

东莞理工学院

操作系统课程设计报告

院 系： 计算机学院

班 级： 14 软卓

姓 名： 赖键锋

学 号： 201441402130

指导老师： 李伟

日 期： 2016.6 - 2016.7

一、 相关说明

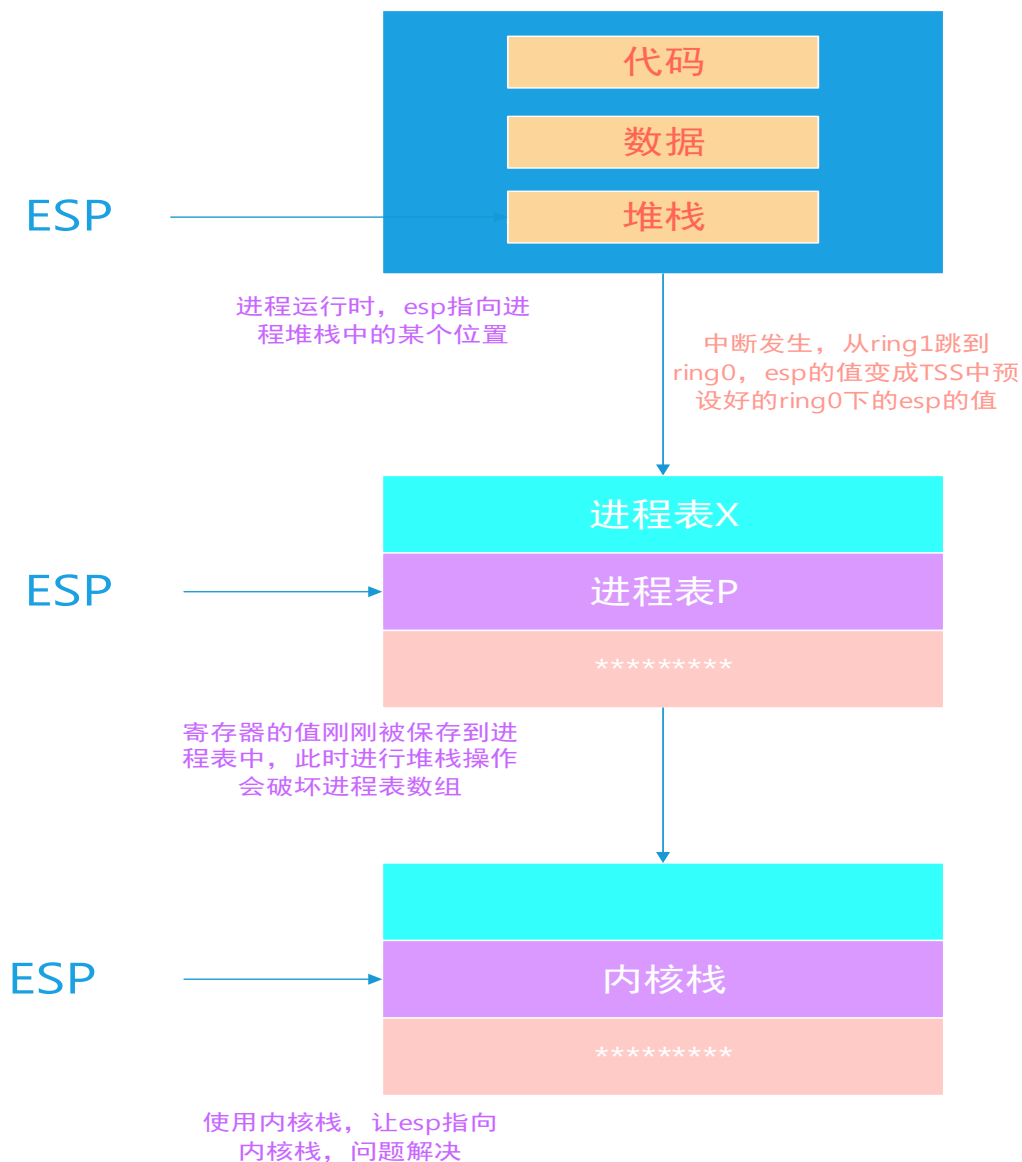


二、 相关知识的记录和说明

进程的概念

1. 系统中有多个进程，每个进程是为了实现某个功能或者完成某个目标，它们都有自己独立的代码、数据和堆栈段，CPU 通过进程调度让这些进程轮流执行，使计算机看起来是同时在运行多个进程（并发）。
2. 进程的实现需要有：时钟中断处理程序，进程调度模块，进程体。
3. 进程的四个要素：进程表，进程体，GDT，TSS。
4. 要保存的进程状态主要是寄存器的值，一条指令即可保存许多寄存器的值：pushad；进程表是存放这些寄存器值的地方，所有的进程表构成一个进程表数组，便于进行进程的管理。

5. Esp 的指向：进程栈，进程表，内核栈



6. 恢复进程，先 pop 恢复寄存器的值，然后执行 iretd 指令。

7. 用 iretd 启动第一个进程：

P_proc_ready: 指向进程表的指针；

S_stackframe: 存放进程的状态值的结构体；

S_proc: 进程表结构体，由 S_stackframe 和第一个 LDT 在 GDT 中的基地址，LDT 选择子，进程名称等信息组成；

iretd 实现从 ring0 到 ring1

```
354 ; =====
355 ; restart
356 ; =====
357 restart:
358     mov esp, [p_proc_ready]
359     lldt     [esp + P_LDT_SEL]
360     lea eax, [esp + P_STACKTOP]
361     mov dword [tss + TSS3_S_SP0], eax
362 restart_reenter:
363     dec dword [k_reenter]
364     pop gs
365     pop fs
366     pop es
367     pop ds
368     popad
369     add esp, 4
370     iretd
371
```

