

哈工大操作系统-L12核心级线程实现实例

哈工大操作系统-L12核心级线程实现实例

- 1.五段论--1中断入口、5中断出口
- 2.五段论--2~4schedule中间发生了什么?
 - 2.1TSS切换
 - 2.2创建一个线程--把线程做成能切换的样子
 - 2.3子进程如何执行自己的代码

核心级线程的两套栈，核心是内核栈。

1.五段论--1中断入口、5中断出口

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);}
```

进入系统调用前，先将内核栈和用户栈关联到一起

■ 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:

reschedule:
pushl $ret_from_sys_call
jmp _schedule
```

内核栈

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

call了系统调用之后，会对比线程当前的状态是否阻塞，阻塞则呼叫reschedule函数进行切换

然后还要用counter函数检查是否时间片用完，也要进行切换

之后ret_from_sys_call是一个中断后的处理函数

- 1.中断入口：在进入中断之前，先把用户栈以及用户态的各种东西保存在内核栈中，建立用户栈和内核栈之间的关联。
- 5.中断出口：reschedule返回之后，一定是回到我们的ret_from_sys_call
 - ret_from_sys_call是中断出口，会把内核栈中的所有东西都pop出来，同时返回到某个线程调用中断的命令之后的第一个命令
- 的

2.五段论--2~4schedule中间发生了什么？

```
reschedule:pushl $ret_from_sys_call
jmp _schedule

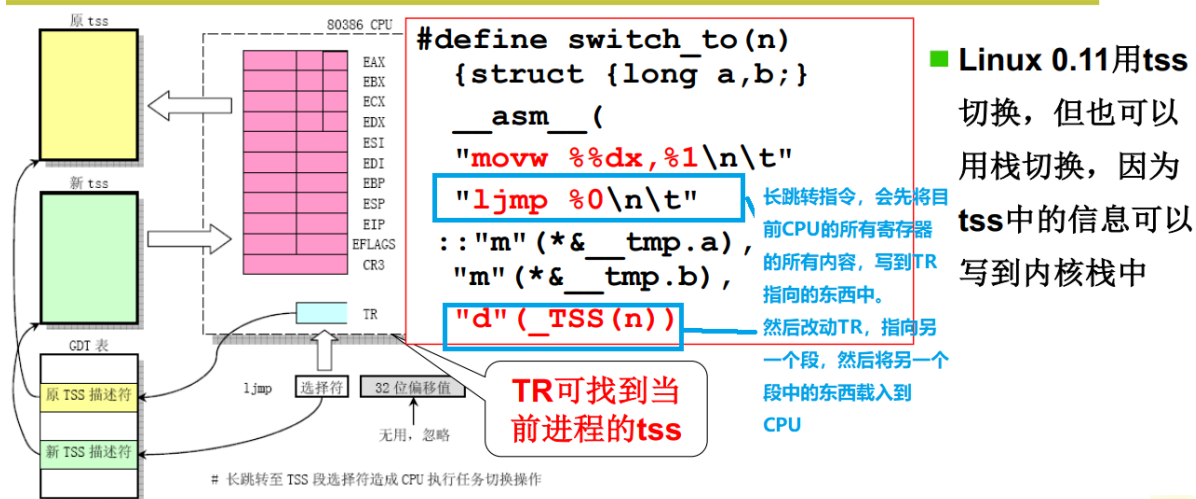
void schedule(void){ next=i;
switch_to(next); }
```

- 首先我们可以看到，进入调度程序之后，会把当前线程的中断出口函数保存在当前线程的内核栈中。
- 而后，真正进入调度程序schedule，找到应该调度的下一个线程的TCB在next中保存，然后进行switch_to
 - 如何找到next是一个调度的问题

- 然后就可以直接切换到下一个线程的TCB
- 然后内核会根据TCB完成内核栈的切换
- 内核紧接着会根据切换完成的内核栈，完成需要执行的指令序列的切换

2.1 TSS切换

切换五段论中的switch_to



但TSS切换效率低

2.2 创建一个线程--把线程做成能切换的样子

另一个故事ThreadCreate就顺了...

