江上之明月 L31 2017-01-16 22:12:16 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

基于内核栈切换的进程切换

预备知识

TSS切换

linux0.11 采用TSS和一条指令ljmp就能完成任务切换。 TSS切换 是指进程切换时依靠任务状态段（TSS）的切换来完成。 在x86系统结构中，每个任务（进程和线程）都对应一个独立的TSS，TSS是内存中的一个结构体，里面包含了几乎所有的CPU寄存器的映像。有一个任务寄存器（Task Register，简称TR）指向当前进程对应的TSS结构体。所谓的TSS切换就将CPU中几乎所有的寄存器都复制到TR指向的那个TSS结构体中保存起来，同时找到一个目标TSS，即要切换到的下一个进程对应的TSS，将其中存放的寄存器映像“扣”CPU上，就完成了执行现场的切换。

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

编写汇编的switch_to：

（1）完成主体框架 （2）完成PCB切换、内核栈切换、LDT切换等 （3）修改fork(),进程基于内核栈切换，所以进程需要创建出能完成内核栈切换的样子 （4）完成PCB，即task_struct结构，增加相应的内容域，同时处理由于task_struct所造成的影响 （5）修改后的linux0.11仍然可以启动、可以正常使用 （6）分析实验3的日志体会修改前后系统运行的差别

具体步骤

进程切换分析如下：

1. 内核创建一个新进程，接口调用如下 fork->int 0x80->sys_call->sys_fork->copy_process
   （1） 用户程序调用fork中通过中断指令INT 0x80 进入内核态。INT指令会把用户态的寄存器的值自动压入栈（SS，ESP,eflags，CS,IP），并根据中断向量表设置CS和IP。用户态的其他寄存器怎么处理？INT指令自动压栈的这个栈是用户的栈还是内核的栈？ （2）sys_call一开始将用户态的寄存器（ds，es，fs，edx，ecx，ebx）压入栈，接着设置好内核的数据段ds，fs。 此时eax存放是还是sys_fork的系统调用号。 然后用call指令调用sys_fork. call指令会自动把返回地址入栈。

（3） sys_fork首先查找新的进程号，并将进程号放入到eax寄存器。接着将用户态的一些寄存器入栈（gs,esi,edi,ebp,eax）,接着调用copy_poress

(4) copy_process是一个C函数，汇编调用C函数会参数默认放在栈中。copy_process的参数与栈中的寄存器的顺序是一一对应的。 首先申请一块4K内存，接着这块内存开始地址为0存放PCB结构体，4K地址处初始化栈指针。内核栈需要copy放到新进程的栈中，存放顺序与进程切换switch_to处理对应。 有一点需要注意的是参数eax是pid号，但是子进程应该用0 替换，因为子进程返回pid为0。 Q： 为什么不一次性把所有的寄存器入栈。

(5) copy_process执行完后eax中存放的是返回值，再加个返回值是pid，返回到sys_fork中调用add $20, Q$：为什么要执行$add20$，%esp?

实验数据

学习时间	287分钟
操作时间	15分钟
按键次数	297次
实验次数	4次
报告字数	13200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

(6) sys_call调用sys_fork返回后，执行pushl %eax，这个eax是sysfork的返回值此时sysfork栈帧是（ds,es,fs,edx,ecx,ebx,eax）,sys_call 最后将栈中的值弹出，并执行iRet。 再次返回到用户态中。至此一个完成的系统调用就完成了。

2. 进程切换switch_to 从时钟中断说进程切换，时钟中断的事件如下 硬件时钟中断->timer_interrupt->do_timer->schedule->switch_to->新进程

（1）硬件时钟中断，硬件自动将寄存器（SS，SP，eflags，cs，ip）入栈，跳转到中断处理函数timer_interrupt处执行。

（2）timer_interrupt首先将寄存器（ds，es,fs,edx，ecx，ebx，eax）入栈，然后设置ds，es，fs，接着调用call指令调用C函数do_timer，其参数也通过栈传输。do_timer返回时，esp+4 相当于丢弃do_timer压入栈的那个参数。

