操作系统实验四

前言

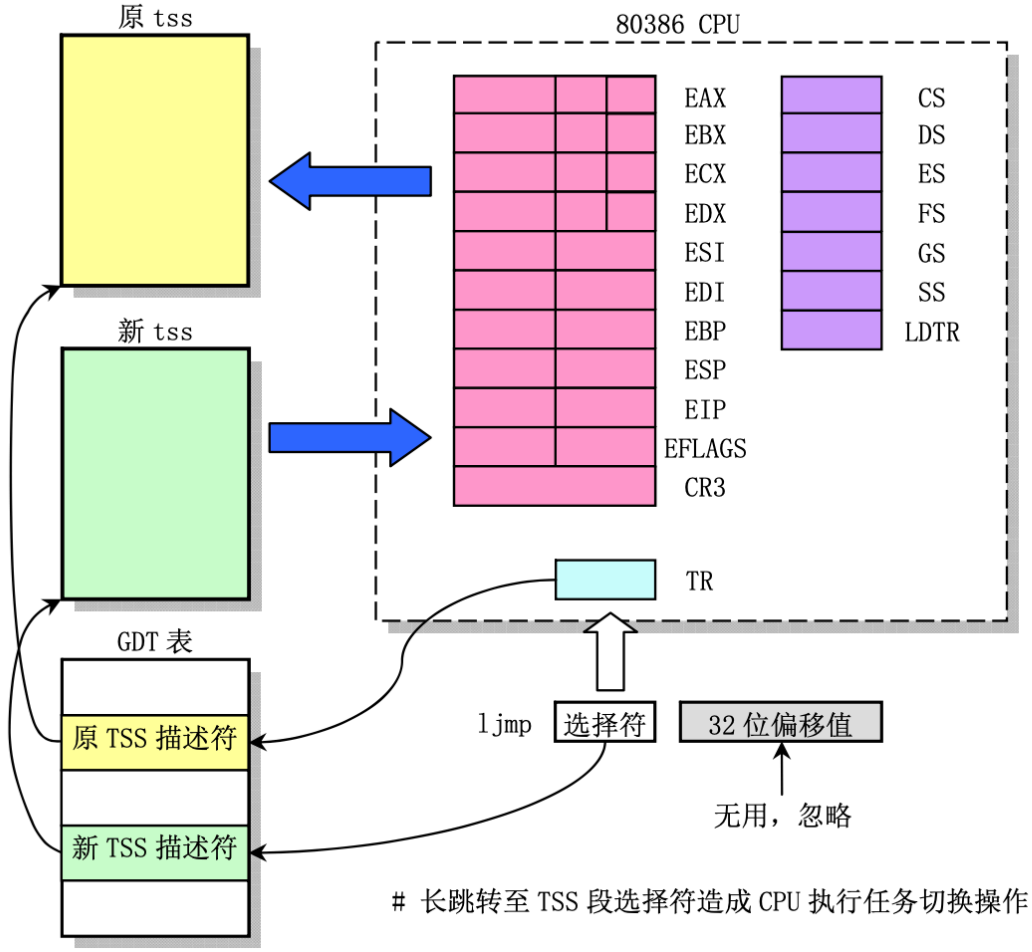
这个实验很难，我做了差不多两天差不多理解了，最后还很奇怪的出现一个问题，这个问题稍后再说，我还不确定是我的实验环境问题还是说这个实验本身切换成内核栈切换进程会导致这个问题，稍后再说，系统还是能正常启动，除了这个以外前面编写的iam，whoami，或者是前面的process程序，还是系统内置的ls等等都可以正常运行，实验手册和网上和课程的相关信息可能不是很完整，这份报告也会再说明一次这个整个流程是怎么实现的。

实验四-基于内核栈的进程切换

这个实验的主要目的是将Linux 0.11现在依靠TSS切换进程切换到自己实现的一个基于内核栈的切换程序，按照实验手册的说明，自己这样实现会有很多好处，这里不再赘述，下面就来看是如何实现的。

