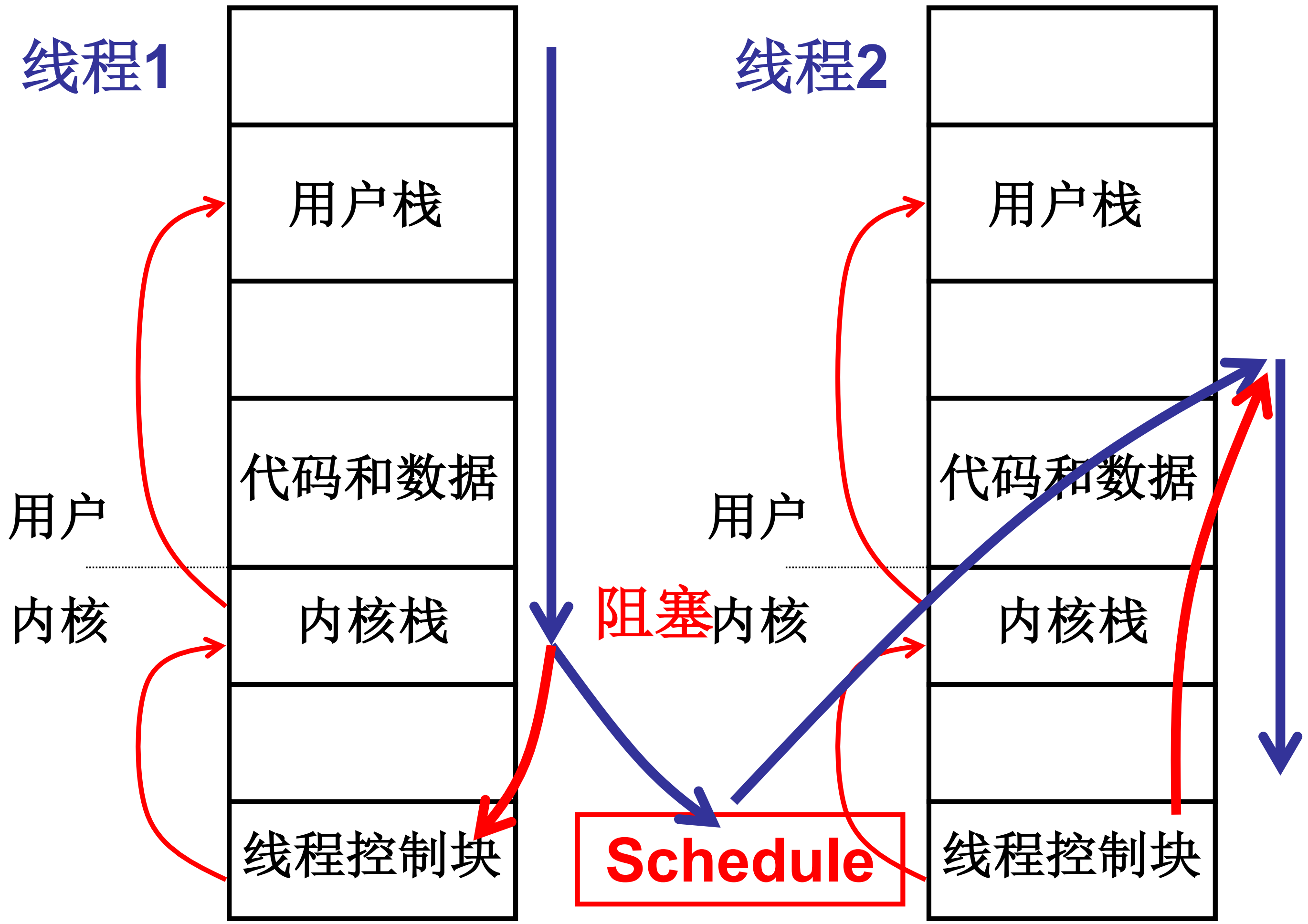
操作系统 Operating Systems

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...

```
main()
{
  A();
  B();
}
A();
{
  fork();
}
```

用户栈

main: ret=exit
A: ret=B

在A()中执行，遇到fork

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

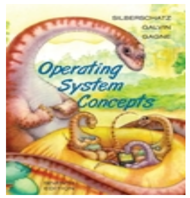
用户栈

main: ret=exit
A: ret=B

内核栈

SS: SP
EFLAGS
ret=??
system_call

fork是系统调用，会引起中断...



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
```

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

状态阻塞，那么重新调度

时间片是否用完了，用完就重新调度

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

内核栈

SS： SP	
EFLAGS	fs
ret=??1	edx
ds	ecx
es	ebx
	??2

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); }
```

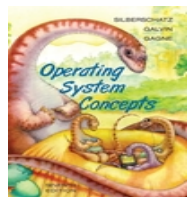
```
ret_from_sys_call:
   popl %eax //返回值 popl %ebx ...
   pop %fs ...
   iret//重要
```

返回到int 0x80后面执行，
mov res,%eax，res=?

