         running = ready_queue[1]
305     ready_queue.pop(0)
306     # 重新排序
307     sort_queue()
308     # 输出
309     if running.name == ready_name:
310         print('process ' + running.name + ' is running.')
311     else:
312         print('process ' + running.name + ' is running.', end = '')
313         print('process ' + ready_name + ' is ready.')
314

```

进程的控制部分包括3个函数，create_process函数，delete_process函数，time_out函数。

create_process函数调用后会创建进程，它首先判断名称是否与存在的进程的名称重复，然后初始化一个新的进程控制块，对于进程状态进行判断。对于当前是否有进程正在运行，ready是否为空，与当前运行进程优先级关系进行判断，判断之后对于进程状态合理安排，并对进程队列调整。然后对ready队列进行排序，并输出。

delete_process函数调用后先判断进程是否运行，然后返还进程的资源，并将其从ready队列、资源的block队列、进程列表依次删除，删除之后判断返还的资源是否能唤醒某进程，如果能够唤醒，对唤醒进程状态和各个队列进行调整。之后对队列排序，并输出。

time_out函数模拟了时钟中断，调用后，它将运行中的进程加入ready队列，将ready队列头放入running，并在队列重新排序后输出运行结果。

D. 资源的控制

```
174 # 请求资源
175 def request_resource(name, num):
176     global running
177     global resource_list
178     global ready_queue
179     # 报错
180     if running == '':
181         print("没有进程运行！")
182         return
183     for resource in resource_list:
184         if resource.name == name:
185             # 假如不够分配
186             if resource.status < num:
187                 running.status = 'blocked'
188                 block_name = running.name
189                 resource.list.append(running)
190                 running = ready_queue[1]
191                 ready_queue.pop(0)
192                 print('process ' + running.name + ' is running.', end = '')
193                 print('process ' + block_name + ' is blocked.')
194             # 假如足够分配
195             else:
196                 running.resources[name] += num
197                 resource.status -= num
198                 print('process ' + running.name + ' requests ' + str(num) + ' ' + name)
199     sort_queue()
200
```

```

201 # 释放资源
202 def release_resource(name):
203     global running
204     global ready_queue
205     global resource_list
206     # 返还资源
207     resource.status += running.resources[name]
208     running.resources[name] = 0
209     # 看看能不能唤醒某进程
210     process = ''
211     for resource in resource_list:
212         if resource.name == name:
213             if resource.list[0].resources[name] <= resource.status:
214                 process = resource.list[0]
215                 # 判断进程状态
216                 if running.priority >= process.priority:
217                     ready_queue.append(process)
218                     process.status = 'ready'
219                 else:
220                     ready_queue.append(process)
221                     running.status = 'ready'
222                     running = process
223                     process.status = 'running'
224     # 对队列进行排序
225     sort_queue()
226     # 输出
227     if process == '':
228         print('release ' + name)
229     else:
230         print('release ' + name + '. wake up process ' + process.name)
231

```

资源的控制包括2个函数，request_resource函数和release_resource函数。

request_resource函数首先对当前运行进程进行判断，如果没有进程运行则报错。然后判断请求资源数是否能被满足，如果不能满足，则阻塞进程，并对队列进行调整；如果可以满足，就分配资源，依次调整。最后排序。

release_resource函数调用后先返还资源，然后判断返还的资源是否能唤醒某进程。如果能够唤醒，则为该进程和队列进行调整。最后对调整后的状态进行排序和输出。

E. 展示模块

```
135 # 展示ready队列
136 def show_process():
137     global ready_queue
138     # 输出ready队列
139     for i in reversed(range(3)):
140         flag = 0
141         print(str(i) + ': ', end = '')
142         for pro in ready_queue:
143             if pro.priority == i:
144                 if flag == 0:
145                     print(pro.name, end='')
146                 else:
147                     print('-' + pro.name, end='')
148             flag += 1
149         print()
150
160 # 展示block的进程
161 def show_block():
162     global resource_list
163     for resource in resource_list:
164         flag = 0
165         print(resource.name + ' ', end = '')
166         for pro in resource.list:
167             if flag == 0:
168                 print(pro.name, end='')
169             else:
170                 print('-' + pro.name, end = '')
171             flag += 1
172         print()
173
234 # 展示资源
235 def show_resource():
236     global resource_list
237     for resource in resource_list:
238         print(resource.name + ' ' + str(resource.status))
239
```

