KERNEL/system_call.s 文档简要注释

Willweb 于 2003-4-2

简述:

系统调用是 linux 内核的用户程序的接口,用户程序可以通过系统调用陷入到 linux 内核执行相关的功能和操作,同时,用户空间的进程也可以通过系统调用深入到内核空间执行,平时我们用到最多的 fork,exit,execve 等函数和操作实际上都是通过系统调用来执行的.

在 linux0.01 内核中,很多以 sys_开头的函数实际上就是系统调用执行的函数体,它在内核空间执行,虽然它支持的系统调用不如 2.4 内核的多,在内核代码中可以看到很多 sys_函数都是空的(这是因为它需要按照 posix 标准来做),但是它的整体结构和流程和 2.4 等更高版本的内核差不多.

下面可以通过一个简单的用户调用来看看 0.01 的系统调用究竟是如何工作的,这里假设用户在程序中执行一条创建一个新进程的函数----fork(),这里与平时编程的函数调用不同,这里找不到 fork 的函数体,实际上是通过 unistd.h 中的_syscall 宏来将当前的 fork 操作转化,具体转化的方式是通过 syscall0(int,fork)宏将 fork 的系统调用号 3 装入 eax 寄存器中,然后就通过中断 0x80 来进入 kernel/systemcall.s 中的_system_call 执行.这里注意,在 sched_init 中将_system_call 注入到 idt 中,且对应的是 0x80 这个中断向量号.

当通过 0x80 中断之后,cpu 自动保存一些寄存器的值到堆栈中,系统调用可以在用户态和核心态调用,但是它的执行是在核心态执行,所以,如果是在用户态调用的话,int 0x80 的时候会自动的将用户态的 ss 和 esp 保存到堆栈中,同时将 eflags,cs 和 eip 也保存返回地址,如果是在内核态调用系统调用,那么就不需要保存 ss 和 esp 了,因为执行完系统调用之后堆栈段和堆栈指针并不切换.