SS：SP
EFLAGS
ret=??1

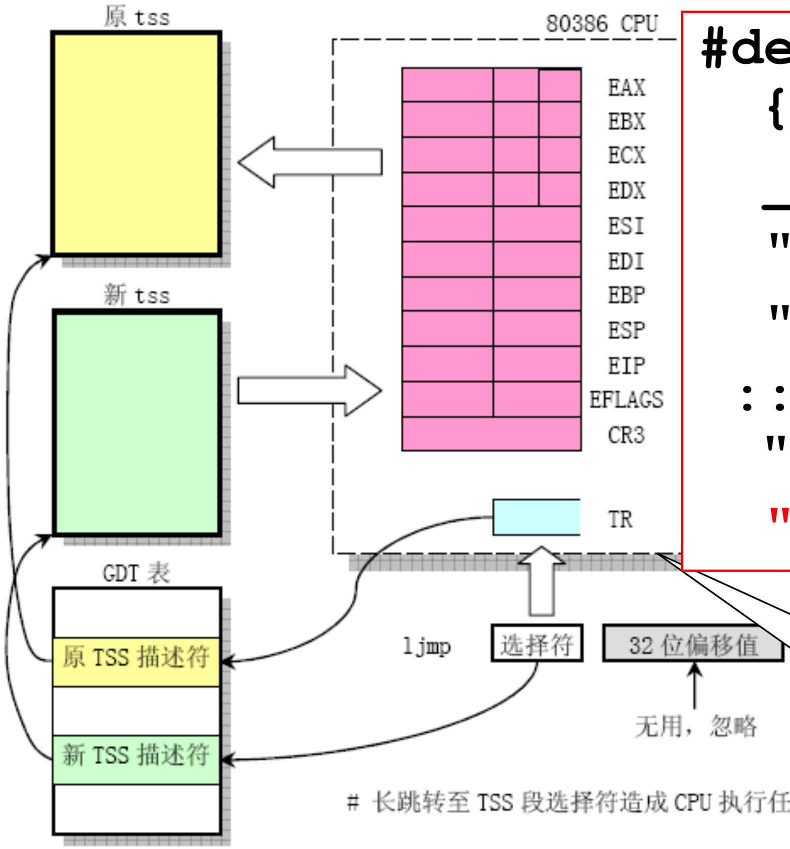
ds,es,fs
edx,ecx,ebx
eax?
??3

进入schedule时的栈



切换五段论中的switch_to

是基于TSS来进行的



长跳转至 TSS 段选择符造成 CPU 执行任务切换操作

```
#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**

另一个故事ThreadCreate就顺了…

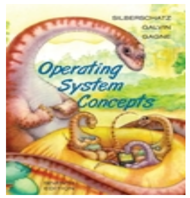
- 从sys_fork开始CreateThread

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

SS： SP
EFLAGS
ret=??1
ds,es,fs
edx,ecx,ebx
??2

gs
esi,edi
ebp
eax
??4

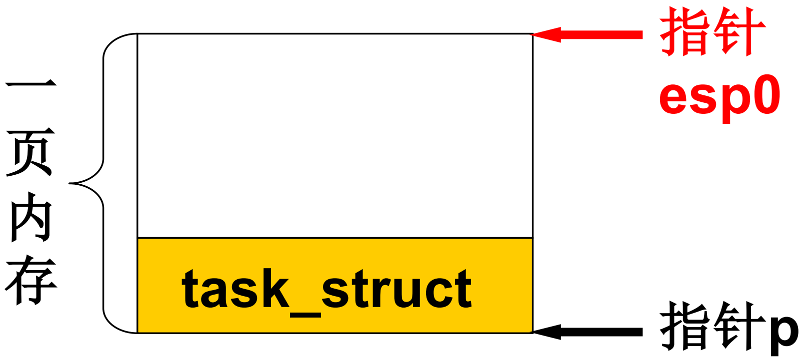
```
int copy_process(int nr,long ebp,
  long edi,long esi,long gs,long
  none,long ebx,long ecx,long edx, long
  fs,long es,long ds,long eip,long
  cs,long eflags,long esp,long ss)
```

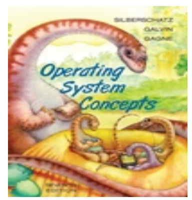


copy_process的细节：创建栈

```
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；





copy_process的细节：执行前准备

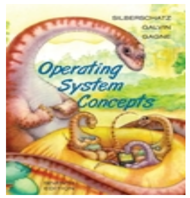
```
p->tss.eip = eip;
p->tss.cs = cs & 0xffff;
//将执行地址cs:eip放在tss中
p->tss.eax = 0;
p->tss.ecx = ecx;
//执行时的寄存器也放进去了
p->tss.ldt = _LDT(nr);
set_tss_desc(gdt+(nr<<1) +
FIRST_TSS_ENTRY, &(p->tss));
set_ldt_desc(gdt+(nr<<1) +
FIRST_LDT_ENTRY, &(p->ldt));
    //内存跟着切换
p->state = TASK_RUNNING;
```

```
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…



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)

SS： SP
EFLAGS
ret=??

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？

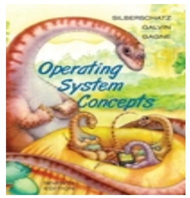
```
mov  %eax, __NR_fork
INT 0x80
mov res,%eax
```

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;
```



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

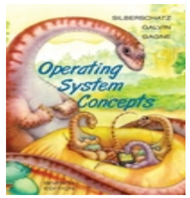
```
if(!fork()) {exec(cmd);}
```

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

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

```
EIP = 0x1C
```

+40	SS: SP
+36	EFLAGS
+28	ret=??1
+24	ds
+20	es
+16	fs
+12	edx
+8	ecx
+4	ebx
esp →	??2
	esp+0x1C



终于可以让A执行了...