指向进程表

弹出进程的状态值

iretd进入进程

8. 第一个进程的启动:

1) 准备进程体

2) 初始化进程表: 寄存器, LDT 选择子和 LDT 的初始化 (把进程的 ds, es, fs, ss, cs, eip, eflags 等寄存器状态值填进进程表)

3) 准备 GDT, TSS

4) 最后一步, 让 esp 指向栈顶, 指令 iretd, 将各个值弹出, 完成 ring0 - ring1 跳转

9. 增加中断处理: (时钟中断处理)

1) 设置打开中断: 如 8259A 的时钟中断

2) 用 push 保护好进程执行状态, 即寄存器的值

3) 切换堆栈

```

155
156 ALIGN    16
157 hwint00:      ; Interrupt routine for irq 0 (the clock)
158     sub esp, 4
159     pushad    ; \
160     push     ds ; |
161     push     es ; | 保存原寄存器值
162     push     fs ; |
163     push     gs ; /
164     mov dx, ss
165     mov ds, dx
166     mov es, dx
167     inc byte [gs:0] ; 改变屏幕第 0 行, 第 0 列的字符
168     mov al, EOI      ; \ .reenable
169     out INT_M_CTL, al ; / master 8259
170     inc dword [k_reenter]
171     cmp dword [k_reenter], 0
172     jne .re_enter
173     mov esp, StackTop ; 切到内核栈
174
175     sti
176
177     push    clock_int_msg
178     call    disp_str
179     add esp, 4
180
181     cli
182
183     mov esp, [p_proc_ready] ; 离开内核栈
184     lea eax, [esp + P_STACKTOP]
185     mov dword [tss + TSS3_S_SP0], eax
186     .re_enter: ; 如果 (k_reenter != 0), 会跳转到这里
187     dec dword [k_reenter]
188     pop gs ; \
189     pop fs ; |
190     pop es ; | 恢复原寄存器值
191     pop ds ; |
192     popad  ; /
193     add esp, 4
194
195     iretd

```

令ds,es,ss具有相同的段,为什么?

控制中断的多重性,
防止无节制的中断嵌套

真正的 中断处理

ring0的 TSS.esp0

- 4) 开中断, 以支持多重中断
- 5) 执行中断处理、
- 6) 关中断
- 7) 设置 ring0 下的堆栈 esp0, 为下一次进程中断使用

10. 实现多个进程：处理好相应的变量和依赖关系

- 1) 添加一个进程体
- 2) 在 Task_table 中增加一个任务项 (global.h)
- 3) NR_TASKS 任务个数增一，进程表数组多一个表项给新进程用。
- 4) 为新进程定义任务堆栈 (proc.h)
- 5) 修改 STACK_SIZE_TOTLE，增一 (proc.h)
- 6) 添加新任务执行体的函数声明 (proto.h)

时钟中断发生时调用中断处理函数：

```
182     push    0
183     call    clock_handler
184     add esp, 4
185
```

最简单的：中断处理函数每次是 p_proc_ready 指向进程表的下一个，中断结束后恢复的进程就是下一个进程，这里有点进程调度的感觉。

```
20     PUBLIC void clock_handler(int irq)
21     {
22         disp_str("#");
23         p_proc_ready++;
24         if (p_proc_ready >= proc_table + NR_TASKS)
25             p_proc_ready = proc_table;
26     }
27
```

11. 高级一点的时间中断切换进程：

时间中断处理程序：

如果不是重入，则 save 函数中，切换到内核栈，将 restart 压栈；如果是重入，则将 restart_recenter 压栈。

```

157 ALIGN    16
158 hwint00:      ; Interrupt routine for irq 0 (the clock).
159     call    save
160
161     in     al, INT_M_CTLMASK    ; \.
162     or     al, 1                ; | 不允许再发生时钟中断
163     out    INT_M_CTLMASK, al    ; /
164
165     mov    al, EOI              ; \. reenabler
166     out    INT_M_CTL, al        ; / master 8259
167
168     sti
169     push    0
170     call    clock_handler
171     add     esp, 4
172     cli
173
174     in     al, INT_M_CTLMASK    ; \.
175     and     al, 0xFE            ; | 又允许时钟中断发生
176     out    INT_M_CTLMASK, al    ; /
177
178     ret
179
180 ALIGN    16
319 save:
320     pushad                ; \.
321     push     ds            ; |
322     push     es            ; | 保存原寄存器值
323     push     fs            ; |
324     push     gs            ; /
325     mov     dx, ss
326     mov     ds, dx
327     mov     es, dx
328
329     mov     eax, esp        ; eax = 进程表起始地址
330
331     inc     dword [k_reenter] ; k_reenter++;
332     cmp     dword [k_reenter], 0 ; if(k_reenter == 0)
333     jne     .1              ; {
334     mov     esp, StackTop    ; mov esp, StackTop <-- 切换到内核栈
335     push     restart         ; push restart
336     jmp     [eax + RETADR - P_STACKBASE]; return;
337 .1:
338     push     restart_reenter ; push restart_reenter
339     jmp     [eax + RETADR - P_STACKBASE]; return;

