程序 8-3 linux/kernel/sys_call.s

```
/*
 *  linux/kernel/system_call.s
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *  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- and flopppy-interrupts are 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':
 *
 *       0(%esp) - %eax
 *       4(%esp) - %ebx
 *       8(%esp) - %ecx
 *       C(%esp) - %edx
 *      10(%esp) - original %eax        (-1 if not system call)
 *      14(%esp) - %fs
 *      18(%esp) - %es
 *      1C(%esp) - %ds
 *      20(%esp) - %eip
 *      24(%esp) - %cs
 *      28(%esp) - %eflags
 *      2C(%esp) - %oldesp
 *      30(%esp) - %oldss
 */
/*
 * system_call.s 文件包含系统调用（system-call）底层处理子程序。由于有些代码比较类似，
 * 所以同时也包括时钟中断处理（timer-interrupt）句柄。硬盘和软盘的中断处理程序也在这里。
 *
 * 注意：这段代码处理信号（signal）识别，在每次时钟中断和系统调用之后都会进行识别。一般
 * 中断过程并不处理信号识别，因为会给系统造成混乱。
 *
 * 从系统调用返回（'ret_from_system_call'）时堆栈的内容见上面 19-30 行。
 */
# 上面 Linus 原注释中一般中断过程是指除了系统调用中断（int 0x80）和时钟中断（int 0x20）
# 以外的其他中断。这些中断会在内核态或用户态随机发生，若在这些中断过程中也处理信号识别
# 的话，就有可能与系统调用中断和时钟中断过程中对信号的识别处理过程相冲突，，违反了内核
# 代码非抢占原则。因此系统既无必要在这些“其他”中断中处理信号，也不允许这样做。

SIG_CHLD        = 17            # 定义 SIG_CHLD 信号（子进程停止或结束）。

EAX             = 0x00          # 堆栈中各个寄存器的偏移位置。
EBX             = 0x04
ECX             = 0x08
EDX             = 0x0C
```

```
40 ORIG_EAX        = 0x10              # 如果不是系统调用（是其它中断）时，该值为-1。
41 FS              = 0x14
42 ES              = 0x18
43 DS              = 0x1C
44 EIP             = 0x20              # 44 -- 48 行 由 CPU 自动入栈。
45 CS              = 0x24
46 EFLAGS          = 0x28
47 OLDESP          = 0x2C              # 当特权级变化时，原堆栈指针也会入栈。
48 OLDSS           = 0x30
49
   # 以下这些是任务结构（task_struct）中变量的偏移值，参见 include/linux/sched.h，105 行开始。
50 state   = 0            # these are offsets into the task-struct.  # 进程状态码。
51 counter = 4            # 任务运行时间计数(递减)（滴答数），运行时间片。
52 priority = 8           # 运行优先数。任务开始运行时 counter=priority，越大则运行时间越长。
53 signal  = 12           # 是信号位图，每个比特位代表一种信号，信号值=位偏移值+1。
54 sigaction = 16         # MUST be 16 (=len of sigaction) # sigaction 结构长度必须是 16 字节。
55 blocked = (33*16)      # 受阻塞信号位图的偏移量。
56
   # 以下定义在 sigaction 结构中的偏移量，参见 include/signal.h，第 55 行开始。
57 # offsets within sigaction
58 sa_handler = 0                     # 信号处理过程的句柄（描述符）。
59 sa_mask = 4                        # 信号屏蔽码。
60 sa_flags = 8                       # 信号集。
61 sa_restorer = 12                   # 恢复函数指针，参见 kernel/signal.c 程序说明。
62
63 nr_system_calls = 82               # Linux 0.12 版内核中的系统调用总数。
64
65 ENOSYS = 38                        # 系统调用号出错码。
66
67 /*
68  * Ok, I get parallel printer interrupts while using the floppy for some
69  * strange reason. Urgel. Now I just ignore them.
