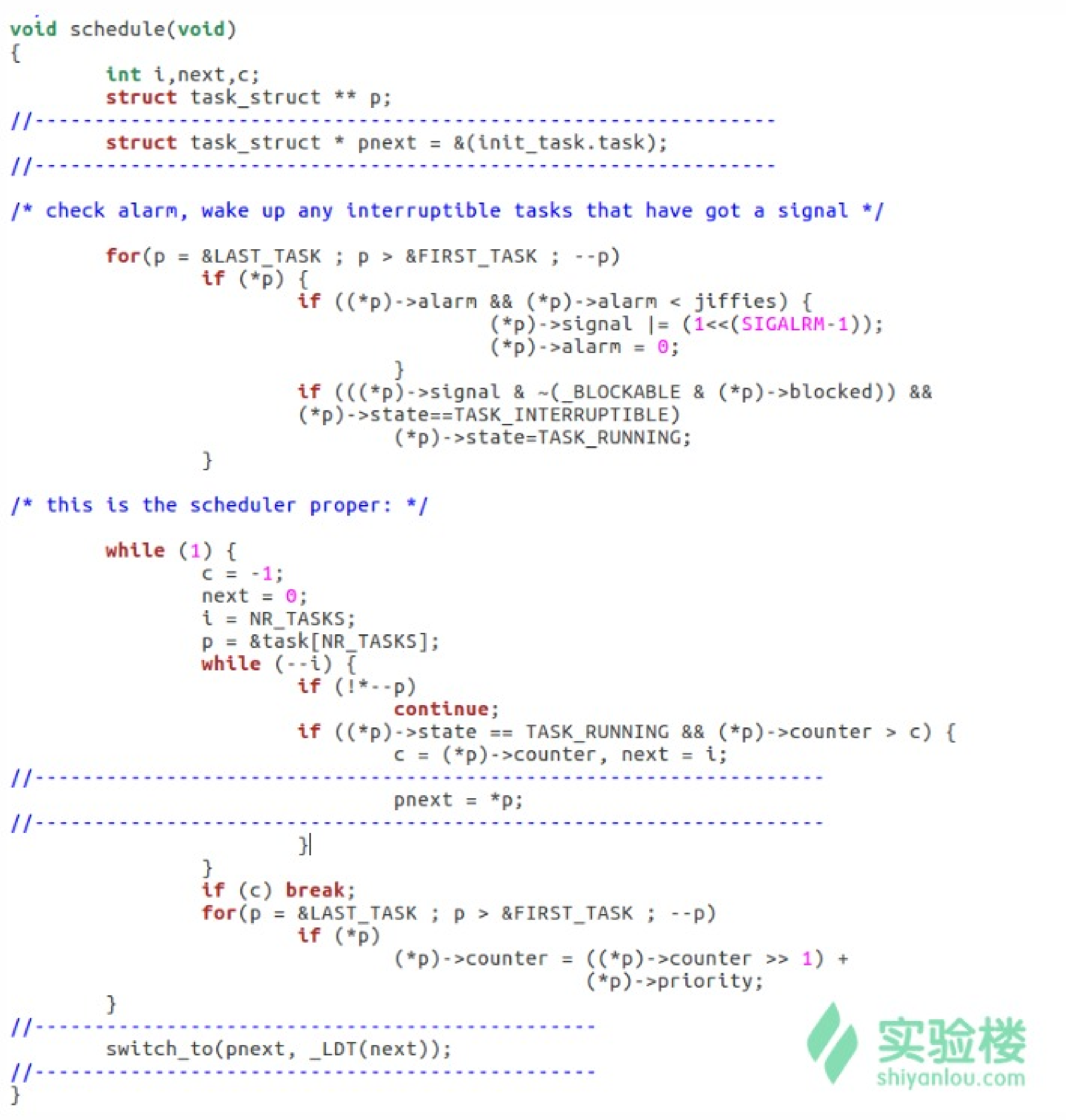
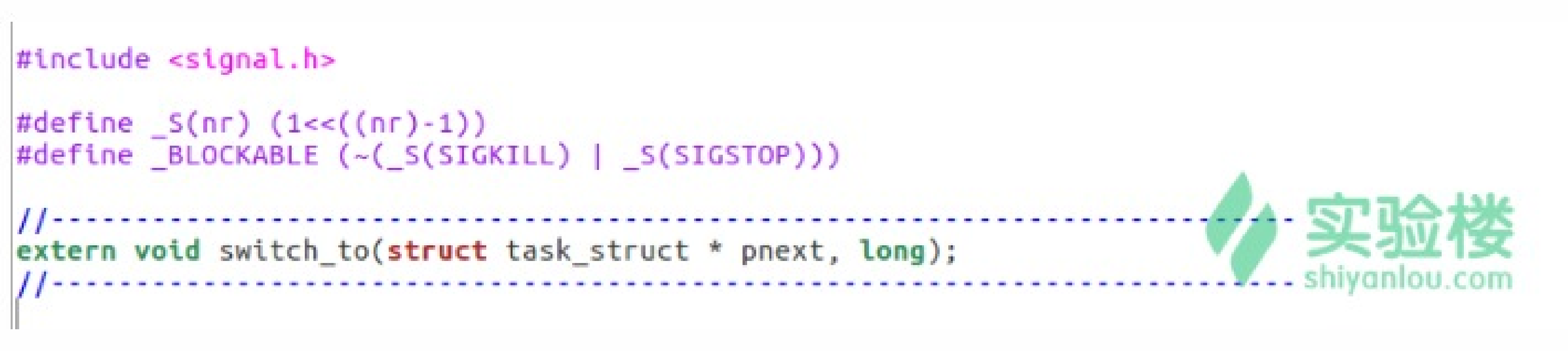
造的进程，就不单单是switch_to的事情了，还需要修改fork，尤其是其中的copy_process，让新进程能够和新switch_to配套使用。