```

如果是非重入，则执行 restart，切换内核栈，保存当前进程的 LDT 和 TSS 的 esp0，（这里理解不够清楚）

```

345 restart:
346     mov esp, [p_proc_ready]
347     lldt     [esp + P_LDT_SEL]
348     lea eax, [esp + P_STACKTOP]
349     mov dword [tss + TSS3_S_SP0], eax
350 restart_reenter:
351     dec dword [k_reenter]
352     pop gs
353     pop fs
354     pop es
355     pop ds
356     popad
357     add esp, 4
358     iretd

```

12. 中断处理接口

Irq_table[int irq]: 中断函数指针数组，通过 irq 来调用第 irq 个中断。

Put_irq_handler(int irq, irq_handler handler):为对应的 irq 中断指定处理函数。

通过 Disable_irq 和 enable_irq 两个函数来控制 8259A 对中断的接收情况。

13. 【怎样设置 8259A 还不清楚】

14. 实现一个系统调用：get_ticks()：获取已经发生的中断次数

系统调用函数 sys_call 看成是一个中断，编号为 0x90，初始化系统调用描述符，

在 init_prot() 中，sys_call 的描述符跟其他的中断描述符一起初始化：

```

161
162     init_idt_desc(INT_VECTOR_SYS_CALL, DA_386IGate,
163                  sys_call, PRIVILEGE_USER);
164

```

这样只要 int sys_call，即是函数 sys_call (eax)，就会调用 sys_call_table[eax]，系统调用表中的第 eax 个系统调用；sys_call_table[eax] 指向一个处理函数；把 sys_call_table[0] 初始化为

sys_get_ticks(),就可以实现用 0 号系统调用来调用 sys_get_ticks()。

调用中断的过程:

- a) 用户调用系统调用的封装函数 sys_call
- b) 封装函数赋值 eax , 执行中断
- c) 中断处理程序在 sys_call_table 数组中根据 eax 找到最终要执行的程序, 并且执行之。

15. 在 syscall.asm 中函数 get_ticks() 中封装了 sys_get_ticks() 系统调用, 所以普通函数才能直接使用 get_ticks() 来调用 sys_call_ticks ()

```
18 get_ticks:
19     mov eax, _NR_get_ticks
20     int INT_VECTOR_SYS_CALL
21     ret
```

16. 8253/8254 PIT (Programmable Interval timer)



计数器 0 在 PC 上的频率是 1193180Hz, 在每个时钟周期 (CLK), 计数器会减 1, 当减到 0 时就会触发一个输出。而计数器是 16 位的最大值是 65535, 默认的时钟中断的发生频率是 $1193180 / 65535 = 18.2\text{Hz}$, 我们可以通过编程设置计数器开始的值来控制时钟中断发生频率。譬如: 如果我们要让时钟中断频率为 100Hz, 则设置 Counter0 计数器的开始值为 $1193180 / 100 = 11931$.

17. 比较精确的延迟函数

设置好时钟中断频率后，每 10ms 发生一次时钟中断。

Milli_delay (milli_sec) 会等 milli_sec ms 后才停止。

```
40 PUBLIC void milli_delay(int milli_sec)
41 {
42     int t = get_ticks();
43
44     while(((get_ticks() - t) * 1000 / HZ) < milli_sec) {}
45 }
46
```

18. 调度函数

```
19 PUBLIC void schedule()
20 {
21     PROCESS* p;
22     int greatest_ticks = 0;
23
24     while (!greatest_ticks) {
25         for (p = proc_table; p < proc_table+NR_TASKS; p++) {
26             if (p->ticks > greatest_ticks) {
27                 greatest_ticks = p->ticks;
28                 p_proc_ready = p;
29             }
30         }
31
32         if (!greatest_ticks) {
33             for (p = proc_table; p < proc_table+NR_TASKS; p++) {
34                 p->ticks = p->priority;
35             }
36         }
37     }
38 }
```

从所有进程中找出优先级最大的进程

所有优先级都运行完，则重置优先级，相当于唤醒吧

时钟中断处理函数

```
20 PUBLIC void clock_handler(int irq)
21 {
22     ticks++;
23     p_proc_ready->ticks--;
24
25     if (k_reenter != 0) {
26         return;
27     }
28
29     if (p_proc_ready->ticks > 0) {
30         return;
31     }
32
33     schedule();
34 }
35
```