```

240 # 打印PCB
241 def print_pcb(name):
242     global process_list
243     global resource_list
244     # 判断是否存在
245     exist = 0
246     for pro in process_list:
247         if pro.name == name:
248             exist = 1
249     if exist == 0:
250         print("没有该进程！")
251         return
252     # 输出PCB
253     for pro in process_list:
254         if pro.name == name:
255             # 输出PID
256             print('PID: ' + name)
257             # 输出进程占用资源
258             print('Resources: ', end = '')
259             occu_resource = ''
260             for key, value in pro.resources.items():
261                 if value != 0:
262                     occu_resource = key
263                     print(key)
264             # 输出运行状态
265             print('Status: ' + pro.status)
266             # 输出对应队列
267             if pro.status == 'ready':
268                 print('Ready List:')
269                 show_process()
270             elif pro.status == 'blocked':
271                 print('Block List:')
272                 for resource in resource_list:
273                     if resource.name == occu_resource:
274                         flag = 0
275                         print(resource.name + ' ', end = '')
276                         for pro in resource.list:
277                             if flag == 0:
278                                 print(pro.name, end='')
279                             else:
280                                 print('-', end = '')
281                                 print(pro.name, end = '')
282                                 flag += 1
283                         print()
284             # 输出树形结构
285             print('Parent: ' + pro.parent)
286             print('Children: ', end='')
287             for child in pro.children:
288                 print(child + ' ', end = '')
289             print()
290             # 输出优先级
291             print('Priority: ' + str(pro.priority))

```

展示模块包含4个函数，show_process函数，show_block函数，show_resource函数，print_pcb函数。

show_process函数实际功能是输出ready_queue队列，它将ready_queue队列中的进程按优先级、按顺序排列，逐个输出。

show_block函数访问资源列表，对每一个资源的阻塞队列进行输出。

show_resource函数访问资源列表，将每一个资源数量打印输出。

print_pcb函数首先判断要打印的进程是否存在，如果存在，则将PID，进程占用资源，运行状态，所在队列，父进程，子进程，优先级依次打印输出。

F. 辅助模块

```
337 # 排列队列
338 def sort_queue():
339     global ready_queue
340     # 排列ready队列
341     tmp_1 = []
342     for pro in ready_queue:
343         if pro.priority == 2:
344             tmp_1.append(pro)
345     for pro in ready_queue:
346         if pro.priority == 1:
347             tmp_1.append(pro)
348     for pro in ready_queue:
349         if pro.priority == 0:
350             tmp_1.append(pro)
351     ready_queue = tmp_1
```

辅助模块只有sort_queue一个函数。

sort_queue访问就绪队列中每个进程，并将所有进程按优先级、进入顺序，从高到低、从早到晚依次排列。

sort_queue被create_process, delete_process, request_resource, release_resource, time_out五个函数调用, 在这五个函数调整队列后, 帮助对调整后的队列进行排序。

九、实验数据及结果分析:

测试流程:

1. 读取命令测试:

我们依次输入一些指令, 测试输出。我们测试的指令包括 cr 指令, list ready 指令, pr 指令, req 指令, rel 指令。结合后面的文件测试, 完成了所有指令的测试。

首先我们看直接读取命令的测试结果。


```
PS C:\Users\GZJZ0\OneDrive\2020.8\os_experiment> python shell.py
process init is running
shell>cr x 1
process x is running
shell>cr p 1
process x is running
shell>cr 1 1
process x is running
shell>cr r 1
process x is running
shell>list ready
2:
1:x-p-1-r
0:init
shell>pr x
PID: x
Resources: Status: running
Parent: init
Children: p 1 r
Priority: 1
shell>req R2 1
process x requests 1 R2
shell>rel R2 1
release R2
shell>
```

对于不同命令, 用不同的颜色方框进行标注。可以看到创建进程后, 我们进程的队列自动创建成功, 且能够输出进程的 PCB, PCB 中

正确输出了父进程、子进程、进程状态、优先级等信息。

2. 读取文件测试

测试文件：

 input.txt

Unsaved changes (cannot

```
1  init
2  cr x 1
3  cr p 1
4  cr q 1
5  cr r 1
6  list ready
7  to
8  req R2 1
9  to
10 req R3 3
11 to
12 req R4 3
13 list res
14 to
15 to
16 req R3 1
17 req R4 2
18 req R2 2
19 list block
20 to
21 de q
22 to
23 to
```

结果输出