现在的Linux 0.11是怎么切换的

先给出一个图片，大致上来说明Intel给的奇妙指令：



这就是任务切换示意图，其中有很重要的表是GDT表，应该是全局描述符表，里面有TSS描述符，TSS从途中大致上我们可以知道，TSS在任务切换的时候，相当于几乎将所有的寄存器都保存进去，让我们看一下TSS在代码中的的声明：

```
struct tss_struct
{
    long back_link; /* 16 high bits zero */
    long esp0;
    long ss0; /* 16 high bits zero */
    long esp1;
    long ss1; /* 16 high bits zero */
    long esp2;
    long ss2; /* 16 high bits zero */
    long cr3;
    long eip;
    long eflags;
    long eax, ecx, edx, ebx;
    long esp;
    long ebp;
    long esi;
    long edi;
    long es;          /* 16 high bits zero */
    long cs;          /* 16 high bits zero */
    long ss;          /* 16 high bits zero */
    long ds;          /* 16 high bits zero */
    long fs;          /* 16 high bits zero */
    long gs;          /* 16 high bits zero */
    long ldt;         /* 16 high bits zero */
    long trace_bitmap; /* bits: trace 0, bitmap 16-31 */
    struct i387_struct i387;
};
```

可以看到起几乎保存了所有的寄存器，相当于给现在的状态拍了一张快照，之后任务切换的时候就靠这个来切换不同的任务，我好像记得这个i387是协处理器之类的，其中值得注意的是esp0，和ss0，要想形成一套栈，需要依靠这两个寄存器来在内核栈和用户栈建立联系，总体上要实现下面的步骤：

1. 读`TR`寄存器，里面保存了`TSS`段的选择子，利用该选择子在GDT表中找到运行进程的TSS段的内存位置。
2. 在`TSS`段中找到新特权级相关的栈段和栈指针即`ss0`和`esp0`，将它们装载到`ss`和`esp`寄存器。
3. 在新的栈中保存`ss`和`esp`以前的值，这些值定义了旧特权级相关的栈的逻辑地址。形象点说，就是在新栈和旧栈之间拉了一条线，形成了一套栈。
4. 如果故障已发生，用引起异常的指令地址装载`cs`和`eip`寄存器，从而使得这条指令能再次执行。
5. 在栈中保存`eflags`,`cs`,`eip`的内容。
6. 装载`cs`和`eip`寄存器，其值分别是IDT表中由中断号指定的门描述符的段选择符和偏移量字段。这些值给出了中断或异常处理程序的第一条指令的逻辑地址。

这样就实现了任务的切换，但是看到这里，还是不会理解内核那些代码是在玩什么，于是还要仔细分析看代码，首先我们给出Linux 0.11中关于任务切换的代码，也就是老switch_to：

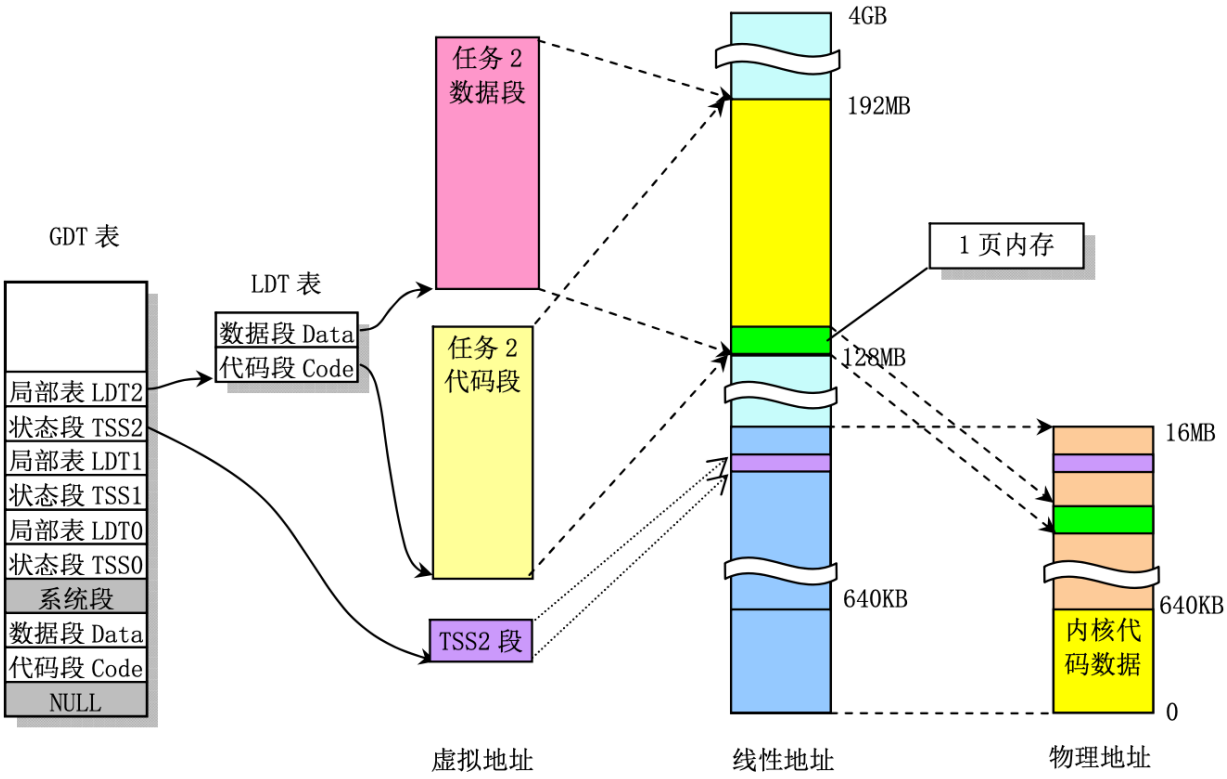
```
#define switch_to(n)                                            \
    {                                                           \
        struct                                                  \
        {                                                       \
            long a, b;                                          \
        } __tmp;                                                \
        __asm__("cmpl %%ecx,current\n\t"                        \
                "je 1f\n\t"                                     \
                "movw %%dx,%1\n\t"                              \
                "xchgl %%ecx,current\n\t"                       \
                "ljmp *%0\n\t"                                  \
                "cmpl %%ecx,last_task_used_math\n\t"            \
                "jne 1f\n\t"                                    \
                "clts\n"                                        \
                "1:" ::"m"(*&__tmp.a),                          \
                "m"(*&__tmp.b),                                 \
                "d"(_TSS(n)), "c"((long)task[n]));              \
    }
```