首先要知道，想要使用堆栈来切换进程，需要用到进程的PCB，因为PCB存储着关于进程的信息，包括内核栈指针等。LDT是用来给进程寻址的，每个进程都会有一个。 如果使用TSS来切换进程的话，其实只需要能够找到每个进程相应的TSS，就能够顺利切换额，因为TSS记录着进程运行的所有信息，一步就将所有全数恢复了。但是现在不能使用TSS，所以进程的PCB和LDT由我们自己来分别传入。 此处修改schedule（）函数，传入下一进程的PCB和LDT。



```
void schedule(void)
{
        int i,next,c;
        struct task_struct ** p;
//-------------------------------------------------------------
        struct task_struct * pnext = &(init_task.task);
//-------------------------------------------------------------

/* check alarm, wake up any interruptible tasks that have got a signal */

        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                if (*p) {
                        if ((*p)->alarm && (*p)->alarm < jiffies) {
                                        (*p)->signal |= (1<<(SIGALRM-1));
                                        (*p)->alarm = 0;
                                }
                        if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
                        (*p)->state==TASK_INTERRUPTIBLE)
                                (*p)->state=TASK_RUNNING;
                }

/* this is the scheduler proper: */

        while (1) {
                c = -1;
                next = 0;
                i = NR_TASKS;
                p = &task[NR_TASKS];
                while (--i) {
                        if (!*--p)
                                continue;
                        if ((*p)->state == TASK_RUNNING && (*p)->counter > c) {
                                c = (*p)->counter, next = i;
//-----------------------------------------------------------------
                                pnext = *p;
//-----------------------------------------------------------------
                        }
                }
                if (c) break;
                for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                        if (*p)
                                (*p)->counter = ((*p)->counter >> 1) +
                                                (*p)->priority;
        }
//----------------------------------------
        switch_to(pnext, _LDT(next));
//----------------------------------------
}
```

