

















## 第5章 内核代码(kernel)

### 5.1 概述

linux/kernel/目录下共包括 10 个 C 语言文件和 2 个汇编语言文件以及一个 kernel 下编译文件的管理配置文件 Makefile。见列表 5-1 所示。其中三个子目录中代码的注释将放在后续章节中进行。本章主要对这 13 个代码文件进行注释。首先我们对所有程序的基本功能进行概括性地总体介绍,以便一开始就对这 12 个文件所实现的功能和它们之间的相互调用关系有个大致的了解,然后逐一对代码进行详细地注释。

列表 5-1 linux/kernel/目录

文件名	大小	最后修改时间(GMT)	说明
 <a href="#">blk_drv/</a>		1991-12-08 14:09:29	
 <a href="#">chr_drv/</a>		1991-12-08 18:36:09	
 <a href="#">math/</a>		1991-12-08 14:09:58	
 <a href="#">Makefile</a>	3309 bytes	1991-12-02 03:21:37	m
 <a href="#">asm.s</a>	2335 bytes	1991-11-18 00:30:28	m
 <a href="#">exit.c</a>	4175 bytes	1991-12-07 15:47:55	m
 <a href="#">fork.c</a>	3693 bytes	1991-11-25 15:11:09	m
 <a href="#">mktime.c</a>	1461 bytes	1991-10-02 14:16:29	m
 <a href="#">panic.c</a>	448 bytes	1991-10-17 14:22:02	m
 <a href="#">printk.c</a>	734 bytes	1991-10-02 14:16:29	m
 <a href="#">sched.c</a>	8242 bytes	1991-12-04 19:55:28	m
 <a href="#">signal.c</a>	2651 bytes	1991-12-07 15:47:55	m
 <a href="#">sys.c</a>	3706 bytes	1991-11-25 19:31:13	m
 <a href="#">system call.s</a>	5265 bytes	1991-12-04 13:56:34	m
 <a href="#">traps.c</a>	4951 bytes	1991-10-30 20:20:40	m
 <a href="#">vsprintf.c</a>	4800 bytes	1991-10-02 14:16:29	m

### 5.2 总体功能描述

该目录下的代码文件从功能上可以分为三类,一类是硬件(异常)中断处理程序文件,一类是系统调用服务处理程序文件,另一类是进程调度等通用功能文件,参见图 1.5。我们现在根据这个分类方式,从实现的功能上进行更详细的说明。

### 5.2.1 硬件中断处理类程序

主要包括两个代码文件：`asm.s` 和 `traps.c` 文件。`asm.s` 用于实现大部分硬件异常所引起的中断的汇编语言处理过程。而 `traps.c` 程序则实现了 `asm.s` 的中断处理过程中调用的 `c` 函数。另外几个硬件中断处理程序在文件 `system_call.s` 和 `mm/page.s` 中实现。

在用户程序（进程）将控制权交给中断处理程序之前，CPU 会首先将至少 12 字节的信息压入中断处理程序的堆栈中。这种情况与一个长调用（段间子程序调用）比较相象。CPU 会将代码段选择符和返回地址的偏移值压入堆栈。另一个与段间调用比较相象的地方是 80386 将信息压入到了目的代码的堆栈上，而不是被中断代码的堆栈。因此当发生中断时，使用的是目的代码的内核态堆栈。另外，CPU 还总是将标志寄存器 `EFLAGS` 的内容压入堆栈。如果优先级级别发生了变化，比如从用户级改变到内核系统级，CPU 还会将原代码的堆栈段值和堆栈指针压入中断程序的堆栈中。对于具有优先级改变时堆栈的内容示意图见图 5-1 所示。

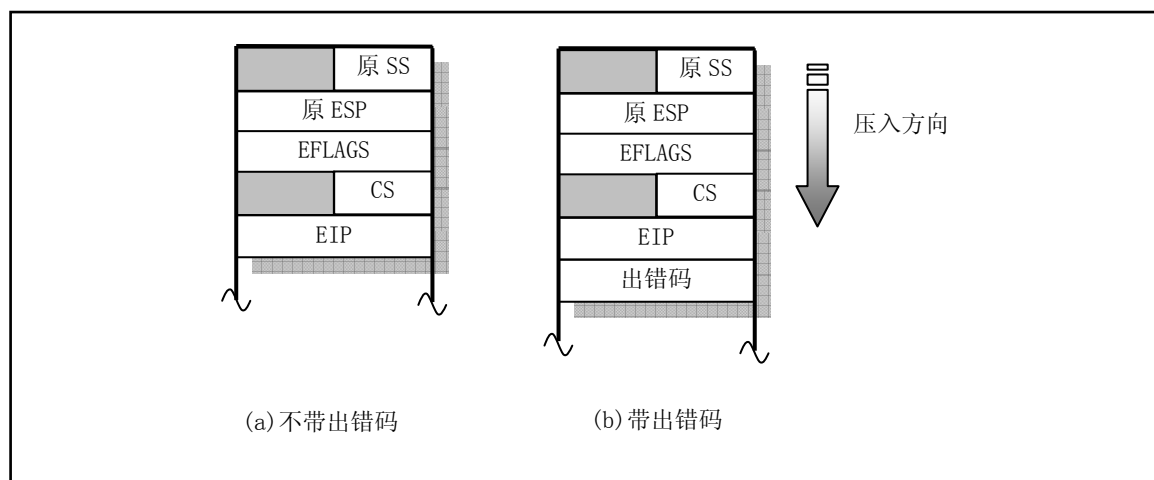


图 5-1 发生中断时堆栈中的内容

`asm.s` 代码文件主要涉及对 Intel 保留中断 `int0--int16` 的处理，其余保留的中断 `int17-int31` 由 Intel 公司留作今后扩充使用。对应于中断控制器芯片各 `IRQ` 发出的 `int32-int47` 的 16 个处理程序将分别在各种硬件（如时钟、键盘、软盘、数学协处理器、硬盘等）初始化程序中处理。Linux 系统调用中断 `int128(0x80)` 的处理则将在 `kernel/system_call.s` 中给出。各个中断的具体定义见代码注释后其它信息一节中的说明。

由于有些异常引起中断时，CPU 内部会产生一个出错代码压入堆栈（异常中断 `int 8` 和 `int10 - int 14`），见图 5-1 (b) 所示，而其它的中断却并不带有这个出错代码（例如被零除出错和边界检查出错等），因此，`asm.s` 程序中将所有中断的处理根据是否携带出错代码而分别进行处理。但处理流程还是一样的。

对一个硬件异常所引起的中断的处理过程见图 5-2 所示。

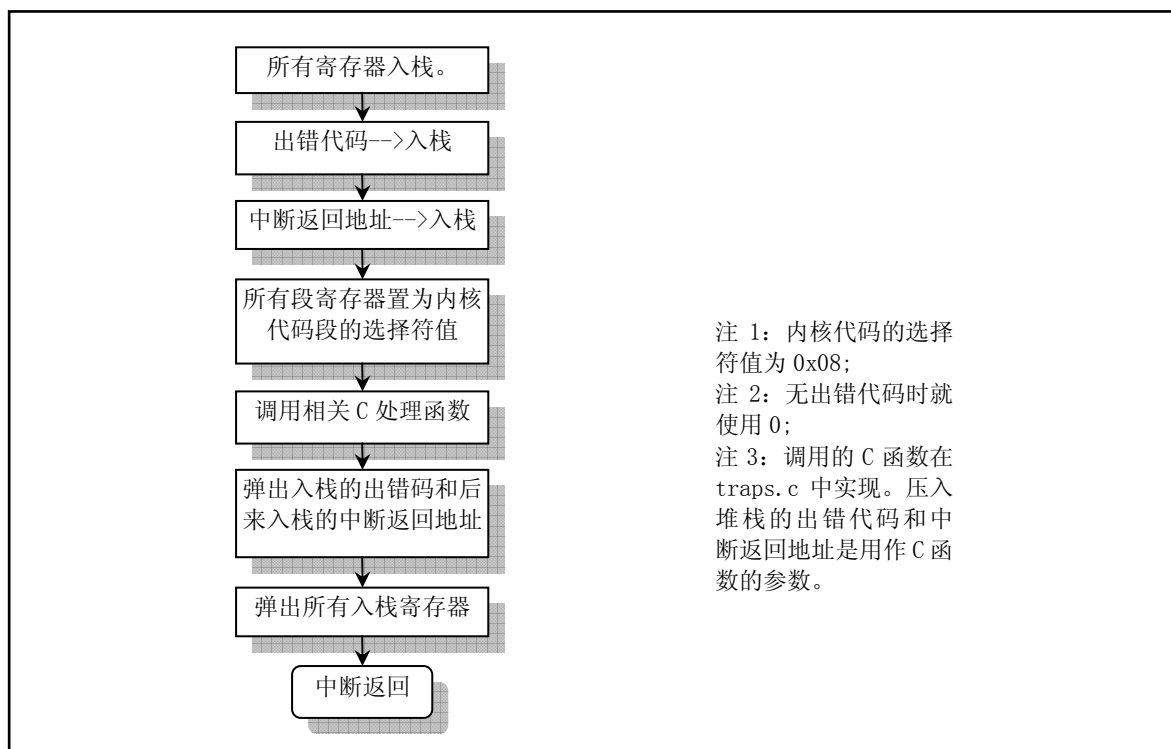


图 5-2 硬件异常（故障、陷阱）所引起的中断处理流程

### 5.2.2 系统调用处理相关程序

Linux 中应用程序调用内核的功能是通过中断调用 int 0x80 进行的，寄存器 eax 中放调用号。因此该中断调用被称为系统调用。实现系统调用的相关文件包括 system\_call.s、fork.c、signal.c、sys.c 和 exit.c 文件。

system\_call.s 程序的作用类似于硬件中断处理中的 asm.s 程序的作用，另外还对时钟中断和硬盘、软盘中断进行处理。而 fork.c 和 signal.c 中的一个函数则类似于 traps.c 程序的作用，为系统中断调用提供 C 处理函数。fork.c 程序提供两个 C 处理函数：find\_empty\_process()和 copy\_process()。signal.c 程序还提供一个处理有关进程信号的函数 do\_signal()，在系统调用中断处理过程中被调用。另外还包括 4 个系统调用 sys\_xxx()函数。

sys.c 和 exit.c 程序实现了其它一些 sys\_xxx()系统调用函数。这些 sys\_xxx()函数都是相应系统调用所需调用的处理函数，有些是使用汇编语言实现的，如 sys\_execve()；而另外一些则用 C 语言实现（例如 signal.c 中的 4 个系统调用函数）。

我们可以根据这些函数的简单命名规则这样来理解：通常以'do\_'开头的中断处理过程中调用的 C 函数，要么是系统调用处理过程中通用的函数，要么是某个系统调用专用的；而以'sys\_'开头的系统调用函数则是指定的系统调用的专用处理函数。例如，do\_signal()函数基本上是所有系统调用都要执行的函数，而 do\_hd()、do\_execve()则是某个系统调用专用的 C 处理函数。

### 5.2.3 其它通用类程序

这些程序包括 schedule.c、mktime.c、panic.c、printk.c 和 vsprintf.c。

schedule.c 程序包括内核调用最频繁的 schedule()、sleep\_on()和 wakeup()函数，是内核的核心调度程序，用于对进程的执行进行切换或改变进程的执行状态。mktime.c 程序中仅包含一个内核使用的时间函数 mktime()，仅在 init/main.c 中被调用一次。panic.c 中包含一个 panic()函数，用于在内核运行出现错误时显示出错信息并停机。printk.c 和 vsprintf.c 是内核显示信息的支持程序，实现了内核专用显示函数

printf()和字符串格式化输出函数 vsprintf()。

## 5.3 Makefile 文件

### 5.3.1 功能简介

编译 linux/kernel/下程序的 make 配置文件，不包括三个子目录。该文件的组成格式与第一章中列表 1.2 的基本相同，在阅读时可以参考列表 1.2 中的有关注释。

### 5.3.2 文件注释

程序 5-1 linux/kernel/Makefile

```

1 #
2 # Makefile for the FREAX-kernel.
3 #
4 # Note! Dependencies are done automagically by 'make dep', which also
5 # removes any old dependencies. DON'T put your own dependencies here
6 # unless it's something special (ie not a .c file).
7 #
8 # FREAX 内核的 Makefile 文件。
9 #
10 # 注意！依赖关系是由'make dep'自动进行的，它也会自动去除原来的依赖信息。不要把你自己的
11 # 依赖关系信息放在这里，除非是特别文件的（也即不是一个.c文件的信息）。
12 # (Linux最初的名字叫FREAX，后来被ftp.funet.fi的管理员改成Linux这个名字)
13
14 AR      =gar      # GNU 的二进制文件处理程序，用于创建、修改以及从归档文件中抽取文件。
15 AS      =gas      # GNU 的汇编程序。
16 LD      =gld      # GNU 的连接程序。
17 LDFLAGS =-s -x    # 连接程序所有的参数，-s 输出文件中省略所有符号信息。-x 删除所有局部符号。
18 CC      =gcc      # GNU C 语言编译器。
19 CFLAGS  =-Wall -O -fstrength-reduce -fomit-frame-pointer -fcombine-regs \
20         -finline-functions -mstring-insns -nostdinc -I../include
21 # C 编译程序选项。-Wall 显示所有的警告信息；-O 优化选项，优化代码长度和执行时间；
22 # -fstrength-reduce 优化循环执行代码，排除重复变量；-fomit-frame-pointer 省略保存不必要
23 # 的框架指针；-fcombine-regs 合并寄存器，减少寄存器类的使用；-finline-functions 将所有简
24 # 单短小的函数代码嵌入调用程序中；-mstring-insns Linux 自己添加的优化选项，以后不再使用；
25 # -nostdinc -I../include 不使用默认路径中的包含文件，而使用这里指定目录中的(../include)。
26 CPP     =gcc -E -nostdinc -I../include
27 # C 前处理选项。-E 只运行 C 前处理，对所有指定的 C 程序进行预处理并将处理结果输出到标准输
28 # 出设备或指定的输出文件中；-nostdinc -I../include 同前。
29
30 # 下面的规则指示 make 利用下面的命令将所有的.c文件编译生成.s汇编程序。该规则的命令
31 # 指使 gcc 采用 CFLAGS 所指定的选项对 C 代码编译后不进行汇编就停止（-S），从而产生与
32 # 输入的各个 C 文件对应的汇编代码文件。默认情况下所产生的汇编程序文件名是原 C 文件名
33 # 去掉.c而加上.s后缀。-o 表示其后是输出文件的名称。其中$*.s（或$@）是自动目标变量，
34 # $<代表第一个先决条件，这里即是符合条件*.c的文件。
35
36 .c.s:
37     $(CC) $(CFLAGS) \
38     -S -o $*.s $<
39 # 下面规则表示将所有.s汇编程序文件编译成.o目标文件。22行是实现该操作的具体命令。

```

```

21 .s.o:
22     $(AS) -c -o $*.o $<
23 .c.o:          # 类似上面, *.c 文件→*.o 目标文件。不进行连接。
24     $(CC) $(CFLAGS) \
25     -c -o $*.o $<
26
27 OBJS = sched.o system_call.o traps.o asm.o fork.o \    # 定义目标文件变量 OBJS。
28     panic.o printk.o vsprintf.o sys.o exit.o \
29     signal.o mktime.o
30
31 kernel.o: $(OBJS)          # 在有了先决条件 OBJS 后使用下面的命令连接成目标 kernel.o
32     $(LD) -r -o kernel.o $(OBJS)
33     sync
34
35 # 下面的规则用于清理工作。当执行'make clean'时, 就会执行 36--40 行上的命令, 去除所有编译
36 # 连接生成的文件。'rm'是文件删除命令, 选项-f 含义是忽略不存在的文件, 并且不显示删除信息。
37 clean:
38     rm -f core *.o *.a tmp_make keyboard.s
39     for i in *.c;do rm -f `basename $$i .c`.s;done
40     (cd chr_drv; make clean)    # 进入 chr_drv/目录; 执行该目录 Makefile 中的 clean 规则。
41     (cd blk_drv; make clean)
42     (cd math; make clean)
43
44 # 下面得目标或规则用于检查各文件之间的依赖关系。方法如下:
45 # 使用字符串编辑程序 sed 对 Makefile 文件 (这里即是自己) 进行处理, 输出为删除 Makefile
46 # 文件中'### Dependencies'行后面的所有行 (下面从 51 开始的行), 并生成 tmp_make
47 # 临时文件 (43 行的作用)。然后对 kernel/目录下的每一个 C 文件执行 gcc 预处理操作。
48 # -M 标志告诉预处理程序输出描述每个目标文件相关性的规则, 并且这些规则符合 make 语法。
49 # 对于每一个源文件, 预处理程序输出一个 make 规则, 其结果形式是相应源程序文件的目标
50 # 文件名加上其依赖关系--该源文件中包含的所有头文件列表。把预处理结果都添加到临时
51 # 文件 tmp_make 中, 然后将该临时文件复制成新的 Makefile 文件。
52 dep:
53     sed '/\##\## Dependencies/q' < Makefile > tmp_make
54     (for i in *.c;do echo -n `echo $$i | sed 's,\.c,\.s,``" "; \
55     $(CPP) -M $$i;done) >> tmp_make
56     cp tmp_make Makefile
57     (cd chr_drv; make dep)    # 对 chr_drv/目录下的 Makefile 文件也作同样的处理。
58     (cd blk_drv; make dep)
59
60 ### Dependencies:
61 exit.s exit.o : exit.c ../include/errno.h ../include/signal.h \
62     ../include/sys/types.h ../include/sys/wait.h ../include/linux/sched.h \
63     ../include/linux/head.h ../include/linux/fs.h ../include/linux/mm.h \
64     ../include/linux/kernel.h ../include/linux/tty.h ../include/termios.h \
65     ../include/asm/segment.h
66 fork.s fork.o : fork.c ../include/errno.h ../include/linux/sched.h \
67     ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
68     ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \
69     ../include/asm/segment.h ../include/asm/system.h
70 mktime.s mktime.o : mktime.c ../include/time.h
71 panic.s panic.o : panic.c ../include/linux/kernel.h ../include/linux/sched.h \
72     ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
73     ../include/linux/mm.h ../include/signal.h