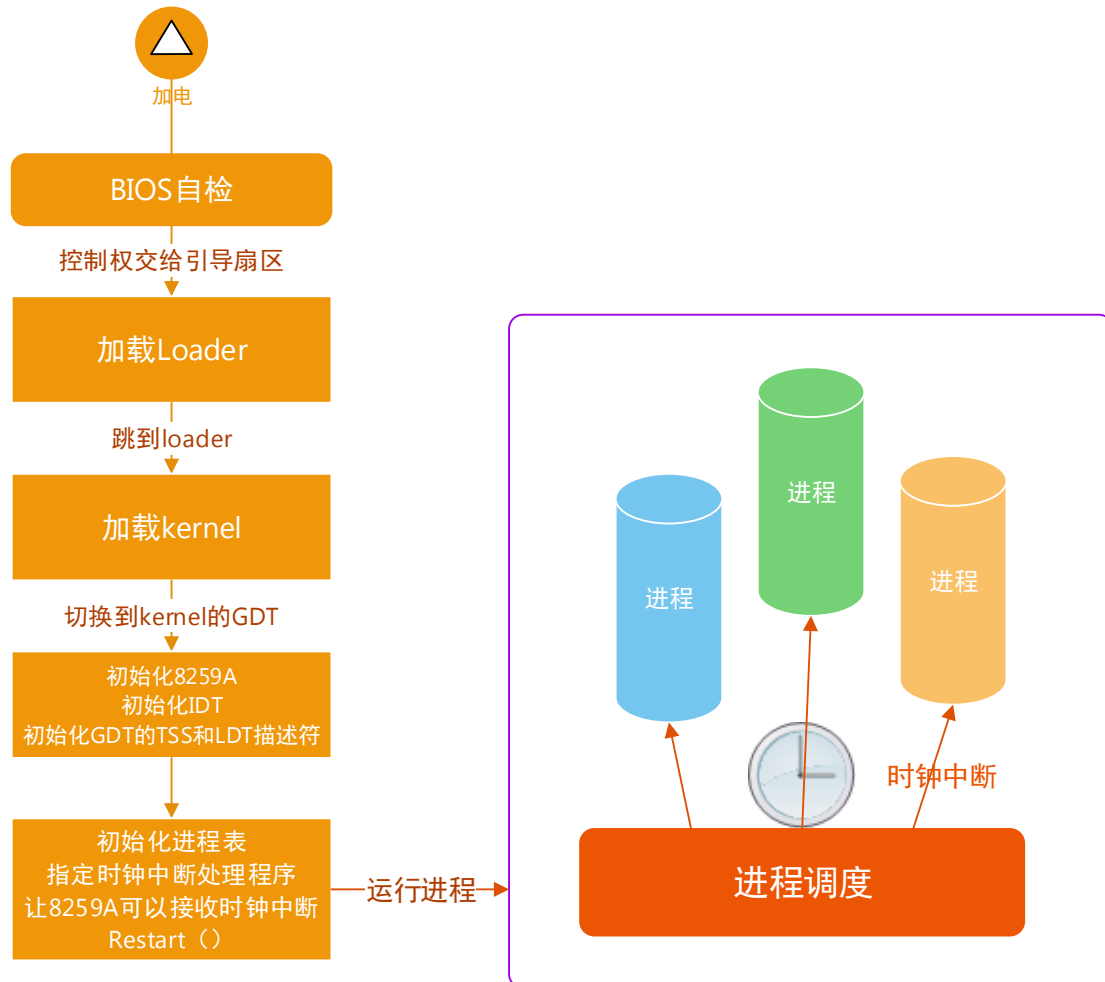
重入

当前进程未运行完不进行调度

是非重入且当前进程一次性运行完，进行进程调度

三、 程序关系图或流程图

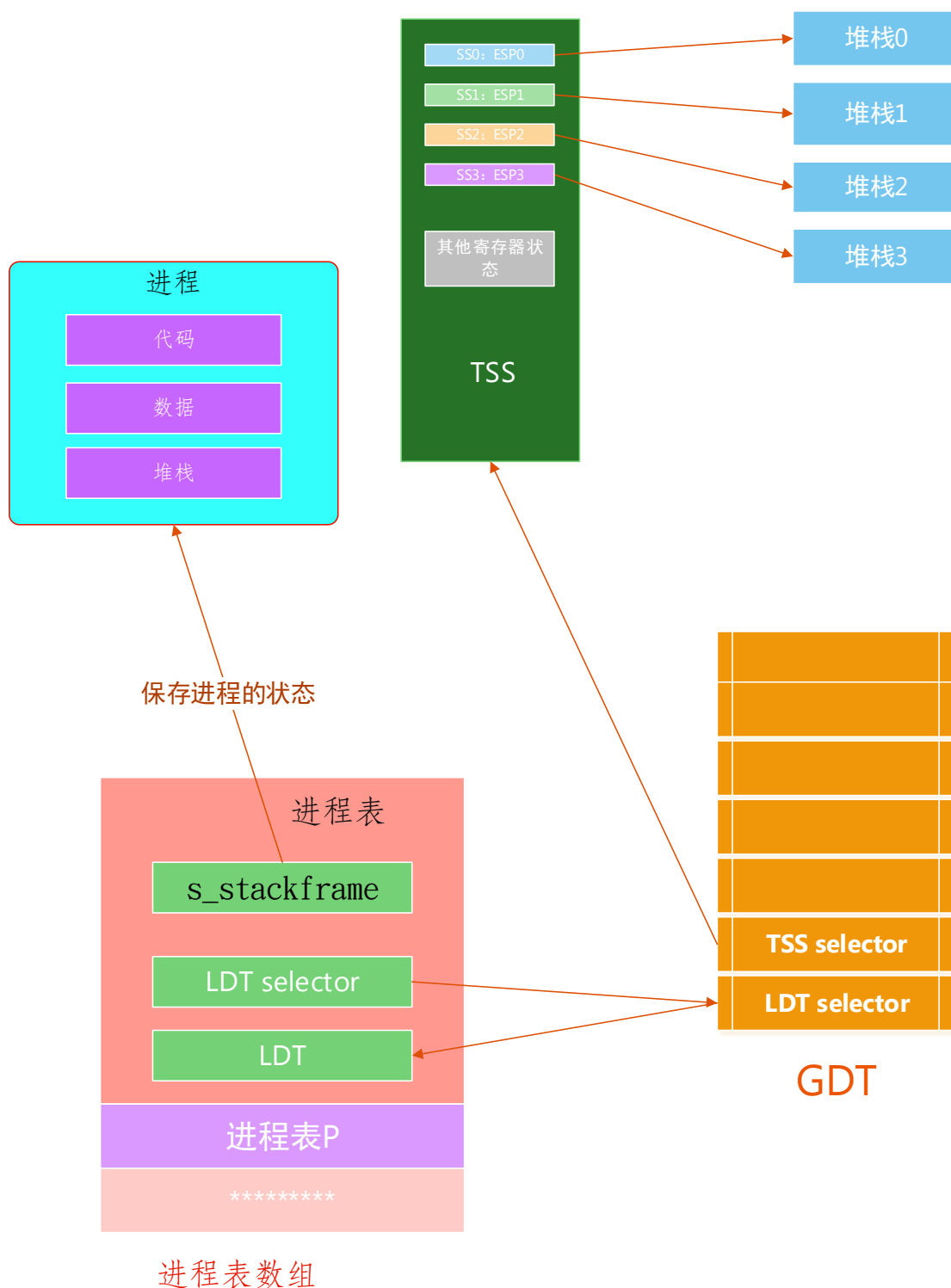
a) 程序的运行流程：



b) 进程的进程表、进程体、GDT、TSS 的关系

- 进程表是进程的描述，进程运行过程中如果被中断，各个寄存器的值都会被保存到进程表中。
- 进程表里的 LDT Selector 指向 GDT 中的一个 LDT 描述符，而这个描述符所指向的内存空间就存在与进程表内。