70  */
   /*
    * 好了，在使用软驱时我收到了并行打印机中断，很奇怪。呵，现在不管它。
    */
71 .globl _system_call,_sys_fork,_timer_interrupt,_sys_execve
72 .globl _hd_interrupt,_floppy_interrupt,_parallel_interrupt
73 .globl _device_not_available, _coprocessor_error
74
   # 系统调用号错误时将返回出错码-ENOSYS。
75 .align 2                           # 内存 4 字节对齐。
76 bad_sys_call:
77         pushl $-ENOSYS             # eax 中置-ENOSYS。
78         jmp ret_from_sys_call

   # 重新执行调度程序入口。调度程序 schedule()在（kernel/sched.c，119 行处开始。
   # 当调度程序 schedule()返回时就从 ret_from_sys_call 处（107 行）继续执行。
79 .align 2
80 reschedule:
81         pushl $ret_from_sys_call   # 将 ret_from_sys_call 的地址入栈（107 行）。
82         jmp _schedule
```

```
#### int 0x80 --linux 系统调用入口点（调用中断 int 0x80，eax 中是调用号）。
.align 2
_system_call:
        push %ds                # 保存原段寄存器值。
        push %es
        push %fs
        pushl %eax              # save the orig_eax    # 保存 eax 原值。

# 一个系统调用最多可带有 3 个参数，也可以不带参数。下面入栈的 ebx、ecx 和 edx 中放着系统
# 调用相应 C 语言函数（见第 99 行）的调用参数。这几个寄存器入栈的顺序是由 GNU gcc 规定的，
# ebx 中可存放第 1 个参数，ecx 中存放第 2 个参数，edx 中存放第 3 个参数。
# 系统调用语句可参见头文件 include/unistd.h 中第 150 到 200 行的系统调用宏。
        pushl %edx
        pushl %ecx              # push %ebx,%ecx,%edx as parameters
        pushl %ebx              # to the system call

# 在保存过段寄存器之后，让 ds,es 指向内核数据段，而 fs 指向当前局部数据段，即指向执行本
# 次系统调用的用户程序的数据段。注意，在 Linux 0.12 中内核给任务分配的代码和数据内存段
# 是重叠的，它们的段基址和段限长相同。参见 fork.c 程序中 copy_mem()函数。
        movl $0x10,%edx         # set up ds,es to kernel space
        mov %dx,%ds
        mov %dx,%es
        movl $0x17,%edx         # fs points to local data space
        mov %dx,%fs

        cmpl _NR_syscalls,%eax  # 调用号如果超出范围的话就跳转。
        jae bad_sys_call

# 下面这句操作数的含义是：调用地址=[_sys_call_table + %eax * 4]。参见程序后的说明。
# sys_call_table[]是一个指针数组，定义在 include/linux/sys.h 中，该数组中设置了内核
# 所有 82 个系统调用 C 处理函数的地址。
        call _sys_call_table(,%eax,4)   # 间接调用指定功能 C 函数。
        pushl %eax                      # 把系统调用返回值入栈。

# 下面 101-106 行查看当前任务的运行状态。如果不在就绪状态（state 不等于 0）就去执行调度
# 程序。如果该任务在就绪状态，但是其时间片已经用完（counter=0），则也去执行调度程序。
# 例如当后台进程组中的进程执行控制终端读写操作时，那么默认条件下该后台进程组所有进程
# 会收到 SIGTTIN 或 SIGTTOU 信号，导致进程组中所有进程处于停止状态。而当前进程则会立刻
# 返回。
2:
        movl _current,%eax      # 取当前任务（进程）数据结构指针➔eax。
        cmpl $0,state(%eax)     # state
        jne reschedule
        cmpl $0,counter(%eax)   # counter
        je reschedule

# 以下这段代码执行从系统调用 C 函数返回后，对信号进行识别处理。其他中断服务程序退出时也
# 将跳转到这里进行处理后才退出中断过程，例如后面 131 行上的处理器出错中断 int 16。
# 首先判别当前任务是否是初始任务 task0，如果是则不必对其进行信号量方面的处理，直接返回。
# 109 行上的_task 对应 C 程序中的 task[]数组，直接引用 task 相当于引用 task[0]。
ret_from_sys_call:
        movl _current,%eax
        cmpl _task,%eax         # task[0] cannot have signals
```

```
110         je 3f                       # 向前(forward)跳转到标号 3 处退出中断处理。

    # 通过对原调用程序代码选择符的检查来判断调用程序是否是用户任务。如果不是则直接退出中断。
    # 这是因为任务在内核态执行时不可抢占。否则对任务进行信号量的识别处理。这里比较选择符是
    # 否为用户代码段的选择符 0x000f（RPL=3，局部表，代码段）来判断是否为用户任务。如果不是
    # 则说明是某个中断服务程序（例如中断 16）跳转到第 107 行执行到此，于是跳转退出中断程序。
    # 另外，如果原堆栈段选择符不为 0x17（即原堆栈不在用户段中），也说明本次系统调用的调用者
    # 不是用户任务，则也退出。
111         cmpw $0x0f,CS(%esp)         # was old code segment supervisor ?