```
PS C:\Users\GZJZ0\OneDrive\2020.8\os\os_experiment> python shell.py
process init is running
shell>read
process x is running
process x is running
process x is running
process x is running
2:
1:x-p-q-r
0:init
process p is running.process x is ready.
process p requests 1 R2
process q is running.process p is ready.
process q requests 3 R3
process r is running.process q is ready.
process r requests 3 R4
R1 1
R2 1
R3 0
R4 1
process x is running.process r is ready.
process p is running.process x is ready.
process q is running.process p is blocked.
process r is running.process q is blocked.
process x is running.process r is blocked.
R1
R2 r
R3 p
R4 q
process x is running.
release R3. wake up process p
process p is running.process x is ready.
process x is running.process p is ready.
shell>
```

经过比对，该结果与我们的预期结果一致，实验成功。下面我们对具体过程进行分析。

结果分析：

首先输入命令 `python shell.py` 调用脚本，此时自动完成了 `init` 进程创建和资源的分配，并开始读取指令。

我们输入指令 `read`，自动调用函数，将我们存储好的 `input.txt` 逐行读取，并运行指令。

`init` 命令已完成，所以这是重复的，程序不会运行。

读取 `cr x 1` 指令，创建进程 `x`，优先级 1，由于 `x` 优先级高于 `init`，所以直接运行 `x` 进程。

读取 `cr p 1` 指令，`cp q 1` 指令，`cr r 1` 指令，分别创建进程 `p`，`q`，`r`，且由于优先级都为 1，不高于 `x`，所以加入优先级为 1 的就绪队列。

输入 `list ready`，打印 `ready` 队列。注意我们为保证与指导书一致，将正在运行的 `x` 也放入了这里。（其实可以不放）

输入命令 `to`，进程 `x` 时间片用完，回到就绪队列，此时运行 `p`。

输入命令 `req R2 1`，为当前运行的 `p` 申请 1 个 `R2` 资源，可以正常申请。

输入命令 `to`，进程 `p` 时间片用完，回到就绪队列，此时运行 `q`。

输入命令 `req R3 3`，为当前运行的 `q` 申请 3 个 `R3` 资源，正常申请。

输入命令 `to`，进程 `q` 时间片用完，回到就绪队列，进程调度执行进程 `r`。

输入命令 `req R4 3`，为进程 `r` 申请 3 个 `R4` 资源，正常申请。

输入命令 `list res` 之后，打印输出各个资源剩余数量。

