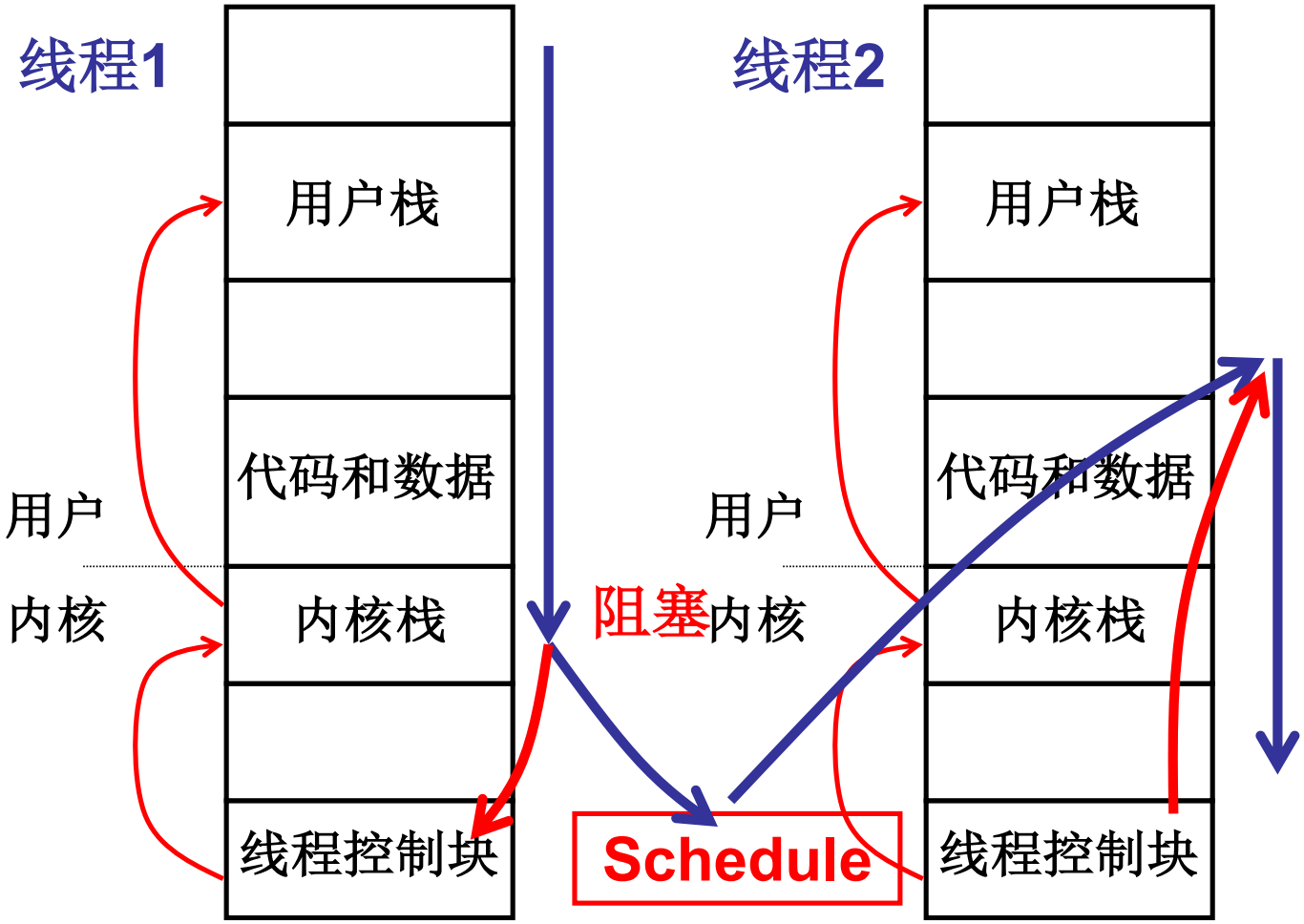
操作系统 Operating Systems

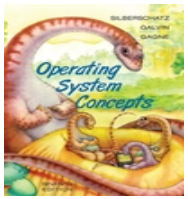
L12 内核级线程实现

Create Kernel Threads