被奇怪的vscode格式化乱了，但是大致上还是能看懂是啥，这里有一段核心的内嵌汇编的代码，差不多意思是这样：

```
                //比较当前要切换的进程是否是当前运行的进程
__asm__("cmpl %%ecx,current\n\t"                        \
                //如果是则不调度直接退出，也就是到下面的1
                "je 1f\n\t"                                     \
                //将要调度的tss指针存到tmp.b中，也即是新任务TSS的16位选择符
                "movw %%dx,%1\n\t"                              \
                //交换pcb值，要调度的存储在ecx中
                //current = task[n]; ecx=被切换出去的任务
                "xchgl %%ecx,current\n\t"                       \
                //进行长跳转，即切换tss，执行长跳转到 *&_tmp
                "ljmp *%0\n\t"                                  \
                //进行切换后的处理协处理器
                "cmpl %%ecx,last_task_used_math\n\t"            \
                //处理完了，还是跳到1部分
                "jne 1f\n\t"                                    \
                "clts\n"                                        \
                "1:" ::"m"(*&__tmp.a),                          \
                "m"(*&__tmp.b),                                 \
                "d"(_TSS(n)), "c"((long)task[n]));              \
    }
```

这样就更进一步，知道了现在的Linux 0.11是怎么切换进程的，但是实际上还是懵，因为不知道的东西还有很多，首先来看，长跳转的时候这个地址是怎么算出来的，这就需要我们知道GDT表的结构：

GDT 表的结构如下图所示，所以第一个 TSS 表项，即 0 号进程的 TSS 表项在第 4 个位置上，4<<3，即 `4 * 8`，相当于 TSS 在 GDT 表中开始的位置，TSS（n）找到的是进程 n 的 TSS 位置，所以还要再加上 n<<4，即 `n * 16`，因为每个进程对应有 1 个 TSS 和 1 个 LDT，**每个描述符的长度都是 8 个字节**，所以是乘以 16，其中 LDT 的作用就是上面论述的那个映射表，关于这个表的详细论述要等到内存管理一章。`TSS(n) = n * 16 + 4 * 8`，得到就是进程 n（切换到的目标进程）的 TSS 选择子，将这个值放到 dx 寄存器中，并且又放置到结构体 tmp 中 32 位长整数 b 的前 16 位，现在 64 位 tmp 中的内容是前 32 位为空，这个 32 位数字是段内偏移，就是 `jmpi 0, 8` 中的 0；接下来的 16 位是 `n * 16 + 4 * 8`，这个数字是段选择子，就是 `jmpi 0, 8` 中的 8，再接下来的 16 位也为空。所以 swith_to 的核心实际上就是 `ljmp 空, n*16+4*8`，现在和前面给出的基于 TSS 的进程切换联系在一起了。



之后反应到系统代码呢，我们发现这个`_LDT(n)`这个宏，追溯回去，其实是：

```
1 #define _LDT(n) ((((unsigned long)n) << 4) + (FIRST_LDT_ENTRY << 3))
```