（3）do_timer函数中检查当前进程的时间片是否为0，假如为0则调用schedule()进行任务切换。

（4）C函数shedule里面，首先对所有进程的信号进行处理，如果进程不阻塞则把进程状态修改等待。 然后根据调度算法找出下一个要运行的进程。 接着调用汇编switch_to切换到下一个进程。 此时内核栈的内容时（ss,sp,eflags,cs,eip, ds,es,fs,edx,ecx,ebx,eax+do_timer+schedule） Q： C函数调用汇编代码如何传参数?

（5）switch_to首先处理c函数调用汇编将ebp入栈，并将当前esp保存到ebp，接着讲寄存器（ecx，ebx，eax）入栈，此时栈帧的内容是（ebp，ecx，ebx，eax）。假如当前进程与切换的进程是同一个进程则不需要切换，弹出栈中的4个寄存器，返回schedule->do_timer->timer_interrupt。

（6）如果不是同一个进程则需要切换进程，切换步骤： 切换PCB->切换内核栈指针->切换LDT

a.切换PCB：将current 和switch_to传的pnext换内容，现在current指向的是pnext进程，ebx指向pnex进程的内存开始位置。

b.TSS中内核栈指针的重写：中断发送时，需要把当前进程的（ss,esp,eflag,cs,eip）压入当前进程的内核栈中。如何找到内核栈的指针呢？intel的中断机制是这么处理的，取TR寄存器存指向的TSS结构体中内核栈指针作为当前进程的内核栈指针。所有的进程都共用这一个TSS，所以一个进程允许前应该设置好tss中的内核栈指针。

c.切换内核栈：将esp寄存器保存到当前进程的PCB中，同时将下一个进程的内核栈指针赋值给寄存器esp。 注意：原来的PCB中并没有内核栈指针，增加这个成员变量后，需要修改与这个结构体相关的硬编码。

d.切换LDT：切换完ldt表，下一个进程执行用户态程序时使用的映射表就是自己的LDT表，地址空间实现了分离。

d.切换PC：switch_to最后退出需要返回下一个子进程的用户态，current指向下一个进程的pcb，寄存器esp指向下一个进程的内核栈。先是执行弹出switch_to压入的寄存器（ebp,ecx,ebx,eax），所以copy_process的栈也要对应的压入这些寄存器。接着ret，ret的地址也应该继续弹栈将（ds,es,fs,gs,esi,edi,edx）弹栈，最后执行iret将终端压入的（ss,esp,eflags,cs,eip）弹出返回用户态。

问题回答

1. 针对下面的代码片段： movl tss,%ecx addl $4096,%ebx movl %ebx,ESP0(%ecx) 回答问题：（1）为什么要加4096；（2）为什么没有设置tss中的ss0。 答：（1）ebx是进程pcb的存放的开始地址，而栈在一页4K内存地址的最后。中断还需要 （2）改用内核栈切换后调用init指令寄存器ss的值会自动压到内核栈中，调用iret返回时，恢复ss寄存器的值。所以tss中的ss0不再需要。

2. 针对代码片段：

*(--krnstack) = ebp; *(--krnstack) = ecx; *(--krnstack) = ebx; *(--krnstack) = 0; 回答问题：（1）子进程第一次执行时，eax=？为什么要等于这个数？哪里的工作让eax等于这样一个数？（2）这段代码中的ebx和ecx来自哪里，是什么含义，为什么要通过这些代码将其写到子进程的内核栈中？（3）这段代码中的ebp来自哪里，是什么含义，为什么要做这样的设置？可以不设置吗？为什么？ 答：（1）eax=0，子程序第一次执行，fork返回0表示子程序返回。父进程的fork返回子程序的pid。 （2）ebx，ecx来自于switch_to,表示的是switch_to函数的参数，写入到子进程的内核栈是进程切换switch （3）C调用汇编要的处理要求。