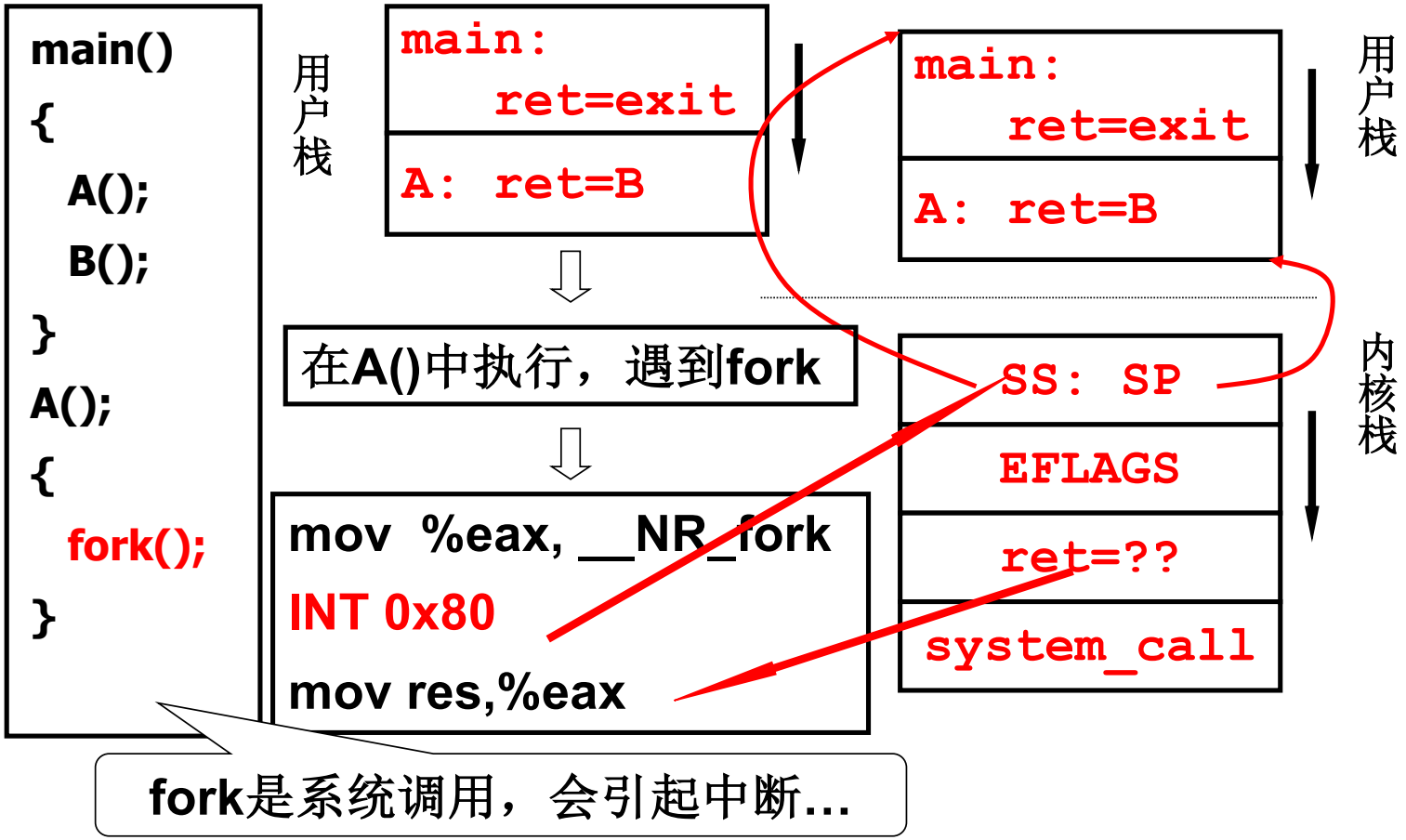
授课教师：李治军 lizhijun_os@hit.edu.cn
综合楼411室

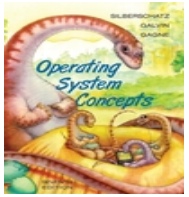
核心级线程的两套栈，核心是内核栈...