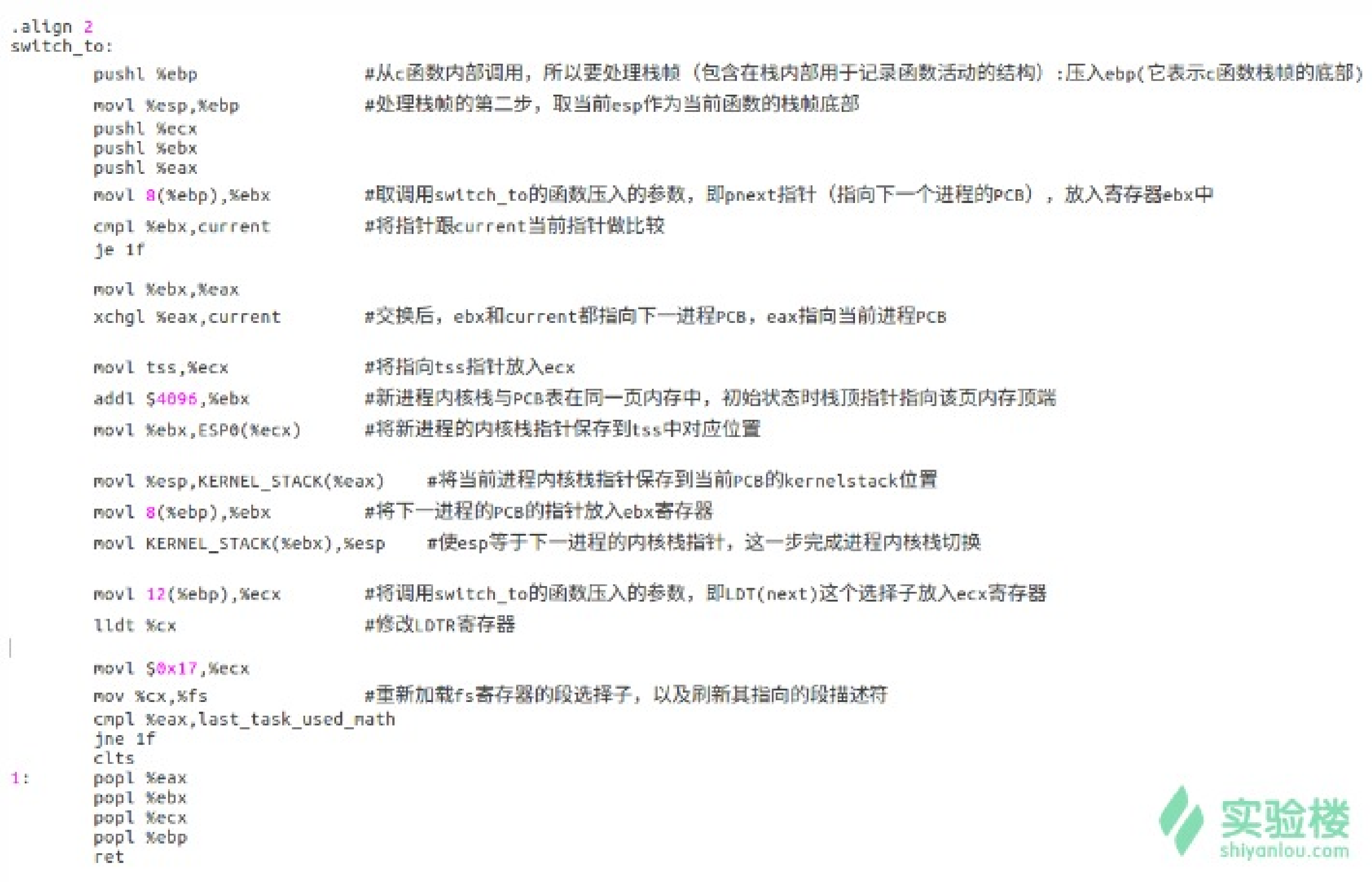
还要注意声明switch_to是外部函数



```
#include <signal.h>

#define _S(nr) (1<<((nr)-1))
#define _BLOCKABLE (~(_S(SIGKILL) | _S(SIGSTOP)))

//-----------------------------------------------------------------------
extern void switch_to(struct task_struct * pnext, long);
//-----------------------------------------------------------------------
```

注意此处pnext一定要初始化（参考了别的同学的作业才知道这个点），这里初始化为进程0的PCB指针。我认为必须初始化的原因是：pnext不初始化，如果在进程队列中没有可以切换的进程，那么它作为一个未初始化的指针被传入switch_to，就会出错。但是实际上是不是这个原因我也不确定，反正我没初始化之前系统死掉了就对了。。。。

