第6课 多任务的实现原理

《跟着瓦利哥学写OS》

联系方式

- 自己动手写操作系统QQ群：82616767。
- 申请考试邮箱：sangwf@gmail.com
- 所有课程的代码都在Github：

https://github.com/sangwf/walleclass

思考题回顾

- 我们给中断向量表初始化的默认处理程序int_ignore对哪类异常搞不定？为什么？

解决上节课的遗憾

- 我们只将前2M物理内存的页表映射好，当访问2M~4M之间的地址时，在缺页异常中，将对应的物理页码映射上去。打印输出写入的字符到屏幕。

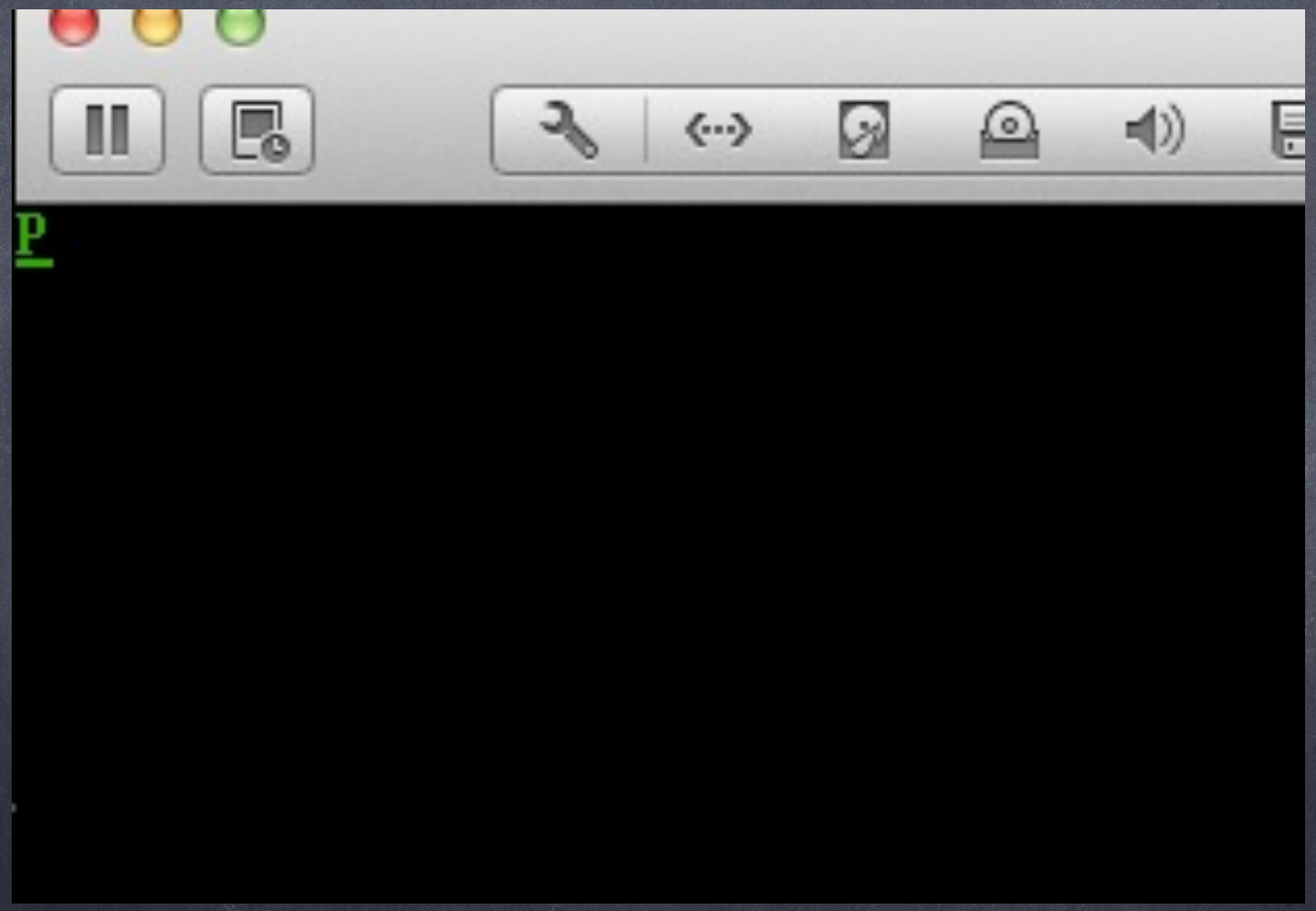
问题出在哪里?

```
1 KERNEL_SEL equ 0x08
2 SCREEN_SEL equ 0x10
3 DATA_SEL   equ 0x18
4
5 bits 32 ;这是32位的指令
6
7 mov eax, DATA_SEL ;先设置数据段
8 mov ds, eax
9 mov gs, eax ;这里不赋值，后果很严重
10
11
12 ;初始化栈
13 lss esp, [init_stack]
```