- GDT 中需要有一个描述符来指向 TSS。
- TSS (任务状态段, 104 字节): 在任务切换过程中起着重要作用，

通过它实现任务的挂起和恢复。

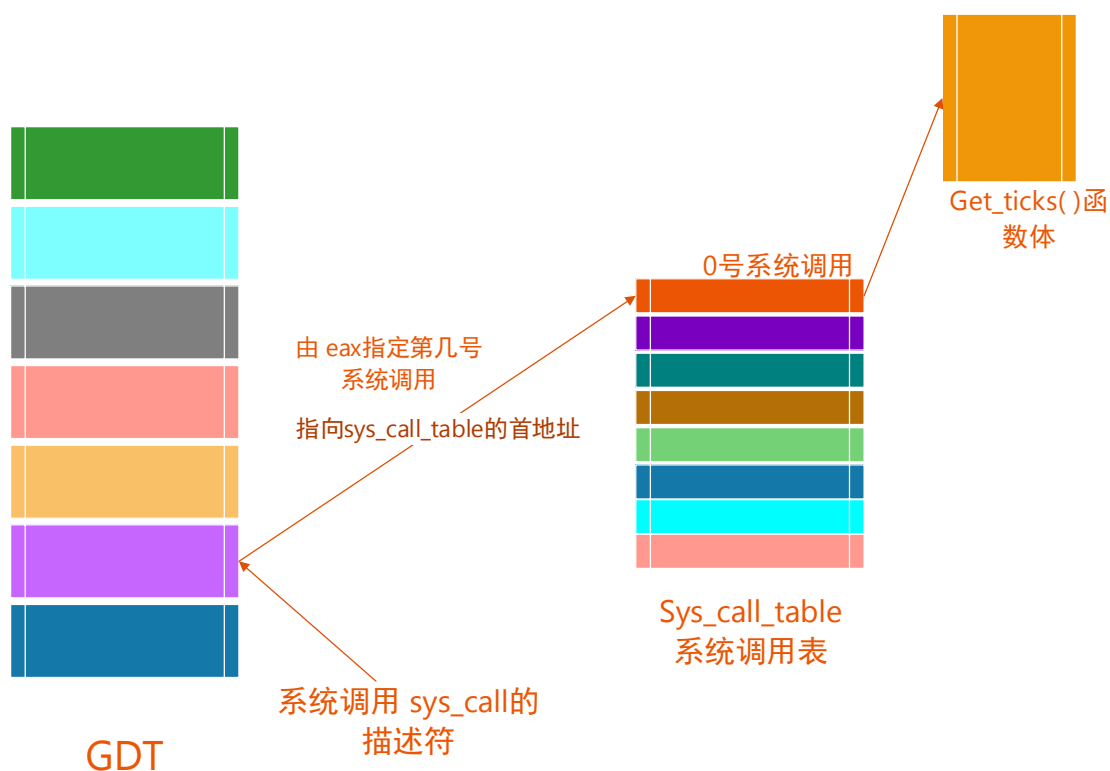


- 任务切换是指：挂起当前正在执行的任务，恢复或启动另一任务的执行。在任务切换过程中，首先，处理器中各寄存器的当前值被自动保存到 TR 所指定的 TSS 中；然后，下一任务的 TSS 的选

择子被装入 TR；最后，从 TR 所指定的 TSS 中取出各寄存器的值送到处理器的各寄存器中。由此可见，通过在 TSS 中保存任务现场各寄存器状态的完整映象，实现任务的切换。