接下来再来看switch_to具体如何实现，它写在system_call.s 中。



```
.align 2
switch_to:
        pushl %ebp              #从c函数内部调用，所以要处理栈帧（包含在栈内部用于记录函数活动的结构）:压入ebp(它表示c函数栈帧的底部)
        movl %esp,%ebp          #处理栈帧的第二步，取当前esp作为当前函数的栈帧底部
        pushl %ecx
        pushl %ebx
        pushl %eax
        movl 8(%ebp),%ebx       #取调用switch_to的函数压入的参数，即pnext指针（指向下一个进程的PCB），放入寄存器ebx中
        cmpl %ebx,current       #将指针跟current当前指针做比较
        je 1f

        movl %ebx,%eax
        xchgl %eax,current      #交换后，ebx和current都指向下一进程PCB，eax指向当前进程PCB

        movl tss,%ecx           #将指向tss指针放入ecx
        addl $4096,%ebx         #新进程内核栈与PCB表在同一页内存中，初始状态时栈顶指针指向该页内存顶端
        movl %ebx,ESP0(%ecx)    #将新进程的内核栈指针保存到tss中对应位置

        movl %esp,KERNEL_STACK(%eax)    #将当前进程内核栈指针保存到当前PCB的kernelstack位置
        movl 8(%ebp),%ebx       #将下一进程的PCB的指针放入ebx寄存器
        movl KERNEL_STACK(%ebx),%esp    #使esp等于下一进程的内核栈指针，这一步完成进程内核栈切换

        movl 12(%ebp),%ecx      #将调用switch_to的函数压入的参数，即LDT(next)这个选择子放入ecx寄存器
        lldt %cx                #修改LDTR寄存器

        movl $0x17,%ecx
        mov %cx,%fs             #重新加载fs寄存器的段选择子，以及刷新其指向的段描述符
        cmpl %eax,last_task_used_math
        jne 1f
        clts
1:      popl %eax
        popl %ebx
        popl %ecx
        popl %ebp
        ret
```

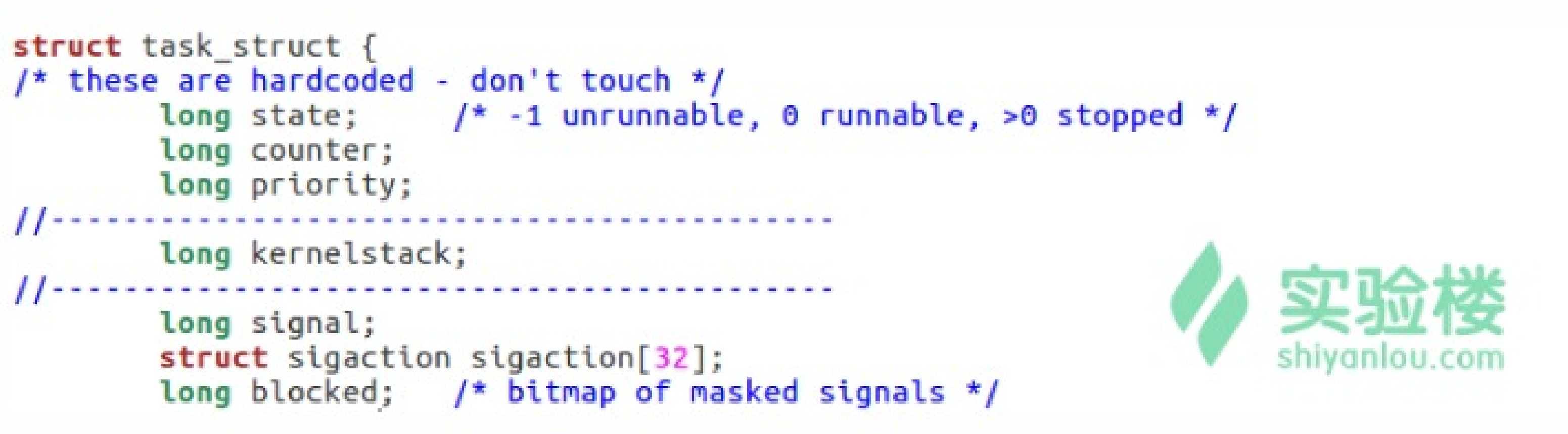
这里有几个需要注意的点，第一个是TSS，为什么不用TSS切换还要处理它，将新进程内核栈的指针记录在TSS中呢？这里的原因是，TSS虽然不用于切换进程，但是每个进程要工作，仍需要用到TSS（比如进程在日常工作切换用户栈和内核栈的时候，硬件需要依赖TSS来进行切换）。在这里，所有新进程都与进程0共同使用一个TSS，只需要在切换进程的时候把TSS记录的内核栈指针换成相应进程的指针即可。所有内核栈共用一个ss0段选择子，也就是同一套内核栈基地址和界限，所以没有设置ss0。而加上4096的原因是：进程内核栈的底部位于PCB表同一页内存的顶部。还要注意给定义tss，这个变量原来是不存在的。

```
//-------------------------------------------------------------
struct tss_struct *tss = &(init_task.task.tss);
//-------------------------------------------------------------
```