谜团依旧没解开。。。

- 使用gs之前，都给它专门赋值过。似乎gs会被某个地方隐含的调用到，期待答案！

迟到的运行结果



回顾操作系统课程

- 操作系统的多任务（进程/线程）是怎么实现的?
- 是不是有一个超级的调度进程？那这个进程挂了又怎么办？

多任务的解决之道

- 答案就在时钟中断！
- 通过时钟中断，我们可以将任务状态保存，并跳转到新的任务。

实验1

- 实现一个时钟中断，每次在屏幕上打印一个字符'T'，循环显示。

关键代码1

```
40 ;重新初始化时钟中断0x08
41 mov eax, 0x00080000 ;段选择符0x08
42 lea edx, [int_timer]
43 mov ax, dx
44
45 mov edx, 0x8E00 ;P DPL ...
46 mov ecx, 0x08 ;时钟中断号
47 lea edi, [idt + ecx * 8] ;中断的偏移地址放到edi
48 mov [edi], eax
49 mov [edi + 4], edx
```

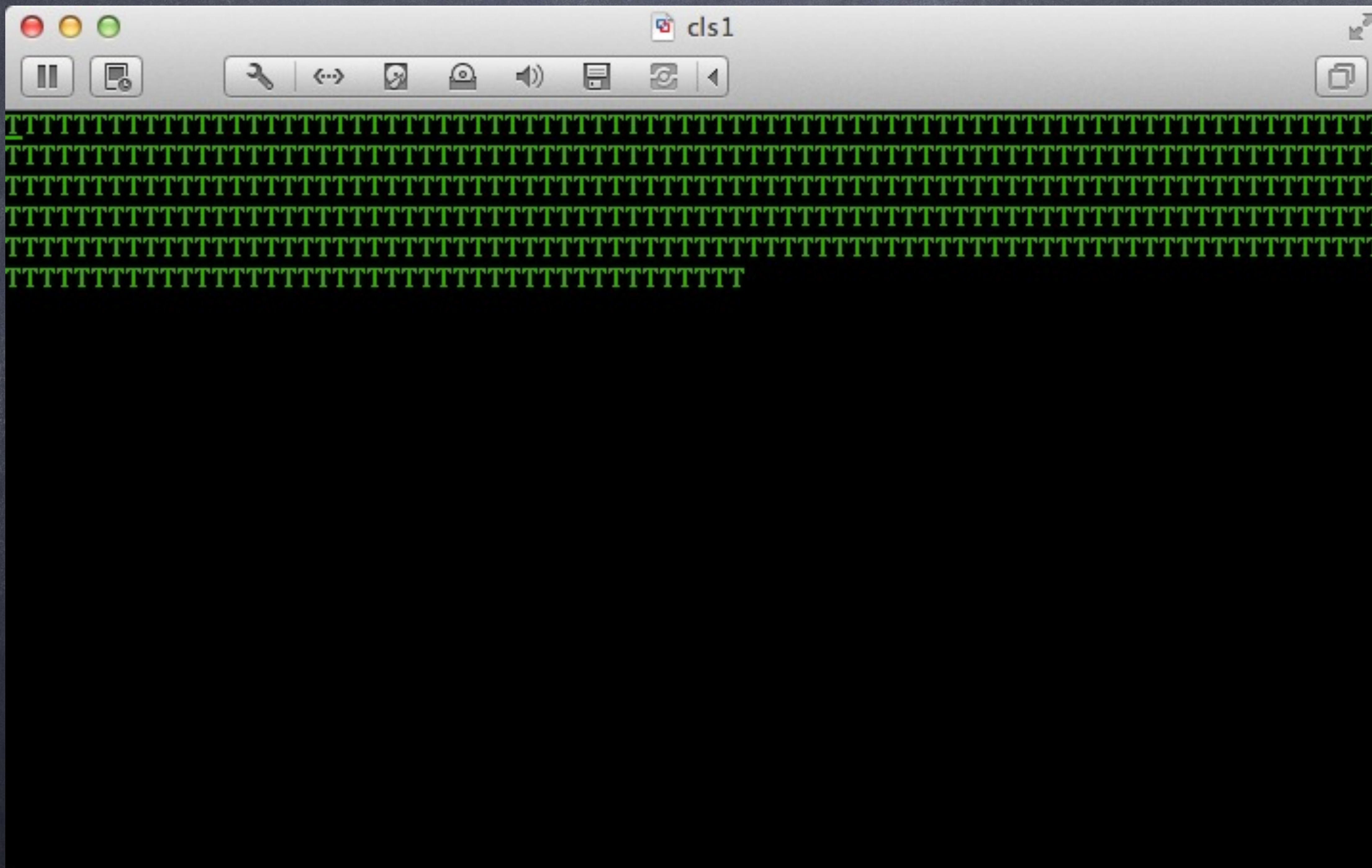
关键代码2

```
55 ;中断屏蔽管理
