_refrain L73 2018-09-28 02:55:46 基于内核栈切换的进程切换



“操作系统原理与实践”实验报告

基于内核栈切换的进程切换

# 实验目的

- 深入理解进程和进程切换的概念；
- 综合应用进程、CPU管理、PCB、LDT、内核栈、内核态等知识解决实际问题；
- 开始建立系统认识。

# 实验内容

现在的Linux 0.11采用TSS（后面会有详细论述）和一条指令就能完成任务切换，虽然简单，但这指令的执行时间却很长，在实现任务切换时大概需要 200 多个时钟周期。而通过堆栈实现任务切换可能要更快，而且采用堆栈的切换还可以使用指令流水的并行优化技术，同时又使得CPU的设计变得简单。所以无论是 Linux还是 Windows，进程/线程的切换都没有使用 Intel 提供的这种TSS切换手段，而都是通过堆栈实现的。

本次实践项目就是将Linux 0.11中采用的TSS切换部分去掉，取而代之的是基于堆栈的切换程序。具体的说，就是将Linux 0.11中的switch_to实现去掉，写成一段基于堆栈切换的代码。

本次实验包括如下内容：

- 编写汇编程序switch_to：
- 完成主体框架；
- 在主体框架下依次完成PCB切换、内核栈切换、LDT切换等；
- 修改fork()，由于是基于内核栈的切换，所以进程需要创建出能完成内核栈切换的样子。
- 修改PCB，即task_struct结构，增加相应的内容域，同时处理由于修改了task_struct所造成的影响。
- 用修改后的Linux 0.11仍然可以启动、可以正常使用。
- （选做）分析实验3的日志体会修改前后系统运行的差别。

# 实验步骤

## 修改schedual()函数

原来传入switch_to函数中的变量为next，即GDT中对应TSS的位置。现在不用TSS了，所以要传入指向目标进程PCB的指针，及现在进程的LDT。即修改为：

```
struct task_struct * pnext = current;#这里将pnext初始化为当前进程。因为当不需要调度的时候，当前进程counter最大，还是要切到当前进程。其他同学写的切到0进程可能有点问题。
 .......
 if ((*p)->state == TASK\_RUNNING \&\& (*p)->counter > c) c = (*p)->counter, next = i, pnext = *p;

 .......

switch_to(pnext, _LDT(next));#由于.h中定义和讲义中不太一样
```

在sched.h中删去原来的switch_to函数，并加入自己写的switch_to的声明：

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

## 实现switch_to

在system_call.s中加入switch_to函数：

```
.align 2
switch_to:
    pushl    %ebp
    movl    %esp,%ebp
    pushl    %ecx
    pushl    %ebx
    pushl    %eax
    movl    8(%ebp),%ebx#取出pnext
    cmpl    %ebx,current#比较pnext和current，与之前的pnext初始化为current对应
    je    1f#如果相等，直接跳过切换过程
    切换PCB
    TSS中内核栈指针重写
    切换内核栈
    切换LDT
    movl    $0x17,%ecx
    mov    %cx,%fs
    cmpl    %eax,last_task_used_math
    jne    1f
    clts
1:    popl    %eax
    popl    %ebx
    popl    %ecx
    popl    %ebp
    ret
```

## 切换PCB指针

可以利用实验指导中的代码：

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

## 内核栈指针的重写

由于CPU原有的CPU机制，TSS结构仍需要保留，但在切换进程时可以让所有进程共用tss0。现在ebx为pnext，即PCB位置。加上4K后即为内核栈的位置。所以在tss0的ESP0偏移位置就保存了内核栈的指针。

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

加入tss的定义：

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

## 内核栈指针的切换

首先要在task_struck中添加kernelstack的声明：

```
    long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    long counter;
    long priority;
    long kernelstack;
    long signal;
```

在init_task中添加kernelstack的初始值

```
 #define INIT_TASK { 0,15,15,PAGE_SIZE+(long)&init_task, 0,{{},},0,...,
```

在system_call.s中定义一些常量：

```
ESP0 =4
KERNEL_STACK = 12 #添加实验指导需要的常量

state    = 0        # these are offsets into the task-struct.
counter    = 4
priority = 8
kernelstack = 12     # 添加定义
signal    = 16
sigaction = 20
blocked = (33*16+4) #由于添加了一个long，所以偏移量加4
```

然后加入切换内核栈指针的代码：

```
 movl %esp,KERNEL_STACK(%eax) #将当前栈顶指针存到当前PCB的kernelstack位置
 movl 8(%ebp),%ebx #再取一下ebx，因为前面修改过ebx的值
 movl KERNEL_STACK(%ebx),%esp #将下一个进程的PCB中的kernelstack取出，赋给esp，完成切换
```

### LDT切换

```
    movl    12(%ebp),%ecx #取出LDT(next)
    lldt    %cx #完成LDT的切换
    #finish
    movl    $0x17,%ecx
    mov    %cx,%fs #问题3
```