而我们再找相关的定义，发现这个：

```
1 #define FIRST_TSS_ENTRY 4
2 #define FIRST_LDT_ENTRY (FIRST_TSS_ENTRY + 1)
```

终于我们知道了这里是在干什么，要不然是会很奇怪，这个地址是怎么算出来的。

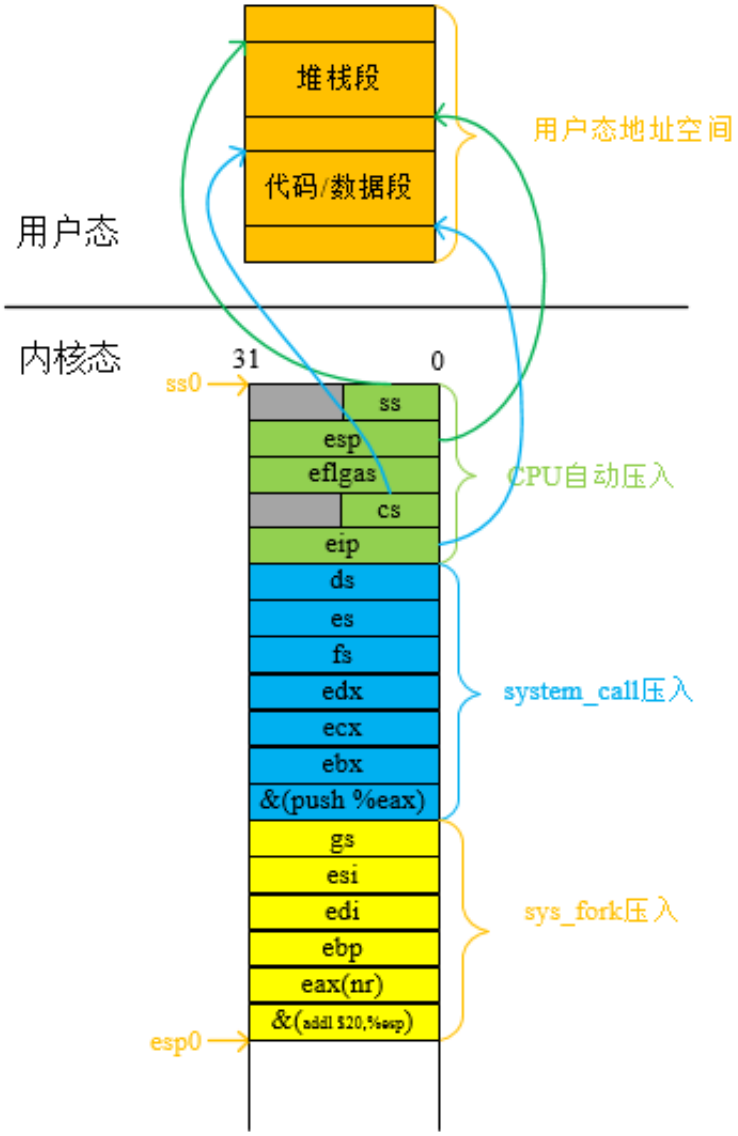
我们要怎么修改

这里为了理解我认为不能使用实验手册的顺序看，很容易就不懂手册里面switch_to到底是在干什么。

首先第一步要知道，除了操作系统第一个进程是操作系统手工创建的以外，其他的进程都是通过fork得到的，在main函数中可以清晰的看到，操作系统会首先move到用户空间，之后调用fork来进行操作，之后回忆fork，fork是一个系统调用，之后就可以回想起来一长串事情，比如int 0x80中断进行系统调用。这样首先要来看system_call在干什么：