当进入到_system_call 之后,就执行下面的操作了.

```
/*
 *  system_call.s  contains the system-call low-level handling routines.
 * This also contains the timer-interrupt handler, as some of the code is
 * the same. The hd-interrupt is also here.
 *
 * NOTE: This code handles signal-recognition, which happens every time
 * after a timer-interrupt and after each system call. Ordinary interrupts
 * don't handle signal-recognition, as that would clutter them up totally
 * unnecessarily.
 *
 * Stack layout in 'ret_from_system_call':
 *//下面是在系统调用过程中内核堆栈的参数顺序,系统调用和平时的应用程序中的函数调用不同,平时我们的函数调用是通过堆栈传值,但是系统调用是通过寄存器传值,因为这里可能需要更改堆栈段
 *     0(%esp) - %eax
 *     4(%esp) - %ebx
 *     8(%esp) - %ecx
 *     C(%esp) - %edx
 *    10(%esp) - %fs
 *    14(%esp) - %es
 *    18(%esp) - %ds
 *    1C(%esp) - %eip
 *    20(%esp) - %cs
 *    24(%esp) - %eflags
 *    28(%esp) - %oldesp
 *    2C(%esp) - %oldss
 */

//下面的指明了一些参数,其中的 EAX,EBX,ECX,EDX,FS,ES,DS,CS,EFLAGS,OLDESP 和 OLDSS 实际上是他们在堆栈中相对于 esp 的偏移
SIG_CHLD    = 17
EAX             = 0x00
EBX             = 0x04
ECX             = 0x08
EDX             = 0x0C
FS          = 0x10
ES          = 0x14
DS          = 0x18
EIP         = 0x1C
CS          = 0x20
EFLAGS          = 0x24
OLDESP          = 0x28
OLDSS           = 0x2C
```

```
//下面的这些字段是定义的在进程结构中他们的偏移地址
state = 0          # these are offsets into the task-struct.
counter    = 4
priority = 8
signal      = 12
restorer = 16          # address of info-restorer
sig_fn      = 20       # table of 32 signal addresses

//总的系统调用个数
nr_system_calls = 67

.globl _system_call,_sys_fork,_timer_interrupt,_hd_interrupt,_sys_execve

.align 2
//如果系统调用号 EAX 超过了总的系统调用个数,那么参数不对,将返回值为-1 返回
bad_sys_call:
        movl $-1,%eax
        iret
.align 2
//系统调用完成之后,如果当前进程的状态不为运行状态或者时间片刚好用光,那么重新选择一个进程运行
reschedule:
        pushl $ret_from_sys_call
        jmp _schedule
.align 2

//系统调用的入口
_system_call:
        //比较调用号是否超出了总的系统调用界限
        cmpl $nr_system_calls-1,%eax
        ja bad_sys_call

        //将所有的段和寄存器都保存到堆栈中,因为他们需要在系统调用过程中传递参数,前面提到了,ss,esp,eflags,cs 和 eip 是系统进入中断的时候硬件自动完成的
        push %ds
        push %es
        push %fs
        pushl %edx
        pushl %ecx             # push %ebx,%ecx,%edx as parameters
        pushl %ebx             # to the system call
        //下面将 ds 和 es 置为内核代码段
        movl $0x10,%edx          # set up ds,es to kernel space
        mov %dx,%ds
        mov %dx,%es
```

```
        //同时把 fs 设置为用户数据段,因为后面可以看到它用来保存一些值
        movl $0x17,%edx         # fs points to local data space
        mov %dx,%fs
        //调用系统调用表中的对应函数,因为一个系统调用函数的地址是 4 个字节的,所以这里是 eax*4 了
        call _sys_call_table(,%eax,4)
        //当系统调用函数执行完之后,自动将返回值保存到 eax 中
        pushl %eax
        //比较当前进程是否在运行和时间片是否到了,如果不能继续运行,那么就重新调度进程
        movl _current,%eax
        cmpl $0,state(%eax)             # state
        jne reschedule
        cmpl $0,counter(%eax)           # counter
        je reschedule
//返回系统调用,在返回系统调用的时候,检查进程是否有信号,如果有,那么就处理对应的函数
ret_from_sys_call:
        movl _current,%eax              # task[0] cannot have signals
        //init 进程不能够接收任何信号,所以如果是 init 进程就不需要检查信号位图
        cmpl _task,%eax
        je 3f
        //检查系统调用结束后是否需要返回用户态,如果不返回用户态,那么就不会执行信号处理函数,因为信号处理函数是在用户态执行的,而且是进程返回用户态的是否就执行
        movl CS(%esp),%ebx          # was old code segment supervisor
        testl $3,%ebx               # mode? If so - don't check signals
        je 3f
        //同样检查堆栈段
        cmpw $0x17,OLDSS(%esp)          # was stack segment = 0x17 ?
        jne 3f
        //如果要返回用户态,那么就逐个检查 signal 对应的 bit
2:      movl signal(%eax),%ebx              # signals (bitmap, 32 signals)
        //检查 ebx 中是否有对应位为 1 的,也就是是否 有信号,如果有,那么索引放到 ecx 中
        bsfl %ebx,%ecx                  # %ecx is signal nr, return if none
        //如果没有,那么就不执行
        je 3f
        //清对应的位