112         jne 3f
113         cmpw $0x17,OLDSS(%esp)      # was stack segment = 0x17 ?
114         jne 3f

    # 下面这段代码（115-128）用于处理当前任务中的信号。首先取当前任务结构中的信号位图（32 位，
    # 每位代表 1 种信号），然后用任务结构中的信号阻塞（屏蔽）码，阻塞不允许的信号位，取得数值
    # 最小的信号值，再把原信号位图中该信号对应的位复位（置 0），最后将该信号值作为参数之一调
    # 用 do_signal()。do_signal()在（kernel/signal.c,128）中，其参数包括 13 个入栈的信息。
    # 在 do_signal()或信号处理函数返回之后，若返回值不为 0 则再看看是否需要切换进程或继续处理
    # 其它信号。
115         movl signal(%eax),%ebx      # 取信号位图➔ebx，每 1 位代表 1 种信号，共 32 个信号。
116         movl blocked(%eax),%ecx     # 取阻塞（屏蔽）信号位图➔ecx。
117         notl %ecx                   # 每位取反。
118         andl %ebx,%ecx              # 获得许可的信号位图。
119         bsfl %ecx,%ecx              # 从低位（位 0）开始扫描位图，看是否有 1 的位，
                                        # 若有，则 ecx 保留该位的偏移值（即第几位 0--31）。
120         je 3f                       # 如果没有信号则向前跳转退出。
121         btrl %ecx,%ebx              # 复位该信号（ebx 含有原 signal 位图）。
122         movl %ebx,signal(%eax)      # 重新保存 signal 位图信息➔current->signal。
123         incl %ecx                   # 将信号调整为从 1 开始的数（1--32）。
124         pushl %ecx                  # 信号值入栈作为调用 do_signal 的参数之一。
125         call _do_signal             # 调用 C 函数信号处理程序（kernel/signal.c，128）。
126         popl %ecx                   # 弹出入栈的信号值。
127         testl %eax, %eax            # 测试返回值，若不为 0 则跳转到前面标号 2（101 行）处。
128         jne 2b                      # see if we need to switch tasks, or do more signals

129 3:      popl %eax                   # eax 中含有第 100 行入栈的系统调用返回值。
130         popl %ebx
131         popl %ecx
132         popl %edx
133         addl $4, %esp               # skip orig_eax     # 跳过（丢弃）原 eax 值。
134         pop %fs
135         pop %es
136         pop %ds
137         iret
138
    #### int16 -- 处理器错误中断。  类型：错误；无错误码。
    # 这是一个外部的基于硬件的异常。当协处理器检测到自己发生错误时，就会通过 ERROR 引脚
    # 通知 CPU。下面代码用于处理协处理器发出的出错信号。并跳转去执行 C 函数 math_error()
    # （kernel/math/error.c 11）。返回后将跳转到标号 ret_from_sys_call 处继续执行。
139 .align 2
140 _coprocessor_error:
141         push %ds
142         push %es
```

```
143         push %fs
144         pushl $-1                   # fill in -1 for orig_eax  # 填-1，表明不是系统调用。
145         pushl %edx
146         pushl %ecx
147         pushl %ebx
148         pushl %eax
149         movl $0x10,%eax             # ds,es 置为指向内核数据段。
150         mov %ax,%ds
151         mov %ax,%es
152         movl $0x17,%eax             # fs 置为指向局部数据段（出错程序的数据段）。
153         mov %ax,%fs
154         pushl $ret_from_sys_call    # 把下面调用返回的地址入栈。
155         jmp _math_error             # 执行 math_error()（kernel/math/error.c，11）。
156
    #### int7 -- 设备不存在或协处理器不存在。  类型：错误；无错误码。
    # 如果控制寄存器 CR0 中 EM（模拟）标志置位，则当 CPU 执行一个协处理器指令时就会引发该
    # 中断，这样 CPU 就可以有机会让这个中断处理程序模拟协处理器指令（181 行）。