输入命令 `to` 之后，进程 `r` 时间片用完，回到就绪队列，此时运行进程 `x`。

输入命令 `to` 之后，进程 `x` 时间片用完，回到就绪队列，此时运行

进程 p。

输入命令 `req R3 1`，为进程 p 请求 1 个 R3 资源，此时 R3 资源为 0，不能满足申请，p 进程阻塞，唤醒 q 进程运行。

输入命令 `req R4 2`，为进程 q 请求 2 个 R4 资源，此时 R4 资源为 1，不能满足申请，q 进程阻塞，唤醒 r 进程运行。

输入命令 `req R2 2`，为进程 r 请求 2 个 R2 资源，此时 R2 资源为 1，不能满足申请，r 进程阻塞，唤醒 x 进程运行。

输入命令 `list block`，此时输出阻塞队列，发现 r 在 R2 上，p 在 R3 上，q 在 R4 上阻塞。

输入命令 `to` 之后，此时只有 x 进程能运行，所以继续运行进程 x。

输入命令 `de q` 之后，删除 q 进程，并释放其占有的 3 个 R3 资源，所以唤醒了进程 p，它之前申请 1 个 R3 资源。此时就绪或运行的用户进程只有 x 和 p。

输入命令 `to` 之后，进程 x 时间片用完，回到就绪队列，此时运行进程 p。

输入命令 `to` 之后，进程 p 时间片用完，回到就绪队列，此时运行进程 x。

十、实验结论：

通过本次实验，用到了课上学过的关于进程和资源管理的内容，设计并实现了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，进程撤销，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，完成对时钟中断的处理，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。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。

最后设计的进程与资源管理器通过了各条命令的输入测试，得到了预期的输出。

十一、总结及心得体会：

通过本次实验，完成了设计和实现进程与资源管理，并完成 Test shell 的编写的工作。过程中建立了系统的进程管理、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。这将在我未来的研究或者工作实践中发挥作用。

十二、对本实验过程及方法、手段的改进建议：

实验 1 指导书有几处错误，第一处：在图 6 的输入输出参考示例中，list ready 指令的输出，优先级为 1 时不应输出 x，因为 x 正在运行。本次实验我们将错就错，和这个保持一致。第二处：本次实验书提到进程优先级不发生变化，但是第 7 页 Test shell 输出示例 B 的优先级发生了变化从 2 到 1，如果假设时间片结束要变化的话，也是不合理的，因为后续测试文件与其矛盾。

报告评分：

指导教师签字：

电子科技大学 计算机 学院

实 验 报 告

(实验) 课程名称 计算机操作系统

电子科技大学

实验报告

学生姓名：郭志猛 学号：2017080201005 指导教师：刘杰彦

实验地点：家中 实验时间：2020 年 5 月

一、实验室名称：计算机实验室

二、实验项目名称：内存地址转换实验

三、实验学时：2 学时

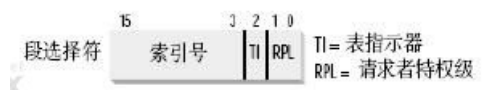
四、实验原理：

1. 逻辑地址到线性地址的转换