整个故事要从进入内核开始——某个中断开始…





切换五段论中的中断入口和中断出口

```
void sched_init(void)
{set_system_gate(0x80,&system_call);}
```

- 初始化时将各种中断处理设置好

```
_system_call:
 push %ds..%fs
 pushl %edx...
 call sys_fork
 pushl %eax
```

中断返回

```
movl _current,%eax
cmpl $0,state(%eax)
jne reschedule
cmpl $0,counter(%eax)
je reschedule
ret_from_sys_call:
```

```
reschedule:
 pushl $ret_from_sys_call
 jmp _schedule
```

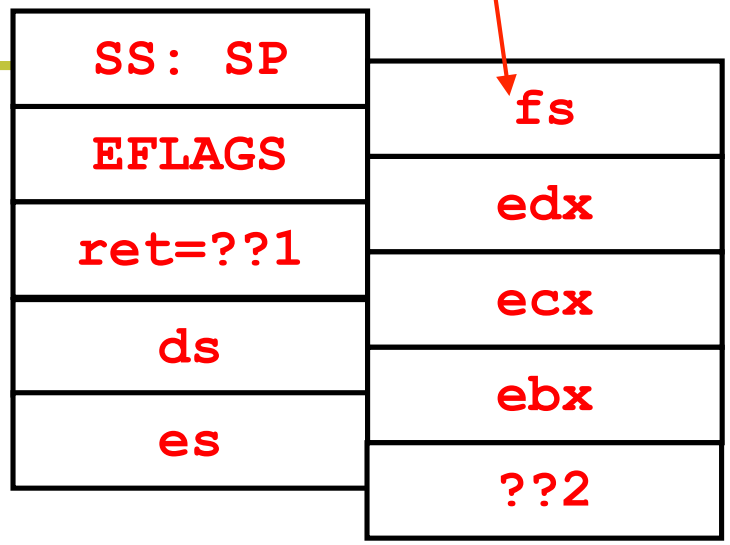
如果非零的话，则进行调度

C函数入口

PCB中的状态，非零表示阻塞

判断时间片是不是用光了

将用户态的各个寄存器压入内核栈，用于之后恢复现场



在内核中可能会引发线程切换，比如磁盘读写。

切换五段论中的schedule和中断出口

```
_system_call:
 call sys_fork
 pushl %eax
```

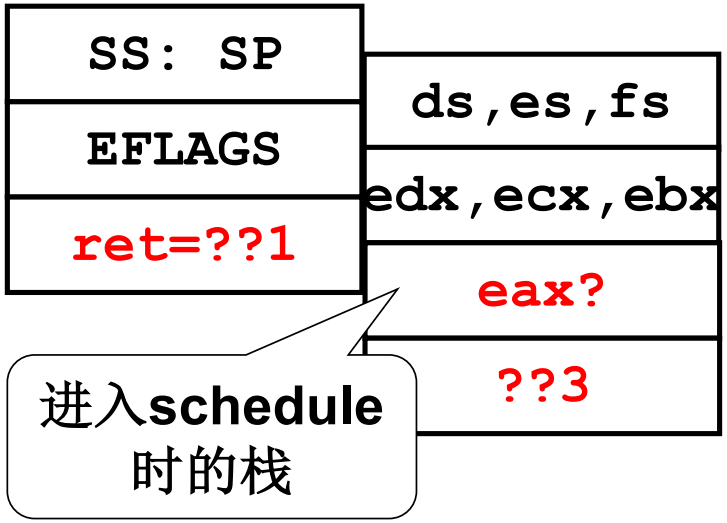
```
cmpl $0,counter(%eax)
je reschedule
ret_from_sys_call:
```

```
reschedule:pushl $ret_from_sys_call
 jmp _schedule
```

```
void schedule(void){ next=i;
  switch_to(next); }
```

```
ret_from_sys_call:
  popl %eax //返回值 popl %ebx ...
  pop %fs ...
  iret//重要
```

返回到int 0x80后面执行，
mov res,%eax，res=?