56 mov al, 0xFE ;只放开时钟中断
57 out 0x21, al
58 mov al, 0xFF
59 out 0xA1, al
60
61 ;设置时钟中断频率
62 mov al, 0x36
63 out 0x43, al
64
65 mov eax, LATCH
66 out 0x40, al
67 mov al, ah
68 out 0x40, al
```

关键代码3

```
int_timer:  ;时钟中断处理函数
    push eax

    mov al, 0x20
    out 0x20, al

    mov al, 'T'
    call func_write_char

    pop eax
    iret
```

运行结果


cls1
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT

任务状态段（TSS）

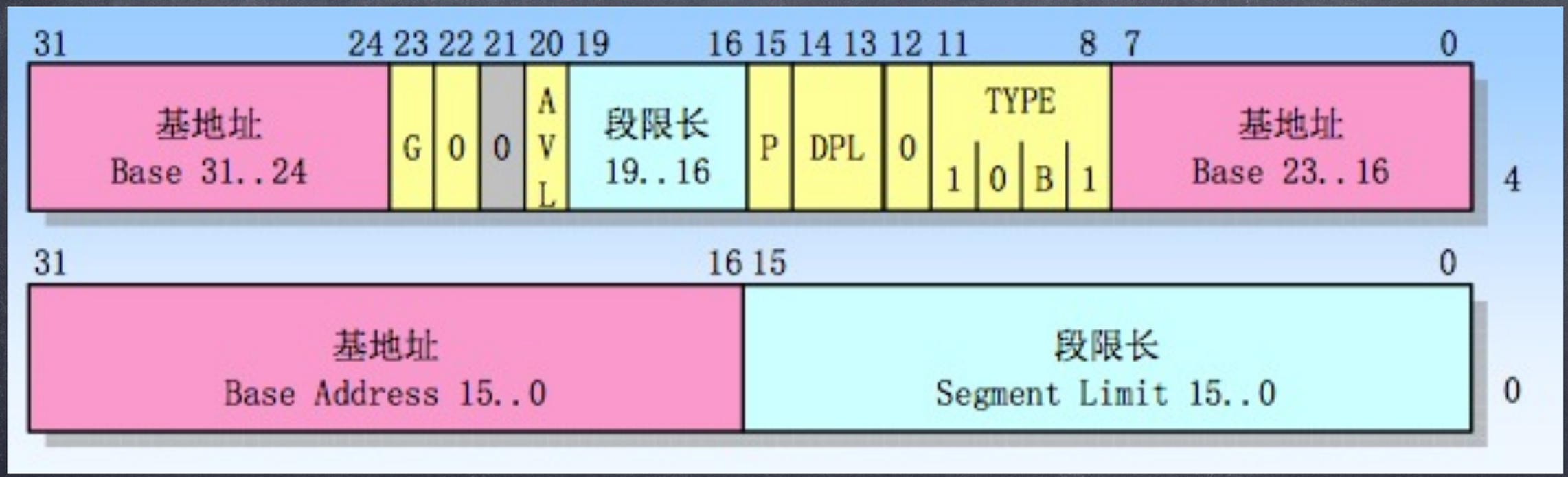
- TSS（Task State Segment）用于存放任务的执行状态信息，如EIP、SS、EAX等。这样只要保存了TSS，就可以把任务的时间静止了。然后换一个任务执行。