逻辑地址：

在 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]。

其中，段标识符也称为段选择符，属于逻辑地址的构成部分，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。如图。



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。

TI 字段:TI=0, 表示相应的段描述符在 GDT 中, TI=1 表示相应的段描述符在 LDT 中。

段表:

系统中的段表有两类: GDT 和 LDT。

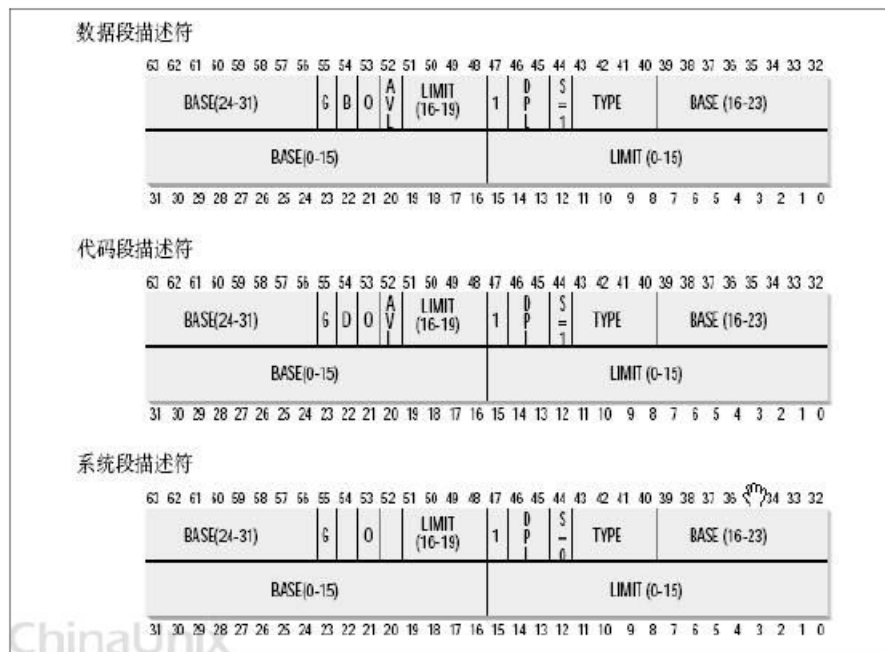
GDT:全局段描述符表,整个系统一个,GDT 表中存放了共享段的描述符,以及 LDT 的描述符(每个 LDT 本身被看作一个段)。

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。

段描述符:

段描述符(即段表项: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。





相关寄存器：

GDTR：存放 GDT 在内存中的起始地址和大小。

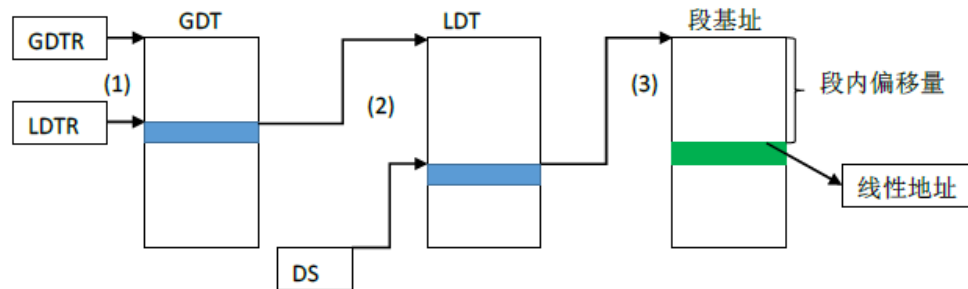
LDTR：当 TI=1，，LDT 的其实地址存放在 GDT 中，此时 LDTR 存放的就是 LDT 在 GDT 中的索引；当 TI=0，表示段描述符在 GDT 中，通过 GDTR 找到 GDT。

段选择符：如在 DS，SS 等寄存器中存储，取高 13 位作为在相应段表中的索引。

线性地址：

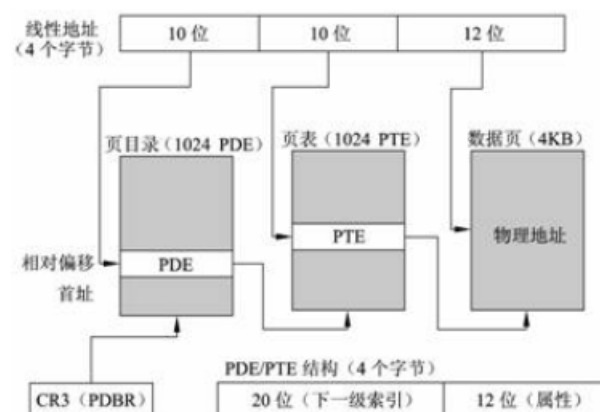
线性地址：段标识符用来标明一个段的编号,具体的,我们需要通过段的编号,查找段表,来获得这个段的起始地址,即段基址。如前所述,这里的段基 址,不是相应的段在内存中的起始地址,而是程序编译链接以后,这个段在逻辑地址空间里的起始位置。进一步的,段基地址段内偏移量,就得到线性地址(即 要访问的数据在整个程序逻辑(虚拟)地址空间中的位置)。

从逻辑地址到线性地址的转换过程,(以 TI=1 为例,此时段选择符 DS 中分离出段索引号(高 13 位)和 TI 字段,TI=1,表明段描述符存放在 LDT 中);



- (1) 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。

2. 线性地址到物理地址的转换



线性地址结构如图所示。

转换过程:

- (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(第一级页表)的起始地址(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(第一级页表)中,提到对应的索引项,因为引入了二级管理模式,线性地址的前十位,是第一级页表中的索引值,根据该索引,查找页目录表中对应的项,该项即保存了一个第二级页表的起始地址。

(3)、查找第二级页表,根据线性地址的中间十位,在该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(即线性地址最后 12 位)相加,得到最终我们想要的物理地址:

五、实验目的:

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