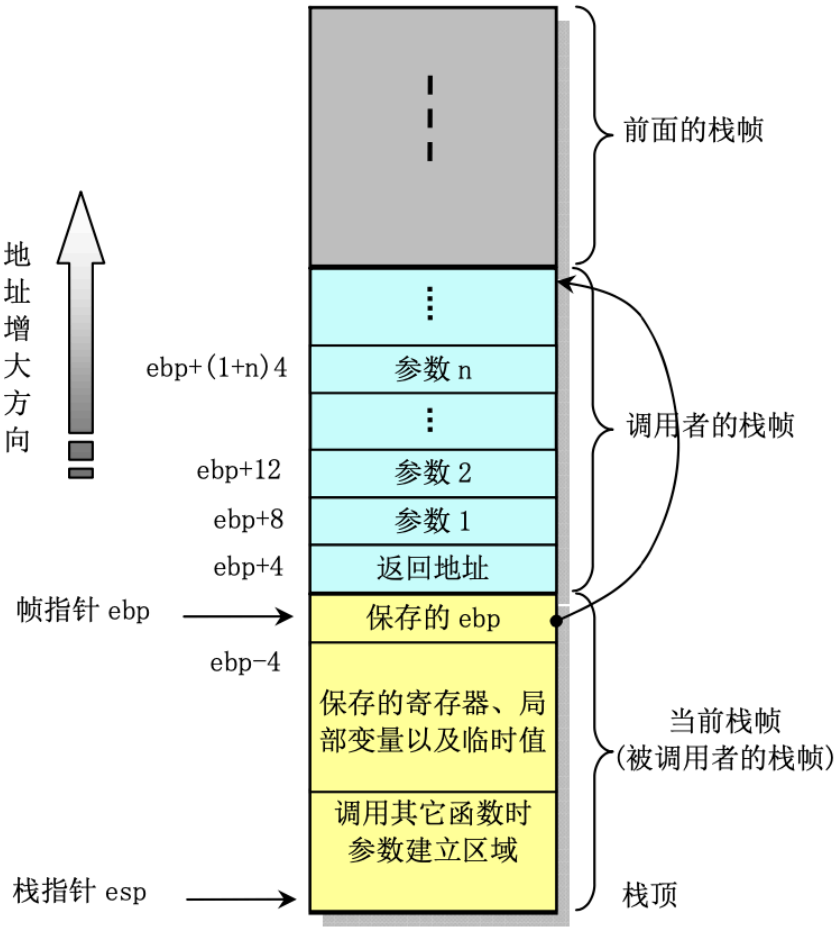
```
system_call:
    cmpl $nr_system_calls-1,%eax
    ja bad_sys_call
    push %ds
    push %es
    push %fs
    pushl %edx
    pushl %ecx      # push %ebx,%ecx,%edx as parameters
    pushl %ebx      # to the system call
    movl $0x10,%edx     # set up ds,es to kernel space
    mov %dx,%ds
    mov %dx,%es
    movl $0x17,%edx     # fs points to local data space
    mov %dx,%fs
    call sys_call_table(,%eax,4)
    pushl %eax
    movl current,%eax
    cmpl $0,state(%eax)     # state
    jne reschedule
    cmpl $0,counter(%eax)       # counter
    je reschedule
```

这就是system_call，首先确定一个事情，就是CPU在int 0x80中断之后，会自动按照下面顺序压栈：SS-ESP-EFLAG-CS-EIP，下面给出一个图来说明这些寄存器是干什么的，（致谢 Wangzhike）：



这幅图很好的说明了ss，esp，cs，eip是干什么的，但是还是需要在细化一点，实际上EIP是在int 0x80的下一条语句，相当于到返回到这个地方，而int 0x80在这里压内核栈的时候，ss这些都是用户态中的数据，相当于压进去了用户态下堆栈的位置，这样才能知道，为什么这里说是建立了联系，练了线成了一套栈，之后就到了system_call中的操作了，找调用这些操作，这里又出现了一个问题，是怎么调用函数的呢，这又是一个新问题，需要理解操作系统的栈帧结构：

大部分CPU上程序使用栈来支持函数调用操作，栈用来传递函数参数，存储返回信息，临时保存寄存器原有值以备回复已经用来存储局部数据。单个函数调用操作所使用的栈的部分被称作栈帧结构，其通常结构件下图：



栈帧结构两端用两个指针来制定，其中ebp常用来作为帧指针，而esp用作栈指针，在函数执行中，栈指针esp会随着数据的入栈和出栈而移动，对函数大部分的访问都基于ebp进行。

比如函数A调用函数B，传递给B的参数包含在A的栈帧中，当A调用B时，函数A的返回地址（其实就是调用之后要去那里执行）被压入栈中，栈中该位置也明确指明了A栈帧的结束处，而B的栈帧从随后的栈部分开始，即途中保存帧指针ebp的地方开始（这里小心，注意地址是向上增大，但是指针是向下的，后面栈指针用--原因差不多也是一样的）。

之后回忆主体，我们是在说fork啊，先不考虑fork实现，假设我们已经又了一个健全的fork函数，之后现在也有了新的进程，之后在system_call中，就会开始分析，这个进程需不需要被调度，如果需要调度的话，这个时候我们switch_to的故事就要展开了：

```
# 需要调度
reschedule:
    pushl $ret_from_sys_call
    jmp schedule
```