    # CR0 的交换标志 TS 是在 CPU 执行任务转换时设置的。TS 可以用来确定什么时候协处理器中的
    # 内容与 CPU 正在执行的任务不匹配了。当 CPU 在运行一个协处理器转义指令时发现 TS 置位时，
    # 就会引发该中断。此时就可以保存前一个任务的协处理器内容，并恢复新任务的协处理器执行
    # 状态（176 行）。参见 kernel/sched.c，92 行。该中断最后将转移到标号 ret_from_sys_call
    # 处执行下去（检测并处理信号）。
157 .align 2
158 _device_not_available:
159         push %ds
160         push %es
161         push %fs
162         pushl $-1                   # fill in -1 for orig_eax  # 填-1，表明不是系统调用。
163         pushl %edx
164         pushl %ecx
165         pushl %ebx
166         pushl %eax
167         movl $0x10,%eax             # ds,es 置为指向内核数据段。
168         mov %ax,%ds
169         mov %ax,%es
170         movl $0x17,%eax             # fs 置为指向局部数据段（出错程序的数据段）。
171         mov %ax,%fs
    # 清 CR0 中任务已交换标志 TS，并取 CR0 值。若其中协处理器仿真标志 EM 没有置位，说明不是
    # EM 引起的中断，则恢复任务协处理器状态，执行 C 函数 math_state_restore()，并在返回时
    # 去执行 ret_from_sys_call 处的代码。
172         pushl $ret_from_sys_call    # 把下面跳转或调用的返回地址入栈。
173         clts                        # clear TS so that we can use math
174         movl %cr0,%eax
175         testl $0x4,%eax             # EM (math emulation bit)
176         je _math_state_restore      # 执行 math_state_restore()（kernel/sched.c，92 行）。

    # 若 EM 标志置位，则去执行数学仿真程序 math_emulate()。
177         pushl %ebp
178         pushl %esi
179         pushl %edi
180         pushl $0                    # temporary storage for ORIG_EIP
181         call _math_emulate          # 调用 C 函数（math/math_emulate.c，476 行）。
182         addl $4,%esp                # 丢弃临时存储。
```

```
183         popl %edi
184         popl %esi
185         popl %ebp
186         ret                         # 这里的 ret 将跳转到 ret_from_sys_call(107 行)。
187
    #### int32 -- (int 0x20) 时钟中断处理程序。中断频率设置为 100Hz(include/linux/sched.h,4),
    # 定时芯片 8253/8254 是在(kernel/sched.c,438)处初始化的。因此这里 jiffies 每 10 毫秒加 1。
    # 这段代码将 jiffies 增 1，发送结束中断指令给 8259 控制器，然后用当前特权级作为参数调用
    # C 函数 do_timer(long CPL)。当调用返回时转去检测并处理信号。
188 .align 2
189 _timer_interrupt:
190         push %ds                    # save ds,es and put kernel data space
191         push %es                    # into them. %fs is used by _system_call
192         push %fs                    # 保存 ds、es 并让其指向内核数据段。fs 将用于 system_call。
193         pushl $-1                   # fill in -1 for orig_eax  # 填-1，表明不是系统调用。

    # 下面我们保存寄存器 eax、ecx 和 edx。这是因为 gcc 编译器在调用函数时不会保存它们。这里也
    # 保存了 ebx 寄存器，因为在后面 ret_from_sys_call 中会用到它。
194         pushl %edx                  # we save %eax,%ecx,%edx as gcc doesn't
195         pushl %ecx                  # save those across function calls. %ebx
196         pushl %ebx                  # is saved as we use that in ret_sys_call
197         pushl %eax