六、实验内容:

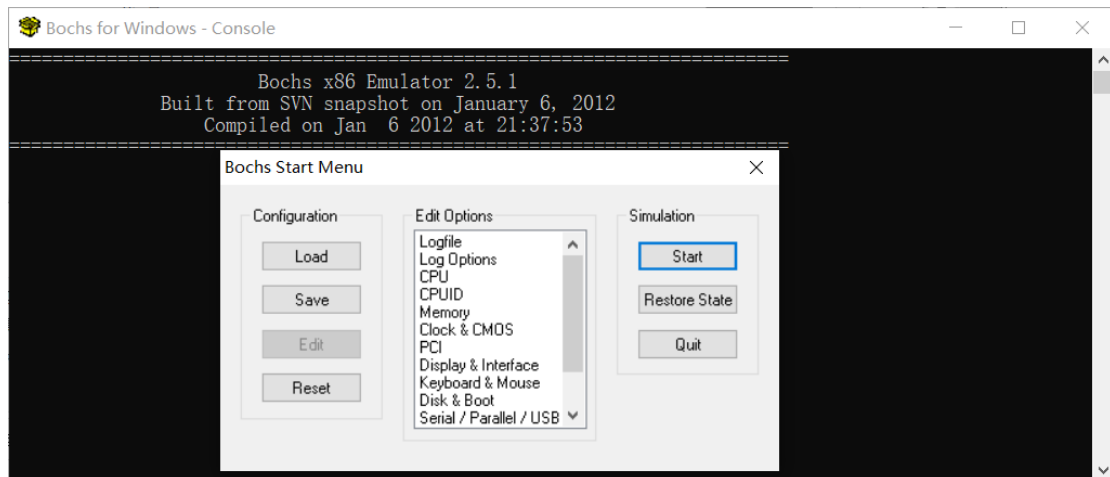
本实验运行了一个设置了全局变量的循环程序,通过查看段寄存器, LDT 表, GDT 表等信息,经过一系列段、页地址转换,找到程序中该全局变量的物理地址。

七、实验器材(设备、元器件):

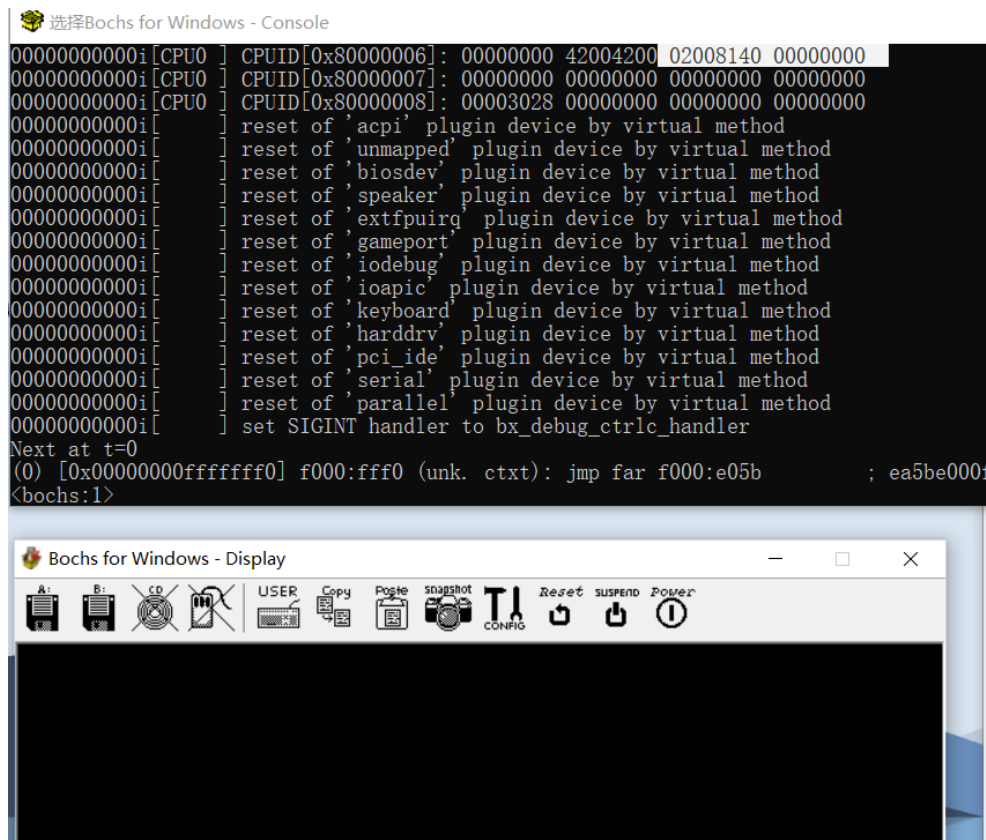
个人 PC, win10 操作系统, Linux 内核+Bochs 虚拟机。

八、实验步骤：

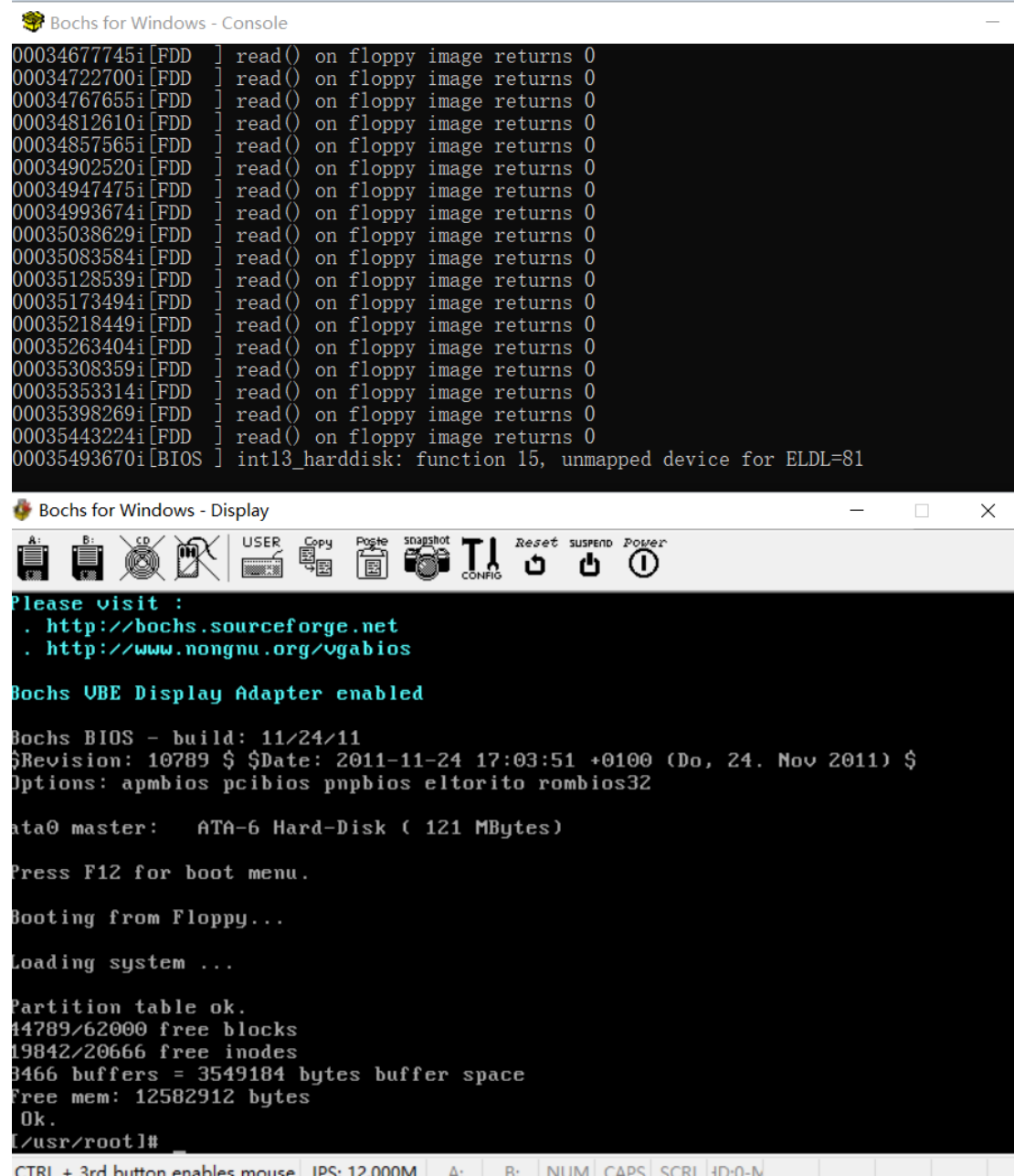
1. 安装 bochs。
2. 拷贝文件到安装目录。
3. 运行 bochsdbg.exe 程序。



4. 启动虚拟机。
5. 出现两个窗口。



6. 加载 Linux 操作系统。



The screenshot shows two windows from the Bochs emulator. The top window, titled "Bochs for Windows - Console", displays a series of log messages indicating that the BIOS is reading data from a floppy disk image. The messages are in the format: [address] [FDD] read() on floppy image returns 0. The addresses range from 00034677745i to 00035493670i. The bottom window, titled "Bochs for Windows - Display", shows the BIOS boot screen. It includes a toolbar with icons for A:, B:, CD, and other functions. The text on the screen reads: "Please visit : . http://bochs.sourceforge.net . http://www.nongnu.org/vgabios", "Bochs VBE Display Adapter enabled", "Bochs BIOS - build: 11/24/11", "\$Revision: 10789 \$ \$Date: 2011-11-24 17:03:51 +0100 (Do, 24. Nov 2011) \$", "Options: apmbios pcibios pnpbios eltorito rombios32", "ata0 master: ATA-6 Hard-Disk (121 MBytes)", "Press F12 for boot menu.", "Booting from Floppy...", "Loading system ...", "Partition table ok.", "44789/62000 free blocks", "19842/20666 free inodes", "3466 buffers = 3549184 bytes buffer space", "Free mem: 12582912 bytes", "Ok.", and the prompt "[usr/root]#".

