“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

一.修改switch_to

1.因为本身0.11上的switch_to(n)是个宏展开成的内嵌汇编,所以就以为修改后的switch_to也应该是内嵌汇编.于是就这么做了. 但是编译出的OS却无法成功加载————在执行到init/main.c的if (!fork()) {},也就是要fork出1号进程,即OS第一次调用fork()时, 就会蜜汁宕机重启,显然,是sys_fork()相关的copy_process()出了问题.但到底是什么问题,当时毫无头绪.于是开始下面列举的 胡乱尝试:

- 是在copy_process()中把内核栈里各存放的寄存器顺序排错了吗?几顿折腾发现不是.
- 又尝试把switch_to()放在fork.c中(本来放在sched.c中),发现没卵用.
- 又把first_return_from_kernel这个汇编标号指向的那堆指令实现成一个内联函数(本来是直接放在system_call.s中的一段汇编),然而无论是放在fork.c还是sched.c都尝试过了, 还是没卵用.关于这一点,个人认为原因是这样的:根据static inline函数类似于宏展开的性质,只有在真正调用它的时候,编译器才会 把函数的代码直接嵌入在它被调用的地方,然而在这里,并没有直接的调用first_return_from_kernel,而是在copy_process()中 把first_return_from_kernel()的入口地址填入内核栈,于是,我认为first_return_from_kernel这个函数名指向的位置并不存在该函数 的代码————只是弱渣的个人猜测,说错了轻喷.

```
static inline void first_return_from_kernel(void)
{
    __asm__("popl %edx\n\t"
                    "popl %edi\n\t"
                    "popl %esi\n\t"
                    "pop %gs\n\t"
                    "pop %fs\n\t"
                    "pop %es\n\t"
                    "pop %ds\n\t"
                    "iret\n\t");
}
```

- 直到最后,我选择把switch_to()放在system_call.s中————也就是直接写成一段汇编,而且**和first_return_from_kernel放在一个文件中** ,就有卵用了 为什么?汇编标号一般不是绝对地址,只是个偏移量吧,所以当switch_to()最后的ret把内核栈中first_return_from_kernel这个汇编标号对应的地址弹出时, 若switch_to()和first_return_from_kernel不在一个文件中,也就不在同一个段内,就会使CS:IP指向错误的地方.汇编是一年前学的,忘了很多东西说错了轻喷

2.switch_to()和first_return_from_kernel的代码(都放在system_call.s中):

```
.align 2
switch_to:
    pushl %ebp              #把当前%ebp入栈
    movl %esp,%ebp          #再把%ebp指向当前栈顶
    pushl %ecx
    pushl %ebx
    pushl %eax              #依次把%ecx,%ebx,%eax都入栈
    movl 8(%ebp),%ebx      #取出函数参数pnext,放在%ebx中
    cmpl %ebx,current      #把pnext与current对比,如果pnext==current,就什么都不做,直接跳到标号1处返回
    je 1f

    #切换PCB
    movl %ebx,%eax          #令%eax也为pnext
    xchgl %eax,current      #xchgl指令使%eax指向当前进程,同时把current设为pnext对应的进程,即切换PCB

    #TSS中内核栈指针重写
    movl tss,%ecx          #tss是个全局变量,用来指向0号进程的TSS部分.令%ecx=tss
    addl $4096,%ebx          #4096bytes即为一页内存的大小,这条指令使%ebx指向pnext的内核栈栈顶
    movl %ebx,ESP0(%ecx)     #用%ebx中的值更新tss中的内核栈指针esp0

    #切换内核栈
    movl %esp,KERNEL_STACK(%eax)     #把当前进程内核栈的栈顶保存PCB->kernelstack中
    movl 8(%ebp),%ebx      #再令ebx=pnext,因为前面修改过ebx的值
    movl KERNEL_STACK(%ebx),%esp     #把pnext->kernelstack放到%esp中,至此,内核栈切换完毕

    # 切换LDT
    movl 12(%ebp),%ec
```

```
.align 2
first_return_from_kernel:
    popl %edx
    popl %edi
    popl %esi
    pop %gs
    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.