第二个是内核栈指针的保存，要知道每个PCB保存的内核栈指针在被保存之前都停留在内核栈中有eax，ebx，ecx，ebp和switch_to下一条语句所在地址的状态。当然也会有其他信息保存，这取决于进程被切换前（或者刚创建时）在进行什么工作。

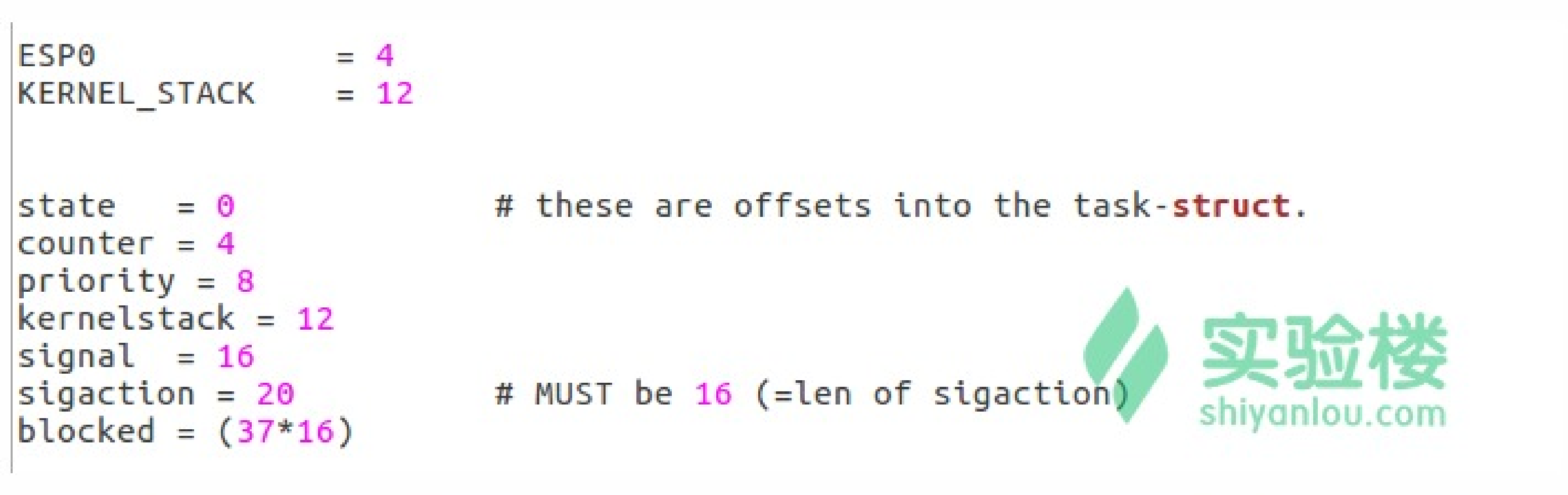
第三个是mov %cx,%fs这条语句的必要性。由于段寄存器实际上分为两个部分，一个部分用来存放段选择子，另一个部分用来存放基地址和段界限。尽管切换进程，fs的段选择子是一样的，但是其对应的基地址和段界限由于LDT的改变已经改变了，所以必须通过重新传入段选择子来刷新基地址和段界限的内容。

当然还要给PCB表加上kernelstack来存放内核栈指针。



```
struct task_struct {
/* these are hardcoded - don't touch */
        long state;     /* -1 unrunnable, 0 runnable, >0 stopped */
        long counter;
        long priority;
//----------------------------------------
        long kernelstack;
//----------------------------------------
        long signal;
        struct sigaction sigaction[32];
        long blocked;   /* bitmap of masked signals */
```

相应地修改一些常量



```
ESP0            = 4
KERNEL_STACK    = 12


state   = 0             # these are offsets into the task-struct.
counter = 4
priority = 8
kernelstack = 12
signal  = 16
sigaction = 20          # MUST be 16 (=len of sigaction)
blocked = (37*16)
```

在进程队列中旧有的进程依靠上述的修改就可以实现利用内核栈切换了。但是新创建的进程要顺利运行还需要修改fork，因为原有的fork是对父进的TSS的复制（除了eax）。要能够利用内核栈，则必须创造出内核在能够使用的结构才行。更具体地说，就是忘栈中填写一些特定的内容，使得弹栈和ret/iret配合新switch_to。 由于栈是后进先出的，所以从顺序上来说应该是： 从first_return_from_kernel到0对应switch_to的最后几个弹栈语句和ret。 从ds到edx是为了将父进程工作环境复制成为子进程的工作环境。 从ss到eip，是为了创造出从内核态到用户态的切换桥梁，中断返回iret弹出这些数据，并回到用户态。