198         movl $0x10,%eax             # ds,es 置为指向内核数据段。
199         mov %ax,%ds
200         mov %ax,%es
201         movl $0x17,%eax             # fs 置为指向局部数据段（程序的数据段）。
202         mov %ax,%fs
203         incl _jiffies
    # 由于初始化中断控制芯片时没有采用自动 EOI，所以这里需要发指令结束该硬件中断。
204         movb $0x20,%al              # EOI to interrupt controller #1
205         outb %al,$0x20

    # 下面从堆栈中取出执行系统调用代码的选择符（CS 段寄存器值）中的当前特权级别(0 或 3)并压入
    # 堆栈，作为 do_timer 的参数。do_timer()函数执行任务切换、计时等工作，在 kernel/sched.c，
    # 324 行实现。
206         movl CS(%esp),%eax
207         andl $3,%eax                # %eax is CPL (0 or 3, 0=supervisor)
208         pushl %eax
209         call _do_timer              # 'do_timer(long CPL)' does everything from
210         addl $4,%esp                # task switching to accounting ...
211         jmp ret_from_sys_call
212
    #### 这是 sys_execve()系统调用。取中断调用程序的代码指针作为参数调用 C 函数 do_execve()。
    # do_execve()在 fs/exec.c，207 行。
213 .align 2
214 _sys_execve:
215         lea EIP(%esp),%eax          # eax 指向堆栈中保存用户程序 eip 指针处。
216         pushl %eax
217         call _do_execve
218         addl $4,%esp                # 丢弃调用时压入栈的 EIP 值。
219         ret
220
    #### sys_fork()调用，用于创建子进程，是 system_call 功能 2。原形在 include/linux/sys.h 中。
```

```
    # 首先调用 C 函数 find_empty_process()，取得一个进程号 last_pid。若返回负数则说明目前任务
    # 数组已满。然后调用 copy_process()复制进程。
221 .align 2
222 _sys_fork:
223         call _find_empty_process    # 为新进程取得进程号 last_pid。（kernel/fork.c，143）。
224         testl %eax,%eax             # 在 eax 中返回进程号。若返回负数则退出。
225         js 1f
226         push %gs
227         pushl %esi
228         pushl %edi
229         pushl %ebp
230         pushl %eax
231         call _copy_process          # 调用 C 函数 copy_process()（kernel/fork.c，68）。
232         addl $20,%esp               # 丢弃这里所有压栈内容。
233 1:      ret
234
    #### int 46 -- (int 0x2E) 硬盘中断处理程序，响应硬件中断请求 IRQ14。
    # 当请求的硬盘操作完成或出错就会发出此中断信号。(参见 kernel/blk_drv/hd.c)。
    # 首先向 8259A 中断控制从芯片发送结束硬件中断指令(EOI)，然后取变量 do_hd 中的函数指针放入 edx
    # 寄存器中，并置 do_hd 为 NULL，接着判断 edx 函数指针是否为空。如果为空，则给 edx 赋值指向
    # unexpected_hd_interrupt()，用于显示出错信息。随后向 8259A 主芯片送 EOI 指令，并调用 edx 中
    # 指针指向的函数: read_intr()、write_intr()或 unexpected_hd_interrupt()。
235 _hd_interrupt:
236         pushl %eax
237         pushl %ecx
238         pushl %edx
239         push %ds
240         push %es
241         push %fs
242         movl $0x10,%eax             # ds,es 置为内核数据段。
243         mov %ax,%ds
244         mov %ax,%es
245         movl $0x17,%eax             # fs 置为调用程序的局部数据段。
246         mov %ax,%fs
    # 由于初始化中断控制芯片时没有采用自动 EOI，所以这里需要发指令结束该硬件中断。
247         movb $0x20,%al
248         outb %al,$0xA0              # EOI to interrupt controller #1 # 送从 8259A。