```

---

```

64 printk.s printk.o : printk.c ../include/stdarg.h ../include/stddef.h \
65  ../include/linux/kernel.h
66 sched.s sched.o : sched.c ../include/linux/sched.h ../include/linux/head.h \
67  ../include/linux/fs.h ../include/sys/types.h ../include/linux/mm.h \
68  ../include/signal.h ../include/linux/kernel.h ../include/linux/sys.h \
69  ../include/linux/fdreg.h ../include/asm/system.h ../include/asm/io.h \
70  ../include/asm/segment.h
71 signal.s signal.o : signal.c ../include/linux/sched.h ../include/linux/head.h \
72  ../include/linux/fs.h ../include/sys/types.h ../include/linux/mm.h \
73  ../include/signal.h ../include/linux/kernel.h ../include/asm/segment.h
74 sys.s sys.o : sys.c ../include/errno.h ../include/linux/sched.h \
75  ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
76  ../include/linux/mm.h ../include/signal.h ../include/linux/tty.h \
77  ../include/termios.h ../include/linux/kernel.h ../include/asm/segment.h \
78  ../include/sys/times.h ../include/sys/utsname.h
79 traps.s traps.o : traps.c ../include/string.h ../include/linux/head.h \
80  ../include/linux/sched.h ../include/linux/fs.h ../include/sys/types.h \
81  ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \
82  ../include/asm/system.h ../include/asm/segment.h ../include/asm/io.h
83 vsprintf.s vsprintf.o : vsprintf.c ../include/stdarg.h ../include/string.h

```

---

## 5.4 asm.s 程序

### 5.4.1 功能描述

asm.s 汇编程序中包括大部分 CPU 探测到的异常故障处理的底层代码，也包括数学协处理器（FPU）的异常处理。该程序与 kernel/traps.c 程序有着密切的关系。该程序的主要处理方式是在中断处理程序中调用相应的 C 函数程序，显示出错位置和出错号，然后退出中断。

在阅读这段代码时参照图 5-3 中堆栈变化示意图将是很有帮助的，图中每个行代表 4 个字节。在开始执行程序之前，堆栈指针 esp 指在中断返回地址一栏(图中 esp0 处)。当把将要调用的 C 函数 do\_divide\_error()或其它 C 函数地址入栈后，指针位置是 esp1 处,此时通过交换指令，该函数的地址被放入 eax 寄存器中，而原来 eax 的值被保存到堆栈上。在把一些寄存器入栈后，堆栈指针位置在 esp2 处。当正式调用 do\_divide\_error()之前，程序将开始执行时的 esp0 堆栈指针值压入堆栈，放到了 esp3 处，并在中断返回弹出栈的寄存器之前指针通过加上 8 又回到 esp2 处。

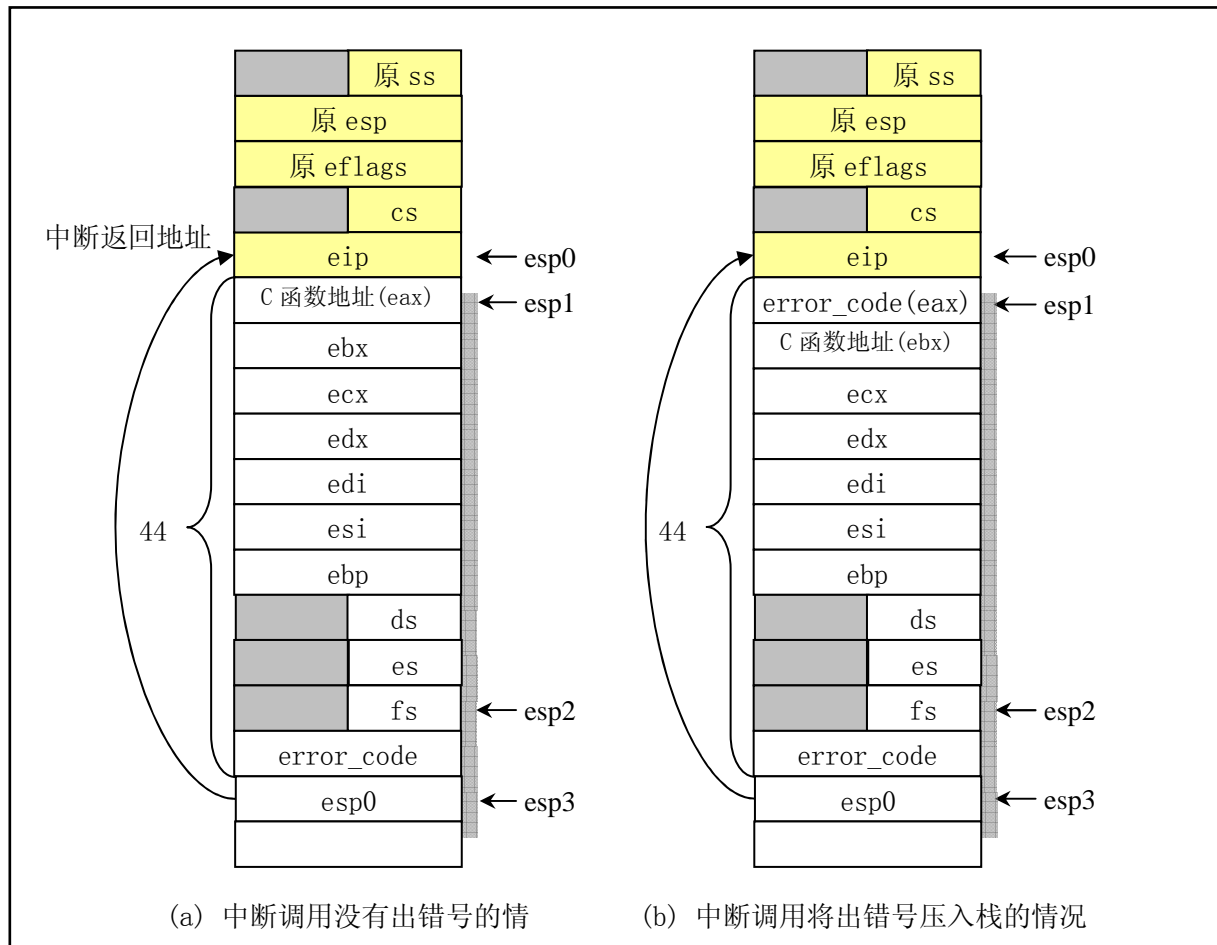


图 5-3 出错处理堆栈变化示意图

正式调用 `do_divide_error()` 之前把出错代码以及 `esp0` 入栈的原因是为了作为调用 C 函数 `do_divide_error()` 的参数。在 `traps.c` 中该函数的原形为：

```
void do_divide_error(long esp, long error_code)。
```

因此在这个 C 函数中就可以打印出出错的位置和错误号。程序中其余异常出错的处理过程与这里描述的过程基本类似。

## 5.4.2 代码注释

程序 5-2 linux/kernel/asm.s

```

1 /*
2  * linux/kernel/asm.s
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 /*
8  * asm.s contains the low-level code for most hardware faults.
9  * page_exception is handled by the mm, so that isn't here. This
10  * file also handles (hopefully) fpu-exceptions due to TS-bit, as

```

```

11 * the fpu must be properly saved/resored. This hasn't been tested.
12 */
/*
   * asm.s 程序中包括大部分的硬件故障（或出错）处理的底层次代码。页异常是由内存管理程序
   * mm 处理的，所以不在这里。此程序还处理（希望是这样）由于 TS-位而造成的 fpu 异常，
   * 因为 fpu 必须正确地进行保存/恢复处理，这些还没有测试过。
   */
13
   # 本代码文件主要涉及对 Intel 保留的中断 int0--int16 的处理（int17-int31 留作今后使用）。
   # 以下是一些全局函数名的声明，其原形在 traps.c 中说明。
14 .globl _divide_error, _debug, _nmi, _int3, _overflow, _bounds, _invalid_op
15 .globl _double_fault, _coprocessor_segment_overnrun
16 .globl _invalid_TSS, _segment_not_present, _stack_segment
17 .globl _general_protection, _coprocessor_error, _irq13, _reserved
18
   # int0 -- （下面这段代码的含义参见图 5.3(a)）。
   # 下面是被零除出错(divide_error)处理代码。标号'_divide_error'实际上是 C 语言函
   # 数 divide_error() 编译后所生成模块中对应的名称。'_do_divide_error' 函数在 traps.c 中。
19 _divide_error:
20     pushl $_do_divide_error # 首先把将要调用的函数地址入栈。这段程序的出错号为 0。
21 no_error_code:             # 这里是无出错号处理的入口处，见下面第 55 行等。
22     xchgl %eax, (%esp)      # _do_divide_error 的地址 → eax, eax 被交换入栈。
23     pushl %ebx
24     pushl %ecx
25     pushl %edx
26     pushl %edi
27     pushl %esi
28     pushl %ebp
29     push %ds                # !! 16 位的段寄存器入栈后也要占用 4 个字节。
30     push %es
31     push %fs
32     pushl $0                # "error code" # 将出错码入栈。
33     lea 44(%esp), %edx      # 取原调用返回地址处堆栈指针位置，并压入堆栈。
34     pushl %edx
35     movl $0x10, %edx        # 内核代码数据段选择符。
36     mov %dx, %ds
37     mov %dx, %es
38     mov %dx, %fs            # 下行上的 '*' 号表示是绝对调用操作数，与程序指针 PC 无关。
39     call *%eax              # 调用 C 函数 do_divide_error()。
40     addl $8, %esp           # 让堆栈指针重新指向寄存器 fs 入栈处。
41     pop %fs
42     pop %es
43     pop %ds
44     popl %ebp
45     popl %esi
46     popl %edi
47     popl %edx
48     popl %ecx
49     popl %ebx
50     popl %eax               # 弹出原来 eax 中的内容。
51     iret
52
   # int1 -- debug 调试中断入口点。处理过程同上。

```



```

53 _debug:
54     pushl $_do_int3          # _do_debug C 函数指针入栈。以下同。
55     jmp no_error_code
56
57     # int2 -- 非屏蔽中断调用入口点。
58     _nmi:
59     pushl $_do_nmi
60     jmp no_error_code
61
62     # int3 -- 同_debug。
63     _int3:
64     pushl $_do_int3
65     jmp no_error_code
66
67     # int4 -- 溢出出错处理中断入口点。
68     _overflow:
69     pushl $_do_overflow
70     jmp no_error_code
71
72     # int5 -- 边界检查出错中断入口点。
73     _bounds:
74     pushl $_do_bounds
75     jmp no_error_code
76
77     # int6 -- 无效操作指令出错中断入口点。
78     _invalid_op:
79     pushl $_do_invalid_op
80     jmp no_error_code
81
82     # int9 -- 协处理器段超出出错中断入口点。
83     _coprocessor_segment_overrun:
84     pushl $_do_coprocessor_segment_overrun
85     jmp no_error_code
86
87     # int15 - 保留。
88     _reserved:
89     pushl $_do_reserved
90     jmp no_error_code
91
92     # int45 -- ( = 0x20 + 13 ) 数学协处理器 (Coprocesor) 发出的中断。
93     # 当协处理器执行完一个操作时就会发出 IRQ13 中断信号，以通知 CPU 操作完成。
94     _irq13:
95     pushl %eax
96     xorb %al,%al             # 80387 在执行计算时，CPU 会等待其操作的完成。
97     outb %al,$0xF0           # 通过写 0xF0 端口，本中断将消除 CPU 的 BUSY 延续信号，并重新
98                               # 激活 80387 的处理器扩展请求引脚 PEREQ。该操作主要是为了确保
99                               # 在继续执行 80387 的任何指令之前，响应本中断。
100
101     movb $0x20,%al
102     outb %al,$0x20           # 向 8259 主中断控制芯片发送 EOI (中断结束) 信号。
103     jmp 1f                   # 这两个跳转指令起延时作用。
104 1:    jmp 1f
105 1:    outb %al,$0xA0          # 再向 8259 从中断控制芯片发送 EOI (中断结束) 信号。
106     popl %eax

```

```

95         jmp _coprocessor_error # _coprocessor_error 原来在本文件中，现在已经放到
                                           # (kernel/system_call.s, 131)

96
# 以下中断在调用时会在中断返回地址之后将出错号压入堆栈，因此返回时也需要将出错号弹出。
# int8 -- 双出错故障。（下面这段代码的含义参见图 5.3(b)）。
97 _double_fault:
98     pushl $_do_double_fault           # C 函数地址入栈。
99 error_code:
100     xchgl %eax, 4(%esp)               # error code <-> %eax, eax 原来的值被保存在堆栈上。
101     xchgl %ebx, (%esp)               # &function <-> %ebx, ebx 原来的值被保存在堆栈上。
102     pushl %ecx
103     pushl %edx
104     pushl %edi
105     pushl %esi
106     pushl %ebp
107     push %ds
108     push %es
109     push %fs
110     pushl %eax                       # error code # 出错号入栈。
111     lea 44(%esp), %eax               # offset # 程序返回地址处堆栈指针位置值入栈。
112     pushl %eax
113     movl $0x10, %eax                # 置内核数据段选择符。
114     mov %ax, %ds
115     mov %ax, %es
116     mov %ax, %fs
117     call *%ebx                      # 调用相应的 C 函数，其参数已入栈。
118     addl $8, %esp                   # 堆栈指针重新指向栈中放置 fs 内容的位置。
119     pop %fs
120     pop %es
121     pop %ds
122     popl %ebp
123     popl %esi
124     popl %edi
125     popl %edx
126     popl %ecx
127     popl %ebx
128     popl %eax
129     iret
130
# int10 -- 无效的任务状态段(TSS)。
131 _invalid_TSS:
132     pushl $_do_invalid_TSS
133     jmp error_code
134
# int11 -- 段不存在。
135 _segment_not_present:
136     pushl $_do_segment_not_present
137     jmp error_code
138
# int12 -- 堆栈段错误。
139 _stack_segment:
140     pushl $_do_stack_segment
141     jmp error_code

```

142

# int13 -- 一般保护性出错。

143 \_general\_protection:

144       pushl \$\_do\_general\_protection

145       jmp error\_code

146

# int7 -- 设备不存在(\_device\_not\_available)在(kernel/system\_call.s, 148)

# int14 -- 页错误(\_page\_fault)在(mm/page.s, 14)

# int16 -- 协处理器错误(\_coprocessor\_error)在(kernel/system\_call.s, 131)

# 时钟中断 int 0x20 (\_timer\_interrupt)在(kernel/system\_call.s, 176)

# 系统调用 int 0x80 (\_system\_call)在(kernel/system\_call.s, 80)

## 5.4.3 其它信息

### 5.4.3.1 Intel 保留中断向量的定义

这里给出了 Intel 保留中断向量具体含义的说明，见表 5-1 所示。

表 5-1 Intel 保留的中断号含义

中断号	名称	类型	信号	说明
0	Devide error	故障	SIGFPE	当进行除以零的操作时产生。
1	Debug	陷阱 故障	SIGTRAP	当进行程序单步跟踪调试时，设置了标志寄存器 eflags 的 T 标志时产生这个中断。
2	nmi	硬件		由不可屏蔽中断 NMI 产生。
3	Breakpoint	陷阱	SIGTRAP	由断点指令 int3 产生，与 debug 处理相同。
4	Overflow	陷阱	SIGSEGV	eflags 的溢出标志 OF 引起。
5	Bounds check	故障	SIGSEGV	寻址到有效地址以外时引起。
6	Invalid Opcode	故障	SIGILL	CPU 执行时发现一个无效的指令操作码。
7	Device not available	故障	SIGSEGV	设备不存在，指协处理器。在两种情况下会产生该中断：(a)CPU 遇到一个转意指令并且 EM 置位时。在这种情况下处理程序应该模拟导致异常的指令。(b)MP 和 TS 都在置位状态时，CPU 遇到 WAIT 或一个转意指令。在这种情况下，处理程序在必要时应该更新协处理器的状态。
8	Double fault	异常中止	SIGSEGV	双故障出错。
9	Coprocessor segment overrun	异常中止	SIGFPE	协处理器段超出。
10	Invalid TSS	故障	SIGSEGV	CPU 切换时发觉 TSS 无效。
11	Segment not present	故障	SIGBUS	描述符所指的段不存在。
12	Stack segment	故障	SIGBUS	堆栈段不存在或寻址越出堆栈段。
13	General protection	故障	SIGSEGV	没有符合 80386 保护机制（特权级）的操作引起。
14	Page fault	故障	SIGSEGV	页不在内存。
15	Reserved			
16	Coprocessor error	故障	SIGFPE	协处理器发出的发出的出错信号引起。

## 5.5 traps.c 程序

### 5.5.1 功能描述

traps.c 程序主要包括一些在处理异常故障（硬件中断）的底层代码 asm.s 中调用的相应 C 函数。用于显示出错位置和出错号等调试信息。其中的 die() 通用函数用于在中断处理中显示详细的出错信息，而代码最后的初始化函数 trap\_init() 是在前面 init/main.c 中被调用，用于硬件异常处理中断向量（陷阱门）的初始化，并设置允许中断请求信号的到来。在阅读本程序时需要参考 asm.s 程序。

### 5.5.2 代码注释

程序 5-3 linux/kernel/traps.c

```

1  /*
2   *  linux/kernel/traps.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   *  'Traps.c' handles hardware traps and faults after we have saved some
9   *  state in 'asm.s'.  Currently mostly a debugging-aid, will be extended
10  *  to mainly kill the offending process (probably by giving it a signal,
11  *  but possibly by killing it outright if necessary).
12  */
13  /*
14   *  在程序 asm.s 中保存了一些状态后，本程序用来处理硬件陷阱和故障。目前主要用于调试目的，
15   *  以后将扩展用来杀死遭损坏的进程（主要是通过发送一个信号，但如果必要也会直接杀死）。
16   */
17  #include <string.h>          // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
18
19  #include <linux/head.h>      // head 头文件，定义了段描述符的简单结构，和几个选择符常量。
20  #include <linux/sched.h>     // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
21                               // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
22  #include <linux/kernel.h>    // 内核头文件。含有一些内核常用函数的原形定义。
23  #include <asm/system.h>      // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
24  #include <asm/segment.h>     // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
25  #include <asm/io.h>          // 输入/输出头文件。定义硬件端口输入/输出宏汇编语句。
26
27  // 以下语句定义了三个嵌入式汇编宏语句函数。有关嵌入式汇编的基本语法见列表后或参见附录。
28  // 取段 seg 中地址 addr 处的一个字节。
29  // 用圆括号括住的组合语句（花括号中的语句）可以作为表达式使用，其中最后的__res 是其输出值。
30  #define get_seg_byte(seg, addr) ({ \
31      register char __res; \
32      __asm__ ("push %%fs;mov %%ax,%%fs;movb %%fs:%2,%%al;pop %%fs" \
33              : "=a" (__res): "" (seg), "m" (*(addr))); \
34      __res;})
35
36  // 取段 seg 中地址 addr 处的一个长字（4 字节）。
37  #define get_seg_long(seg, addr) ({ \