■ 从sys_fork开始CreateThread

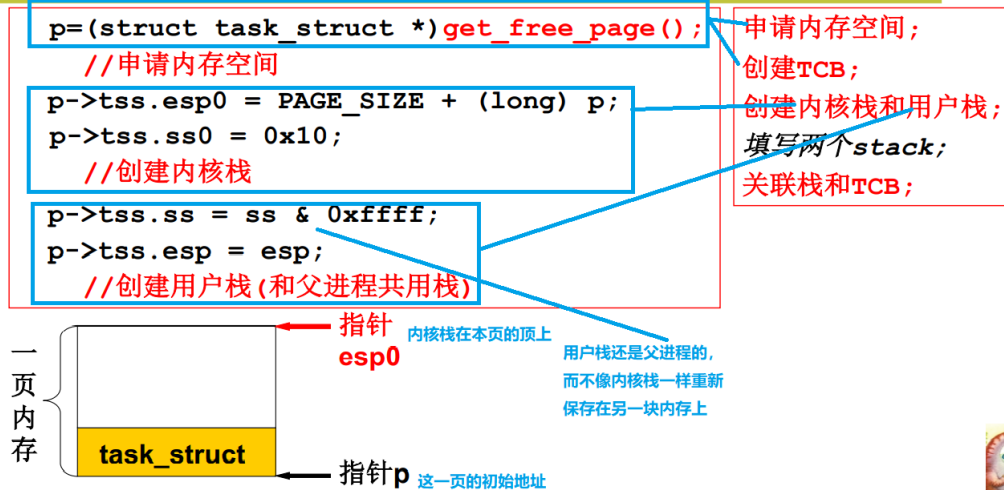
```
_sys_fork:
push %gs; pushl %esi
...
pushl %eax
call _copy_process
addl $20,%esp
ret
```

SS: SP	
EFLAGS	gs
ret=??1	esi,edi
ds,es,fs	ebp
edx,ecx,ebx	eax
??2	??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

得到一页空闲的内存，然后进行强制的类型转换，实际上创建了一个TCB



2.3子进程如何执行自己的代码

结构：子进程进入A，父进程等待...

子进程创建后，故事要从exec这个系统调用开始
fork这边返回是0，
因此会执行exec

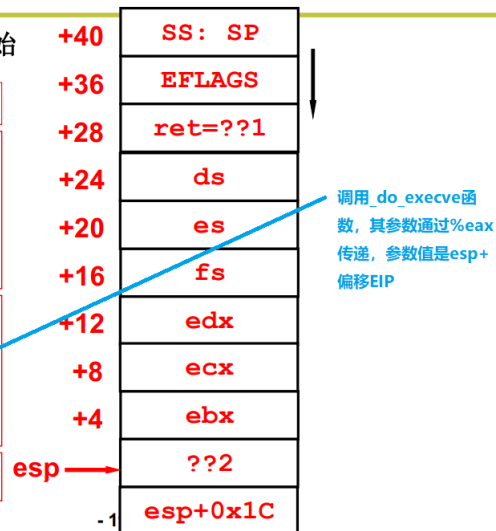
```
if(!fork()) {exec(cmd);}
```

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

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

EIP = 0x1C

Operating Systems



终于可以让A执行了...

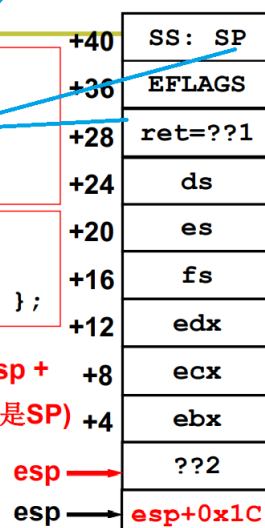
记住我们传进来的eip是刚刚%eax中的参数，为esp+0x1C

```
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)

将ret中的内容放置成为ex.a_entry
子进程就开始执行自己的代码了



ex.a_entry是可执行程序入口地址，产生可执行文件时写入...

每个进程的代码中应该有一句话if(!fork()){。这样子就能区分，如果是从该进程创建了子进程，那么子进程会执行什么东西。