## 修改fork

### 修改copy_process()

```
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;
    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;
    p->utime = p->stime = 0;
    p->cutime = p->cstime = 0;
    p->start_time = jiffies;
    /*
    p->tss.back_link = 0;
    p->tss.esp0 = PAGE_SIZE + (long) p;
    p->tss.ss0 = 0x10;
    p->tss.eip = eip;
    p->tss.eflags = eflags;
    p->tss.eax = 0;
    p->tss.ecx = ecx;
    p->tss.edx = edx;
    p->tss.ebx = ebx;
    p->tss.esp = esp;
    p->tss.ebp = ebp;
    p->tss.esi = esi;
    p->tss.edi = edi;
    p->tss.es = es & 0xffff;
    p->tss.cs = cs & 0xffff;
    p->tss.ss = ss & 0xffff;
    p->tss.ds = ds & 0xffff;
    p->tss.fs = fs & 0xffff;
    p->tss.gs = gs & 0xffff;
    p->tss.ldt = _LDT(nr);
    p->tss.trace_bitmap = 0x80000000;
    if (last_task_used_math == current)
        __asm__("clts ; fnsave %0"::"m" (p->tss.i387));
    */#有关tss设置的内容全部注释掉
    long* krnstack = (long*)(PAGE_SIZE + (long)p);#初始化krnstack，指向子进程的内核
栈
    *(--krnstack) = ss & 0xffff;
    *(--krnstack) = esp;
    *(--krnstack) = eflags;
    *(--krnstack) = cs & 0xffff;
    *(--krnstack) = eip;
#拷贝父进程中的五个参数
    *(--krnstack) = (long) first_return_from_kernel;#处理switch_to返回的位置
    *(--krnstack) = ebp;
    *(--krnstack) = ecx;
    *(--krnstack) = ebx;
    *(--krnstack) = 0;
#把switch_to中要pop的东西存进去。
    p->kernelstack = krnstack;

    if (copy_mem(nr,p)) {
        task[nr] = NULL;
        free_page((long) p);
        return -EAGAIN;
    }
    for (i=0; i<NR_OPEN;i++)
        if ((f=p->filp[i]))
            f->f_count++;
    if (current->pwd)
        current->pwd->i_count++;
    if (current->root)
        current->root->i_count++;
    if (current->executable)
        current->executable->i_count++;
    set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
    set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->ldt));
    p->state = TASK_RUNNING;
```

```
    return last_pid;
}
```

### 增加first_return_from_kernel

这里要回复这一堆寄存器的值：

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

在kernel_stack中也要push进去：

```
    *(--krnstack) = ds & 0xffff;
    *(--krnstack) = es & 0xffff;
    *(--krnstack) = fs & 0xffff;
    *(--krnstack) = gs & 0xffff;
    *(--krnstack) = esi;
    *(--krnstack) = edi;
    *(--krnstack) = edx;
```

在system_call.s中设置first_return_from_kernel为全局可见：

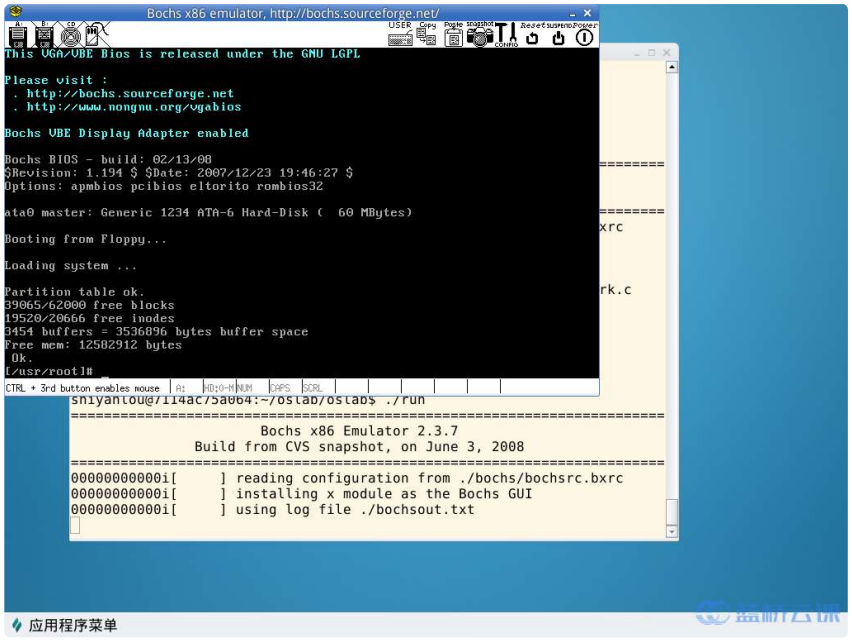
```
.globl switch_to, first_return_from_kernel
```

在fork.c中添加函数声明：

```
extern void first_return_from_kernel(void);
```

## 实验结果

终于可以运行了！




##实验思考 1.针对：

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

(1)为什么要加4096？ 这里ebx指向的是下一个进程的PCB。加上4096后，即为一个进程分配4K的空间，栈顶即为内核堆栈的指针，栈底为进程的PCB。 (2)因为进程的切换不靠tss进行，但CPU的机制造成对每个进程仍然会有TR寄存器、tss的设置等内容，所以可以让所有进程都共用tss0的空间。所以不需要设置tss0。 2.针对：