```

```

29 register unsigned long __res; \
30 __asm__( "push %%fs;mov %%ax,%%fs;movl %%fs:%2,%%eax;pop %%fs" \
31         : "=a" (__res): "" (seg), "m" (*(addr))); \
32 __res;})
33
// 取 fs 段寄存器的值 (选择符)。
34 #define _fs() ({ \
35 register unsigned short __res; \
36 __asm__( "mov %%fs,%%ax": "=a" (__res):); \
37 __res;})
38
// 以下定义了一些函数原型。
39 int do_exit(long code); // 程序退出处理。(kernel/exit.c, 102)
40
41 void page_exception(void); // 页异常。实际是 page_fault (mm/page.s, 14)
42
// 以下定义了一些中断处理程序原型, 代码在 (kernel/asm.s 或 system_call.s) 中。
43 void divide_error(void); // int0 (kernel/asm.s, 19)。
44 void debug(void); // int1 (kernel/asm.s, 53)。
45 void nmi(void); // int2 (kernel/asm.s, 57)。
46 void int3(void); // int3 (kernel/asm.s, 61)。
47 void overflow(void); // int4 (kernel/asm.s, 65)。
48 void bounds(void); // int5 (kernel/asm.s, 69)。
49 void invalid_op(void); // int6 (kernel/asm.s, 73)。
50 void device_not_available(void); // int7 (kernel/system_call.s, 148)。
51 void double_fault(void); // int8 (kernel/asm.s, 97)。
52 void coprocessor_segment_overrun(void); // int9 (kernel/asm.s, 77)。
53 void invalid_TSS(void); // int10 (kernel/asm.s, 131)。
54 void segment_not_present(void); // int11 (kernel/asm.s, 135)。
55 void stack_segment(void); // int12 (kernel/asm.s, 139)。
56 void general_protection(void); // int13 (kernel/asm.s, 143)。
57 void page_fault(void); // int14 (mm/page.s, 14)。
58 void coprocessor_error(void); // int16 (kernel/system_call.s, 131)。
59 void reserved(void); // int15 (kernel/asm.s, 81)。
60 void parallel_interrupt(void); // int39 (kernel/system_call.s, 280)。
61 void irq13(void); // int45 协处理器中断处理 (kernel/asm.s, 85)。
62
// 该子程序用来打印出错中断的名称、出错号、调用程序的 EIP、EFLAGS、ESP、fs 段寄存器值、
// 段的基址、段的长度、进程号 pid、任务号、10 字节指令码。如果堆栈在用户数据段, 则还
// 打印 16 字节的堆栈内容。
63 static void die(char * str, long esp_ptr, long nr)
64 {
65     long * esp = (long *) esp_ptr;
66     int i;
67
68     printk("%s: %04x\n|r", str, nr&0xffff);
69     printk("EIP: %04x:%p\nEFLAGS: %04x:%p\nESP: %04x:%p\n",
70           esp[1], esp[0], esp[2], esp[4], esp[3]);
71     printk("fs: %04x\n", _fs());
72     printk("base: %p, limit: %p\n", get_base(current->ldt[1]), get_limit(0x17));
73     if (esp[4] == 0x17) {
74         printk("Stack: ");
75         for (i=0; i<4; i++)

```

```

76         printk( "%p ", get_seg_long(0x17, i+(long *)esp[3]));
77         printk( "\n");
78     }
79     str(i);
80     printk( "Pid: %d, process nr: %d\n\r", current->pid, 0xffff & i);
81     for(i=0; i<10; i++)
82         printk( "%02x ", 0xff & get_seg_byte(esp[1], (i+(char *)esp[0])));
83     printk( "\n\r");
84     do_exit(11);          /* play segment exception */
85 }
86
87 // 以下这些以 do_开头的函数是对应名称中断处理程序调用的 C 函数。
88 void do_double_fault(long esp, long error_code)
89 {
90     die( "double fault", esp, error_code);
91 }
92 void do_general_protection(long esp, long error_code)
93 {
94     die( "general protection", esp, error_code);
95 }
96 void do_divide_error(long esp, long error_code)
97 {
98     die( "divide error", esp, error_code);
99 }
100 }
101 void do_int3(long * esp, long error_code,
102             long fs, long es, long ds,
103             long ebp, long esi, long edi,
104             long edx, long ecx, long ebx, long eax)
105 {
106     int tr;
107
108     __asm__( "str %%ax": "=a" (tr): "" (0)); // 取任务寄存器值→tr。
109     printk( "eax|t|tebx|t|tecx|t|tedx|n|r%8x|t%8x|t%8x|t%8x|n|r",
110            eax, ebx, ecx, edx);
111     printk( "esi|t|tedi|t|tebp|t|tesp|n|r%8x|t%8x|t%8x|t%8x|n|r",
112            esi, edi, ebp, (long) esp);
113     printk( "n|rds|tes|tfs|ttr|n|r%4x|t%4x|t%4x|t%4x|n|r",
114            ds, es, fs, tr);
115     printk( "EIP: %8x   CS: %4x   EFLAGS: %8x|n|r", esp[0], esp[1], esp[2]);
116 }
117 }
118 void do_nmi(long esp, long error_code)
119 {
120     die( "nmi", esp, error_code);
121 }
122 }
123 void do_debug(long esp, long error_code)
124 {
125     die( "debug", esp, error_code);
126 }
127 }

```

```
128
129 void do\_overflow(long esp, long error_code)
130 {
131     die("overflow", esp, error_code);
132 }
133
134 void do\_bounds(long esp, long error_code)
135 {
136     die("bounds", esp, error_code);
137 }
138
139 void do\_invalid\_op(long esp, long error_code)
140 {
141     die("invalid operand", esp, error_code);
142 }
143
144 void do\_device\_not\_available(long esp, long error_code)
145 {
146     die("device not available", esp, error_code);
147 }
148
149 void do\_coprocessor\_segment\_overnrun(long esp, long error_code)
150 {
151     die("coprocessor segment overrun", esp, error_code);
152 }
153
154 void do\_invalid\_TSS(long esp, long error_code)
155 {
156     die("invalid TSS", esp, error_code);
157 }
158
159 void do\_segment\_not\_present(long esp, long error_code)
160 {
161     die("segment not present", esp, error_code);
162 }
163
164 void do\_stack\_segment(long esp, long error_code)
165 {
166     die("stack segment", esp, error_code);
167 }
168
169 void do\_coprocessor\_error(long esp, long error_code)
170 {
171     if (last\_task\_used\_math != current)
172         return;
173     die("coprocessor error", esp, error_code);
174 }
175
176 void do\_reserved(long esp, long error_code)
177 {
178     die("reserved (15, 17-47) error", esp, error_code);
179 }
180
```

```

// 下面是异常（陷阱）中断程序初始化子程序。设置它们的中断调用门（中断向量）。
// set_trap_gate()与 set_system_gate()的主要区别在于前者设置的特权级为 0，后者是 3。因此
// 断点陷阱中断 int3、溢出中断 overflow 和边界出错中断 bounds 可以由任何程序产生。
// 这两个函数均是嵌入式汇编宏程序(include/asm/system.h, 第 36 行、39 行)。
181 void trap_init(void)
182 {
183     int i;
184
185     set_trap_gate(0,&divide_error); // 设置除操作出错的中断向量值。以下雷同。
186     set_trap_gate(1,&debug);
187     set_trap_gate(2,&nmi);
188     set_system_gate(3,&int3);        /* int3-5 can be called from all */
189     set_system_gate(4,&overflow);
190     set_system_gate(5,&bounds);
191     set_trap_gate(6,&invalid_op);
192     set_trap_gate(7,&device_not_available);
193     set_trap_gate(8,&double_fault);
194     set_trap_gate(9,&coprocessor_segment_overrun);
195     set_trap_gate(10,&invalid_TSS);
196     set_trap_gate(11,&segment_not_present);
197     set_trap_gate(12,&stack_segment);
198     set_trap_gate(13,&general_protection);
199     set_trap_gate(14,&page_fault);
200     set_trap_gate(15,&reserved);
201     set_trap_gate(16,&coprocessor_error);
// 下面将 int17-48 的陷阱门先均设置为 reserved，以后每个硬件初始化时会重新设置自己的陷阱门。
202     for (i=17;i<48;i++)
203         set_trap_gate(i,&reserved);
204     set_trap_gate(45,&irq13);        // 设置协处理器的陷阱门。
205     outb_p(inb_p(0x21)&0xfb,0x21);    // 允许主 8259A 芯片的 IRQ2 中断请求。
206     outb(inb_p(0xA1)&0xdf,0xA1);      // 允许从 8259A 芯片的 IRQ13 中断请求。
207     set_trap_gate(39,&parallel_interrupt); // 设置并行口的陷阱门。
208 }
209

```

## 5.5.3 其它信息

### 5.5.3.1 嵌入式汇编的基本格式

本节是第一次在内核源程序中接触到 C 语言中的嵌入式汇编代码。由于我们在通常的 C 语言程序的编制过程中一般是不会使用嵌入式汇编程序的，因此这里有必要对其基本格式进行简单的描述，详细的说明可参见 GNU gcc 手册中[5]第 4 章的内容（Extensions to the C Language Family），或见参考文献[20]（Using Inline Assembly with gcc）。

具有输入和输出参数的嵌入汇编的基本格式为：

```

asm(“汇编语句”
    : 输出寄存器
    : 输入寄存器
    : 会被修改的寄存器 );

```



其中，“汇编语句”是你写汇编指令的地方；“输出寄存器”表示当这段嵌入汇编执行完之后，哪些寄存器用于存放输出数据。此地，这些寄存器会分别对应一 C 语言表达式或一个内存地址；“输入寄存器”表示在开始执行汇编代码时，这里指定的一些寄存器中应存放的输入值，它们也分别对应着一 C 变量或常数值。下面我们用例子来说明嵌入汇编语句的使用方法。

我们在下面列出了前面代码中第 22 行开始的一段代码作为例子来详细解说，为了能看清楚我们将这段代码进行了重新编排和编号。

---

```

01 #define get_seg_byte(seg,addr) \
02 ({ \
03     register char __res; \
04     __asm__("push %%fs; \
05             mov %%ax,%%fs; \
06             movb %%fs:%2,%%al; \
07             pop %%fs" \
08             : "a" (__res) \
09             : "" (seg), "m" (*(addr))); \
10     __res;})

```

---

这段 10 行代码定义了一个嵌入汇编语言宏函数。通常使用汇编语句最方便的方式是把它们放在一个宏内。用圆括号括住的组合语句（花括号中的语句）可以作为表达式使用，其中最后的变量\_\_res（第 10 行）是该表达式的输出值。

因为是宏语句，需要在一行上定义，因此这里使用反斜杠\将这些语句连成一行。这条宏定义将被替换到宏名称在程序中被引用的地方。第 1 行定义了宏的名称，也即是宏函数名称 get\_seg\_byte(seg,addr)。第 3 行定义了一个寄存器变量\_\_res。第 4 行上的\_\_asm\_\_表示嵌入汇编语句的开始。从第 4 行到第 7 行的 4 条 AT&T 格式的汇编语句。

第 8 行即是输出寄存器，这句的含义是在这段代码运行结束后将 `eax` 所代表的寄存器的值放入\_\_res 变量中，作为本函数的输出值，“=a”中的“a”称为加载代码，“=”表示这是输出寄存器。第 9 行表示在这段代码开始运行时将 `seg` 放到 `eax` 寄存器中，“”表示使用与上面同个位置的输出相同的寄存器。而`*(addr)`表示一个内存偏移地址值。为了在上面汇编语句中使用该地址值，嵌入汇编程序规定把输出和输入寄存器统一按顺序编号，顺序是从输出寄存器序列从左到右从上到下以“%0”开始，分别记为%0、%1、...%9。因此，输出寄存器的编号是%0（这里只有一个输出寄存器），输入寄存器前一部分(“” (seg))的编号是%1，而后部分的编号是%2。上面第 6 行上的%2 即代表`*(addr)`这个内存偏移量。

现在我们来研究 4—7 行上的代码的作用。第一句将 `fs` 段寄存器的内容入栈；第二句将 `eax` 中的段值赋给 `fs` 段寄存器；第三句是把 `fs:*(addr)`所指定的字节放入 `al` 寄存器中。当执行完汇编语句后，输出寄存器 `eax` 的值将被放入\_\_res，作为该宏函数（块结构表达式）的返回值。很简单，不是吗？

通过上面分析，我们知道，宏名称中的 `seg` 代表一指定的内存段值，而 `addr` 表示一内存偏移地址量。到现在为止，我们应该很清楚这段程序的功能了吧！该宏函数的功能是从指定段和偏移值的内存地址处取一个字节。

在看下一个例子。

---

```

01 asm("cld\n\t"
02      "rep\n\t"
03      "stol"

```

---

```

04      : /* 没有输出寄存器 */
05      : "c"(count-1), "a"(fill_value), "D"(dest)
06      : "%ecx", "%edi");

```

1-3 行这三句是通常的汇编语句，用以清方向位，重复保存值。第 4 行说明这段嵌入汇编程序没有用到输出寄存器。第 5 行的含义是：将 `count-1` 的值加载到 `ecx` 寄存器中（加载代码是“c”），`fill_value` 加载到 `eax` 中，`dest` 放到 `edi` 中。为什么要让 `gcc` 编译程序去做这样的寄存器值的加载，而不让我们自己做呢？因为 `gcc` 在它进行寄存器分配时可以进行某些优化工作。例如 `fill_value` 值可能已经在 `eax` 中。如果是在一个循环语句中的话，`gcc` 就可能在整个循环操作中保留 `eax`，这样就可以在每次循环中少用一个 `movl` 语句。

最后一行的作用是告诉 `gcc` 这些寄存器中的值已经改变了。很古怪吧？不过在 `gcc` 知道你拿这些寄存器做些什么后，这确实能够对 `gcc` 的优化操作有所帮助。表 5-2 中是一些你可能会用到的寄存器加载代码及其具体的含义。

表 5-2 常用寄存器加载代码说明

代码	说明	代码	说明
a	使用寄存器 <code>eax</code>	m	使用内存地址
b	使用寄存器 <code>ebx</code>	o	使用内存地址并可以加偏移值
c	使用寄存器 <code>ecx</code>	I	使用常数 0-31
d	使用寄存器 <code>edx</code>	J	使用常数 0-63
S	使用 <code>esi</code>	K	使用常数 0-255
D	使用 <code>edi</code>	L	使用常数 0-65535
q	使用动态分配字节可寻址寄存器 ( <code>eax</code> 、 <code>ebx</code> 、 <code>ecx</code> 或 <code>edx</code> )	M	使用常数 0-3
r	使用任意动态分配的寄存器	N	使用 1 字节常数 (0-255)
g	使用通用有效的地址即可 ( <code>eax</code> 、 <code>ebx</code> 、 <code>ecx</code> 、 <code>edx</code> 或内存变量)	O	使用常数 0-31
A	使用 <code>eax</code> 与 <code>edx</code> 联合(64 位)		

下面的例子不是让你自己指定哪个变量使用哪个寄存器，而是让 `gcc` 为你选择。

```

01  asm("leal (%1, %1, 4), %0"
02      : "=r"(y)
03      : "0"(x));

```

第一句汇编语句 `leal (r1, r2, 4), r3` 语句表示  $r1+r2*4 \rightarrow r3$ 。这个例子可以非常快地将 `x` 乘 5。其中 “%0”, “%1” 是指 `gcc` 自动分配的寄存器。这里 “%1” 代表输入值 `x` 要放入的寄存器， “%0” 表示输出值寄存器。输出寄存器代码前一定要加等于号。如果输入寄存器的代码是 0 或为空时，则说明使用与相应输出一样的寄存器。所以，如果 `gcc` 将 `r` 指定为 `eax` 的话，那么上面汇编语句的含义即为：

```
"leal (eax, eax, 4), eax"
```

注意：在执行代码时，如果不希望汇编语句被 gcc 优化而挪动地方，就需要在 `asm` 符号后面添加 `volatile` 关键词：

```
asm volatile (.....);
```

或者更详细的说明为：

```
__asm__ __volatile__ (.....);
```

下面在具一个较长的例子，如果能看得懂，那就说明嵌入汇编代码对你来说基本没问题了。这段代码是从 `include/string.h` 文件中摘取的，是 `strncmp()` 字符串比较函数的一种实现。需要注意的是，其中每行中的“`\n\t`”是用于 gcc 预处理程序输出列表好看而设置的，含义与 C 语言中相同。

---

```

//// 字符串 1 与字符串 2 的前 count 个字符进行比较。
// 参数: cs - 字符串 1, ct - 字符串 2, count - 比较的字符数。
// %0 - eax(__res)返回值, %1 - edi(cs)串 1 指针, %2 - esi(ct)串 2 指针, %3 - ecx(count)。
// 返回: 如果串 1 > 串 2, 则返回 1; 串 1 = 串 2, 则返回 0; 串 1 < 串 2, 则返回-1。
extern inline int strncmp(const char * cs, const char * ct, int count)
{
register int __res ;                // __res 是寄存器变量。
__asm__( "cld\n"                    // 清方向位。
        "1:\tdecl %3\n\t"           // count--。
        "js 2f\n\t"                 // 如果 count<0, 则向前跳转到标号 2。
        "lods b\n\t"                // 取串 2 的字符 ds:[esi]→al, 并且 esi++。
        "scas b\n\t"                // 比较 al 与串 1 的字符 es:[edi], 并且 edi++。
        "jne 3f\n\t"                // 如果不相等, 则向前跳转到标号 3。
        "testb %al, %al\n\t"        // 该字符是 NULL 字符吗?
        "jne 1b\n\t"                // 不是, 则向后跳转到标号 1, 继续比较。
        "2:\txorl %%eax, %%eax\n\t" // 是 NULL 字符, 则 eax 清零 (返回值)。
        "jmp 4f\n\t"                // 向前跳转到标号 4, 结束。
        "3:\tmovl $1, %%eax\n\t"    // eax 中置 1。
        "jl 4f\n\t"                 // 如果前面比较中串 2 字符<串 1 字符, 则返回 1, 结束。
        "negl %%eax\n\t"            // 否则 eax = -eax, 返回负值, 结束。
        "4:"
        : "=a" (__res) : "D" (cs), "S" (ct), "c" (count) : "si", "di", "cx");
return __res;                       // 返回比较结果。
}