249         jmp 1f                      # give port chance to breathe    # 这里 jmp 起延时作用。
250 1:      jmp 1f
    # do_hd 定义为一个函数指针，将被赋值 read_intr()或 write_intr()函数地址。放到 edx 寄存器后
    # 就将 do_hd 指针变量置为 NULL。然后测试得到的函数指针，若该指针为空，则赋予该指针指向 C
    # 函数 unexpected_hd_interrupt()，以处理未知硬盘中断。
251 1:      xorl %edx,%edx
252         movl %edx,_hd_timeout       # hd_timeout 置为 0。表示控制器已在规定时间内产生了中断。
253         xchgl _do_hd,%edx
254         testl %edx,%edx
255         jne 1f                      # 若空，则让指针指向 C 函数 unexpected_hd_interrupt()。
256         movl $_unexpected_hd_interrupt,%edx
257 1:      outb %al,$0x20              # 送 8259A 主芯片 EOI 指令（结束硬件中断）。
258         call *%edx                  # "interesting" way of handling intr.
259         pop %fs                     # 上句调用 do_hd 指向的 C 函数。
260         pop %es
261         pop %ds
```

```
262         popl %edx
263         popl %ecx
264         popl %eax
265         iret
266
    #### int38 -- (int 0x26) 软盘驱动器中断处理程序，响应硬件中断请求 IRQ6。
    # 其处理过程与上面对硬盘的处理基本一样。（kernel/blk_drv/floppy.c）。
    # 首先向 8259A 中断控制器主芯片发送 EOI 指令，然后取变量 do_floppy 中的函数指针放入 eax
    # 寄存器中，并置 do_floppy 为 NULL，接着判断 eax 函数指针是否为空。如为空，则给 eax 赋值指向
    # unexpected_floppy_interrupt ()，用于显示出错信息。随后调用 eax 指向的函数: rw_interrupt,
    # seek_interrupt,recal_interrupt,reset_interrupt 或 unexpected_floppy_interrupt。
267 _floppy_interrupt:
268         pushl %eax
269         pushl %ecx
270         pushl %edx
271         push %ds
272         push %es
273         push %fs
274         movl $0x10,%eax             # ds,es 置为内核数据段。
275         mov %ax,%ds
276         mov %ax,%es
277         movl $0x17,%eax             # fs 置为调用程序的局部数据段。
278         mov %ax,%fs
279         movb $0x20,%al              # 送主 8259A 中断控制器 EOI 指令（结束硬件中断）。
280         outb %al,$0x20              # EOI to interrupt controller #1
    # do_floppy 为一函数指针，将被赋值实际处理 C 函数指针。该指针在被交换放到 eax 寄存器后就将
    # do_floppy 变量置空。然后测试 eax 中原指针是否为空，若是则使指针指向 C 函数
    # unexpected_floppy_interrupt()。
281         xorl %eax,%eax
282         xchgl _do_floppy,%eax
283         testl %eax,%eax             # 测试函数指针是否=NULL?
284         jne 1f                  # 若空，则使指针指向 C 函数 unexpected_floppy_interrupt()。
285         movl $_unexpected_floppy_interrupt,%eax
286 1:      call *%eax                  # "interesting" way of handling intr.    # 间接调用。
287         pop %fs                     # 上句调用 do_floppy 指向的函数。
288         pop %es
289         pop %ds
290         popl %edx
291         popl %ecx
292         popl %eax
293         iret
294
    #### int 39 -- (int 0x27) 并行口中断处理程序，对应硬件中断请求信号 IRQ7。
    # 本版本内核还未实现。这里只是发送 EOI 指令。
295 _parallel_interrupt:
296         pushl %eax
297         movb $0x20,%al
298         outb %al,$0x20
299         popl %eax
300         iret
```