```
00034677745i[FDD] read() on floppy image returns 0
00034722700i[FDD] read() on floppy image returns 0
00034767655i[FDD] read() on floppy image returns 0
00034812610i[FDD] read() on floppy image returns 0
00034857565i[FDD] read() on floppy image returns 0
00034902520i[FDD] read() on floppy image returns 0
00034947475i[FDD] read() on floppy image returns 0
00034993674i[FDD] read() on floppy image returns 0
00035038629i[FDD] read() on floppy image returns 0
00035083584i[FDD] read() on floppy image returns 0
00035128539i[FDD] read() on floppy image returns 0
00035173494i[FDD] read() on floppy image returns 0
00035218449i[FDD] read() on floppy image returns 0
00035263404i[FDD] read() on floppy image returns 0
00035308359i[FDD] read() on floppy image returns 0
00035353314i[FDD] read() on floppy image returns 0
00035398269i[FDD] read() on floppy image returns 0
00035443224i[FDD] read() on floppy image returns 0
00035493670i[BIOS] int13_harddisk: function 15, unmapped device for ELDL=81
```

Bochs for Windows - Display

Please visit :
 . <http://bochs.sourceforge.net>
 . <http://www.nongnu.org/vgabios>

Bochs VBE Display Adapter enabled

Bochs BIOS - build: 11/24/11
\$Revision: 10789 \$ \$Date: 2011-11-24 17:03:51 +0100 (Do, 24. Nov 2011) \$
Options: apmbios pcibios pnpbios eltorito rombios32

ata0 master: ATA-6 Hard-Disk (121 MBytes)

Press F12 for boot menu.

Booting from Floppy...

Loading system ...

Partition table ok.
44789/62000 free blocks
19842/20666 free inodes
3466 buffers = 3549184 bytes buffer space
Free mem: 12582912 bytes
Ok.
[usr/root]#

7. 生成 mytest 可执行文件。