3. 为什么要在切换完LDT之后要重新设置fs=0x17？而且为什么重设操作要出现在切换完LDT之后，出现在LDT之前又会怎么样？ 答： 待定

代码补丁

```
diff --git a/linux-0.11/include/linux/sched.h b/linux-
0.11/include/linux/sched.h
index 21ab8ea..3482f65 100644
--- a/linux-0.11/include/linux/sched.h
+++ b/linux-0.11/include/linux/sched.h
@@ -82,6 +82,7 @@ struct task_struct {
     long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
     long counter;
     long priority;
+    long kernelstack;
     long signal;
     struct sigaction sigaction[32];
     long blocked;    /* bitmap of masked signals */
@@ -114,6 +115,7 @@ struct task_struct {
  */
 #define INIT_TASK \
 /* state etc */    { 0,15,15, \
+/* kernel stack*/ PAGE_SIZE+(long)&init_task,\
 /* signals */    0,{{},},0, \
 /* ec,brk... */    0,0,0,0,0,0, \
 /* pid etc.. */    0,-1,0,0,0, \
diff --git a/linux-0.11/kernel/Makefile b/linux-0.11/kernel/Makefile
index 0afa1dc..c73e97c 100644
--- a/linux-0.11/kernel/Makefile
+++ b/linux-0.11/kernel/Makefile
@@ -24,7 +24,7 @@ CPP    =gcc-3.4 -E -nostdinc -I../include
     $(CC) $(CFLAGS) \
     -c -o $*.o $<

-OBJS  = sched.o system_call.o traps.o asm.o fork.o \
+OBJS  =sched.o system_call.o traps.o asm.o fork.o \
     panic.o printk.o vsprintf.o sys.o exit.o \
     signal.o mktime.o

diff --git a/linux-0.11/kernel/fork.c b/linux-0.11/kernel/fork.c
index 2486b13..405c971 100644
--- a/linux-0.11/kernel/fork.c
+++ b/linux-0.11/kernel/fork.c
@@ -17,6 +17,8 @@
 #include <asm/segment.h>
 #include <asm/system.h>

+extern void first_return_from_kernel(void);
+
 extern void write_verify(unsigned long address);

 long last_pid=0;
@@ -66,6 +68,7 @@ int copy_mem(int nr,struct task_struct * p)
  * information (task[nr]) and sets up the necessary registers. It
  * also copies the data segment in it's entirety.
  */
+#if 0
 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,
@@ -90,6 +93,7 @@ int copy_process(int nr,long ebp,long edi,long esi,long
gs,long none,
     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;
@@ -113,6 +117,8 @@ int copy_process(int nr,long ebp,long edi,long esi,long
gs,long none,
     p->tss.trace_bitmap = 0x80000000;
     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);
@@ -132,6 +138,108 @@ int copy_process(int nr,long ebp,long edi,long esi,long
gs,long none,
     p->state = TASK_RUNNING;    /* do this last, just in case */
     return last_pid;
 }
+#else
+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;
+    long *krnstack;
+
+    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;        /* process leadership doesn't inherit */
+    p->utime = p->stime = 0;
+    p->cutime = p->cstime = 0;
+    p->start_time = jiffies;
+
+#if 1
+    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;
+    *(--krnstack) = ebp;
+    *(--krnstack) = ecx;
+    *(--krnstack) = ebx;
+    *(--krnstack) = 0;
+    p->kernelstack = krnstack;
+
+
+
+//#else
+    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));
+#endif
+
+    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;
+}
+#endif

 int find_empty_process(void)
 {
diff --git a/linux-0.11/kernel/sched.c b/linux-0.11/kernel/sched.c
index 15d839b..e8c05c1 100644
--- a/linux-0.11/kernel/sched.c
+++ b/linux-0.11/kernel/sched.c
@@ -64,6 +64,8 @@ struct task_struct *last_task_used_math = NULL;

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

+struct tss_struct *tss = &(init_task.task.tss);
+
 long user_stack [ PAGE_SIZE>>2 ] ;

 struct {
@@ -105,6 +107,7 @@ void schedule(void)
 {
     int i,next,c;
     struct task_struct ** p;
+    struct task_struct *pnext=&init_task.task;

 /* check alarm, wake up any interruptible tasks that have got a signal */

@@ -130,7 +133,7 @@ void schedule(void)
             if (!*--p)
                 continue;
             if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
-                c = (*p)->counter, next = i;
+                c = (*p)->counter, next = i,pnext = *p;
         }
         if (c) break;
         for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
@@ -138,7 +141,8 @@ void schedule(void)
                 (*p)->counter = ((*p)->counter >> 1) +
                         (*p)->priority;
     }
-    switch_to(next);
+    //switch_to(next);
+    switch_to_by_stack(pnext, _LDT(next));
 }

 int sys_pause(void)
diff --git a/linux-0.11/kernel/system_call.s b/linux-0.11/kernel/system_call.s
index 05891e1..e89a1ec 100644
--- a/linux-0.11/kernel/system_call.s
+++ b/linux-0.11/kernel/system_call.s
@@ -45,12 +45,16 @@ EFLAGS        = 0x24
 OLDESP        = 0x28
 OLDSS         = 0x2C

+ESP0 = 4 #
+KERNEL_STACK = 12  #add kernel stack offset
+
 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)
+kernalstack = 12
+signal    = 16   # 12
+sigaction =20  # 16        # MUST be 16 (=len of sigaction)
+blocked = ((33*16)+4)

 # offsets within sigaction
 sa_handler = 0
@@ -67,6 +71,7 @@ nr_system_calls = 72
```