TSS的全貌（104 bytes）

内核栈与用户栈

- TSS中，包括SS0/ESP0、SS1/ESP1、SS2/ESP2，分别对应于CPU特权级中的Ring 0、Ring 1、Ring 2所使用的栈。
- 在Linux中，只使用了Ring 0和Ring 3，Ring 3所对应的栈就是用户栈。

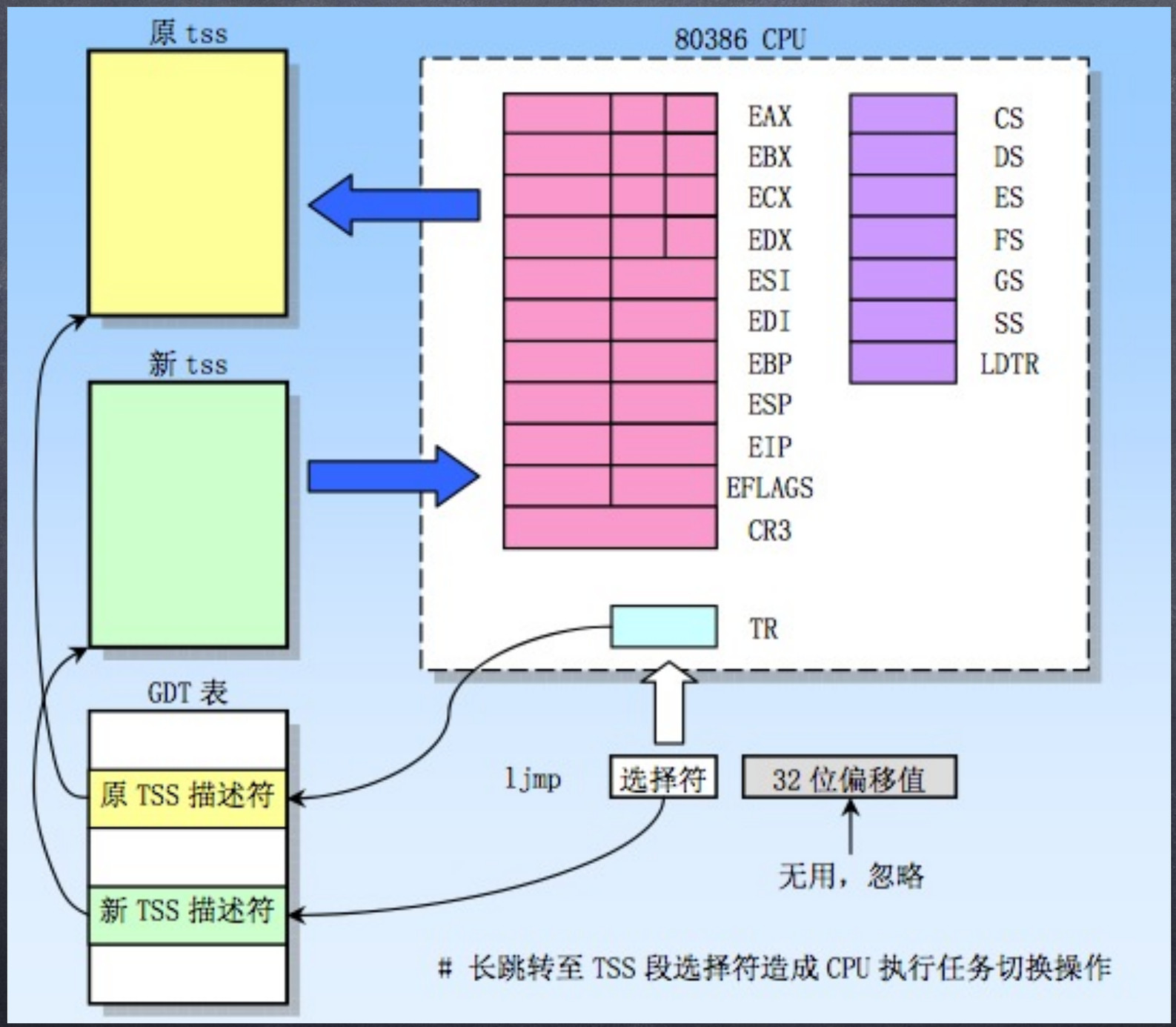
TSS描述符（存放于GDT中）



任务寄存器

- TR（Task Register）存放了16位的GDT中的TSS选择符，以及在不可见部分存放了整个TSS的描述符（出于效率考虑）。
- 通过jmp TSS选择符：0x0000000000，可以实现TR中存放的TSS选择符的切换。后面的0x0000000000是无效的。

任务切换示意图


原 tss
新 tss
80386 CPU
EAX
EBX
ECX
EDX
ESI
EDI
EBP
ESP
EIP
EFLAGS
CR3
CS
DS
ES
FS
GS
SS
LDTR
TR
GDT 表
原 TSS 描述符
新 TSS 描述符
ljmp
选择符
32 位偏移值
无用，忽略
长跳转至 TSS 段选择符造成 CPU 执行任务切换操作