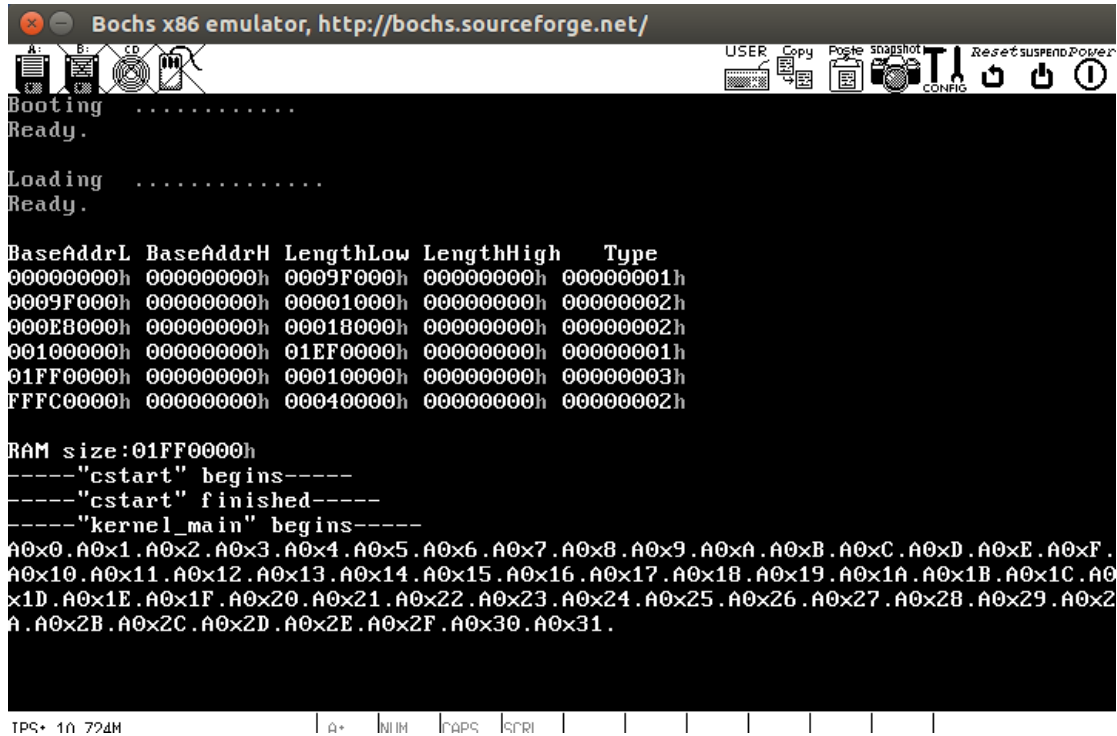
● TSS 和进程表 切换进程的区别？

c) 系统调用关系图：（中断的调用过程与此类似）



四、 运行过程及理解

6.a: 第一个进程：出现了字符 A 和不断增加的数字。



The screenshot shows the Bochs x86 emulator window. The title bar reads "Bochs x86 emulator, http://bochs.sourceforge.net/". The menu bar includes "USER", "Copy", "Paste", "snapshot", "CONFIG", "Reset", "suspend", and "Power". The main window displays the following text:

```
Booting .....
Ready.

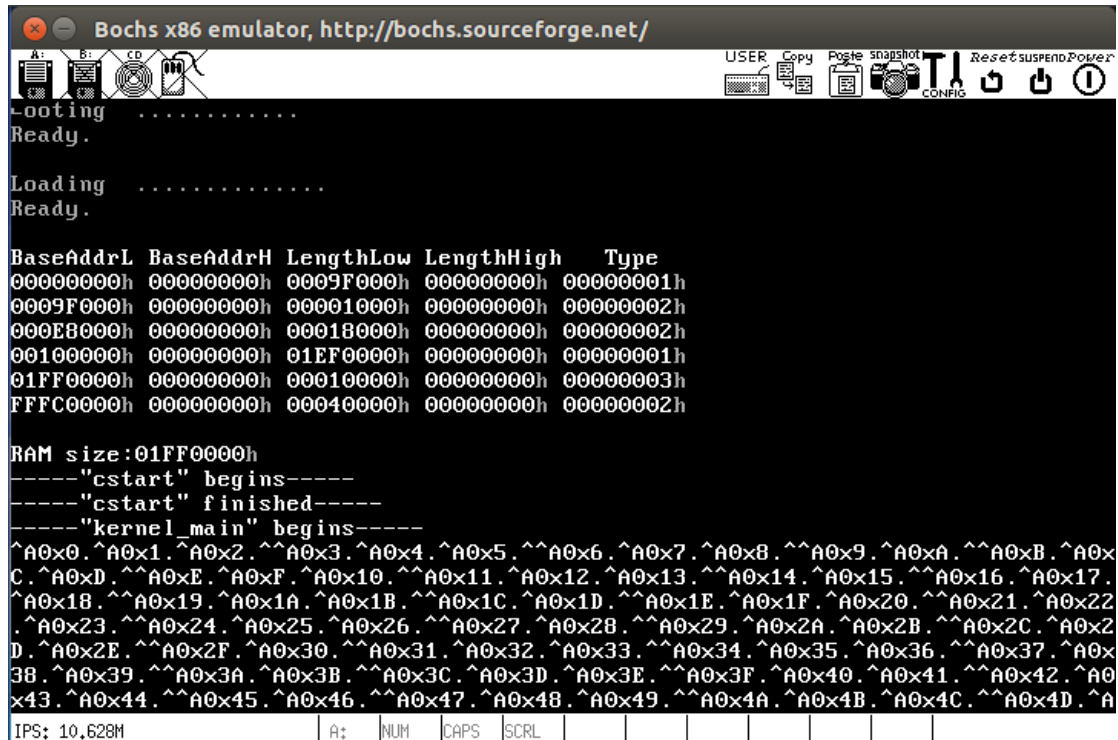
Loading .....
Ready.

BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0009F000h 00000000h 00000001h
0009F000h 00000000h 00001000h 00000000h 00000002h
000E8000h 00000000h 00018000h 00000000h 00000002h
00100000h 00000000h 01EF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h

RAM size:01FF0000h
-----"cstart" begins-----
-----"cstart" finished-----
-----"kernel_main" begins-----
A0x0.A0x1.A0x2.A0x3.A0x4.A0x5.A0x6.A0x7.A0x8.A0x9.A0xA.A0xB.A0xC.A0xD.A0xE.A0xF.
A0x10.A0x11.A0x12.A0x13.A0x14.A0x15.A0x16.A0x17.A0x18.A0x19.A0x1A.A0x1B.A0x1C.A0
x1D.A0x1E.A0x1F.A0x20.A0x21.A0x22.A0x23.A0x24.A0x25.A0x26.A0x27.A0x28.A0x29.A0x2
A.A0x2B.A0x2C.A0x2D.A0x2E.A0x2F.A0x30.A0x31.
```