```

---

## 5.6 system\_call.s 程序

### 5.6.1 功能描述

本程序主要实现系统调用(system\_call)中断 int 0x80 的入口处理过程以及信号检测处理(从代码第 80 行开始), 同时给出了两个系统功能的底层接口, 分别是 `sys_execve` 和 `sys_fork`。还列出了处理过程类似的协处理器出错(int 16)、设备不存在(int7)、时钟中断(int32)、硬盘中断(int46)、软盘中断(int38)的中断处理程序。

对于软中断(system\_call、coprocessor\_error、device\_not\_available), 处理过程基本上是首先为调用相应 C 函数处理程序作准备, 将一些参数压入堆栈, 然后调用 C 函数进行相应功能的处理, 处理返回后再

去检测当前任务的信号位图，对值最小的一个信号进行处理并复位信号位图中的该信号。系统调用的 C 语言处理函数分布在整个 linux 内核代码中，由 include/linux/sys.h 头文件中的系统函数指针数组表来匹配。

对于硬件中断请求信号 IRQ 发来的中断，其处理过程首先是向中断控制芯片 8259A 发送结束硬件中断控制字指令 EOI，然后调用相应的 C 函数处理程序。对于时钟中断也要对当前任务的信号位图进行检测处理。

对于系统调用(int 0x80)的中断处理过程，可以把它看作是一个“接口”程序。实际每个系统调用功能的处理基本上都是通过调用相应的 C 函数进行的。即所谓的“Bottom half”函数。

这个程序在刚进入时，首先检查 `eax` 中的功能号是否有效（在给定的范围内），然后保存会用到的寄存器。Linux 内核默认把 `ds,es` 用于内核数据段，而 `fs` 用于用户数据段。接着通过一个地址跳转表（`sys_call_table`）调用相应系统调用的 C 函数。在 C 函数返回后，保存（push）了调用返回值。

接下来，该程序查看执行本次调用进程的状态。如果由于上面 C 函数的操作或其它情况而使进程的状态从执行态变成了其它状态，或者由于时间片已经用完（`counter==0`），则调用进程调度函数 `schedule()`（`jmp_schedule`）。由于在执行“`jmp_schedule`”之前已经把返回地址 `ret_from_sys_call` 入栈，因此在执行完 `schedule()` 后最终会返回到 `ret_from_sys_call` 处继续执行。

从 `ret_from_sys_call` 标号处开始的代码执行一些系统调用的后处理工作。主要判断当前进程是否是初始进程 0 就直接退出此次系统调用，中断返回。否则再根据代码段描述符和所使用的堆栈来判断调用本系统调用的进程是否是一个普通进程，若不是则说明是内核进程（例如初始进程 1）或其它。则也立刻弹出堆栈内容退出系统调用中断。末端的一块代码用来处理调用系统调用进程的信号。若进程结构的信号位图表明该进程有接收到信号，则调用信号处理函数 `do_signal()`。

最后，该程序恢复保存的寄存器内容，退出此次中断处理过程并返回调用程序。若有信号时会首先“返回”到相应信号处理函数中去执行，然后返回调用 `system_call` 的程序。

系统调用处理过程的整个流程见图 5-4 所示。

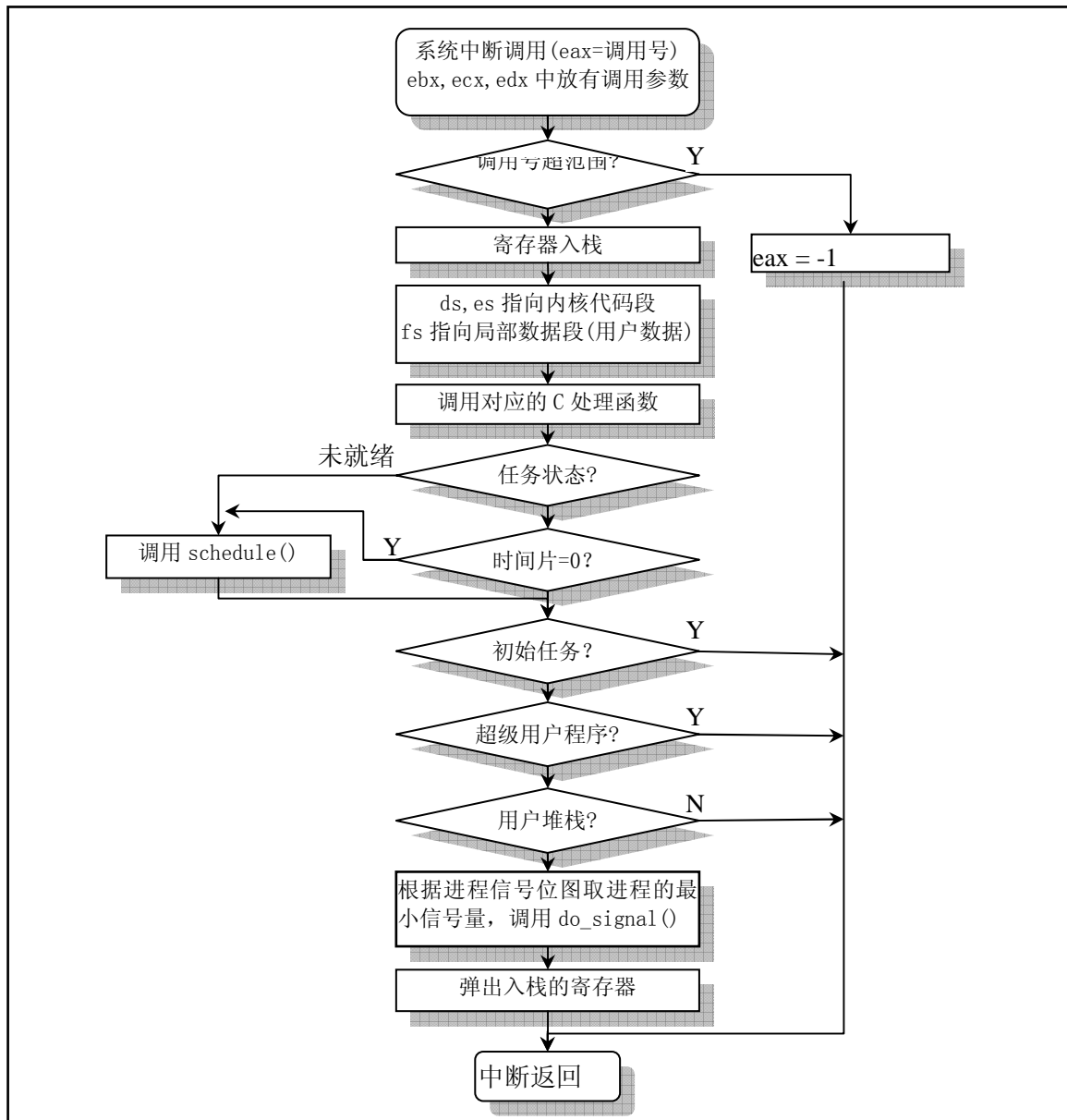


图 5-4 系统中断调用处理流程

## 5.6.2 代码注释

程序 5-4 linux/kernel/system\_call.s

```

1 /*
2  * linux/kernel/system_call.s
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 /*
8  * system_call.s contains the system-call low-level handling routines.
9  * This also contains the timer-interrupt handler, as some of the code is
10 * the same. The hd- and floppy-interrupts are also here.
11 *
12 * NOTE: This code handles signal-recognition, which happens every time

```

```

13 * after a timer-interrupt and after each system call. Ordinary interrupts
14 * don't handle signal-recognition, as that would clutter them up totally
15 * unnecessarily.
16 *
17 * Stack layout in 'ret_from_system_call':
18 *
19 *     0(%esp) - %eax
20 *     4(%esp) - %ebx
21 *     8(%esp) - %ecx
22 *    C(%esp) - %edx
23 *    10(%esp) - %fs
24 *    14(%esp) - %es
25 *    18(%esp) - %ds
26 *    1C(%esp) - %eip
27 *    20(%esp) - %cs
28 *    24(%esp) - %eflags
29 *    28(%esp) - %oldesp
30 *    2C(%esp) - %oldss
31 */
/*
* 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
* 中断信号并不处理信号识别，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。
*/

32
33 SIG_CHLD      = 17          # 定义 SIG_CHLD 信号（子进程停止或结束）。
34
35 EAX           = 0x00        # 堆栈中各个寄存器的偏移位置。
36 EBX           = 0x04
37 ECX           = 0x08
38 EDX           = 0x0C
39 FS            = 0x10
40 ES            = 0x14
41 DS            = 0x18
42 EIP           = 0x1C
43 CS            = 0x20
44 EFLAGS        = 0x24
45 OLDESP        = 0x28        # 当有特权级变化时。
46 OLDSS         = 0x2C
47
48 # 以下这些是任务结构(task_struct)中变量的偏移值，参见 include/linux/sched.h, 77 行开始。
49 state = 0                # these are offsets into the task-struct. # 进程状态码
50 counter = 4              # 任务运行时间计数(递减)(滴答数)，运行时间片。
51 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。
52 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。
53 sigaction = 16           # MUST be 16 (=len of sigaction) // sigaction 结构长度必须是 16 字节。
54                          // 信号执行属性结构数组的偏移值，对应信号将要执行的操作和标志信息。
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 48 行开始。
55 # offsets within sigaction
56 sa_handler = 0           // 信号处理过程的句柄（描述符）。
57 sa_mask = 4             // 信号量屏蔽码
58 sa_flags = 8            // 信号集。
59 sa_restorer = 12        // 恢复函数指针，参见 kernel/signal.c。
60
61 nr_system_calls = 72    # Linux 0.11 版内核中的系统调用总数。
62
63 /*
64  * Ok, I get parallel printer interrupts while using the floppy for some
65  * strange reason. Urgel. Now I just ignore them.
66  */
67 /*
68  * 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。
69  */
70 # 定义入口点。
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
75 # 错误的系统调用号。
76 .align 2                 # 内存 4 字节对齐。
77 bad_sys_call:
78     movl $-1,%eax        # eax 中置-1，退出中断。
79     iret
80 # 重新执行调度程序入口。调度程序 schedule 在(kernel/sched.c, 104)。
81 .align 2
82 reschedule:
83     pushl $ret_from_sys_call # 将 ret_from_sys_call 的地址入栈（101 行）。
84     jmp _schedule
85 ##### int 0x80 --linux 系统调用入口点(调用中断 int 0x80, eax 中是调用号)。
86 .align 2
87 _system_call:
88     cmpl $nr_system_calls-1,%eax # 调用号如果超出范围的话就在 eax 中置-1 并退出。
89     ja bad_sys_call
90     push %ds              # 保存原段寄存器值。
91     push %es
92     push %fs
93     pushl %edx            # ebx, ecx, edx 中放着系统调用相应的 C 语言函数的调用参数。
94     pushl %ecx            # push %ebx, %ecx, %edx as parameters
95     pushl %ebx            # to the system call
96     movl $0x10,%edx       # set up ds, es to kernel space
97     mov %dx, %ds          # ds, es 指向内核数据段(全局描述符表中数据段描述符)。
98     mov %dx, %es
99     movl $0x17,%edx       # fs points to local data space
100    mov %dx, %fs           # fs 指向局部数据段(局部描述符表中数据段描述符)。
101    # 下面这句操作数的含义是：调用地址 = _sys_call_table + %eax * 4。参见列表后的说明。
102    # 对应的 C 程序中的 sys_call_table 在 include/linux/sys.h 中，其中定义了一个包括 72 个
103    # 系统调用 C 处理函数的地址数组表。
104    call _sys_call_table(,%eax, 4)
105    pushl %eax             # 把系统调用返回值入栈。
106    movl _current,%eax     # 取当前任务（进程）数据结构地址→eax。

```



```

# 下面 97-100 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,但其时间片已用完(counter=0),则也去执行调度程序。
97      cmpl $0,state(%eax)      # state
98      jne reschedule
99      cmpl $0,counter(%eax)    # counter
100     je reschedule
# 以下这段代码执行从系统调用 C 函数返回后,对信号量进行识别处理。
101 ret_from_sys_call:
# 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。
# 103 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]。
102     movl _current,%eax      # task[0] cannot have signals
103     cmpl _task,%eax
104     je 3f                  # 向前(forward)跳转到标号 3。
# 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内核任务(例如任务 1)。如果是则直接
# 退出中断。否则对于普通进程则需进行信号量的处理。这里比较选择符是否为普通用户代码段的选择
# 符 0x000f (RPL=3, 局部表, 第 1 个段(代码段)),如果不是则跳转退出中断程序。
105     cmpw $0x0f,CS(%esp)     # was old code segment supervisor ?
106     jne 3f
# 如果原堆栈段选择符不为 0x17(也即原堆栈不在用户数据段中),则也退出。
107     cmpw $0x17,OLDSS(%esp)  # was stack segment = 0x17 ?
108     jne 3f
# 下面这段代码(109-120)的用途是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,82)中,其参数包括 13 个入栈的信息。
109     movl signal(%eax),%ebx   # 取信号位图→ebx,每 1 位代表 1 种信号,共 32 个信号。
110     movl blocked(%eax),%ecx  # 取阻塞(屏蔽)信号位图→ecx。
111     notl %ecx                # 每位取反。
112     andl %ebx,%ecx           # 获得许可的信号位图。
113     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)。
114     je 3f                    # 如果没有信号则向前跳转退出。
115     btrl %ecx,%ebx           # 复位该信号(ebx 含有原 signal 位图)。
116     movl %ebx,signal(%eax)   # 重新保存 signal 位图信息→current->signal。
117     incl %ecx                # 将信号调整为从 1 开始的数(1-32)。
118     pushl %ecx               # 信号值入栈作为调用 do_signal 的参数之一。
119     call _do_signal           # 调用 C 函数信号处理程序(kernel/signal.c,82)
120     popl %eax                # 弹出信号值。
121 3:    popl %eax
122     popl %ebx
123     popl %ecx
124     popl %edx
125     pop %fs
126     pop %es
127     pop %ds
128     iret
129
#### int16 -- 下面这段代码处理协处理器发出的出错信号。跳转执行 C 函数 math_error()
# (kernel/math/math_emulate.c,82),返回后将跳转到 ret_from_sys_call 处继续执行。
130 .align 2
131 _coprocessor_error:
132     push %ds
133     push %es

```



```

134     push %fs
135     pushl %edx
136     pushl %ecx
137     pushl %ebx
138     pushl %eax
139     movl $0x10,%eax          # ds,es 置为指向内核数据段。
140     mov %ax,%ds
141     mov %ax,%es
142     movl $0x17,%eax          # fs 置为指向局部数据段（出错程序的数据段）。
143     mov %ax,%fs
144     pushl $ret_from_sys_call # 把下面调用返回的地址入栈。
145     jmp _math_error          # 执行 C 函数 math_error() (kernel/math/math_emulate.c, 37)
146
#### int7 -- 设备不存在或协处理器不存在 (Coprocessor not available)。
# 如果控制寄存器 CR0 的 EM 标志置位，则当 CPU 执行一个 ESC 转义指令时就会引发该中断，这样就
# 可以有机会让这个中断处理程序模拟 ESC 转义指令（169 行）。
# CR0 的 TS 标志是在 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（上下文）
# 与 CPU 正在执行的任务不匹配了。当 CPU 在运行一个转义指令时发现 TS 置位了，就会引发该中断。
# 此时就应该恢复新任务的协处理器执行状态（165 行）。参见 (kernel/sched.c, 77) 中的说明。
# 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）。
147 .align 2
148 _device_not_available:
149     push %ds
150     push %es
151     push %fs
152     pushl %edx
153     pushl %ecx
154     pushl %ebx
155     pushl %eax
156     movl $0x10,%eax          # ds,es 置为指向内核数据段。
157     mov %ax,%ds
158     mov %ax,%es
159     movl $0x17,%eax          # fs 置为指向局部数据段（出错程序的数据段）。
160     mov %ax,%fs
161     pushl $ret_from_sys_call # 把下面跳转或调用的返回地址入栈。
162     clls                     # clear TS so that we can use math
163     movl %cr0,%eax
164     testl $0x4,%eax          # EM (math emulation bit)
                                # 如果不是 EM 引起的中断，则恢复新任务协处理器状态，
165     je _math_state_restore   # 执行 C 函数 math_state_restore() (kernel/sched.c, 77)。
166     pushl %ebp
167     pushl %esi
168     pushl %edi
169     call _math_emulate        # 调用 C 函数 math_emulate(kernel/math/math_emulate.c, 18)。
170     popl %edi
171     popl %esi
172     popl %ebp
173     ret                      # 这里的 ret 将跳转到 ret_from_sys_call(101 行)。
174
#### int32 -- (int 0x20) 时钟中断处理程序。中断频率被设置为 100Hz(include/linux/sched.h, 5)，
# 定时芯片 8253/8254 是在 (kernel/sched.c, 406) 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。

```

```

175 .align 2
176 _timer_interrupt:
177     push %ds                # save ds, es and put kernel data space
178     push %es                # into them. %fs is used by _system_call
179     push %fs
180     pushl %edx               # we save %eax, %ecx, %edx as gcc doesn't
181     pushl %ecx               # save those across function calls. %ebx
182     pushl %ebx               # is saved as we use that in ret_sys_call
183     pushl %eax
184     movl $0x10, %eax         # ds, es 置为指向内核数据段。
185     mov %ax, %ds
186     mov %ax, %es
187     movl $0x17, %eax         # fs 置为指向局部数据段（出错程序的数据段）。
188     mov %ax, %fs
189     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。
190     movb $0x20, %al          # EOI to interrupt controller #1
191     outb %al, $0x20          # 操作命令字 OCW2 送 0x20 端口。
    # 下面 3 句从选择符中取出当前特权级别(0 或 3)并压入堆栈，作为 do_timer 的参数。
192     movl CS(%esp), %eax
193     andl $3, %eax            # %eax is CPL (0 or 3, 0=supervisor)
194     pushl %eax
    # do_timer(CPL) 执行任务切换、计时等工作，在 kernel/shched.c, 305 行实现。
195     call _do_timer            # 'do_timer(long CPL)' does everything from
196     addl $4, %esp            # task switching to accounting ...
197     jmp ret_from_sys_call
198
    ##### 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, 182)。
199 .align 2
200 _sys_execve:
201     lea EIP(%esp), %eax
202     pushl %eax
203     call _do_execve
204     addl $4, %esp            # 丢弃调用时压入栈的 EIP 值。
205     ret
206
    ##### 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 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。
207 .align 2
208 _sys_fork:
209     call _find_empty_process  # 调用 find_empty_process() (kernel/fork.c, 135)。
210     testl %eax, %eax
211     js 1f
212     push %gs
213     pushl %esi
214     pushl %edi
215     pushl %ebp
216     pushl %eax
217     call _copy_process        # 调用 C 函数 copy_process() (kernel/fork.c, 68)。
218     addl $20, %esp            # 丢弃这里所有压栈内容。
219 1:     ret