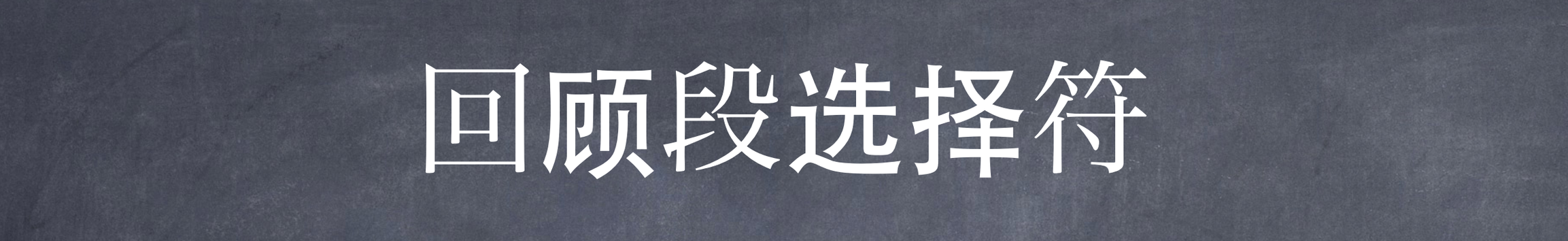
局部描述符表

- LDT（Local Descripter Table）与GDT相对，用于存放用户态任务（Ring 3）的局部使用的描述符，对其他任务不可见。

- LDTR也是TSS中的一个字段。

回顾段选择符

- 请求特权级 RPL（Requested Privilege Level）；
- 表指示标志 TI（Table Index）；
- 索引值（Index）。

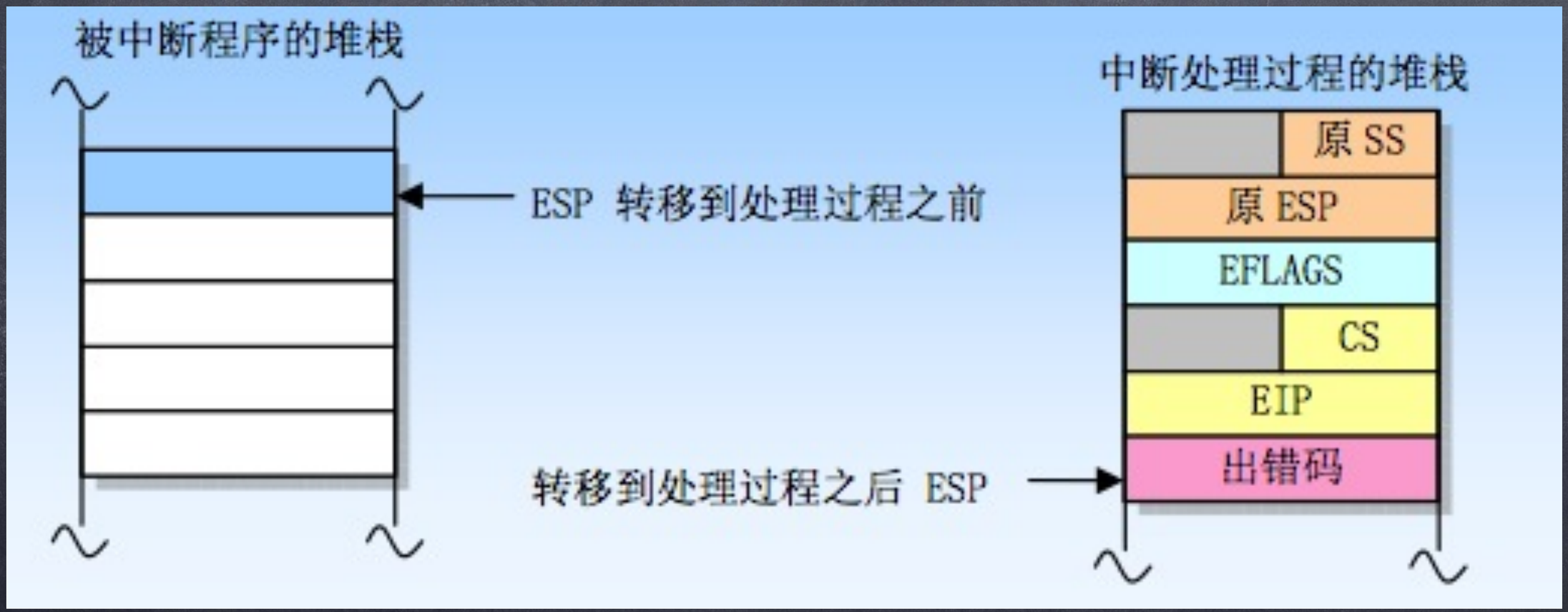


TI：0表示GDT中的描述符，1标识LDT中的描述符

从用户态到内核态

- 硬件中断，或int n。如果是用户态的任务，牵涉到用户栈到内核栈的切换。
- 从当前TSS中，获取到SS0、ESP0，然后将当前SS、ESP等压入该栈，并更新SS、ESP为SS0、ESP0。

用户栈到内核栈的切换


被中断程序的堆栈
ESP 转移到处理过程之前
转移到处理过程之后 ESP
中断处理过程的堆栈
原 SS
原 ESP
EFLAGS
CS
EIP
出错码

从内核态到用户态

- 通过iret。
- 在实验中，我们会构造一个中断压栈的场景，然后调用iret跳转到用户态。

实现多任务的步骤

- 1，设置时钟中断，在时钟中断中，实现任务切换的逻辑。
- 2，设置GDT，在GDT中，设置任务的TSS/LDT描述符。
- 3，设置TR、LDT寄存器，并构造一个中断场景。
- 4，调用iret跳转到用户态任务。

