```
pop    %fs
pop    %es
pop    %ds
iret
```

不要忘了在system_call.s中把它们设为全局可见,而且在sched.c和fork.c这些引用它们的文件中加上声明.

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

```
extern int timer_interrupt(void);
extern int system_call(void);

extern void switch_to(struct task_struct * pnext,unsigned long ldtn);
```

```
extern void write_verify(unsigned long address);
extern void first_return_from_kernel(void);
long last_pid=0;
```

因为在PCB,即结构体task_struct中增加了kernelstack这个field,所以要修改一下system_call.s中的一些硬编码,不过我没按实验指导书上说的把kernelstack 放在第四个field中,而是在blocked之后:

```
struct task_struct {
/* these are hardcoded - don't touch */
    long state;  /* -1 unrunnable, 0 runnable, >0 stopped */
    long counter;
    long priority;
    long signal;
    struct sigaction sigaction[32];
    long blocked; /* bitmap of masked signals */
    long kernelstack;
/* various fields */
    int exit_code;
    unsigned long start_code,end_code,end_data,brk,start_stack;
    long pid,father,pgrp,session,leader;
    unsigned short uid,euid,suid;
    unsigned short gid,egid,sgid;
```

```
ESP0        = 4

state       = 0         # these are offsets into the task-struct.
counter     = 4
priority    = 8
signal      = 12
sigaction   = 16        # MUST be 16 (=len of sigaction)
blocked     = (33*16)
KERNEL_STACK  = (33*16+4)
```

记得在sched.c中初始化全局变量tss:

```
union task_union {
    struct task_struct task;
    char stack[PAGE_SIZE];
};

static union task_union init_task = {INIT_TASK,};

long volatile jiffies=0;
long startup_time=0;
struct task_struct *current = &(init_task.task);
struct tss_struct *tss = &(init_task.task.tss);
struct task_struct *last_task_used_math = NULL;

struct task_struct * task[NR_TASKS] = {&(init_task.task), };

long user_stack [ PAGE_SIZE>>2 ] ;
```

修改sched.h中的宏INIT_TASK

```
/*
 *  INIT_TASK is used to set up the first task table, touch at
 * your own risk!. Base=0, limit=0x9ffff (=640kB)
 */
#define INIT_TASK \
/* state,counter,priority */ { 0,15,15,\
/* signals */  0,{{},},0,PAGE_SIZE+(long)&init_task,\
/* ec,brk... */ 0,0,0,0,0,0, \
/* pid etc.. */ 0,-1,0,0,0, \
/* uid etc */  0,0,0,0,0,0, \
/* alarm */   0,0,0,0,0,0, \
/* math */ 0, \
/* fs info */  -1,0022,NULL,NULL,NULL,0, \
/* filp */ {NULL,}, \
    { \
        {0,0}, \
/* ldt */  {0x9f,0xc0fa00}, \
        {0x9f,0xc0f200}, \
    }, \
/*tss*/  {0,PAGE_SIZE+(long)&init_task,0x10,0,0,0,0,(long)&pg_dir,\
     0,0,0,0,0,0,0,0, \
     0,0,0x17,0x17,0x17,0x17,0x17,0x17, \
     _LDT(0),0x80000000, \
        {} \
    }, \
}
```

二.修改schedule()

直接上代码吧,详见注释:

```
void    schedule(void)
{
        int     i,next,c;
        struct      task_struct         **  p;
        struct      task_struct         *   pnext=&(init_task.task);             //pnext 是指向目标进程的PCB的指针

    /**
            pnext 必须初始化为指向0号进程的PCB的指针, 一开始我并没有初始化pnext,
            就是一个声明, 结果OS不停的宕机重启. 谷歌后发现, 当OS中只有一个0号进程时,
            其会不停的调用schedule(), 而schedeule()最终会调用switch_to(pnext, LDT(n)),
            已知switch_to()有个功能是当发现pnext==current 时, 就什么都不做, 所以若pnext
            初始化指向0号进程的PCB, 0号进程可以一直空转. 若pnext 没初始化, 那pnext 就是个野指针,
            鬼知道switch_to()会切到什么地方去, 自然就宕机了
        **/
/*    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));               //#define       LDT(n)    _LDT(n)
}
```

三.修改copy_process()

直接上代码,详见注释

```
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;

        p   =  (struct        task_struct          *)    get_free_page();
        if    (!p)
              return     -  EAGAIN;
        long     *   krnstack;
        krnstack=(PAGE_SIZE          +  (long)      p);   //krnstack指向内核栈的开端( 即栈中无元素时的栈顶), 记住栈从
高地址往低地址增长
        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;
        p- >signal       =  0;
        p- >alarm      =  0;
        p- >leader       =  0;                /*    process     leadership       doesn't      inherit      */
        p- >utime      =  p- >stime      =  0;
        p- >cutime      =  p- >cstime      =  0;
        p- >start_time          =  jiffies;
        //对tss的操作可以去掉了

/**
        要知道新fork出的进程从未调用过schedule(), switch_to()这些函数来切到别的进程中去,
        所以, 新进程的内核栈中并没有用户栈的栈顶%esp, cs:ip等等这些寄存器的值,
        也没有schedule()的右大括号'}' 的地址, 所以我们要"初始化"它的内核栈, 往里面加一点东西,
        使别的进程调用了schedule(), 并switch_to()到该新进程时, 可以顺利的从内核态返回
**/

//通过first_return_from_kernel的iret 弹出
        *( --  krnstack)       =  ss   &  0xffff     ; //因为SS只有16位, 而入栈是以32入的
        *( --  krnstack)       =  esp;
        *( --  krnstack)       =  eflags;
```