```

[220](#)

```
##### int 46 -- (int 0x2E) 硬盘中断处理程序，响应硬件中断请求 IRQ14。
# 当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()。
```

[221](#) \_hd\_interrupt:

```
222     pushl %eax
223     pushl %ecx
224     pushl %edx
225     push %ds
226     push %es
227     push %fs
228     movl $0x10,%eax           # ds,es 置为内核数据段。
229     mov %ax,%ds
230     mov %ax,%es
231     movl $0x17,%eax           # fs 置为调用程序的局部数据段。
232     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。
233     movb $0x20,%al
234     outb %al,$0xA0           # EOI to interrupt controller #1  # 送从 8259A。
235     jmp 1f                   # give port chance to breathe
236 1:     jmp 1f                   # 延时作用。
237 1:     xorl %edx,%edx
238     xchgl _do_hd,%edx        # do_hd 定义为一个函数指针，将被赋值 read_intr()或
                                # write_intr()函数地址。(kernel/blk_drv/hd.c)
                                # 放到 edx 寄存器后就将 do_hd 指针变量置为 NULL。
239     testl %edx,%edx          # 测试函数指针是否为 Null。
240     jne 1f                   # 若空，则使指针指向 C 函数 unexpected_hd_interrupt()。
241     movl $_unexpected_hd_interrupt,%edx # (kernel/blk_drv/hdc, 237)。
242 1:     outb %al,$0x20         # 送主 8259A 中断控制器 EOI 指令（结束硬件中断）。
243     call *%edx               # "interesting" way of handling intr.
244     pop %fs                  # 上句调用 do_hd 指向的 C 函数。
245     pop %es
246     pop %ds
247     popl %edx
248     popl %ecx
249     popl %eax
250     iret
251
```

```
##### 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。
```

[252](#) \_floppy\_interrupt:

```
253     pushl %eax
254     pushl %ecx
255     pushl %edx
256     push %ds
257     push %es
```

```

258     push %fs
259     movl $0x10,%eax           # ds,es 置为内核数据段。
260     mov %ax,%ds
261     mov %ax,%es
262     movl $0x17,%eax           # fs 置为调用程序的局部数据段。
263     mov %ax,%fs
264     movb $0x20,%al           # 送主 8259A 中断控制器 EOI 指令（结束硬件中断）。
265     outb %al,$0x20           # EOI to interrupt controller #1
266     xorl %eax,%eax
267     xchgl _do_floppy,%eax     # do_floppy 为一函数指针，将被赋值实际处理 C 函数程序，
                                # 放到 eax 寄存器后就将 do_floppy 指针变量置空。
268     testl %eax,%eax          # 测试函数指针是否=NULL?
269     jne 1f                   # 若空，则使指针指向 C 函数 unexpected_floppy_interrupt()。
270     movl $_unexpected_floppy_interrupt,%eax
271 1:    call *%eax              # "interesting" way of handling intr.
272     pop %fs                  # 上句调用 do_floppy 指向的函数。
273     pop %es
274     pop %ds
275     popl %edx
276     popl %ecx
277     popl %eax
278     iret
279
#### 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。
280 _parallel_interrupt:
281     pushl %eax
282     movb $0x20,%al
283     outb %al,$0x20
284     popl %eax
285     iret

```

## 5.6.3 其它信息

### 5.6.3.1 GNU 汇编语言的 32 位寻址方式

采用的是 AT&T 的汇编语言语法。32 位寻址的正规格式为：

AT&T: `immed32(basepointer, indexpointer, indexscale)`

Intel: `[basepointer + indexpointer*indexscal + immed32]`

该格式寻址位置的计算方式为：`immed32 + basepointer + indexpointer * indexscale`

在应用时，并不需要写出所有这些字段，但 `immed32` 和 `basepointer` 之中必须有一个存在。以下是一些例子。

- o 对一个指定的 C 语言变量寻址：

AT&T: `_booga`

Intel: `[_booga]`

注意：变量前的下划线是从汇编程序中得到静态（全局）C 变量(booga)的方法。

- o 对寄存器内容指向的位置寻址：

AT&T: `(%eax)`

Intel: `[eax]`

- o 通过寄存器中的内容作为基址寻址一个变量：

AT&T: `_variable(%eax)` Intel: `[eax + _variable]`

o 在一个整数数组中寻址一个值（比例值为 4）:

AT&T: `_array(%eax,4)` Intel: `[eax*4 + _array]`

o 使用直接数寻址偏移量:

对于 C 语言: `*(p+1)` 其中 `p` 是字符的指针 `char *`

AT&T: 则 AT&T 格式: `1(%eax)` 其中 `eax` 中是 `p` 的值。 Intel: `[eax+1]`

o 在一个 8 字节为一个记录的数组中寻址指定的字符。其中 `eax` 中是指定的记录号, `ebx` 中是指定字符在记录中的偏移址:

AT&T: `_array(%ebx,%eax,8)` Intel: `[ebx + eax*8 + _array]`

## 5.7 mktime.c 程序

### 5.7.1 功能描述

该程序只有一个函数 `mktime()`, 仅供内核使用。计算从 1970 年 1 月 1 日 0 时起到开机当日经过的秒数, 作为开机时间。

### 5.7.2 代码注释

程序 5-5 linux/kernel/mktime.c 程序

```

1  /*
2   * linux/kernel/mktime.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7 #include <time.h>          // 时间头文件, 定义了标准时间数据结构 tm 和一些处理时间函数原型。
8
9 /*
10 * This isn't the library routine, it is only used in the kernel.
11 * as such, we don't care about years<1970 etc, but assume everything
12 * is ok. Similarly, TZ etc is happily ignored. We just do everything
13 * as easily as possible. Let's find something public for the library
14 * routines (although I think minix times is public).
15 */
16 /*
17 * PS. I hate whoever thought up the year 1970 - couldn't they have gotten
18 * a leap-year instead? I also hate Gregorius, pope or no. I'm grumpy.
19 */
20 /*
21 * 这不是库函数, 它仅供内核使用。因此我们不关心小于 1970 年的年份等, 但假定一切均很正常。
22 * 同样, 时间区域 TZ 问题也先忽略。我们只是尽可能简单地处理问题。最好能找到一些公开的库函数
23 * (尽管我认为 minix 的时间函数是公开的)。
24 * 另外, 我恨那个设置 1970 年开始的人 - 难道他们就不能选择从一个闰年开始? 我恨格里高利历、
25 * 罗马教皇、主教, 我什么都不在乎。我是个脾气暴躁的人。
26 */
27 #define MINUTE 60          // 1 分钟的秒数。

```

---

```

21 #define HOUR (60*MINUTE)           // 1 小时的秒数。
22 #define DAY (24*HOUR)              // 1 天的秒数。
23 #define YEAR (365*DAY)             // 1 年的秒数。
24
25 /* interestingly, we assume leap-years */
26 /* 有趣的是我们考虑进了闰年 */
27 /* 下面以年为界限，定义了每个月开始时的秒数时间数组。
28 static int month[12] = {
29     0,
30     DAY*(31),
31     DAY*(31+29),
32     DAY*(31+29+31),
33     DAY*(31+29+31+30),
34     DAY*(31+29+31+30+31),
35     DAY*(31+29+31+30+31+30),
36     DAY*(31+29+31+30+31+30+31),
37     DAY*(31+29+31+30+31+30+31+30),
38     DAY*(31+29+31+30+31+30+31+31+30+31+30),
39 };
40
41 // 该函数计算从 1970 年 1 月 1 日 0 时起到开机当日经过的秒数，作为开机时间。
42 long kernel_mktime(struct tm * tm)
43 {
44     long res;
45     int year;
46
47     year = tm->tm_year - 70;           // 从 70 年到现在经过的年数(2 位表示方式)，
48                                         // 因此会有 2000 年问题。
49
50 /* magic offsets (y+1) needed to get leapyears right. */
51 /* 为了获得正确的闰年数，这里需要这样一个魔幻偏值(y+1) */
52 res = YEAR*year + DAY*((year+1)/4); // 这些年经过的秒数时间 + 每个闰年时多 1 天
53 res += month[tm->tm_mon];           // 的秒数时间，在加上当年到当月时的秒数。
54 /* and (y+2) here. If it wasn't a leap-year, we have to adjust */
55 /* 以及(y+2)。如果(y+2)不是闰年，那么我们就必须进行调整(减去一天的秒数时间)。*/
56 if (tm->tm_mon>1 && ((year+2)%4))
57     res -= DAY;
58 res += DAY*(tm->tm_mday-1);         // 再加上本月过去的天数的秒数时间。
59 res += HOUR*tm->tm_hour;           // 再加上当天过去的小时数的秒数时间。
60 res += MINUTE*tm->tm_min;          // 再加上 1 小时内过去的分钟数的秒数时间。
61 res += tm->tm_sec;                 // 再加上 1 分钟内已过的秒数。
62 return res;                        // 即等于从 1970 年以来经过的秒数时间。
63 }
64

```

---

## 5.7.3 其它信息

### 5.7.3.1 闰年时间的计算方法

闰年的基本计算方法是：

如果 y 能被 4 除尽且不能被 100 除尽，或者能被 400 除尽，则 y 是闰年。

## 5.8 sched.c 程序

### 5.8.1 功能描述

sched.c 是内核中有关任务调度函数的程序，其中包括有关调度的基本函数(sleep\_on、wakeup、schedule 等)以及一些简单的系统调用函数(比如 getpid())。另外 Linus 为了编程的方便，考虑到软盘驱动程序定时的需要，也将操作软盘的几个函数放到了这里。

这几个基本函数的代码虽然不长，但有些抽象，比较难以理解。好在市面上有许多教科书对此解释得都很清楚，因此可以参考其它书籍对这些函数的讨论。这些也就是教科书上的重点讲述对象，否则理论书籍也就没有什么好讲的了☺。这里仅对调度函数 schedule()作一些说明。

schedule()函数首先对所有任务(进程)进行检测，唤醒任何一个已经得到信号的任务。具体方法是针对任务数组中的每个任务，检查其报警定时值 alarm。如果任务的 alarm 时间已经过期(alarm<jiffies)，则在它的信号位图中设置 SIGALRM 信号，然后清 alarm 值。jiffies 是系统从开机开始算起的滴答数(10ms/滴答)。在 sched.h 中定义。如果进程的信号位图中除去被阻塞的信号外还有其它信号，并且任务处于可中断睡眠状态(TASK\_INTERRUPTIBLE)，则置任务为就绪状态(TASK\_RUNNING)。

随后是调度函数的核心处理部分。这部分代码根据进程的时间片和优先权调度机制，来选择随后要执行的任务。它首先循环检查任务数组中的所有任务，根据每个就绪态任务剩余执行时间的值 counter，选取该值最大的一个任务，并利用 switch\_to()函数切换到该任务。若所有就绪态任务的该值都等于零，表示此刻所有任务的时间片都已经运行完，于是就根据任务的优先权值 priority，重置每个任务的运行时间片值 counter，再重新执行循环检查所有任务的执行时间片值。

另两个值得一提的函数是自动进入睡眠函数 sleep\_on()和唤醒函数 wake\_up()，这两个函数虽然很短，却要比 schedule()函数难理解。这里用图示的方法加以解释。简单地说，sleep\_on()函数的主要功能是一个进程(或任务)所请求的资源正忙或不在内存中时暂时切换出去，放在等待队列中等待一段时间。当切换回来后再继续运行。放入等待队列的方式是利用了函数中的 tmp 指针作为各个正在等待任务的联系。

函数中共牵涉到对三个任务指针操作：\*p、tmp 和 current，\*p 是等待队列头指针，如文件系统内存 i 节点的 i\_wait 指针、内存缓冲操作中的 buffer\_wait 指针等；tmp 是临时指针；current 是当前任务指针。对于这些指针在内存中的变化情况我们可以用图 5-5 的示意图说明。图中的长条表示内存字节序列。

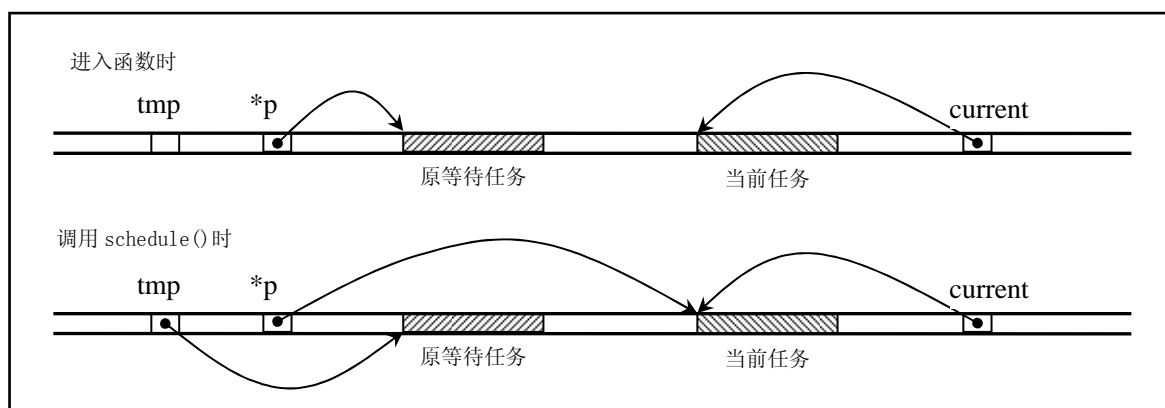


图 5-5 sleep\_on() 函数中指针变化示意图。

当刚进入该函数时，队列头指针 `*p` 指向已经在等待队列中等待的任务结构(进程描述符)。当然，在系统刚开始执行时，等待队列上无等待任务。因此上图中的原等待任务在刚开始时是不存在的，此时



\*p 指向 NULL。通过指针操作，在调用调度程序之前，队列头指针指向了当前任务结构，而函数中的临时指针 tmp 指向了原等待任务。从而通过该临时指针的作用，在几个进程为等待同一资源而多次调用该函数时，程序就隐式地构筑出一个等待队列。从图 5-6 中我们可以更容易地理解 sleep\_on() 函数的等待队列形成过程。图中示出了当向队列头部插入第三个任务时的情况。

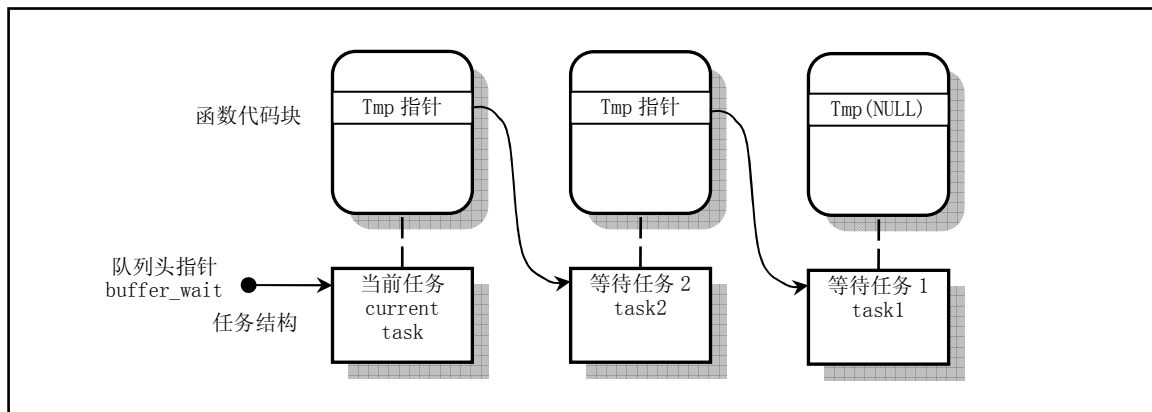


图 5-6 sleep\_on() 函数的隐式任务等待队列。

在插入等待队列后，sleep\_on() 函数就会调用 schedule() 函数去执行别的进程。当进程被唤醒而重新执行时就会执行后续的语句，把比它早进入等待队列的一个进程唤醒。

唤醒操作函数 wake\_up() 把正在等待可用资源的指定任务置为就绪状态。该函数是一个通用唤醒函数。在有些情况下，例如读取磁盘上的数据块，由于等待队列中的任何一个任务都可能被先唤醒，因此还需要把被唤醒任务结构的指针置空。这样，在其后进入睡眠的进程被唤醒而又重新执行 sleep\_on() 时，就无需唤醒该进程了。

还有一个函数 interruptible\_sleep\_on()，它的结构与 sleep\_on() 的基本类似，只是在进行调度之前是把当前任务置成了可中断等待状态，并在本任务被唤醒后还需要队列上是否有后来的等待任务，若有，则调度它们先运行。在内核 0.12 开始，这两个函数被合二为一，仅用任务的状态作为参数来区分这两种情况。

在阅读本文件的代码时，最好同时参考包含文件 include/kernel/sched.h 文件中的注释，以便更清晰地了解内核的调度机理。

## 5.8.2 代码注释

程序 5-6 linux/kernel/sched.c

```

1  /*
2  *  linux/kernel/sched.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *  'sched.c' is the main kernel file. It contains scheduling primitives
9  *  (sleep_on, wakeup, schedule etc) as well as a number of simple system
10 *  call functions (type getpid(), which just extracts a field from
11 *  current-task
12 */
13
14 * 'sched.c' 是主要的内核文件。其中包括有关调度的基本函数(sleep_on、wakeup、schedule 等)以及