        btrl %ecx,%ebx                  # clear it
        movl %ebx,signal(%eax)
        //将进程对应的信号处理函数地址放入 ebx,检查做何种处理
        movl sig_fn(%eax,%ecx,4),%ebx  # %ebx is signal handler address
        cmpl $1,%ebx
        //如果为 0 那么就执行系统默认的信号处理函数
        jb default_signal           # 0 is default signal handler – exit
```

```
        //如果为 1 就忽略这个信号
        je 2b                   # 1 is ignore - find next signal
        //否则就执行次信号处理函数,将其地址放到 eip 中,这样信号切换回来之后就开始运行它
        movl $0,sig_fn(%eax,%ecx,4)     # reset signal handler address
        incl %ecx
        xchgl %ebx,EIP(%esp)        # put new return address on stack
        //因为要用到一些参数,所以需要对参数地址进行检查,以免用户态的地址调用了内核的
        subl $28,OLDESP(%esp)
        movl OLDESP(%esp),%edx      # push old return address on stack
        pushl %eax                  # but first check that it's ok.
        pushl %ecx
        pushl $28
        pushl %edx
        call _verify_area
        popl %edx
        addl $4,%esp
        popl %ecx
        popl %eax
        //将所有用到的一些重要字段保存在 fs 段中
        movl restorer(%eax),%eax
        movl %eax,%fs:(%edx)        # flag/reg restorer
        movl %ecx,%fs:4(%edx)           # signal nr
        movl EAX(%esp),%eax
        movl %eax,%fs:8(%edx)           # old eax
        movl ECX(%esp),%eax
        movl %eax,%fs:12(%edx)          # old ecx
        movl EDX(%esp),%eax
        movl %eax,%fs:16(%edx)          # old edx
        movl EFLAGS(%esp),%eax
        movl %eax,%fs:20(%edx)          # old eflags
        movl %ebx,%fs:24(%edx)          # old return addr
        //返回之前,恢复压入的参数
3:      popl %eax
        popl %ebx
        popl %ecx
        popl %edx
        pop %fs
        pop %es
        pop %ds
        iret

//信号的默认处理函数,如果是 SIG_CHLD 信号,就处理它,否则就 exit
default_signal:
        incl %ecx
```

```
        cmpl $SIG_CHLD,%ecx
        je 2b
        pushl %ecx
        call _do_exit
        addl $4,%esp
        jmp 3b

.align 2
_timer_interrupt:
        push %ds
        push %es
        push %fs
        pushl %edx
        pushl %ecx
        pushl %ebx
        pushl %eax
        movl $0x10,%eax
        mov %ax,%ds
        mov %ax,%es
        movl $0x17,%eax
        mov %ax,%fs
        //在时间中断中将 jiffies 全局变量加 1
        incl _jiffies
        //发 EOI 指令给 EOI 寄存器
        movb $0x20,%al
        outb %al,$0x20
        movl CS(%esp),%eax
        andl $3,%eax        # %eax is CPL (0 or 3, 0=supervisor)
        pushl %eax
        //在这里 eax 就是 do_timer 中的 cpl 了
        call _do_timer      # 'do_timer(long CPL)' does everything from
        addl $4,%esp        # task switching to accounting ...
        jmp ret_from_sys_call

.align 2
_sys_execve:
        lea EIP(%esp),%eax
        pushl %eax
        call _do_execve
        addl $4,%esp
        ret

.align 2
_sys_fork:
```

```
        call _find_empty_process
        testl %eax,%eax
        js 1f
        push %gs
        pushl %esi
        pushl %edi
        pushl %ebp
        pushl %eax
        call _copy_process
        addl $20,%esp
1:      ret

_hd_interrupt:
        pushl %eax
        pushl %ecx
        pushl %edx
        push %ds
        push %es
        push %fs
        movl $0x10,%eax
        mov %ax,%ds
        mov %ax,%es
        movl $0x17,%eax
        mov %ax,%fs
        movb $0x20,%al
        outb %al,$0x20          # EOI to interrupt controller #1
        jmp 1f                  # give port chance to breathe
1:      jmp 1f
1:      outb %al,$0xA0          # same to controller #2
        movl _do_hd,%eax
        testl %eax,%eax
        jne 1f
        movl $_unexpected_hd_interrupt,%eax
1:      call *%eax              # "interesting" way of handling intr.
        pop %fs
        pop %es
        pop %ds
        popl %edx
        popl %ecx
        popl %eax
        iret
```