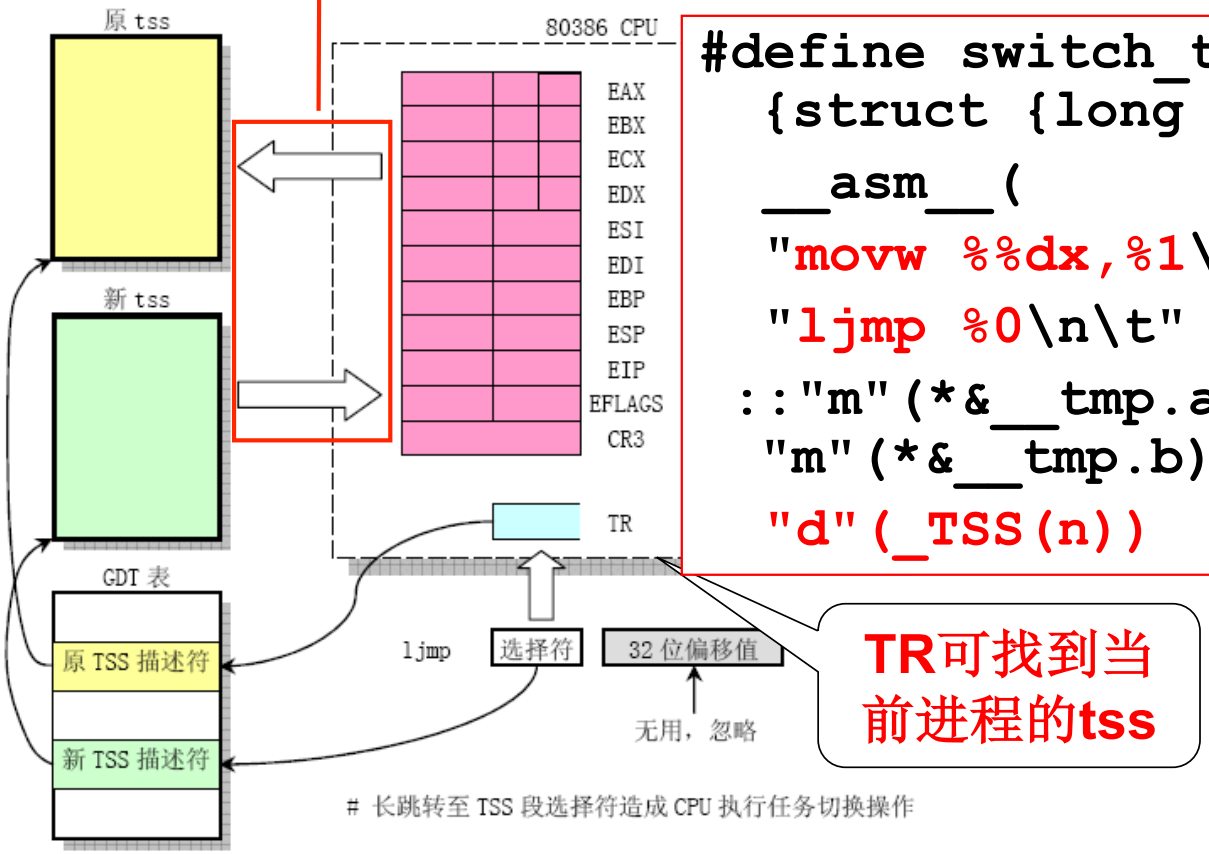


出口需要将之前push进来的寄存器
再pop出去恢复用户态的现场

进行任务的现场的
保存与恢复

tss切换效率比较低，
将tss切换替换为基于内核栈的切换
现在的linux和windows都是基于内核栈
的切换

切换五段论中的switch_to

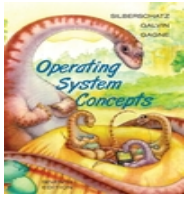



```
#define switch_to(n)
  {struct {long a,b;}
  __asm__(
  "movw %%dx,%1\n\t"
  "ljmp %0\n\t"
  ::"m"(*&__tmp.a),
  "m"(*&__tmp.b),
  "d"(_TSS(n))
```

TR可找到当
前进程的tss

- Linux 0.11用tss切换，但也可以用栈切换，因为tss中的信息可以写到内核栈中

Task struct segment



Fork 叉子分叉

创建一个线程，
就是做出一个能切换的样子！

另一个故事ThreadCreate就顺了…

- 从sys_fork开始CreateThread

```
_sys_fork:
 push %gs; pushl %esi
 ...
 pushl %eax
 call _copy_process
 addl $20,%esp
 ret
```

因为前面的代码已经将栈压好了，
所以这个地方只需要将对应的参数和
栈中各个指针的位置对应上就行了！

```
int copy_process(int nr,long ebp,
  long edi,long esi,long gs,long
  none,long ebx,long ecx,long edx, long
  fs,long es,long ds,long eip,long
  cs,long eflags,long esp,long ss)
```