```



```

    * 一些简单的系统调用函数（比如 getpid(), 仅从当前任务中获取一个字段）。
    */
13 #include <linux/sched.h> // 调度程序头文件。定义了任务结构 task_struct、第 1 个初始任务
    // 的数据。还有一些以宏的形式定义的有关描述符参数设置和获取的
    // 嵌入式汇编函数程序。
14 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
15 #include <linux/sys.h> // 系统调用头文件。含有 72 个系统调用 C 函数处理程序, 以 'sys_' 开头。
16 #include <linux/fdreg.h> // 软驱头文件。含有软盘控制器参数的一些定义。
17 #include <asm/system.h> // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
18 #include <asm/io.h> // io 头文件。定义硬件端口输入/输出宏汇编语句。
19 #include <asm/segment.h> // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
20
21 #include <signal.h> // 信号头文件。定义信号符号常量, sigaction 结构, 操作函数原型。
22
23 #define _S(nr) (1<<((nr)-1)) // 取信号 nr 在信号位图中对应位的二进制数值。信号编号 1-32。
    // 比如信号 5 的位图数值 = 1<<(5-1) = 16 = 00010000b。
24 #define _BLOCKABLE (~(_S(SIGKILL) | _S(SIGSTOP))) // 除了 SIGKILL 和 SIGSTOP 信号以外其它都是
    // 可阻塞的(...1011111111101111111b)。
25
    // 显示任务号 nr 的进程号、进程状态和内核堆栈空闲字节数（大约）。
26 void show_task(int nr, struct task_struct * p)
27 {
28     int i, j = 4096-sizeof(struct task_struct);
29
30     printk("%d: pid=%d, state=%d, ", nr, p->pid, p->state);
31     i=0;
32     while (i<j && !((char *) (p+1))[i]) // 检测指定任务数据结构以后等于 0 的字节数。
33         i++;
34     printk("%d (of %d) chars free in kernel stack\n", i, j);
35 }
36
    // 显示所有任务的任务号、进程号、进程状态和内核堆栈空闲字节数（大约）。
37 void show_stat(void)
38 {
39     int i;
40
41     for (i=0; i<NR_TASKS; i++) // NR_TASKS 是系统能容纳的最大进程（任务）数量（64 个），
42         if (task[i]) // 定义在 include/kernel/sched.h 第 4 行。
43             show_task(i, task[i]);
44 }
45
46 #define LATCH (1193180/HZ) // 定义每个时间片的滴答数⊙。
47
48 extern void mem_use(void); // [??]没有任何地方定义和引用该函数。
49
50 extern int timer_interrupt(void); // 时钟中断处理程序(kernel/system_call.s, 176)。
51 extern int system_call(void); // 系统调用中断处理程序(kernel/system_call.s, 80)。
52
53 union task_union { // 定义任务联合(任务结构成员和 stack 字符数组成员)。
54     struct task_struct task; // 因为一个任务的数据结构与其内核态堆栈放在同一内存页
55     char stack[PAGE_SIZE]; // 中, 所以从堆栈段寄存器 ss 可以获得其数据段选择符。
56 };
    // 每个任务（进程）在内核态运行时都有自己的内核态堆栈。这里定义了任务的内核态堆栈结构。

```

```

57
58 static union task union init_task = {INIT_TASK,}; // 定义初始任务的数据(sched.h 中)。
59
60 long volatile jiffies=0; // 从开机开始算起的滴答数时间值(10ms/滴答)。
// 前面的限定符 volatile, 英文解释是易变、不稳定的意思。这里是要求 gcc 不要对该变量进行优化
// 处理, 也不要挪动位置, 因为也许别的程序会来修改它的值。
61 long startup_time=0; // 开机时间。从 1970:0:0:0 开始计时的秒数。
62 struct task_struct *current = &(init_task.task); // 当前任务指针(初始化为初始任务)。
63 struct task_struct *last_task_used_math = NULL; // 使用过协处理器任务的指针。
64
65 struct task_struct * task[NR_TASKS] = {&(init_task.task), }; // 定义任务指针数组。
66
67 long user_stack [ PAGE_SIZE>>2 ] ; // 定义用户堆栈, 4K。指针指在最后一项。
68
// 该结构用于设置堆栈 ss:esp (数据段选择符, 指针), 见 head.s, 第 23 行。
69 struct {
70     long * a;
71     short b;
72     } stack_start = { & user_stack [PAGE_SIZE>>2] , 0x10 };
73 /*
74  * 'math_state_restore()' saves the current math information in the
75  * old math state array, and gets the new ones from the current task
76  */
77 /*
78  * 将当前协处理器内容保存到老协处理器状态数组中, 并将当前任务的协处理器
79  * 内容加载进协处理器。
80  */
// 当任务被调度交换过以后, 该函数用以保存原任务的协处理器状态(上下文)并恢复新调度进来的
// 当前任务的协处理器执行状态。
77 void math_state_restore()
78 {
79     if (last_task_used_math == current) // 如果任务没变则返回(上一个任务就是当前任务)。
80         return; // 这里所指的"上一个任务"是刚被交换出去的任务。
81     __asm__("fwait"); // 在发送协处理器命令之前要先发 WAIT 指令。
82     if (last_task_used_math) { // 如果上个任务使用了协处理器, 则保存其状态。
83         __asm__("fnsave %0"::"m" (last_task_used_math->tss.i387));
84     }
85     last_task_used_math=current; // 现在, last_task_used_math 指向当前任务,
// 以备当前任务被交换出去时使用。
86     if (current->used_math) { // 如果当前任务用过协处理器, 则恢复其状态。
87         __asm__("frstor %0"::"m" (current->tss.i387));
88     } else { // 否则的话说明是第一次使用,
89         __asm__("fninit"::); // 于是就向协处理器发初始化命令,
90         current->used_math=1; // 并设置使用了协处理器标志。
91     }
92 }
93
94 /*
95  * 'schedule()' is the scheduler function. This is GOOD CODE! There
96  * probably won't be any reason to change this, as it should work well
97  * in all circumstances (ie gives IO-bound processes good response etc).
98  * The one thing you might take a look at is the signal-handler code here.
99  *

```

```

100  * NOTE!! Task 0 is the 'idle' task, which gets called when no other
101  * tasks can run. It can not be killed, and it cannot sleep. The 'state'
102  * information in task[0] is never used.
103  */
/*
 * 'schedule()' 是调度函数。这是个很好的代码！没有任何理由对它进行修改，因为它可以在所有的
 * 环境下工作（比如能够对 IO-边界处理很好的响应等）。只有一件事值得留意，那就是这里的信号
 * 处理代码。
 * 注意！！任务 0 是个闲置('idle')任务，只有当没有其它任务可以运行时才调用它。它不能被杀
 * 死，也不能睡眠。任务 0 中的状态信息'state'是从来不用了的。
 */
104 void schedule(void)
105 {
106     int i,next,c;
107     struct task_struct ** p;          // 任务结构指针的指针。
108
109     /* check alarm, wake up any interruptible tasks that have got a signal */
    /* 检测 alarm（进程的报警定时值），唤醒任何已得到信号的可中断任务 */
110
    // 从任务数组中最后一个任务开始检测 alarm。
111     for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
112         if (*p) {
113             // 如果设置过任务的定时值 alarm，并且已经过期(alarm<jiffies),则在信号位图中置 SIGALRM 信号，
114             // 即向任务发送 SIGALRM 信号。然后清 alarm。该信号的默认操作是终止进程。
115             // jiffies 是系统从开机开始算起的滴答数（10ms/滴答）。定义在 sched.h 第 139 行。
116             if ((*p)->alarm && (*p)->alarm < jiffies) {
117                 (*p)->signal |= (1<<(SIGALRM-1));
118                 (*p)->alarm = 0;
119             }
120             // 如果信号位图中除被阻塞的信号外还有其它信号，并且任务处于可中断状态，则置任务为就绪状态。
121             // 其中'~(BLOCKABLE & (*p)->blocked)'用于忽略被阻塞的信号，但 SIGKILL 和 SIGSTOP 不能被阻塞。
122             if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
123                 (*p)->state==TASK_INTERRUPTIBLE)
124                 (*p)->state=TASK_RUNNING;      //置为就绪（可执行）状态。
125         }
126
127     /* this is the scheduler proper: */
    /* 这里是调度程序的主要部分 */
128
129     while (1) {
130         c = -1;
131         next = 0;
132         i = NR_TASKS;
133         p = &task[NR_TASKS];
134         // 这段代码也是从任务数组的最后一个任务开始循环处理，并跳过不含任务的数组槽。比较每个就绪
135         // 状态任务的 counter（任务运行时间的递减滴答计数）值，哪一个值大，运行时间还不长，next 就
136         // 指向哪个的任务号。
137         while (--i) {
138             if (!*--p)
139                 continue;
140             if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
141                 c = (*p)->counter, next = i;
142         }
143     }

```

```

// 如果比较得出有 counter 值大于 0 的结果，则退出 124 行开始的循环，执行任务切换（141 行）。
135         if (c) break;
// 否则就根据每个任务的优先权值，更新每一个任务的 counter 值，然后回到 125 行重新比较。
// counter 值的计算方式为 counter = counter / 2 + priority。这里计算过程不考虑进程的状态。
136         for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
137             if (*p)
138                 (*p)->counter = ((*p)->counter >> 1) +
139                     (*p)->priority;
140     }
// 切换到任务号为 next 的任务运行。在 126 行 next 被初始化为 0。因此若系统中没有任何其它任务
// 可运行时，则 next 始终为 0。因此调度函数会在系统空闲时去执行任务 0。此时任务 0 仅执行
// pause() 系统调用，并又会调用本函数。
141     switch_to(next); // 切换到任务号为 next 的任务，并运行之。
142 }
143
//// pause() 系统调用。转换当前任务的状态为可中断的等待状态，并重新调度。
// 该系统调用将导致进程进入睡眠状态，直到收到一个信号。该信号用于终止进程或者使进程调用
// 一个信号捕获函数。只有当捕获了一个信号，并且信号捕获处理函数返回，pause() 才会返回。
// 此时 pause() 返回值应该是-1，并且 errno 被置为 EINTR。这里还没有完全实现（直到 0.95 版）。
144 int sys_pause(void)
145 {
146     current->state = TASK_INTERRUPTIBLE;
147     schedule();
148     return 0;
149 }
150
// 把当前任务置为不可中断的等待状态，并让睡眠队列头的指针指向当前任务。
// 只有明确地唤醒时才会返回。该函数提供了进程与中断处理程序之间的同步机制。
// 函数参数*p 是放置等待任务的队列头指针。
151 void sleep_on(struct task_struct **p)
152 {
153     struct task_struct *tmp;
154
// 若指针无效，则退出。（指针所指的对象可以是 NULL，但指针本身不会为 0）。
155     if (!p)
156         return;
157     if (current == &(init_task.task)) // 如果当前任务是任务 0，则死机(impossible!)。
158         panic("task[0] trying to sleep");
159     tmp = *p; // 让 tmp 指向已经在等待队列上的任务(如果有的话)。
160     *p = current; // 将睡眠队列头的等待指针指向当前任务。
161     current->state = TASK_UNINTERRUPTIBLE; // 将当前任务置为不可中断的等待状态。
162     schedule(); // 重新调度。
// 只有当这个等待任务被唤醒时，调度程序才又返回到这里，则表示进程已被明确地唤醒。
// 既然大家都在等待同样的资源，那么在资源可用时，就有必要唤醒所有等待该资源的进程。该函数
// 嵌套调用，也会嵌套唤醒所有等待该资源的进程。
163     if (tmp) // 若在其前还存在等待的任务，则也将其置为就绪状态（唤醒）。
164         tmp->state=0;
165 }
166
// 将当前任务置为可中断的等待状态，并放入*p 指定的等待队列中。
167 void interruptible_sleep_on(struct task_struct **p)
168 {
169     struct task_struct *tmp;

```

```

170
171     if (!p)
172         return;
173     if (current == &(init_task.task))
174         panic("task[0] trying to sleep");
175     tmp=*p;
176     *p=current;
177 repeat: current->state = TASK_INTERRUPTIBLE;
178     schedule();
    // 如果等待队列中还有等待任务，并且队列头指针所指向的任务不是当前任务时，则将该等待任务置为
    // 可运行的就绪状态，并重新执行调度程序。当指针*p 所指向的不是当前任务时，表示在当前任务被放
    // 入队列后，又有新的任务被插入等待队列中，因此，就应该同时也将所有其它的等待任务置为可运行
    // 状态。
179     if (*p && *p != current) {
180         (**p).state=0;
181         goto repeat;
182     }
    // 下面一句代码有误，应该是*p = tmp，让队列头指针指向其余等待任务，否则在当前任务之前插入
    // 等待队列的任务均被抹掉了。当然，同时也需删除 192 行上的语句。
183     *p=NULL;
184     if (tmp)
185         tmp->state=0;
186 }
187
    // 唤醒指定任务*p。
188 void wake_up(struct task_struct **p)
189 {
190     if (p && *p) {
191         (**p).state=0;    // 置为就绪（可运行）状态。
192         *p=NULL;
193     }
194 }
195
196 /*
197  * OK, here are some floppy things that shouldn't be in the kernel
198  * proper. They are here because the floppy needs a timer, and this
199  * was the easiest way of doing it.
200  */
    /*
    * 好了，从这里开始是一些有关软盘的子程序，本不应该放在内核的主要部分中的。将它们放在这里
    * 是因为软驱需要一个时钟，而放在这里是最方便的办法。
    */
    // 这里用于软驱定时处理的代码是 201 - 262 行。在阅读这段代码之前请先看一下块设备一章中有关
    // 软盘驱动程序（floppy.c）后面的说明。或者到阅读软盘块设备驱动程序时在来看这段代码。
    // 其中时间单位：1 个滴答 = 1/100 秒。
    // 下面数组存放等待软驱马达启动到正常转速的进程指针。数组索引 0-3 分别对应软驱 A-D。
201 static struct task_struct * wait_motor[4] = {NULL, NULL, NULL, NULL};
    // 下面数组分别存放各软驱马达启动所需要的滴答数。程序中默认启动时间为 50 个滴答（0.5 秒）。
202 static int mon_timer[4]={0, 0, 0, 0};
    // 下面数组分别存放各软驱在马达停转之前需维持的时间。程序中设定为 10000 个滴答（100 秒）。
203 static int moff_timer[4]={0, 0, 0, 0};
    // 对应软驱控制器中当前数字输出寄存器。该寄存器每位的定义如下：
    // 位 7-4：分别控制驱动器 D-A 马达的启动。1 - 启动；0 - 关闭。

```

```

// 位 3 : 1 - 允许 DMA 和中断请求; 0 - 禁止 DMA 和中断请求。
// 位 2 : 1 - 启动软盘控制器; 0 - 复位软盘控制器。
// 位 1-0: 00 - 11, 用于选择控制的软驱 A-D。
204 unsigned char current_DOR = 0x0C; // 这里设置初值为: 允许 DMA 和中断请求、启动 FDC。
205
// 指定软驱启动到正常运转状态所需等待时间。
// nr -- 软驱号(0-3), 返回值为滴答数。
206 int ticks_to_floppy_on(unsigned int nr)
207 {
208     extern unsigned char selected; // 选中软驱标志(kernel/blk_drv/floppy.c, 122)。
209     unsigned char mask = 0x10 << nr; // 所选软驱对应数字输出寄存器中启动马达比特位。
210     // mask 高 4 位是各软驱启动马达标志。
211     if (nr>3)
212         panic("floppy_on: nr>3"); // 系统最多有 4 个软驱。
// 首先预先设置好指定软驱 nr 停转之前需要经过的时间(100 秒)。然后取当前 DOR 寄存器值到
// 临时变量 mask 中, 并把指定软驱的马达启动标志置位。
213     moff_timer[nr]=10000; // 100 s = very big :-) */ // 停转维持时间。
214     cli(); // use floppy_off to turn it off */
215     mask |= current_DOR;
// 如果当前没有选择软驱, 则首先复位其它软驱的选择位, 然后置指定软驱选择位。
216     if (!selected) {
217         mask &= 0xFC;
218         mask |= nr;
219     }
// 如果数字输出寄存器的当前值与要求的值不同, 则向 FDC 数字输出端口输出新值(mask), 并且如果
// 要求启动的马达还没有启动, 则置相应软驱的马达启动定时器值(HZ/2 = 0.5 秒或 50 个滴答)。若
// 已经启动, 则再设置启动定时为 2 个滴答, 能满足下面 do_floppy_timer()中先递减后判断的要求。
// 执行本次定时代码的要求即可。此后更新当前数字输出寄存器变量 current_DOR。
220     if (mask != current_DOR) {
221         outb(mask, FD_DOR);
222         if ((mask ^ current_DOR) & 0xf0)
223             mon_timer[nr] = HZ/2;
224         else if (mon_timer[nr] < 2)
225             mon_timer[nr] = 2;
226         current_DOR = mask;
227     }
228     sti();
229     return mon_timer[nr]; // 最后返回启动马达所需的时间值。
230 }
231
// 等待指定软驱马达启动所需的一段时间, 然后返回。
// 设置指定软驱的马达启动到正常转速所需的延时, 然后睡眠等待。在定时中断过程中会一直递减
// 判断这里设定的延时值。当延时到期, 就会唤醒这里的等待进程。
232 void floppy_on(unsigned int nr)
233 {
234     cli(); // 关中断。
235     while (ticks_to_floppy_on(nr)) // 如果马达启动定时还没到, 就一直把当前进程置
236         sleep_on(nr+wait_motor); // 为不可中断睡眠状态并放入等待马达运行的队列中。
237     sti(); // 开中断。
238 }
239
// 置关闭相应软驱马达停转定时器(3 秒)。
// 若不使用该函数明确关闭指定的软驱马达, 则在马达开启 100 秒之后也会被关闭。

```



```

240 void floppy_off(unsigned int nr)
241 {
242     moff_timer[nr]=3*HZ;
243 }
244
// 软盘定时处理子程序。更新马达启动定时值和马达关闭停转计时值。该子程序会在时钟定时中断
// 过程中被调用，因此系统每经过一个滴答(10ms)就会被调用一次，随时更新马达开启或停转定时
// 器的值。如果某一个马达停转定时到，则将数字输出寄存器马达启动位复位。
245 void do_floppy_timer(void)
246 {
247     int i;
248     unsigned char mask = 0x10;
249
250     for (i=0 ; i<4 ; i++,mask <<= 1) {
251         if (!(mask & current_DOR)) // 如果不是 DOR 指定的马达则跳过。
252             continue;
253         if (mon_timer[i] {
254             if (!--mon_timer[i])
255                 wake_up(i+wait_motor); // 如果马达启动定时到则唤醒进程。
256             } else if (!moff_timer[i]) { // 如果马达停转定时到则
257                 current_DOR &= ~mask; // 复位相应马达启动位，并
258                 outb(current_DOR, FD_DOR); // 更新数字输出寄存器。
259             } else
260                 moff_timer[i]--; // 马达停转计时递减。
261         }
262     }
263
264 #define TIME_REQUESTS 64 // 最多可有 64 个定时器链表（64 个任务）。
265
// 定时器链表结构和定时器数组。
266 static struct timer_list {
267     long jiffies; // 定时滴答数。
268     void (*fn)(); // 定时处理程序。
269     struct timer_list * next; // 下一个定时器。
270 } timer_list[TIME_REQUESTS], * next_timer = NULL;
271
// 添加定时器。输入参数为指定的定时值(滴答数)和相应的处理程序指针。
// 软盘驱动程序(floppy.c)利用该函数执行启动或关闭马达的延时操作。
// jiffies - 以 10 毫秒计的滴答数; *fn()- 定时时间到时执行的函数。
272 void add_timer(long jiffies, void (*fn)(void))
273 {
274     struct timer_list * p;
275
// 如果定时处理程序指针为空，则退出。
276     if (!fn)
277         return;
278     cli();
// 如果定时值<=0，则立刻调用其处理程序。并且该定时器不加入链表中。
279     if (jiffies <= 0)
280         (fn)();
281     else {
// 从定时器数组中，找一个空闲项。
282         for (p = timer_list ; p < timer_list + TIME_REQUESTS ; p++)