```
70 .globl system_call,sys_fork,timer_interrupt,sys_execve,switch_to
71 .globl hd_interrupt,floppy_interrupt,parallel_interrupt
72 .globl device_not_available, coprocessor_error,first_return_from_kernel
73
```



```
50  extern int timer_interrupt(void);
51  extern int system_call(void);
52
53  extern void switch_to(struct task_struct * pnext,unsigned long ldtn);
54
```



```
20  extern void write_verify(unsigned long address);
21  extern void first_return_from_kernel(void);
22  long last_pid=0;
23
```


- 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:

```
81  struct task_struct {
82  /* these are hardcoded - don't touch */
83      long state;  /* -1 unrunnable, 0 runnable, >0 stopped */
84      long counter;
85      long priority;
86      long signal;
87      struct sigaction sigaction[32];
88      long blocked; /* bitmap of masked signals */
89      long kernelstack;
90  /* various fields */
91      int exit_code;
92      unsigned long start_code,end_code,end_data,brk,start_stack;
93      long pid,father,pgrp,session,leader;
94      unsigned short uid,euid,suid;
95      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:

```
55  union task_union {
56      struct task_struct task;
57      char stack[PAGE_SIZE];
58  };
59
60  static union task_union init_task = {INIT_TASK,};
61
62  long volatile jiffies=0;
63  long startup_time=0;
64  struct task_struct *current = &(init_task.task);
65  struct tss_struct *tss = &(init_task.task.tss);
66  struct task_struct *last_task_used_math = NULL;
67
68  struct task_struct * task[NR_TASKS] = {&(init_task.task), };
69
70  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;
    *(--krnstack) = cs & 0xffff;
    *(--krnstack) = eip;

//由first_return_from_kernel前几条pop指令弹出
    *(--krnstack)=ds & 0xffff;
    *(--krnstack)=es & 0xffff;
    *(--krnstack)=fs & 0xffff;
    *(--krnstack)=gs & 0xffff;
    *(--krnstack) = esi;
    *(--krnstack) = edi;
    *(--krnstack) = edx;

//当switch_to的ret被执行后,这里存放的标号first_return_from_kernel代表地址就会弹到的CS:IP中
    *(--krnstack) =(long) first_return_from_kernel;

//这几个寄存器是为switch_to()的结尾的几个popl准备的
    *(--krnstack) = ebp;
    *(--krnstack) = ecx;
    *(--krnstack) = ebx;
    *(--krnstack) = 0; //这里的0其实是%eax,即fork()的返回值

    p->kernelstack=krnstack;//保存好内核栈的栈顶
    if (last_task_used_math == current)
        __asm__("clts ; 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;
}
```

四.实验报告

1.

- 4096bytes是一页内存的大小,已知%ebx存放了参数pnext,即PCB的起始地址,已知对于一个进程,一页中高地址处存放内核栈,低地址处存放PCB,于是这样可以使%ebx 存放pnext所指向的进程的内核栈的开始处.
- 因为已经不用TSS结构来存放大多数寄存器的值了

2.

- %eax中存放fork()系统调用的返回值,令%eax=0可使子进程的fork()返回0,用来区分父进程和子进程.就是在copy_ 卜令子进程的内核栈顶存放0,当子进程被调度 执行时,即switch_to()到子进程的内核栈时,从switch_to()返回前的那一坨p 的第一条popl %eax,就把内核栈顶的0弹到%eax中.
- 这段代码的%ebx,%ecx来自父进程,是copy_process()的参数之一,也就是父进程执行的上下文环境之一.将它们写入子进程内核栈是为了与switch_to()返回前的那几条pop指令 适配
- %ebp同上,就是恢复进程上下文的一种手段,不可以不设置

3.

- 重设fs=0x17有个额外的效果:把GDT表达0x17对应的那个表项的基地址和段限长放在fs的隐藏部分,所以下次用fs生产用户态的内存地址时,若fs没有改过,则不必再去查GDT表, 而是直接利用fs的隐藏部分中的基地址和段限长来得到用户态的内存地址,提高了执行效率.
- 必须在切换了LDT后再重设fs=0x17,否则,fs的隐藏部分存放的就是上一个进程的用户态内存地址的基地址和段限长.