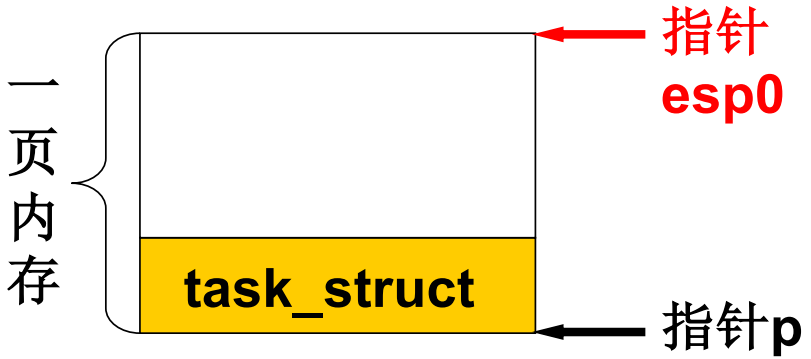
将进程全部先复制一份

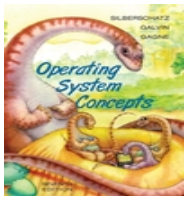
将所有的寄存器当做参数传入

copy_process的细节：创建栈 获取一页内存

```
p=(struct task_struct *)get_free_page();
   //申请内存空间
p->tss.esp0 = PAGE_SIZE + (long) p;
p->tss.ss0 = 0x10;
   //创建内核栈
p->tss.ss = ss & 0xffff;
p->tss.esp = esp;
   //创建用户栈(和父进程共用栈)
```

申请内存空间；
创建TCB；
创建内核栈和用户栈；
填写两个stack；
关联栈和TCB；





父 (eax!=0)

子 (eax=0)

copy_process的细节：执行前准备

```
p->tss.eip = eip;
p->tss.cs = cs & 0xffff;
//将执行地址cs:eip放在tss中
p->tss.eax = 0;
p->tss.ecx = ecx;
//执行时的寄存器也放进去了
p->tss.ldt = _LDT(nr);
set_tss_desc(gdt+(nr<<1) +
FIRST_TSS_ENTRY, &(p->tss));
set_ldt_desc(gdt+(nr<<1) +
FIRST_LDT_ENTRY, &(p->ldt));
    //内存跟着切换
p->state = TASK_RUNNING;
```

```
copy_process( ...,
   long eip,long
   cs,long
   eflags,long
   esp,long ss)
```

```
...
填写两个stack;
```

SS: SP
EFLAGS
ret=??1
ds,es,fs
edx,ecx,ebx

仔细体会tss将要承担的作用…

将eax置为零

前面返回的时候将eax置给ret

```
if(!fork()){
}
```

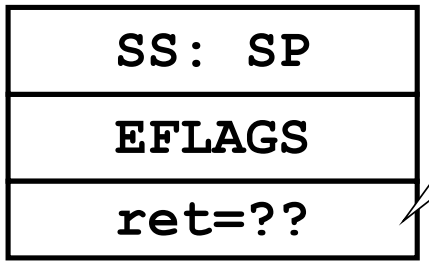
在返回的时候，父进程eax不是零，所以不执行里面的代码而子进程里面是零，所以执行了里面的代码！

子进程利用父进程的壳子，然后跑自己的代码！

第三个故事: 如何执行我们想要的代码?

```
int main(int argc, char * argv[])
{  while(1)  { scanf("%s", cmd);
     if(!fork()) {exec(cmd);}  wait(0); }
```

ThreadCreate(*A)中的A必须体现?
用户输入hello命令，exec(hello)



iret指令前
%eax是什么?

- fork()何时返回0，何时不会? 首先要找到fork()怎么返回?

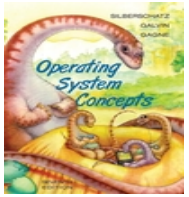
```
mov  %eax, __NR_fork
INT 0x80
mov res,%eax
```

如何到这条指令?

- 父进程用iret，因为要从核心态到用户态；那么子进程呢?仔细想一想...

ljmp tss ⇨

```
p->tss.eip = eip;
p->tss.cs = cs;
p->tss.eax = 0;
```



exec做的工作是将壳子中的内容替换。
在进入之前，父进程和子进程执行的是一段代码。
那么在内核中折腾一段时间返回之后，
应该执行的是子进程自己的代码。这就是这个中断要做的事情。

结构: 子进程进入A，父进程等待...