```

```

283             if (!p->fn)
284                 break;
// 如果已经用完了定时器数组，则系统崩溃☹。
285         if (p >= timer\_list + TIME REQUESTS)
286             panic("No more time requests free");
// 向定时器数据结构填入相应信息。并链入链表头
287         p->fn = fn;
288         p->jiffies = jiffies;
289         p->next = next\_timer;
290         next\_timer = p;
// 链表项按定时值从小到大排序。在排序时减去排在前面需要的滴答数，这样在处理定时器时只要
// 查看链表头的第一项的定时是否到期即可。[[?? 这段程序好象没有考虑周全。如果新插入的定时
// 器值 < 原来头一个定时器值时，也应该将所有后面的定时值均减去新的第 1 个的定时值。]]
291         while (p->next && p->next->jiffies < p->jiffies) {
292             p->jiffies -= p->next->jiffies;
293             fn = p->fn;
294             p->fn = p->next->fn;
295             p->next->fn = fn;
296             jiffies = p->jiffies;
297             p->jiffies = p->next->jiffies;
298             p->next->jiffies = jiffies;
299             p = p->next;
300         }
301     }
302     sti();
303 }
304
//// 时钟中断 C 函数处理程序，在 kernel/system_call.s 中的_timer_interrupt (176 行) 被调用。
// 参数 cpl 是当前特权级 0 或 3，0 表示内核代码在执行。
// 对于一个进程由于执行时间片用完时，则进行任务切换。并执行一个计时更新工作。
305 void do\_timer(long cpl)
306 {
307     extern int beepcount; // 扬声器发声时间滴答数(kernel/chr_drv/console.c, 697)
308     extern void sysbeepstop(void); // 关闭扬声器(kernel/chr_drv/console.c, 691)
309
// 如果发声计数次数到，则关闭发声。(向 0x61 口发送命令，复位位 0 和 1。位 0 控制 8253
// 计数器 2 的工作，位 1 控制扬声器)。
310     if (beepcount)
311         if (!--beepcount)
312             sysbeepstop();
313
// 如果当前特权级(cpl)为 0 (最高，表示是内核程序在工作)，则将内核程序运行时间 stime 递增；
// [ Linux 把内核程序统称为超级用户(supervisor)的程序，见 system_call.s, 193 行上的英文注释]
// 如果 cpl > 0，则表示是一般用户程序在工作，增加 utime。
314     if (cpl)
315         current->utime++;
316     else
317         current->stime++;
318
// 如果有用户的定时器存在，则将链表第 1 个定时器的值减 1。如果已等于 0，则调用相应的处理
// 程序，并将该处理程序指针置为空。然后去掉该项定时器。
319     if (next\_timer) { // next_timer 是定时器链表的头指针(见 270 行)。
320         next\_timer->jiffies--;

```



```

321         while (next_timer && next_timer->jiffies <= 0) {
322             void (*fn)(void); // 这里插入了一个函数指针定义!!! ⊗
323
324             fn = next_timer->fn;
325             next_timer->fn = NULL;
326             next_timer = next_timer->next;
327             (fn)(); // 调用处理函数。
328         }
329     }
    // 如果当前软盘控制器 FDC 的数字输出寄存器中马达启动位有置位的, 则执行软盘定时程序(245 行)。
330     if (current_DOR & 0xf0)
331         do_floppy_timer();
332     if ((--current->counter)>0) return; // 如果进程运行时间还没完, 则退出。
333     current->counter=0;
334     if (!cpl) return; // 对于内核态程序, 不依赖 counter 值进行调度。
335     schedule();
336 }
337
    // 系统调用功能 - 设置报警定时时间值(秒)。
    // 如果参数 seconds>0, 则设置该新的定时值并返回原定时值。否则返回 0。
338 int sys_alarm(long seconds)
339 {
340     int old = current->alarm;
341
342     if (old)
343         old = (old - jiffies) / HZ;
344     current->alarm = (seconds>0)?(jiffies+HZ*seconds):0;
345     return (old);
346 }
347
    // 取当前进程号 pid。
348 int sys_getpid(void)
349 {
350     return current->pid;
351 }
352
    // 取父进程号 ppid。
353 int sys_getppid(void)
354 {
355     return current->father;
356 }
357
    // 取用户号 uid。
358 int sys_getuid(void)
359 {
360     return current->uid;
361 }
362
    // 取 euid。
363 int sys_geteuid(void)
364 {
365     return current->euid;
366 }

```

```

367 // 取组号 gid。
368 int sys_getgid(void)
369 {
370     return current->gid;
371 }
372 // 取 egid。
373 int sys_getegid(void)
374 {
375     return current->egid;
376 }
377 // 系统调用功能 -- 降低对 CPU 的使用优先权（有人会用吗？☺）。
// 应该限制 increment 大于 0，否则的话，可使优先权增大！！
378 int sys_nice(long increment)
379 {
380     if (current->priority-increment>0)
381         current->priority -= increment;
382     return 0;
383 }
384 // 调度程序的初始化子程序。
385 void sched_init(void)
386 {
387     int i;
388     struct desc_struct * p; // 描述符表结构指针。
389
390     if (sizeof(struct sigaction) != 16) // sigaction 是存放有关信号状态的结构。
391         panic("Struct sigaction MUST be 16 bytes");
// 设置初始任务（任务 0）的任务状态段描述符和局部数据表描述符(include/asm/system.h, 65)。
392     set_tss_desc(gdt+FIRST_TSS_ENTRY, &(init_task.task.tss));
393     set_ldt_desc(gdt+FIRST_LDT_ENTRY, &(init_task.task.ldt));
// 清任务数组和描述符表项（注意 i=1 开始，所以初始任务的描述符还在）。
394     p = gdt+2+FIRST_TSS_ENTRY;
395     for(i=1; i<NR_TASKS; i++) {
396         task[i] = NULL;
397         p->a=p->b=0;
398         p++;
399         p->a=p->b=0;
400         p++;
401     }
402 /* Clear NT, so that we won't have troubles with that later on */
// 清除标志寄存器中的位 NT，这样以后就不会有麻烦 */
// NT 标志用于控制程序的递归调用(Nested Task)。当 NT 置位时，那么当前中断任务执行
// iret 指令时就会引起任务切换。NT 指出 TSS 中的 back_link 字段是否有效。
403     asm ("pushfl ; andl $0xffffbfff, (%esp) ; popfl"); // 复位 NT 标志。
404     ltr(0); // 将任务 0 的 TSS 加载到任务寄存器 tr。
405     lldt(0); // 将局部描述符表加载到局部描述符表寄存器。
// 注意！！是将 GDT 中相应 LDT 描述符的选择符加载到 ldtr。只明确加载这一次，以后新任务
// LDT 的加载，是 CPU 根据 TSS 中的 LDT 项自动加载。

// 下面代码用于初始化 8253 定时器。

```

```

406     outb_p(0x36, 0x43);          /* binary, mode 3, LSB/MSB, ch 0 */
407     outb_p(LATCH & 0xff, 0x40); /* LSB */ // 定时值低字节。
408     outb(LATCH >> 8, 0x40);     /* MSB */ // 定时值高字节。
// 设置时钟中断处理程序句柄（设置时钟中断门）。
409     set_intr_gate(0x20, &timer_interrupt);
// 修改中断控制器屏蔽码，允许时钟中断。
410     outb(inb_p(0x21) & ~0x01, 0x21);
// 设置系统调用中断门。
411     set_system_gate(0x80, &system_call);
412 }
413

```

## 5.8.3 其它信息

### 5.8.3.1 软盘控制器编程

在编程时需要访问 4 个端口，分别对应一个或多个寄存器。对于 1.2M 的软盘控制器有以下一些端口。

表 5-3 软盘控制器端口

I/O 端口	读写性	寄存器名称
0x3f2	只写	数字输出寄存器(数字控制寄存器)
0x3f4	只读	FDC 主状态寄存器
0x3f5	读/写	FDC 数据寄存器
0x3f7	只读	数字输入寄存器
0x3f7	只写	磁盘控制寄存器(传输率控制)

数字输出端口（数字控制端口）是一个 8 位寄存器，它控制驱动器马达开启、驱动器选择、启动/复位 FDC 以及允许/禁止 DMA 及中断请求。

FDC 的主状态寄存器也是一个 8 位寄存器，用于反映软盘控制器 FDC 和软盘驱动器 FDD 的基本状态。通常，在 CPU 向 FDC 发送命令之前或从 FDC 获取操作结果之前，都要读取主状态寄存器的状态位，以判别当前 FDC 数据寄存器是否就绪，以及确定数据传送的方向。

FDC 的数据端口对应多个寄存器（只写型命令寄存器和参数寄存器、只读型结果寄存器），但任一时刻只能有一个寄存器出现在数据端口 0x3f5。在访问只写型寄存器时，主状态控制的 DIO 方向位必须为 0（CPU → FDC），访问只读型寄存器时则反之。在读取结果时只有在 FDC 不忙之后才算读完结果，通常结果数据最多有 7 个字节。

软盘控制器共可以接受 15 条命令。每个命令均经历三个阶段：命令阶段、执行阶段和结果阶段。

命令阶段是 CPU 向 FDC 发送命令字节和参数字节。每条命令的第一个字节总是命令字节（命令码）。其后跟着 0-8 字节的参数。

执行阶段是 FDC 执行命令规定的操作。在执行阶段 CPU 是不加干预的，一般是通过 FDC 发出中断请求获知命令执行的结束。如果 CPU 发出的 FDC 命令是传送数据，则 FDC 可以以中断方式或 DMA 方式进行。中断方式每次传送 1 字节。DMA 方式是在 DMA 控制器管理下，FDC 与内存进行数据的传输直至全部数据传送完。此时 DMA 控制器会将传输字节计数终止信号通知 FDC，最后由 FDC 发出中断请求信号告知 CPU 执行阶段结束。

结果阶段是由 CPU 读取 FDC 数据寄存器返回值，从而获得 FDC 命令执行的结果。返回结果数据的长度为 0-7 字节。对于没有返回结果数据的命令，则应向 FDC 发送检测中断状态命令获得操作的状态。

## 5.9 signal.c 程序

### 5.9.1 功能描述

signal.c 程序涉及内核中所有有关信号处理的函数。在 UNIX 系统中，信号是一种“软件中断”处理机制。有许多较为复杂的程序会使用到信号。信号机制提供了一种处理异步事件的方法。例如，用户在终端键盘上键入 `ctrl-C` 组合键来终止一个程序的执行。该操作就会产生一个 `SIGINT` (`SIGnal INTerrupt`) 信号，并被发送到当前前台执行的进程中；当进程设置了一个报警时钟到期时，系统就会向进程发送一个 `SIGALRM` 信号；当发生硬件异常时，系统也会向正在执行的进程发送相应的信号。另外，一个进程也可以向另一个进程发送信号。例如使用 `kill()` 函数向同组的子进程发送终止执行信号。

信号处理机制在很早的 UNIX 系统中就已经有了，但那些早期 UNIX 内核中信号处理的方法并不是那么可靠。信号可能会被丢失，而且在处理紧要区域代码时进程有时很难关闭一个指定的信号，后来 POSIX 提供了一种可靠处理信号的方法。为了保持兼容性，本程序中还是提供了两种处理信号的方法。

通常使用一个无符号长整数 (32 位) 中的比特位表示各种不同信号。因此最多可表示 32 个不同的信号。在本版 Linux 内核中，定义了 22 种不同的信号。其中 20 种信号是 POSIX.1 标准中规定的所有信号，另外 2 种是 Linux 的专用信号：`SIGUNUSED` (未定义) 和 `SIGSTKFLT` (堆栈错)，前者可表示系统目前还不支持的所有其它信号种类。这 22 种信号的具体名称和定义可参考程序后的信号列表，也可参阅 `include/signal.h` 头文件。

对于进程来说，当收到一个信号时，可以由三种不同的处理或操作方式。

1. 忽略该信号。大多数信号都可以被进程忽略。但有两个信号忽略不掉：`SIGKILL` 和 `SIGSTOP`。不能被忽略掉的原因是能为超级用户提供一个确定的方法来终止或停止指定的任何进程。另外，若忽略掉某些硬件异常而产生的信号 (例如被 0 除)，则进程的行为或状态就可能变得不可知了。
2. 捕获该信号。为了进行该操作，我们必须首先告诉内核在指定的信号发生时调用我们自定义的信号处理函数。在该处理函数中，我们可以做任何操作，当然也可以什么不做，起到忽略该信号的同样作用。自定义信号处理函数来捕获信号的一个例子是：如果我们在程序执行过程中创建了一些临时文件，那么我们就可以定义一个函数来捕获 `SIGTERM` (终止执行) 信号，并在该函数中做一些清理临时文件的工作。`SIGTERM` 信号是 `kill` 命令发送的默认信号。
3. 执行默认操作。内核为每种信号都提供一种默认操作。通常这些默认操作就是终止进程的执行。参见程序后信号列表中的说明。

本程序给出了设置和获取进程信号阻塞码 (屏蔽码) 系统调用函数 `sys_ssetmask()` 和 `sys_sgetmask()`、信号处理系统调用 `sys_sigal()` (即传统信号处理函数 `signal()`)、修改进程在收到特定信号时所采取的行动的系统调用 `sys_sigaction()` (既可靠信号处理函数 `sigaction()`) 以及在系统调用中断处理程序中处理信号的函数 `do_signal()`。有关信号操作的发送信号函数 `send_sig()` 和通知父进程函数 `tell_father()` 则被包含在另一个程序 (`exit.c`) 中。程序中的名称前缀 `sig` 均是信号 `signal` 的简称。

`signal()` 和 `sigaction()` 的功能比较类似，都是更改信号原处理句柄 (`handler`，或称为处理程序)。但 `signal()` 就是内核操作上述传统信号处理的方式，在某些特殊时刻可能会造成信号丢失。

在 `include/signal.h` 头文件第 55 行上，`signal()` 函数原型申明如下：

```
void (*signal(int signr, void (*handler)(int)))(int);
```

这个 `signal()` 函数含有两个参数。一个指定需要捕获的信号 `signr`；另外一个新的信号处理函数指针 (新的信号处理句柄) `void (*handler)(int)`。新的信号处理句柄是一个无返回值且具有一个整型参数的函数指针，该整型参数用于当指定信号发生时内核将其传递给处理句柄。`signal()` 函数会返回原信号处理

句柄，这个返回的句柄也是一个无返回值且具有一个整型参数的函数指针。并且在新句柄被调用执行过一次后，信号处理句柄又会被恢复成默认处理句柄值 `SIG_DFL`。

在 `include/signal.h` 文件中（第 45 行起），默认句柄 `SIG_DFL` 和忽略处理句柄 `SIG_IGN` 的定义是：

```
#define SIG_DFL      ((void (*)(int))0)
#define SIG_IGN      ((void (*)(int))1)
```

都分别表示无返回值的函数指针，与 `signal()` 函数中第二个参数的要求相同。指针值分别是 0 和 1。这两个指针值逻辑上讲是实际程序中不可能出现的函数地址值。因此在 `signal()` 函数中就可以根据这两个特殊的指针值来判断是否使用默认信号处理句柄或忽略对信号的处理（当然 `SIGTERM` 和 `SIGSTOP` 是不能被忽略的）。参见下面程序列表中第 94—98 行的处理过程。

当一个程序被执行时，系统会设置其处理所有信号的方式为 `SIG_DFL` 或 `SIG_IGN`。另外，当程序 `fork()` 一个子进程时，子进程会继承父进程的信号处理方式（信号屏蔽码）。因此父进程对信号的设置和处理方式在子进程中同样有效。

为了能连续地捕获一个指定的信号，`signal()` 函数的通常使用方式例子如下。

---

```
void sig_handler(int signr)           // 信号句柄。
{
    signal(SIGINT, sig_handler);      // 为处理下一次信号发生而重新设置自己的处理句柄。
    ...
}

main ()
{
    signal(SIGINT, sig_handler);      // 主程序中设置自己的信号处理句柄。
    ...
}
```

---

`signal()` 函数不可靠的原因在于当信号已经发生而进入自己设置的信号处理函数中，但在重新再一次设置自己的处理句柄之前，在这段时间内有可能又有一个信号发生。但是此时系统已经把处理句柄设置成默认值。因此就有可能造成信号丢失。

`sigaction()` 函数采用了 `sigaction` 数据结构来保存指定信号的信息，它是一种可靠的内核处理信号的机制，它可以让方便地查看或修改指定信号的处理句柄。该函数是 `signal()` 函数的一个超集。该函数在 `include/signal.h` 头文件（第 66 行）中的申明为：

```
int sigaction(int sig, struct sigaction *act, struct sigaction *oldact);
```

其中参数 `sig` 是我们需要查看或修改其信号处理句柄的信号，后两个参数是 `sigaction` 结构的指针。当参数 `act` 指针不是 `NULL` 时，就可以根据 `act` 结构中的信息修改指定信号的行为。当 `oldact` 不为空时，内核就会在该结构中返回信号原来的设置信息。`sigaction` 结构见如下所示：

---

```
48 struct sigaction {
49     void (*sa_handler)(int);           // 信号处理句柄。
50     sigset_t sa_mask;                 // 信号的屏蔽码，可以阻塞指定的信号集。
51     int sa_flags;                     // 信号选项标志。
52     void (*sa_restorer)(void);        // 信号恢复函数指针（系统内部使用）。
53 };
```

---

当修改对一个信号的处理方法时，如果处理句柄 `sa_handler` 不是默认处理句柄 `SIG_DFL` 或忽略处理句柄 `SIG_IGN`，那么在 `sa_handler` 处理句柄可被调用前，`sa_mask` 字段就指定了需要加入到进程信号屏蔽位图中的一个信号集。如果信号处理句柄返回，系统就会恢复进程原来的信号屏蔽位图。这样在一个信号句柄被调用时，我们就可以阻塞指定的一些信号。当信号句柄被调用时，新的信号屏蔽位图会自动地把当前发送的信号包括进去，阻塞该信号的继续发送。从而在我们处理一指定信号期间能确保阻塞同一个信号而不让其丢失，直到此次处理完毕。另外，在一个信号被阻塞期间而又多次发生时通常只保存其一个样例，也即在阻塞解除时对于阻塞的多个同一信号只会再调用一次信号处理句柄。在我们修改了一个信号的处理句柄之后，除非再次更改，否则就一直使用该处理句柄。这与传统的 `signal()` 函数不一样。`signal()` 函数会在一处处理句柄结束后将其恢复成信号的默认处理句柄。