调度

接着上面说的，现在我们就确定我们要开始调度了，这时候我们在看一眼栈：

内核栈	说明
SS	
ESP	
EFLAGS	
CS	
EIP	到这里前面几个是CPU干的
DS	push %ds
ES	push %es
FS	push %fs
EDX	push %eds
ECX	push %ecx
EBX	push %ebx
EAX	pushl %eax
ret_from_sys_call	

我比较菜，这里就用表格来看了，后面的备注说明了是谁把内核栈征程这个样子的，这里很显然，在去调度之前要记录一下返回到那里，也就是返回到ret_from_sys_call这个地方，这样子才能从这整个流程中出来。

之后我们的调度程序找到下一个进程之后，就会想办法切换过去，于是核心函数就来了，要实现这个转换，也就是switch_to函数，首先大体上来说明要怎么切换，首先切换PCB，其实就是前面说的将current指针指向下一个PCB，之后是TSS内核栈指针的重写，虽然我们不使用内置的TSS那个长跳转切换了，但是我们还是要让CPU找到内核栈，之后是切换内核栈，还要切换LDT，之后是PC指针的切换，也就是CS和EIP的切换，首先要注意到的是C语言怎么调用汇编程序传参数，其实是通过EBP来传递的，这其实就是为什么实验手册中给出来的前面几个要压入ebp，其实也是调用函数，这里将最后修改完成的switch_to放在这里：

```
switch_to:
    pushl %ebp
    movl %esp,%ebp
    pushl %ecx
    pushl %ebx
    pushl %eax
    movl 8(%ebp),%ebx
    cmpl %ebx,current
    je 1f
# 切换PCB
    movl %ebx,%eax
        xchgl %eax,current
```

```
# TSS中的内核栈指针的重写
    movl tss,%ecx
        addl $4096,%ebx
        movl %ebx,ESP0(%ecx)
# 切换内核栈，注意这里就真的切换到子进程到内核堆栈了
# 看这里会把堆栈指针移动过去
    movl %esp,KERNEL_STACK(%eax)
# 再取一下 ebx，因为前面修改过 ebx 的值
        movl 8(%ebp),%ebx
        movl KERNEL_STACK(%ebx),%esp
# 切换LDT
        movl 12(%ebp),%ecx
        lldt %cx
# 切换完之后
    movl $0x17,%ecx
    mov %cx,%fs
# 和后面的 clts 配合来处理协处理器，由于和主题关系不大，此处不做论述
    cmpl %eax,last_task_used_math
    jne 1f # viewvos test direct ret not popl
    clts
1:  popl %eax
    popl %ebx
    popl %ecx
    popl %ebp
        ret
```

这里的`movl 8(%ebp),%ebx`就是取出来switch_to的第二个参数，其实就是_LDT(next)，同理12就是第一个参数，前面已经说过了，这个时候我们再看一眼内核栈会变成什么样子：

内核栈	说明
SS	
ESP	
EFLAGA	
CS	
EIP	CPU干的
DS	
ES	
EDX	
ECX	
EBX	
EAX	
ret_from_sys_call	system_call干的
pnext	switch_to第一个参数
_LDT(next)	第二个参数
}	其实C语言这个}相当于返回
ebp	帧指针
ecx	
ebx	
eax	esp也指向了这里

之后就要开始切换，这里就按照实验手册所写的开始切换，首先完成PCB的切换，实际上是将current指向新的PCB：

```
movl %ebx,%eax
xchgl %eax,current
```

之后是TSS的切换，当从用户态进入内核态时，CPU会自动依靠TR寄存器找到当前进程的TSS，然后根据里面ss0和esp0的值找到内核栈的位置，完成用户栈到内核栈的切换。所以仍需要有一个当前TSS，我们需要在schedule.c中定义`struct tss_struct *tss=&(init_task.task.tss)`这样一个全局变量，即0号进程的tss，所有进程都共用这个tss，任务切换时不再发生变化。 虽然所有进程共用一个tss，但不同进程的内核栈是不同的，所以在每次进程切换时，需要更新tss中esp0的值，让它指向新的进程的内核栈，并且要指向新的进程的内核栈的栈底，即要保证此时的内核栈是个空栈，帧指针和栈指

针都指向内核栈的栈底。 这是因为新进程每次中断进入内核时，其内核栈应该是一个空栈。为此我们还需要定义：`ESP0 = 4`，这是TSS中内核栈指针esp0的偏移值，以便可以找到esp0。具体实现代码如下：

```
movl tss,%ecx
addl $4096,%ebx
movl %ebx,ESP0(%ecx)
```