The status bar at the bottom shows "TPS: 10.724M" and various keyboard status indicators: "A+", "NUM", "CAPS", "SCRL", and several function keys.

6.b: 添加中断处理：屏幕第一个字符会随时钟中断产生而不断的跳动。



The screenshot shows the Bochs x86 emulator window with the same title bar and menu bar as in 6.a. The main window displays the following text:

```
Booting .....
Ready.

Loading .....
Ready.

BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0009F000h 00000000h 00000001h
0009F000h 00000000h 00001000h 00000000h 00000002h
000E8000h 00000000h 00018000h 00000000h 00000002h
00100000h 00000000h 01EF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h

RAM size:01FF0000h
-----"cstart" begins-----
-----"cstart" finished-----
-----"kernel_main" begins-----
^A0x0.^A0x1.^A0x2.^A0x3.^A0x4.^A0x5.^A0x6.^A0x7.^A0x8.^A0x9.^A0xA.^A0xB.^A0x
C.^A0xD.^A0xE.^A0xF.^A0x10.^A0x11.^A0x12.^A0x13.^A0x14.^A0x15.^A0x16.^A0x17.
^A0x18.^A0x19.^A0x1A.^A0x1B.^A0x1C.^A0x1D.^A0x1E.^A0x1F.^A0x20.^A0x21.^A0x22
.^A0x23.^A0x24.^A0x25.^A0x26.^A0x27.^A0x28.^A0x29.^A0x2A.^A0x2B.^A0x2C.^A0x2
D.^A0x2E.^A0x2F.^A0x30.^A0x31.^A0x32.^A0x33.^A0x34.^A0x35.^A0x36.^A0x37.^A0x
38.^A0x39.^A0x3A.^A0x3B.^A0x3C.^A0x3D.^A0x3E.^A0x3F.^A0x40.^A0x41.^A0x42.^A0
x43.^A0x44.^A0x45.^A0x46.^A0x47.^A0x48.^A0x49.^A0x4A.^A0x4B.^A0x4C.^A0x4D.^A
```

The status bar at the bottom shows "IPS: 10.628M" and the same keyboard status indicators as in 6.a.

6.e: 多进程， clock_handler (int IRQ) 实现时间轮转法的进程切换。

A+数字和 B+数字和#交替出现。

Bochs x86 emulator, http://bochs.sourceforge.net/

Booting
Ready.

Loading
Ready.

BaseAddrL	BaseAddrH	LengthLow	LengthHigh	Type
00000000h	00000000h	0009F000h	00000000h	00000001h
0009F000h	00000000h	00001000h	00000000h	00000002h
000E8000h	00000000h	00018000h	00000000h	00000002h
00100000h	00000000h	01EF0000h	00000000h	00000001h
01FF0000h	00000000h	00010000h	00000000h	00000003h
FFFC0000h	00000000h	00040000h	00000000h	00000002h

RAM size:01FF0000h
-----"cstart" begins-----
-----"cstart" finished-----
-----"kernel_main" begins-----
A0x0.#B0x1000.#A0x1.#B0x1001.##B0x1002.#A0x2.###A0x3.#B0x1003.#A0x4.#B0x1004.##B0x1005.#A0x5.###A0x6.#B0x1006.##B0x1007.#A0x7.#B0x1008.#A0x8.###A0x9.#B0x1009.##B0x100A.#A0xA.###A0xB.#B0x100B.#A0xC.#B0x100C.##B0x100D.#A0xD.###A0xE.#B0x100E.##B0x100F.#A0xF.#B0x1010.#A0x10.##B0x1011.##B0x1012.#A0x12.#B0x1013.#A0x13.###A0x14.#B0x1014.##B0x1015.#A0x15.###A0x16.#B0x1016.#A0x17.#B0x1017.##B0x1018.#A0x18.###A0x19.#B0x1019.##B0x101A.#A0x1A.#B0x101B.#A0x1B.###A0x1C.#B0x101C.##B0x101D.#A0x1D.###A0x1E.#B0x101E.#A0x1F.#B0x101F.##B0x1020.#A0x20.##

