



“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

基于内核栈切换的进程切换

实验内容:

1. 将TSS+一条指令的切换模式改换为基于堆栈的切换程序.
2. 修改的主要部分: 1. 在system_call.s中填写switch_to函数, 2.修改fork.c中的内容, 初始化新的进程.

• TSS切换的缺点与内核栈切换的优点

在现在的Linux 0.11中, 真正完成进程切换是依靠任务状态段(TSS)的切换来完成的.具体来说, 在设计X86架构时, 每个任务(进程或线程)都对应一个独立的TSS, TSS就是内存中的一个结构体,里面包含可几乎所有的CPU寄存器映像. TR寄存器指向当前进程对应的TSS结构体, 所谓的TSS切换就是将当前CPU中的所有寄存器放在当前的TSS结构体中保存起来,找到下一个切换的进程,将他的寄存器扣到CPU中,这样就完成了上下文切换,过程如图:  进程切换原理图 步骤如下:

- i. 利用TR寄存器找到当前GDT表中当前TSS描述符, 这个描述符其实就是当前TSS的地址
- ii. 将当前CPU中的寄存器映像放到这段内存区域中,相当于拍摄了一个快照.
- iii. ljmp指令的操作数中存放了下一个进程或线程的段选择子,这样就可以从GDT中找到目标TSS描述符.
- iv. 将目标TSS中的寄存器覆盖当前CPU中的寄存器,这样就完成了切换操作.

上面这些工作其实可以用一条指令,"ljmp 段选择子:段内偏移", 相当于执行了如下代码:

```
#define switch_to(n) {  
    struct{long a,b;} tmp;  
    __asm__ ("movw %dx,%1"  
    "ljmp %0" :: "m"(&tmp.a), "m"(&tmp.b), "d"(TSS(n))  
    )  
}  
#define FIRST_TSS_ENTRY 4  
#define TSS(n) (((unsigned long) n) << 4) + (FIRST_TSS_ENTRY << 3))
```

下面从头分析调度过程:

1. fork.c函数

主要是对子进程的内核线程初始化, 在include/linux/sched.h的TASK_STRUCT结构体第四个位置增添域kernelstack,类型为long,这个新添的域就是为了存储内核栈的位置,非常重要. copy_process函数内容如下(+为新增的代码):

实验数据

学习时间	70分钟
操作时间	54分钟
按键次数	5375次
实验次数	3次
报告字数	6281字
是否完成	完成

评分

未评分

下一篇

篇

相关报告

操作系统原理与实践: 熟悉实验环境 实验报告

操作系统原理与实践: 熟悉实验环境 实验报告

操作系统原理与实践: 基于内核栈切换的进程切换 实验报告

操作系统原理与实践: 熟悉实验环境 实验报告

操作系统原理与实践: 信号量的实现和应用 实验报告

```

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;
+   long *krnstack;

+   krnstack = (long)(PAGE_SIZE + (long)p);
+   *(--krnstack) = ss & 0xffff;
+   *(--krnstack) = esp;
+   *(--krnstack) = eflags;
+   *(--krnstack) = cs & 0xffff;
+   *(--krnstack) = eip;
+   *(--krnstack) = ds & 0xffff;
+   *(--krnstack) = es & 0xffff;
+   *(--krnstack) = fs & 0xffff;
+   *(--krnstack) = gs & 0xffff;
+   *(--krnstack) = esi;
+   *(--krnstack) = edi;
+   *(--krnstack) = edx;
+   *(--krnstack) = (long)first_return_from_kernel; //一个函数,汇编语
言写的,为了弹出eax,ebx,ecx,edx,esi,edi,gs,fs,es,ds
+   *(--krnstack) = ebp;
+   *(--krnstack) = ecx;
+   *(--krnstack) = ebx;
+   *(--krnstack) = 0;
+   p->kernelstack = krnstack;

    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;
    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__("cldts ; fnsave %0"::"m" (p->tss.i387));
    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;    /* do this last, just in case */
return last_pid;
}

```

copy_process函数就是为了初始化子进程内核栈中的内容,这个内核栈中必须存放子进程运行起来的所有信息,比如: `eax,ebx,ecx,edx,esi,edi,gs,fs,es,ds, ss, esp, eflags, cs, eip`. 这些内容不就像TSS段中的内容嘛,最后再提一嘴: `int 0x80`, CPU会将SS、ESP、EFLAGS、CS、EIP等寄存器压入内核栈中, `IRET`指令称为中断返回指令,会将这些寄存器弹出. #####2.schedule函数

主要作用: 找出下一个切换的进程, 将该进程PCB和LDT作为参数传递给switch_to_by_stack

```

void schedule(void)
{
    int i,next,c;
    struct task_struct ** p;
+   struct task_struct *pnex = &(init_task.task);    //PCB指针
/* 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, pnex=*p;
        }
        if (c) break;
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) +
                    (*p)->priority;
    }
+   switch_to_by_stack(pnex, _LDT(next));    //PCB, LDT
}

```

3.switch_to_by_stack(称呼为当前进程和目的进程),

KERNEL_STACK = 12, ESP0 = 4

```

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
    TSS中的内核栈指针的重写
    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

```

1. 切换PCB:无非是将current修改为目标PCB, PCB的地址:8(%ebp), LDT的地址: 0xc(%ebp),这是参数传递规律,但这仅仅是在32位机器下, 64位机器下参数少的时候是通过寄存器,参数多的时候才会使用栈来传递参数.

```

movl %ebx, %eax
xchgl %eax, current      #current是一个全局变量

```

2. TSS中的内核栈指针的重写: 其实就是相当与p.tss->esp0 = (long)addr + 4k

```

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

```

3. 切换内核栈:关键部分, 将当前进程的内核栈写回PCB中, 就是esp中的值写回当前进程的PCB的KERNEL_STACK(内核指针域)中,并将目标进程的KERNEL_STACK写到esp中

```

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

```

4. 切换LDT: LDT(Local Descriptor Table),每个进程最多有一个LDT, LDT和GDT本质上是相同的.在TASK_STRUCT结构体中可以观察到struct desc_struct ldt[3];;主要存放某个任务专用的各段的段描述符, 只能是三类进程段的段描述符和调用门描述符,最大表长4GB

```

mov 0xc(%ebp), %ecx
lldt %cx

```

注意:

```

movl $0x17,%ecx
mov %cx,%fs

```

主要作用:重新取一下段寄存器fs的值,这两句话必须要加、也必须要出现在切换完LDT之后. 通过fs访问用户内存,LDT切换完成就意味着切换了分配给进程的用户态内存地址空间.所以前一个fs指向的是上一个进程的用户态内存, 而现在需要执行下一个进程的用户态内存, 所以就需要用这两条指令来重取fs, 另外还有一个有点, 加快了指令的执行速度.####4.

first_return_from_kernel

```

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

```