实验2

- 实现两个用户态任务，分别不断打印A和B，通过时钟中断实现任务切换。

- 遥想当年Linus实现Linux之前，就写了这么一个程序，很激动的拿给自己的亲妹妹看，她看了一眼觉得还不错，可是不理解。这就是传说中的Linux 0.00，是它孕育了后来的Linux。

关键代码1

```
int_timer: ;时钟中断处理函数
    push ds
    push eax

    mov al, 0x20
    out 0x20, al

    mov eax, DATA_SEL
    mov ds, ax

    mov eax, 1
    cmp [current], eax
    je .switch_0
    mov dword [current], 1
    jmp TSS1_SEL: 0 ;注意，执行这句后，马上保存了当前现场，
                    ;并跳转到了任务1之前的现场去执行，也就
                    ;下次切换回来时，是执行了下一句。是在内
                    ;核态时，别切换出去了。
    jmp .switch_finish
    .switch_0:
    mov dword [current], 0
    jmp TSS0_SEL: 0
    .switch_finish:
    pop eax
    pop ds
    iret
```

关键代码2

```
;第四个描述符（0x20），TSS0
dw 0x68
dw tss0
dw 0xe900
dw 0x0

;第五个描述符（0x28），LDT0
dw 0x40
dw ldt0
dw 0xe200
dw 0x0

;第六个描述符（0x30），TSS1
dw 0x68
dw tss1
dw 0xe900
dw 0x0

;第七个描述符（0x38），LDT1
dw 0x40
dw ldt1
dw 0xe200
dw 0x0
```

关键代码3

```
tss0:
    dd 0
    dd krn_stk0, DATA_SEL ; 第1个dd是内核栈
    dd 0, 0, 0, 0, 0
    dd 0, 0, 0, 0, 0
    dd 0
    dd 0, 0, 0, 0 ; 第14个dd是用户栈
    dd 0, 0, 0, 0x17, 0, 0
    dd LDT0_SEL, 0x80000000

times 128 dd 0
krn_stk0:

align 8
ldt0:
    dw 0, 0, 0, 0
    dw 0x03ff, 0x0000, 0xfa00, 0x00c0
    dw 0x03ff, 0x0000, 0xf200, 0x00c0
```

关键代码4

```
tss1:
    dd 0
    dd krn_stk1, DATA_SEL ; 第1个dd是内核栈
    dd 0, 0, 0, 0, 0
    dd task1, 0x200 ;eip, eflags
    dd 0, 0, 0, 0
    dd usr_stk1, 0, 0, 0 ; 第14个dd是用户栈
    dd 0x17, 0x0f, 0x17, 0x17, 0x17, 0x17
    dd LDT1_SEL, 0x80000000

times 128 dd 0
krn_stk1:
times 128 dd 0
usr_stk1:

align 8
ldt1:
    dw 0, 0, 0, 0
    dw 0x03ff, 0x0000, 0xfa00, 0x00c0
    dw 0x03ff, 0x0000, 0xf200, 0x00c0
```