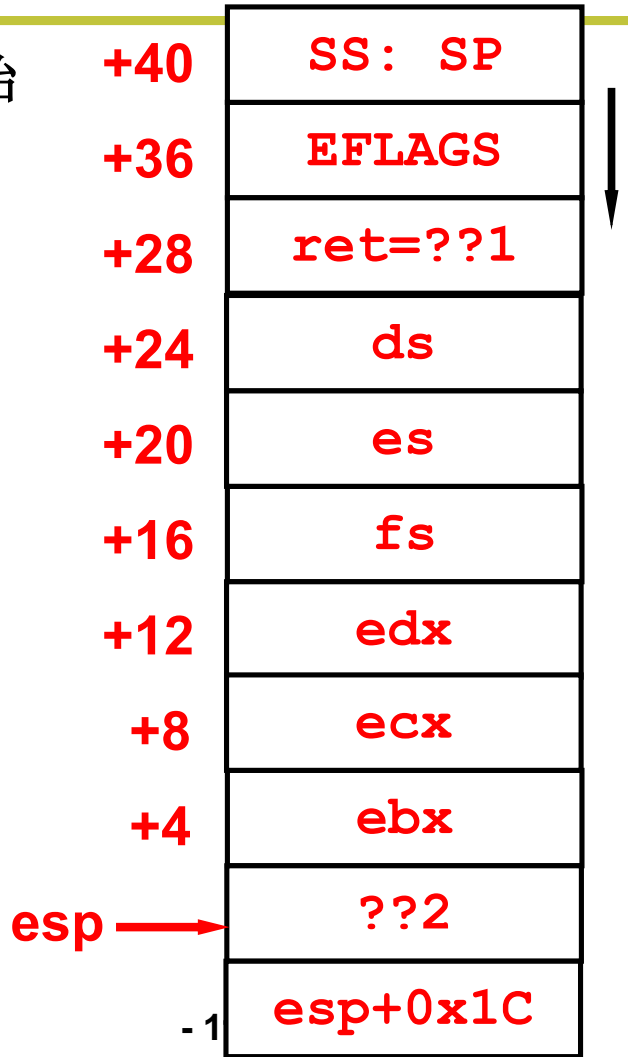
- 故事要从exec这个系统调用开始

```
if(!fork()) {exec(cmd);}
```

```
_system_call:
   push %ds .. %fs
   pushl %edx..
   call sys_execve
```

```
_sys_execve:
  lea EIP(%esp),%eax
  pushl %eax
  call _do_execve
```

```
EIP = 0x1C
```



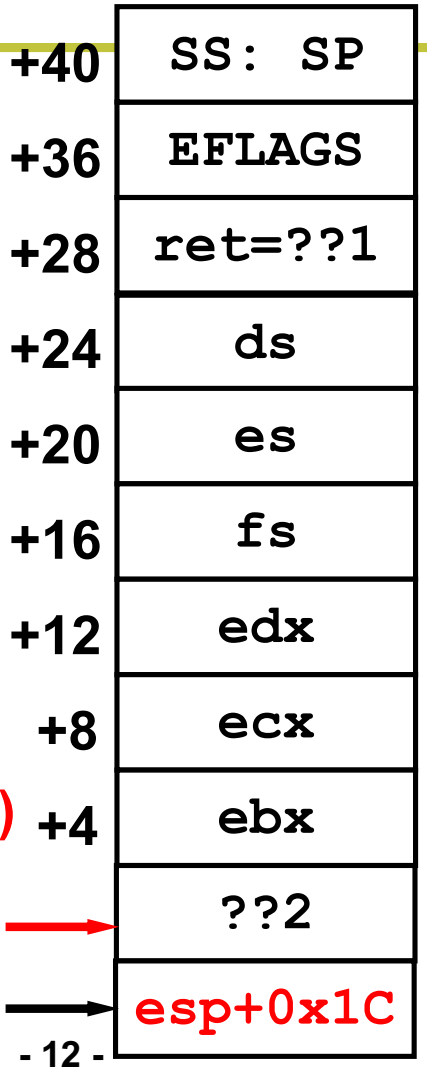
中断返回就是执行iret指令，
就是从栈中弹出地址，并执行这个地址。
所以只需要将子进程的入口地址放到栈中合适的位置即可！

终于可以让A执行了...