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

 .align 2
 bad_sys_call:
@@ -92,7 +97,7 @@ system_call:
     movl $0x17,%edx        # fs points to local data space
     mov %dx,%fs
     call sys_call_table(,%eax,4)
-    pushl %eax
+    pushl %eax            #sub process pid
     movl current,%eax
     cmpl $0,state(%eax)        # state
     jne reschedule
@@ -283,3 +288,55 @@ parallel_interrupt:
     outb %al,$0x20
     popl %eax
     iret
+
+.align 2
+switch_to_by_stack:
+    #C调用汇编，需要处理栈帧
+    pushl     %ebp         # ebp基址寄存器 ：保存进入函数时sp栈顶位置;这里是保存上一个函数的ebp
+    movl     %esp,%ebp    # 当前esp栈指针保存在ebp，这是当前函数的栈帧
+    pushl    %ecx
+    pushl    %ebx
+    pushl    %eax
+
+    #取下一个进程的PCB
+    movl    8(%ebp),%ebx   # ebp+8是pnext的地址
+    cmpl    %ebx,current
+    je        1f
+    #切换PCB
+    movl    %ebx,%eax
+    xchgl    %eax,current
+    #TSS中内核栈指针的重写
+    movl tss,    %ecx    # tss->ecx
+    addl $4096,    %ebx # 栈底=ebx+4096
+    movl %ebx,     ESP0(%ecx) # 设置当前任务的内核栈esp0
+    #切换内核栈
+    movl %esp, KERNEL_STACK(%eax)
+    movl 8(%ebp), %ebx # pnex ->ebx
+    movl KERNEL_STACK(%ebx),%esp
+    #切换LDT
+    #movl    $0x17,%ecx
+    #mov        %cx,%fs
+      movl     12(%ebp),%ecx#取出第二个参数，_LDT(next)
+lldt    %cx#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
+
+first_return_from_kernel:
+     popl %edx
+     popl %edi
+     popl %esi
+     pop  %gs
+     pop  %fs
+     pop  %es
+     pop  %ds
+     iret
+
```