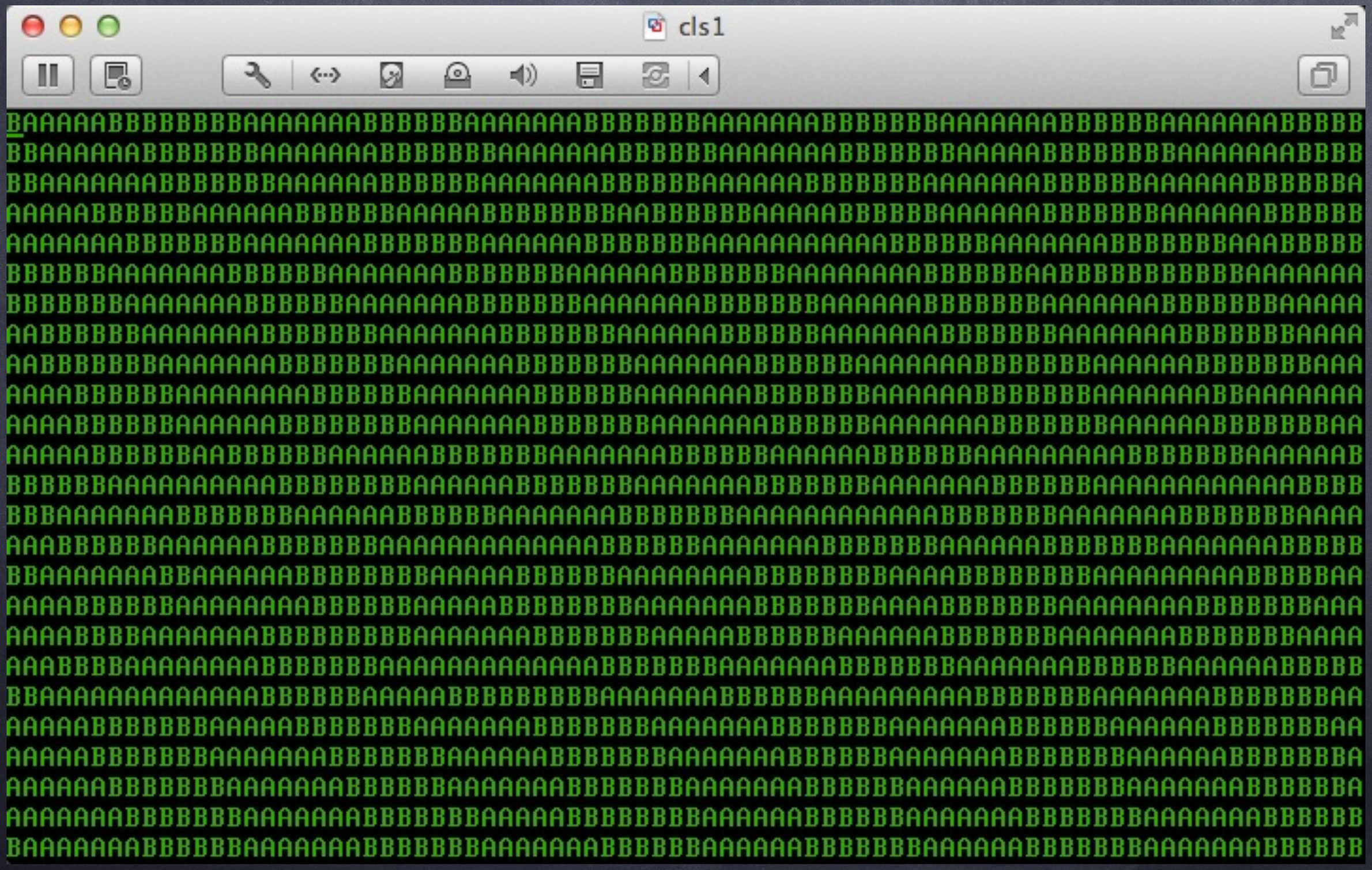
关键代码5

```
pushfd ;复位eflags中的任务嵌套标志，不用关注
and dword [esp], 0xffffbfff
popfd

mov eax, TSS0_SEL
ltr ax
mov eax, LDT0_SEL
lldt ax
mov dword [current], 0
sti ;开中断

push 0x17 ;任务0的局部数据段用于ss
push init_stack
pushfd
push 0x0f ;CS
push task0 ;EIP
iret
```

关键代码6

```
task0:
    mov al, 'A'
    int 0x81
    mov ecx, 0xffffff
    .t0:
    loop .t0
    jmp task0
task1:
    mov al, 'B'
    int 0x81
    mov ecx, 0xffffff
    .t1:
    loop .t1
    jmp task1
```

运行结果



思考

- Linux下，是怎么创建一个新进程的？

谢谢！