6.L. 第一个系统调用：在 A 前面打印 ‘+’ 号

Bochs x86 emulator, http://bochs.sourceforge.net/

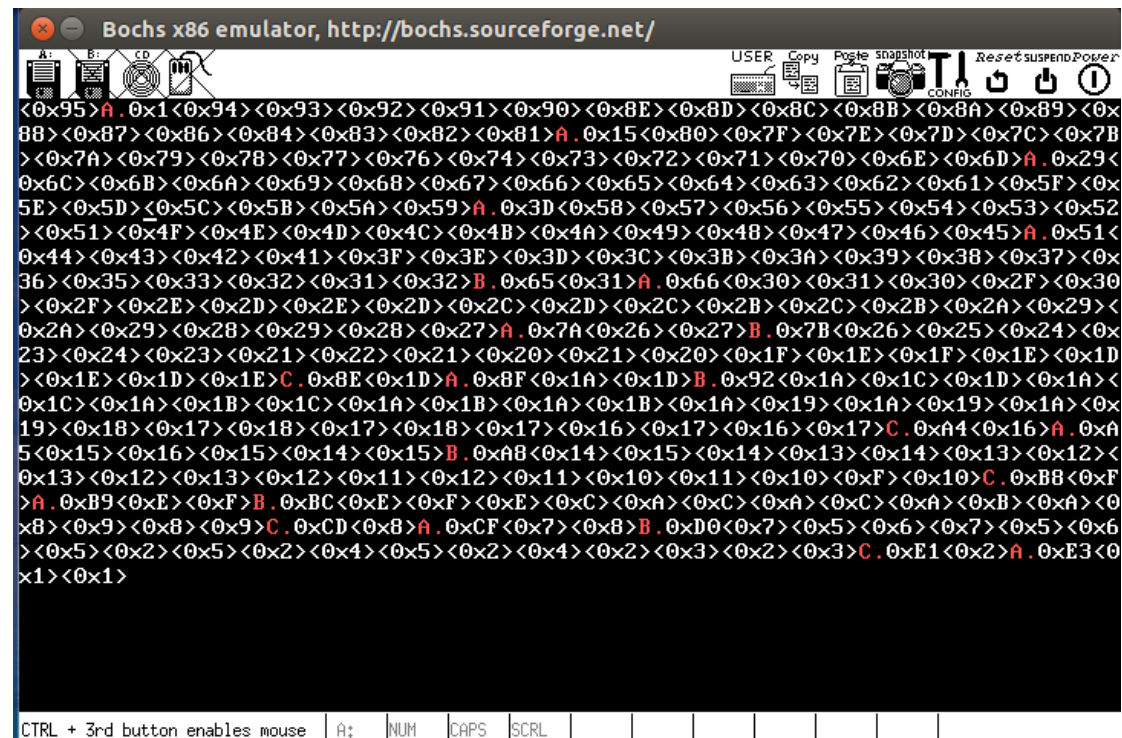
Booting
Ready.

Loading
Ready.

BaseAddrL	BaseAddrH	LengthLow	LengthHigh	Type
00000000h	00000000h	0009F000h	00000000h	00000001h
0009F000h	00000000h	00001000h	00000000h	00000002h
000E8000h	00000000h	00018000h	00000000h	00000002h
00100000h	00000000h	01EF0000h	00000000h	00000001h
01FF0000h	00000000h	00010000h	00000000h	00000003h
FFFC0000h	00000000h	00040000h	00000000h	00000002h

RAM size:01FF0000h
-----"cstart" begins-----
-----"cstart" finished-----
-----"kernel_main" begins-----
+A0x0.#B0x1000.#C0x2000.#+A0x1.#B0x1001.#C0x2001.##B0x1002.#C0x2002.#+A0x2.###A0x3.#B0x1003.#C0x2003.##B0x1004.#C0x2004.#+A0x4.#B0x1005.#C0x2005.#+A0x5.###A0x6.#B0x1006.#C0x2006.##B0x1007.#C0x2007.#+A0x7.#B0x1008.#C0x2008.#+A0x8.###A0x9.#B0x1009.#C0x2009.##B0x100A.#C0x200A.#+A0xA.###A0xB.#B0x100B.#C0x200B.#+A0xC.#B0x100C.#C0x200C.##B0x100D.#C0x200D.#+A0xD.###A0xE.#B0x100E.#C0x200E.##B0x100F.#C0x200F.#+A0xF.#B0x1010.#C0x2010.#+A0x10.###A0x11.#B0x1011.#C0x2011.##B0x1012.#C0x2012.#+A0x12.#B0x1013.#C0x2013.#+A0x13.###A0x14.#B0x1014.#C0x2014.##B0x1015.

6.q. 通过 schedule () 实现进程调度：并有优先级的概念



五、 重点知识总结

- 1) 从零个进入第一个进程
- 2) 从一个进程增加到两个进程
- 3) 从两个进程增加到三个进程
- 4) 进程的切换和中断处理
- 5) 多进程的调度