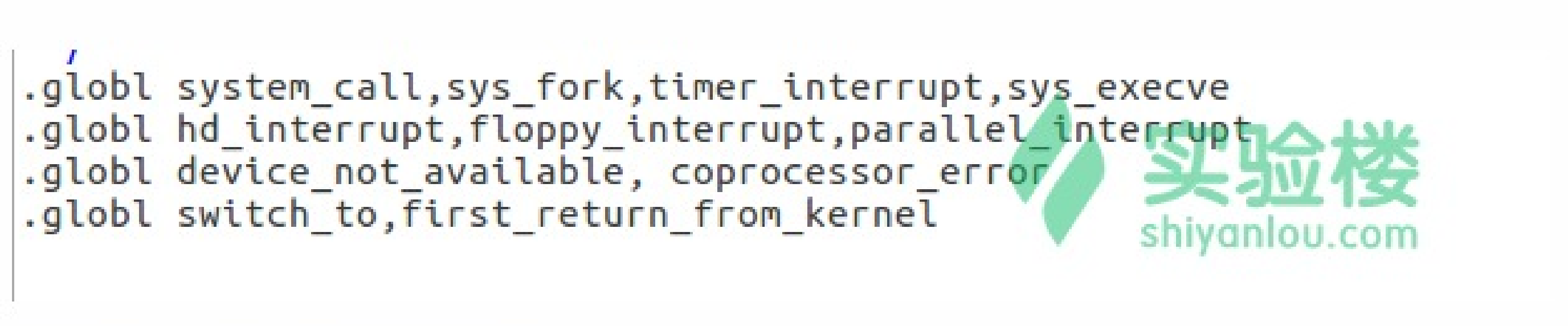
```
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
                long ebx,long ecx,long edx,
                long fs,long es,long ds,
                long eip,long cs,long eflags,long esp,long ss)
{
        struct task_struct *p;
        int i;
        struct file *f;
        long* krnstack;

        p = (struct task_struct *) get_free_page();
        if (!p)
                return -EAGAIN;
        task[nr] = p;
        *p = *current;  /* NOTE! this doesn't copy the supervisor stack */
        p->state = TASK_UNINTERRUPTIBLE;
        p->pid = last_pid;
        p->father = current->pid;
        p->counter = p->priority;

        krnstack = (long)(PAGE_SIZE + (long)p);
        *(--krnstack) = ss & 0xffff;
        *(--krnstack) = esp;
        *(--krnstack) = eflags;
        *(--krnstack) = cs & 0xffff;
        *(--krnstack) = eip;                    /* 上面这5句，是用来为iret从内核栈返回用户态准备的 */
        *(--krnstack) = ds & 0xffff;
        *(--krnstack) = es & 0xffff;
        *(--krnstack) = fs & 0xffff;
        *(--krnstack) = gs & 0xffff;
        *(--krnstack) = esi;
        *(--krnstack) = edi;
        *(--krnstack) = edx;
        *(--krnstack) = (long)first_return_from_kernel;
        *(--krnstack) = ebp;
        *(--krnstack) = ecx;
        *(--krnstack) = ebx;
        *(--krnstack) = 0;                      /* 这个0对应于寄存器eax，它是作为fork的返回值使用的 */
        p->kernelstack = krnstack;              /* 将新进程的内核栈栈顶保存在它的PCB中，将来切换时需要用 */

        p->signal = 0;
```

```
.align 2
first_return_from_kernel:
        popl %edx
        popl %edi
        popl %esi
        pop %gs
        pop %fs
        pop %es
        pop %ds
        iret
```

声明一下first_return_from_kernel函数在外部。

```
//--------------------------------------------------
extern void first_return_from_kernel(void);
//--------------------------------------------------
```

以及注意first_return_from_kernel和switch_to两个汇编子程序都要让外部文件看见。



```
.globl system_call,sys_fork,timer_interrupt,sys_execve
.globl hd_interrupt,floppy_interrupt,parallel_interrupt
.globl device_not_available, coprocessor_error
.globl switch_to,first_return_from_kernel
```

随意写个fork测试一下是否正常运行。

```
#include <stdio.h>

int main()
{
        if (!fork()) {
                printf("This is child\n");
                exit(0);
        }
        printf("This is parent\n");

        return 0;
}
~
~
~
~
~
~
~
~
~
~
~
~
```