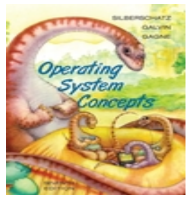
```
int do_execve( * eip,...
{   p += change_ldt(...;
    eip[0] = ex.a_entry;
    eip[3] = p; ...
```

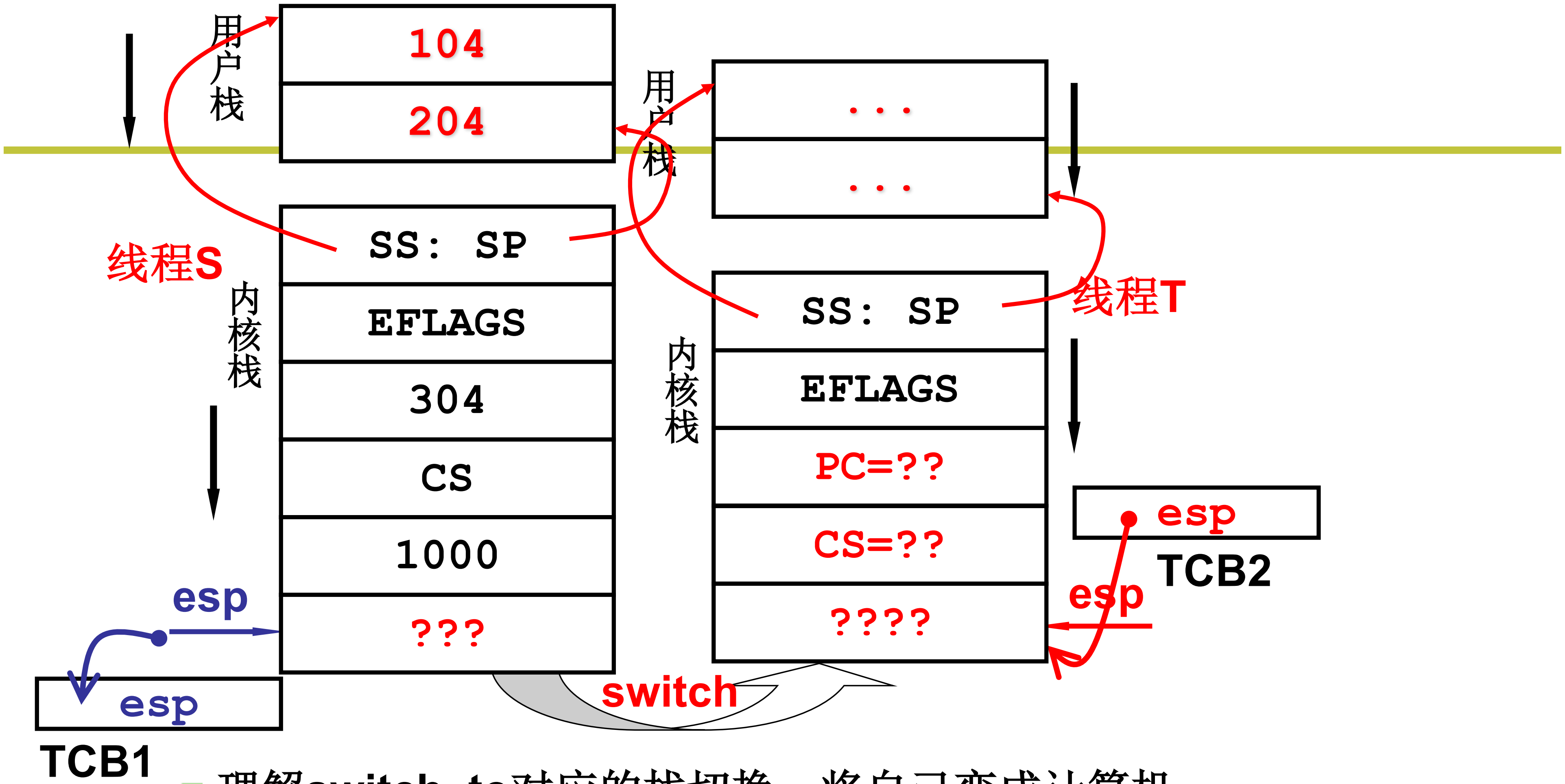
```
struct exec {
 unsigned long a_magic;
 unsigned a_entry; //入口 };
```

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

+40	SS: SP
+36	EFLAGS
+28	ret=??1
+24	ds
+20	es
+16	fs
+12	edx
+8	ecx
+4	ebx
esp →	??2
esp →	esp+0x1C

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