之后是内核栈的切换：

> Linux 0.11的PCB定义中没有保存内核栈指针这个域(kernelstack)，所以需要加上，而宏KERNEL_STACK就是你加的那个位置的偏移值，当然将kernelstack域加在task_struct中的哪个位置都可以，但是在某些汇编文件中（主要是在system_call.s中）有些关于操作这个结构一些汇编硬编码，所以一旦增加了kernelstack，这些硬编码需要跟着修改，由于第一个位置，即long state出现的汇编硬编码很多，所以kernelstack千万不要放置在task_struct中的第一个位置，当放在其他位置时，修改system_call.s中的那些硬编码就可以了。

修改结构体这里就不说了，按照实验手册修改就可以了，值得注意的是在内核栈修改了之后，我们这时候后面就真的针对子进程来操作了，要说明这一点是想后面说fork的时候，fork为什么不只fork 复制父进程里面那前5个CPU自动压入的寄存器呢，这里应该可以看到切换到父进程之后，还需要的操作就需要子进程也有相关的信息了。

```
movl %esp,KERNEL_STACK(%eax)
movl 8(%ebp),%ebx
movl KERNEL_STACK(%ebx),%esp
```

其他细节还有修改0号进程PCB初始化的内容，这里就不细说了。

最后是LDT的切换，一旦修改完成，下一个进程在执行用户态程序时使用的映射表就是自己的LDT表了，地址分离实现了。

```
movl 12(%ebp),%ecx
lldt %cx
```

ire 指令完成用户栈切换

切换是有了，但是我们需要能够被切换的进程，在fork的时候，我们需要自己造一个进程出来，这个时候就需要修改copy_process()函数，对于被切换出去的进程，当它再次被调度执行时，根据被切换出去的进程的内核栈的样子，switch_to的最后一句指令`ret`会弹出switch_to()后面的指令`}`作为返回返回地址继续执行，从而执行`}`从schedule()函数返回，将弹出`ret_from_sys_call`作为返回地址执行ret_from_sys_call，在ret_from_sys_call中进行一些处理，最后执行`iret`指令，进行中断返回，将弹出原来用户态进程被中断地方的指令作为返回地址，继续从被中断处执行。 对于得到CPU的新的进程，我们要修改fork.c中的copy_process()函数，将新的进程的内核栈填写成能进行PC切换的样子。按照实验手册的提示，子进程的栈应该长成这个样子：

内核栈
SS
ESP
EFLAG
CS
EIP
DS
FS
GS
ESI
EDI
EDX
First_return_from_kernel
ebp
ecx
ebx
eax

其实看copy_process这个函数头：

```
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)
```

其实就这么多寄存器，fs是用户空间的，前面说到已经切换到内核栈了，再使用内核栈的时候就是子进程的内核栈了，当然子进程的栈里面就需要保存这些信息，如果子进程的栈中没有这些信息，那么switch_to下面就没有办法运行，后面到1的弹栈也没有办法做，所以才需要保存这么多寄存器。

这样子就可以给出来子进程的堆栈需要有啥：

```
*(--kstack) = ss & 0xffff;
*(--kstack) = esp;
*(--kstack) = eflags;
*(--kstack) = cs & 0xffff;
*(--kstack) = eip;
```

