HUST_AT L32 2016-08-13 09:07 Linux 计算机专业课

实验代码

“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

因为昨天实验快做完时，停电了，所以我实在不想再做第二遍了，大致写下核心代码吧，给出我所理解的注释

下面是两个汇编写的函数

```
.global switch_to,first_return_from_kernel
.align
switch_to:
pushl %ebp
movl %esp,%ebp
pushl %ecx      //保存要用的寄存器
pushl %ebx      //保存要用的寄存器
pushl %eax      //保存要用的寄存器
movl 8(%ebp),%ebx //第一个参数pnext保存到ebx中
cmpl %ebx,current
je 1f     //比较ebx中pnext是否与当前的current PCB相等，如果相等，则不需要switch，所以直接跳到1返回
数
mov %ebx,%eax
xchgl %eax,current//这两句完后，ebx和current的都是pnext，eax存的是以前的老PCB，就是说这句是**核
指令**之一，把current指向了pnext
movl tss,%ecx //因为用堆栈切换的方法中，全局只有一个tss了，它用来描述当前正在执行的进程，所以switch
to还必须把tss与新的进程同步，所以要更新tss结构中的内容
add %4096,%ebx//为什么要加4096 ，因为之前ebx已经指向pnext，就是说是新PCB的开头，但是0.11规定是用4
K-页的内存来保存PCB的，而且栈就紧挨着PCB，于是把pnext+4096就是栈地址，所以现在ebx中存的就是新的进程
的核栈地址
movl %ebx,ESP0(%ecx)//这条指令印证了上条指令，现在我们要把ebx中新进程的内核栈地址同步到TSS结构中
movl %esp,KERNEL_STACK(%eax)//前面的指令这是完成了 current指向下一个pcb，但还没有完成真正的栈
切换，就是说现在的栈依旧是以前老进程的栈
//而这句指令就是为真正切换栈做准备，这条指令的作用是把老进程的栈地址保存到老进程的PCB中，%eax存的是老
程的指针
movl 8(%ebp),%ebx//因为前面把ebx+了4096 ，所以ebx中存的不是pnext了，所以要重新给ebx变成pnext
movl KERNEL_STACK(%ebx),%esp //**这句完成了栈的真正的切换，将pnext所指PCB中的栈地址送入esp，完
成栈的切换
//输入代码**

movl 12(%ebp),%ecx
lldt
movl $0x17,%ecx
mov %cx,%fs
cmpl %eax,last_task_used_math
jne 1f

1:
popl %eax
popl %ebx
popl %ecx
popl %ebp
ret             //这个是switch_to的返回处理，返回保存的寄存器

.align first_return_from_kernel
popl %edx
popl %edi
popl %esi
pop %gs
pop %fs
pop %es
pop %ds       //以上寄存器都是入内核时保存的用户态的寄存器，从内核态返回用户态时要pop他们
iret           //从用户态进入内核的唯一方法是中断，中断会压入用户态寄存器，这个也是pop出用户态时的寄存器
```

下面是对copy_process的修改，参考于mqmelon同学。有部分修改。

```
int copy_process int long long long long long
        long long long
        long long long
        long long long long long
{
    long * krnstack;    //添加内核栈指针变量
    struct task_struct *p;
    int i;
    struct file *f;

    p = (struct task_struct *) get_free_page();
    if (!p)
        return - EAGAIN;
    krnstack=(long *)(PAGE_SIZE+(long *)p);  //实际上进程每次进入内核，栈顶都指向这里。实际上这句C
确，和那句add %4096,%ebx的本质是一样的
    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;
    //初始化内核栈内容，由于系统不再使用tss进行切换，所以内核栈内容要自己安排好 ，内核栈切换的核心思想
就是构造出一个被切换出去的进程。而一个进程无非代码PCB+PCB+栈。所以这个copy_pocess做了，代码 用父进程的
    //PCB的构造，从上面的代码可以看出是如何构造的，接下来就是内核栈的构造，我们要做的就是把这个栈构造
成能进程被切换出去的时候的样子。所以搞清楚一个进程被切换出去时，栈是什么样子很重要，以及栈为什么是这个样
很重要
    //下面部分就是进入内核后int之前入栈内容，即用户态下的cpu现场，进程从用户态进入内核时，会用中断，而
断会自动把ss,esp,eflag,cs,eip入内核栈，所以我们要做的就是构造这样的栈
    *(--krnstack)    = ss  & 0xffff  ;  //保存用户栈段寄存器，这些参数均来自于此次的函数调用，
                    //即父进程压栈内容，看下面关于tss的设置此处和那里一样。
    *(--krnstack)    = esp;   //保存用户栈顶指针
    *(--krnstack)    = eflags;    //保存标识寄存器
    *(--krnstack)    = cs  & 0xffff  ;  //保存用户代码段寄存器
    *(--krnstack)    = eip;   //保存eip指针数据，iret时会出栈使用 ，这里也是子进程运行时的语句地址。即i
f(!fork()==0) 那里的地址，由父进程传递
    //下面是iret时要使用的栈内容，由于调度发生前被中断的进程总是在内核的int中，
    //所以这里也要模拟中断返回现场，
    //根据老师的视频讲义和实验指导，这里保存了段寄存器数据。
    //由switch_to返回后first_return_from_kernel时运行，模拟system_call的返回 ，
    *(--krnstack)    = ds  & 0xffff  ;
    *(--krnstack)    = es  & 0xffff  ;
    *(--krnstack)    = fs  & 0xffff  ;
    *(--krnstack)    = gs  & 0xffff  ;
    *(--krnstack)    = esi;
    *(--krnstack)    = edi;
    *(--krnstack)    = edx;
    //*(--krnstack) = ecx;  //这三句是我根据int返回栈内容加上去的，后来发现不加也可以
                //但如果完全模拟return_from_systemcall的话，这里应该要加上。
    //*(   --krnstack)    = ebx;
    //*(--krnstack) = 0;  //此处应是返回的子进程pid//eax;
                //其意义等同于p->tss.eax=0;因为tss不再被使用，
                //所以返回值在这里被写入栈内，在switch_to返回前被弹出给eax;

    //switch_to的ret语句将会用以下地址做为弹出进址进行运行
    *(--krnstack)    = (long *)first_return_from_kernel;
    //*(--krnstack) = &first_return_from_kernel;  //讨论区中有同学说应该这样写，结果同上
    //这是在switch_to一起定义的一段用来返回用户态的汇编标号,也就是
    //以下是switch_to函数返回时要使用的出栈数据
    //也就是说如果子进程得到机会运行，一定也是先
    //到switch_to的结束部分去运行，因为PCB是在那里被切换的，栈也是在那里被切换的，
    //所以下面的数据一定要事先压到一个要运行的进程中才可以平衡。
    *(--krnstack)    = ebp;
    *(--krnstack)    = eflags;    //新添加
    *(--krnstack)    = ecx;
    *(--krnstack)    = ebx;
    *(--krnstack)    = 0;  //这里的eax=0是switch_to返回时弹出的，而且在后面没有被修改过。
                //此处之所以是0，是因为子进程要返回0。而返回数据要放在eax中，
                //由于switch_to之后eax并没有被修改，所以这个值一直被保留。
                //所以在上面的栈中可以不用再压入eax等数据。
                //将内核栈的栈顶保存到内核指针处
    p->kernelstack=krnstack;        //保存当前栈顶
```

其他部分还有很多细节问题，那些才是这个实验真正学到东西得部分。感觉那些部分非常的坑，毕竟内核不好调试。