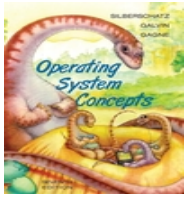
```
int do_execve( * eip,...
{  p += change_ldt(...;
   eip[0] = ex.a_entry;
   eip[3] = p; ...
```

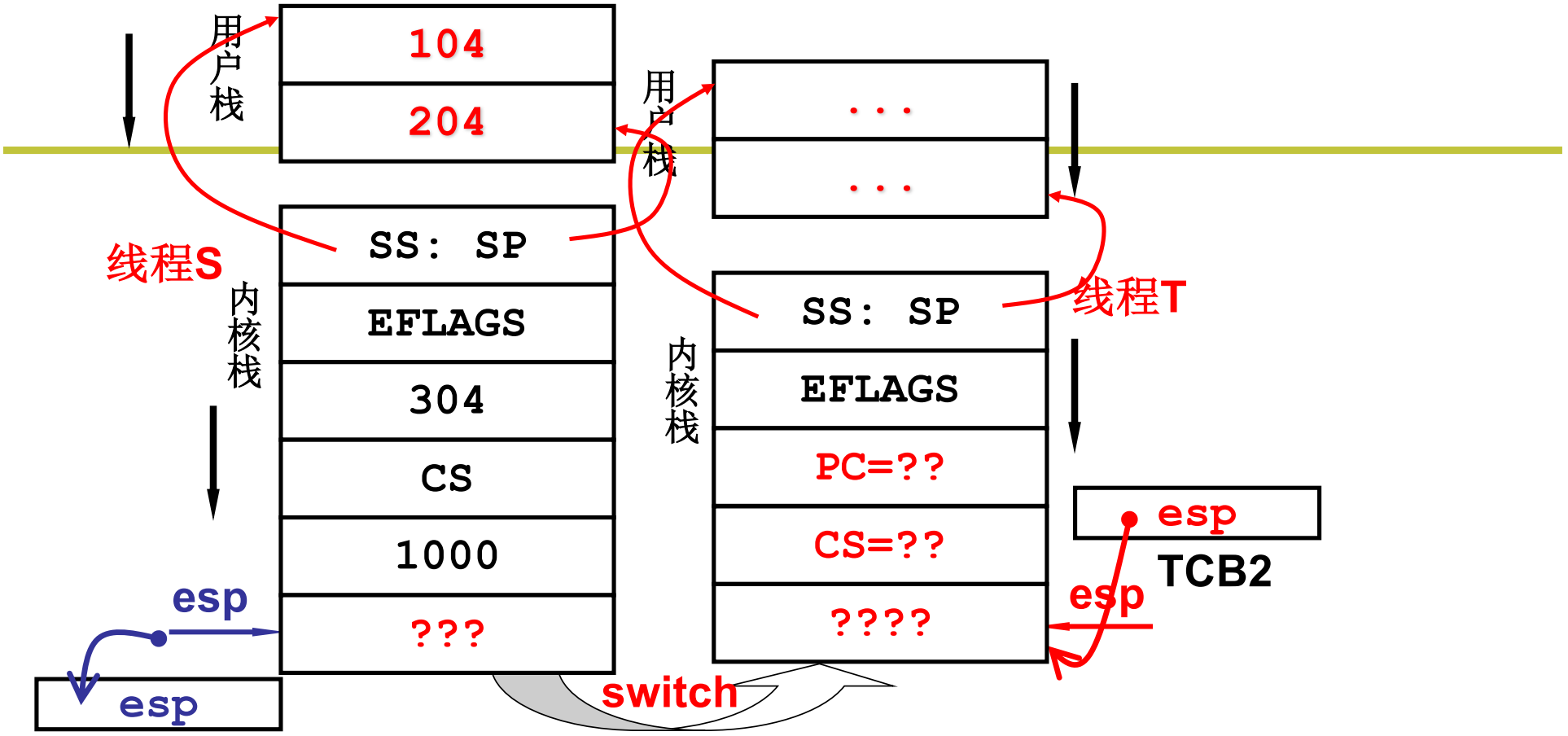
```
struct exec {
 unsigned long a_magic;
 unsigned a_entry; //入口 };
```

- eip[0] = esp + 0x1C; eip[3] = esp + 0x1C+0x0C = esp + 0x28 (正好是SP)



ex.a_entry是可执行程序入口地址，产生可执行文件时写入...





- 理解switch_to对应的栈切换，将自己变成计算机
- ThreadCreate的目的就是初始化这样一套栈