```
Bochs for Windows - Display

A: B: CD USER Copy Paste Snapshot CONFIG Reset SUSPEND Power

#include <stdio.h>

int j=0x80201005;

int main()
{
    printf("the address of j is 0x%x\n",&j);
    while(j);
    printf("program terminated normally!\n");
    return 0;
}

~
~
~
```

```
[/usr/root]# gcc -o mytest mytest.c
[/usr/root]#
```

8. 执行可执行文件。

```
[/usr/root]# gcc -o mytest mytest.c
[/usr/root]# ./mytest
the address of j is 0x3004
```

9. 进入调试状态。

```
00035443224i[FDD ] read() on floppy image returns 0
00035493670i[BIOS ] int13_harddisk: function 15, unmapped device for ELDL=81
02401600000i[ ] Ctrl-C detected in signal handler.
Next at t=2401691372
(0) [0x0000000000faa06c] 000f:000000000000006c (unk. ctxt): jz .+2 (0x10000070) ; 7402
<bochs:2>
```

10. 输入 sreg 命令。

```
选择Bochs for Windows - Console

XMM[15]: 00000000_00000000_00000000_00000000
<bochs:4> 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
ldtr:base=0x00000000000054b8, limit=0x7ff
<bochs:5> xp /2w 0x5cb8+13*8
[bochs:]
```

读 ds 段信息，ds 段为 0x0017=0000 0000 0001 0111，取前 13 为代

表索引号，为 2。TI 位为 1，所以段描述符在 LDT 中，具体为 LDT 表的第 3 项。

11. 查看 LDTR 寄存器。

同步骤 10 图。

读 LDTR 寄存器信息，为 0x0068=0000 0000 0110 1000。其中，高 13 位段索引号，为 13，代表 LDT 其实地址在 GDT 表第 14 项。

12. 查看 GDT 中对应表项，得到 LDT 段描述符。

同步骤 10 图。

读 GDTR 寄存器信息，基址为 0x5cb8。然后再加上 LDT 的偏移量 13，通过基址加偏移我们可以查看 GDT 中对应表项，得到 LDT 段描述符。如下图。

```
(bochs:5> xp /2w 0x5cb8+13*8
[bochs]:
0x00000000000005d20 <bogus+      0>:  0x92d00068      0x000082fd
```

注意到右侧的为高位，左侧的为低位。拼接得到 LDT 的基址为 0x00fd92d0。

13. 查看 LDT 中第 2 项段描述符。

```
[bochs]:
0x0000000000fd92e0 <bogus+      0>:  0x00003fff      0x10c0f300
<bochs:8>
```

发现和 ds 寄存器的 dl、dh 中的数值完全相同。

14. 计算 ds 段基地址。

从上面的 ds 寄存器的值进行拼接，所以 ds 段的基址为 0x1000 0000。与 sreg 得到信息一致。

15. 计算线性地址 $0x10000000+0x3004=0x10003004$ 。

这个地址即为线性地址。我们将其按 10-10-12 划分，得到第一级页表内索引为 $0x40$ ，第二级页表内的索引为 $0x03$ ，页内偏移为 $0x04$ 。

16. 使用 `creg` 查看寄存器 `cr3` 值。

```
<bochs:9> creg
CR0=0x8000001b: PG cd nw ac wp ne ET TS em MP PE
CR2=page fault laddr=0x0000000010002fa8
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 ts
vme
EFER=0x00000000: ffxsr nxe lma lme sce
```

可以看到寄存器 `CR3` 的值为 0 ，即第一级页表起始地址为 0 。

17. 一级页表查看下一级索引。

因为第一级页表其实地址为 0 ，所以我们利用第一级页表的索引 $0x40$ 即可找到第二级页表。

```
<bochs:10> xp /2w 0x40*4
[bochs]:
0x0000000000000100 <bogus+ 0>: 0x00faa027 0x00000000
<bochs:11> 5488080000e[WGUI] enq_scanode: buffer full
```

这里高 20 位为页框号，所以下一级索引为 $0x00faa000$ 。

18. 二级页表查看下一级索引。

```
<bochs:12> xp /2w 0x00faa000+3*4
[bochs]:
0x000000000000faa00c <bogus+ 0>: 0x00fa7067 0x00000000
```

我们可以得到基址 $0x00fa7000$ 。

19. 根据索引找到值。

```
<bochs:13> xp /2w 0x00fa7000+4
[bochs]:
0x000000000000fa7004 <bogus+ 0>: 0x80201005 0x00003084
```

我们发现存储的值 $0x80201005$ 正是之前设置的学号后 8 位。成

功。

20. 设置值为 0.

```
<bochs:14> setpmem 0x00fa7004 4 0
<bochs:15> xp /2w 0x00fa7000+4
[bochs]:
0x0000000000fa7004 <bogus+ 0>: 0x00000000 0x00003084
```

设置之后重新查看该地址，发现值变为了 0. 修改成功。

21. 输入 c 继续运行，显示程序正常结束。

```
[/usr/root]# ./mytest
the address of j is 0x3004
program terminated normally!
```

输入 c 之后 linux 正常运行，返回值为 0，函数正常结束。

九、实验结论：

通过本次实验，我们使用了 Bochs 虚拟机与 Linux 内核，了解了计算机的寻址过程，并且完成了地址转换，其间还学到了计算机各种寄存器的用法。

十、总结及心得体会：

本次实验，我们设置了一个全局变量的循环程序，通过段寄存器、LDT 表、GDT 表等信息，手动地查找地址信息并进行地址转换，找到了全局变量的物理地址。

最后找到自己学号后 8 位，还是挺有成就感的。

十一、对本实验过程及方法、手段的改进建议：

这个实验不适合用这个实验报告模板写，直接依据步骤贴图并进行分析应该就好了。

报告评分：

指导教师签字：