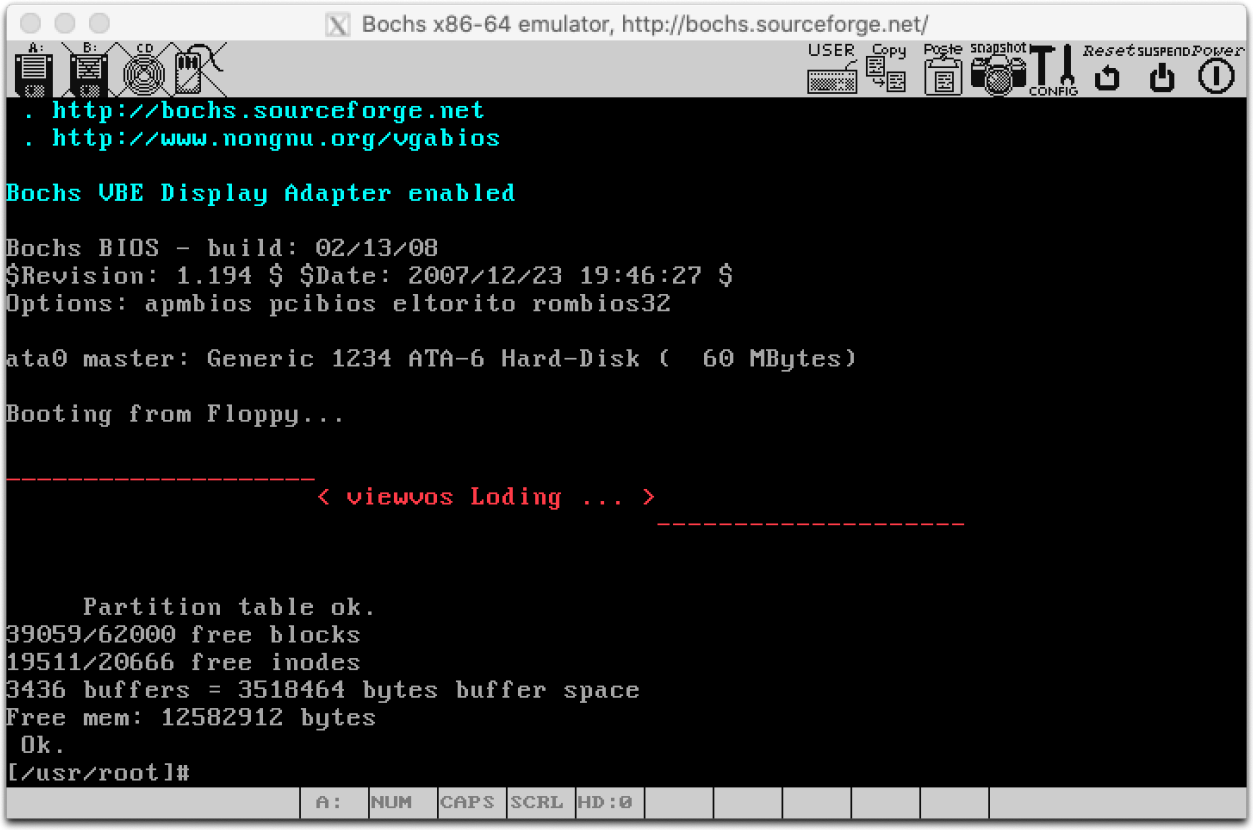
```
*(--kstack) = ds & 0xffff;
*(--kstack) = es & 0xffff;
*(--kstack) = fs & 0xffff;
*(--kstack) = gs & 0xffff;
*(--kstack) = esi;
*(--kstack) = edi;
*(--kstack) = edx;
//* finish copy ass first_return_from_kernel
*(--kstack) = (long)first_return_from_kernel;
*(--kstack) = ebp;
*(--kstack) = ecx;
*(--kstack) = ebx;
*(--kstack) = 0;
p->kernelstack = (long)kstack;
```

最后回答问题的时候再说明一次细节，但是大体上对应前面我们需要的子进程栈，发现就是这样写下来的，至于`first_return_from_kernel`要做些什么呢，PCB 切换完成、内核栈切换完成、LDT 切换完成，接下来应该那个“内核级线程切换五段论”中的最后一段切换了，即完成用户栈和用户代码的切换，依靠的核心指令就是 iret，当然在切换之前应该回复一下执行现场，主要就是`eax,ebx,ecx,edx,esi,edi,gs,fs,es,ds`等寄存器的恢复，其实就是一串pop弹栈：

```
first_return_from_kernel:
    popl %edx
    popl %edi
    popl %esi
    pop %gs
    pop %fs
    pop %es
    pop %ds
    iret
```

这里iret其实就是中断返回，其会返回CS，EFLAG，EIP，SS这些参数，所以栈中需要准备好这些参数切换，因为新进程中没有进过int，所以需要我们准备好这些参数。

大致上就是这么一个过程，还是很难理解，最后能够成功运行，这内部已经切换了好几次了。



问题回答

1. 申请一页内存也就是4K
2. 对于Linux 0.11来说，内核栈段选择子永远都是0x10，这里就不需要修改这个值
3. 子进程第一次执行要返回0，也就是eax=0，才能和父进程进行分支执行
4. ebx，ecx均来自copy_process函数的调用参数，是在sys_fork之前和之中分段进行入栈的。具体到ebx、ecx两个参数是由system_call函数入栈的。是进入内核态前程序在用户态运行时两个寄存器的状态数据。
5. 这段代码的ebp来自sys_fork中的入栈动作，这个ebp再system_call的时候也没有体现，就直接在sysy_fork中入栈，应该ebp仍然是用户态的ebp，最后返回的时候，保证能够返回到用户态栈帧，应该需要保存
6. cpu的段寄存器都存在两类值，一类是显式设置段描述符，另一类是隐式设置的段属性及段限长等值，这些值必须经由movl、lldt、lgdt等操作进行设置，而在设置了ldt后，要将fs显示设置一次才能保证段属性等值正确。