`sigaction` 结构中的 `sa_flags` 用于指定其它一些处理信号的选项，这些选项的定义请参见 `include/signal.h` 文件中（第 36-39 行）的说明。

`sigaction` 结构中的最后一个字段和 `sys_signal()` 函数的参数 `restorer` 是一函数指针。在编译连接程序时由 `Libc` 函数库提供，用于在信号处理程序结束后清理用户态堆栈，并恢复系统调用存放在 `eax` 中的返回值，见下面详细说明。

`do_signal()` 函数是内核系统调用(`int 0x80`)中断处理程序中对信号的预处理程序。在进程每次调用系统调用时，若该进程已收到信号，则该函数就会把信号的处理句柄（即对应的信号处理函数）插入到用户程序堆栈中。这样，在当前系统调用结束返回后就会立刻执行信号句柄程序，然后再继续执行用户的程序，见图 5-7 所示。

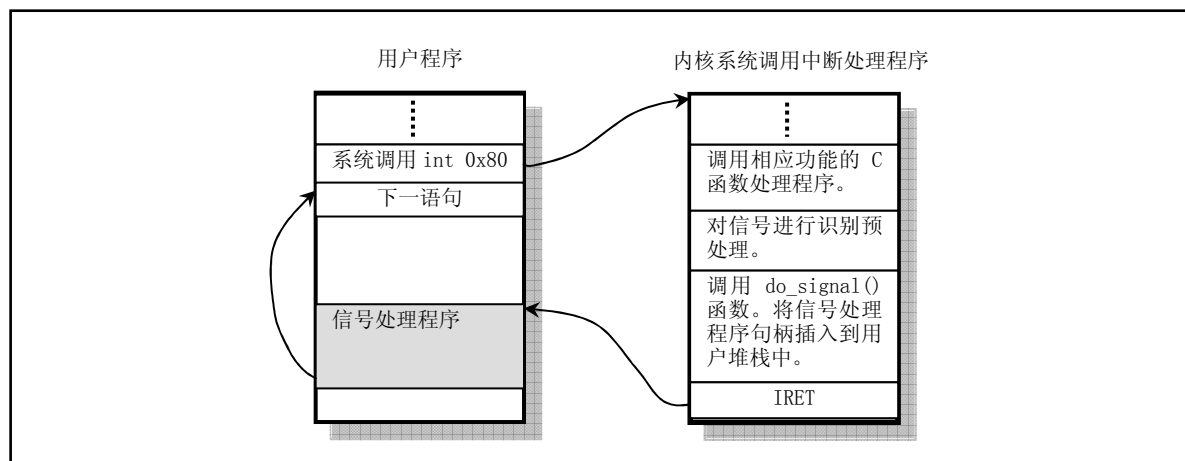


图 5-7 信号处理程序的调用方式。

在 `do_signal()` 函数把信号处理程序的参数插入到用户堆栈中之前，首先会把在用户程序堆栈指针向下扩展 `long` 个长字（参见下面程序中 106 行），然后将相关的参数添入其中，参见图 5-8 所示。由于 `do_signal()` 函数从 104 行开始的代码比较难以理解，下面我们将对其进行详细描述。

在用户程序调用系统调用刚进入内核时，该进程的内核态堆栈上会由 CPU 自动压入如图 5-8 中所示的内容，也即：用户程序的 `SS` 和 `ESP` 以及用户程序中下一条指令的执行点位置 `CS` 和 `EIP`。在处理完此次指定的系统调用功能并准备调用 `do_signal()` 时（也即 `system_call.s` 程序 118 行之后），内核态堆栈中的内容见图 5-9 中左边所示。因此 `do_signal()` 的参数即是这些在内核态堆栈上的内容。

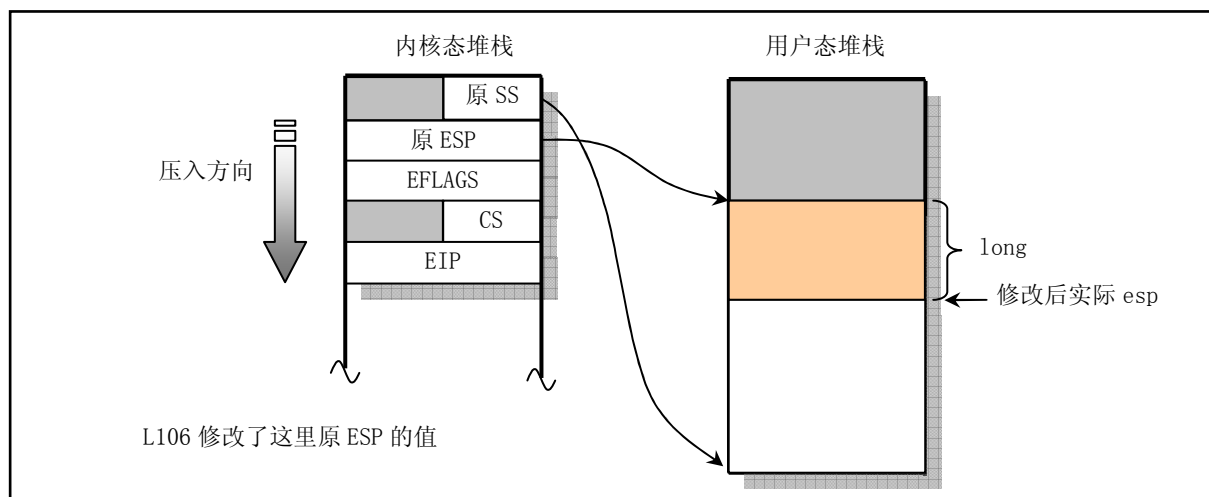


图 5-8 do\_signal() 函数对用户堆栈的修改

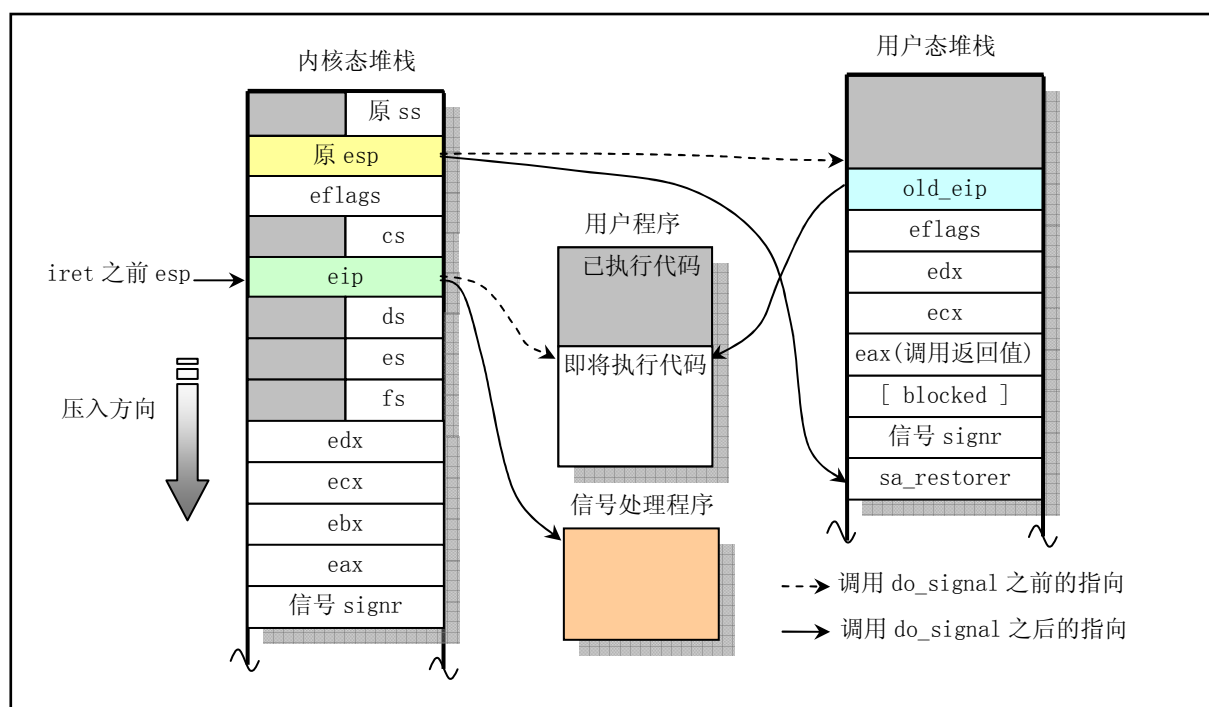


图 5-9 do\_signal() 函数修改用户态堆栈的具体过程

在处理完两个默认信号句柄 (SIG\_IGN 和 SIG\_DFL) 之后, 若用户自定义了信号处理程序 (信号句柄 sa\_handler), 则从 104 行起 do\_signal() 开始准备把用户自定义的句柄插入用户态堆栈中。它首先把内核态堆栈中原用户程序的返回执行点指针 eip 保存为 old\_eip, 然后将该 eip 替换成指向自定义句柄 sa\_handler, 也即让图中内核态堆栈中的 eip 指向 sa\_handler。接下来通过把内核态中保存的“原 esp”减去 longs 值, 把用户态堆栈向下扩展了 7 或 8 个长字空间。最后把内核堆栈上的一些寄存器内容复制到了这个空间中, 见图中右边所示。

总共往用户态堆栈上放置了 7 到 8 个值, 我们现在来说明这些值的含义以及放置这些值的原因。

old\_eip 即是原用户程序的返回地址, 是在内核堆栈上 eip 被替换成信号句柄地址之前保留下来的。eflags、edx 和 ecx 是原用户程序在调用系统调用之前的值, 基本上也是调用系统调用的参数, 在系统调用返回后仍然需要恢复这些用户程序的寄存器值。eax 中保存有系统调用的返回值。如果所处理的信号还允许收到本身, 则堆栈上还存放有该进程的阻塞码 blocked。下一个是信号 signr 值。



最后一个是信号活动恢复函数的指针 `sa_restorer`。这个恢复函数不是由用户设定的，因为在用户定义 `signal()` 函数时只提供了一个信号值 `signr` 和一个信号处理句柄 `handler`。

下面是为 **SIGINT** 信号设置自定义信号处理句柄的一个简单例子，默认情况下，按下 **Ctrl-C** 组合键会产生 **SIGINT** 信号。

---

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void handler(int sig)                // 信号处理句柄。
{
    printf("The signal is %d\n", sig);
    (void) signal(SIGINT, SIG_DFL); // 恢复 SIGINT 信号的默认处理句柄。（实际上内核会
}                                     // 自动恢复默认值，但对于其它系统未必如此）

int main()
{
    (void) signal(SIGINT, handler); // 设置 SIGINT 的用户自定义信号处理句柄。
    while (1) {
        printf("Signal test.\n");
        sleep(1);                    // 等待 1 秒钟。
    }
}
```

---

其中，信号处理函数 `handler()` 会在信号 **SIGINT** 出现时被调用执行。该函数首先输出一条信息，然后会把 **SIGINT** 信号的处理设置成默认信号处理句柄。因此在第二次按下 **Ctrl-C** 组合键时，**SIG\_DFL** 会让该程序结束运行。

那么 `sa_restorer` 这个函数是从哪里来的呢？其实它是由库函数提供的，在 **Libc** 函数库中有它的函数，定义如下：

---

```
.globl __sig_restore
.globl __mask_sig_restore
# 若没有 blocked 则使用这个 restorer 函数
__sig_restore:
    addl $4,%esp        # 丢弃信号值 signr
    popl %eax           # 恢复系统调用返回值。
    popl %ecx           # 恢复原用户程序寄存器值。
    popl %edx
    popfl               # 恢复用户程序时的标志寄存器。
    ret
# 若有 blocked 则使用下面这个 restorer 函数，blocked 供 ssetmask 使用。
__mask_sig_restore:
    addl $4,%esp        # 丢弃信号值 signr
    call __ssetmask     # 设置信号屏蔽码 old blocking
    addl $4,%esp        # 丢弃 blocked 值。
    popl %eax
    popl %ecx
    popl %edx
    popfl
    ret
```

---



该函数的主要作用是为了在信号处理程序结束后，恢复用户程序执行系统调用后的返回值和一些寄存器内容，并清除作为信号处理程序参数的信号值 `signr`。在编译连接用户自定义的信号处理程序时，编译程序会把这个函数插入到用户程序中。由该函数负责清理在信号处理程序执行完后恢复用户程序的寄存器值和系统调用返回值，就好象没有运行过信号处理程序，而是直接从系统调用中返回的。

最后说明一下执行的流程。在 `do_signal()` 执行完后，`system_call.s` 将会把进程内核态上 `eip` 以下的所有值弹出堆栈。在执行了 `iret` 指令之后，CPU 将把内核态堆栈上的 `cs:eip`、`eflags` 以及 `ss:esp` 弹出，恢复到用户态去执行程序。由于 `eip` 已经被替换为指向信号句柄，因此，此刻即会立即执行用户自定义的信号处理程序。在该信号处理程序执行完后，通过 `ret` 指令，CPU 会把控制权移交给 `sa_restorer` 所指向的恢复程序去执行。而 `sa_restorer` 程序会做一些用户态堆栈的清理工作，也即会跳过堆栈上的信号值 `signr`，并把系统调用后的返回值 `eax` 和寄存器 `ecx`、`edx` 以及标志寄存器 `eflags`。完全恢复了系统调用后各寄存器和 CPU 的状态。最后通过 `sa_restorer` 的 `ret` 指令弹出原用户程序的 `eip`（也即堆栈上的 `old_eip`），返回去执行用户程序。

## 5.9.2 代码注释

程序 5-7 linux/kernel/signal.c

```

1  /*
2   *  linux/kernel/signal.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  #include <linux/sched.h> // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                           // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
8  #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
9  #include <asm/segment.h> // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
10
11 #include <signal.h>      // 信号头文件。定义信号符号常量，信号结构以及信号操作函数原型。
12
13 volatile void do_exit(int error_code); // 前面的限定符 volatile 要求编译器不要对其进行优化。
14
15 // 获取当前任务信号屏蔽位图（屏蔽码）。
16 int sys_sgetmask()
17 {
18     return current->blocked;
19 }
20
21 // 设置新的信号屏蔽位图。SIGKILL 不能被屏蔽。返回值是原信号屏蔽位图。
22 int sys_ssetmask(int newmask)
23 {
24     int old=current->blocked;
25
26     current->blocked = newmask & ~(1<<(SIGKILL-1));
27     return old;
28 }
29
30 // 复制 sigaction 数据到 fs 数据段 to 处。。
31 static inline void save_old(char * from, char * to)
32 {
33     int i;

```

```

31
32     verify_area(to, sizeof(struct sigaction)); // 验证 to 处的内存是否足够。
33     for (i=0 ; i< sizeof(struct sigaction) ; i++) {
34         put_fs_byte(*from,to);                // 复制到 fs 段。一般是用户数据段。
35         from++;                                // put_fs_byte() 在 include/asm/segment.h 中。
36         to++;
37     }
38 }
39
40 // 把 sigaction 数据从 fs 数据段 from 位置复制到 to 处。
41 static inline void get_new(char * from, char * to)
42 {
43     int i;
44     for (i=0 ; i< sizeof(struct sigaction) ; i++)
45         *(to++) = get_fs_byte(from++);
46 }
47
48 // signal() 系统调用。类似于 sigaction()。为指定的信号安装新的信号句柄(信号处理程序)。
49 // 信号句柄可以是用户指定的函数，也可以是 SIG_DFL（默认句柄）或 SIG_IGN（忽略）。
50 // 参数 signum -- 指定的信号； handler -- 指定的句柄； restorer - 恢复函数指针，该函数由 Libc
51 // 库提供。用于在信号处理程序结束后恢复系统调用返回时几个寄存器的原有值以及系统调用的返回
52 // 值，就好象系统调用没有执行过信号处理程序而直接返回到用户程序一样。
53 // 函数返回原信号句柄。
54 int sys_signal(int signum, long handler, long restorer)
55 {
56     struct sigaction tmp;
57
58     if (signum<1 || signum>32 || signum==SIGKILL) // 信号值要在 (1-32) 范围内，
59         return -1;                               // 并且不得是 SIGKILL。
60     tmp.sa_handler = (void (*)(int)) handler;      // 指定的信号处理句柄。
61     tmp.sa_mask = 0;                              // 执行时的信号屏蔽码。
62     tmp.sa_flags = SA_ONESHOT | SA_NOMASK;         // 该句柄只使用 1 次后就恢复到默认值，
63                                                     // 并允许信号在自己的处理句柄中收到。
64     tmp.sa_restorer = (void (*)(void)) restorer;  // 保存恢复处理函数指针。
65     handler = (long) current->sigaction[signum-1].sa_handler;
66     current->sigaction[signum-1] = tmp;
67     return handler;
68 }
69
70 // sigaction() 系统调用。改变进程在收到一个信号时的操作。signum 是除了 SIGKILL 以外的任何
71 // 信号。[如果新操作(action)不为空]则新操作被安装。如果 oldaction 指针不为空，则原操作
72 // 被保留到 oldaction。成功则返回 0，否则为-1。
73 int sys_sigaction(int signum, const struct sigaction * action,
74                  struct sigaction * oldaction)
75 {
76     struct sigaction tmp;
77
78     // 信号值要在 (1-32) 范围内，并且信号 SIGKILL 的处理句柄不能被改变。
79     if (signum<1 || signum>32 || signum==SIGKILL)
80         return -1;
81     // 在信号的 sigaction 结构中设置新的操作（动作）。
82     tmp = current->sigaction[signum-1];

```

```

71     get_new((char *) action,
72             (char *) (signum-1+current->sigaction));
// 如果 oldaction 指针不为空的话, 则将原操作指针保存到 oldaction 所指的位置。
73     if (oldaction)
74         save_old((char *) &tmp, (char *) oldaction);
// 如果允许信号在自己的信号句柄中收到, 则令屏蔽码为 0, 否则设置屏蔽本信号。
75     if (current->sigaction[signum-1].sa_flags & SA_NOMASK)
76         current->sigaction[signum-1].sa_mask = 0;
77     else
78         current->sigaction[signum-1].sa_mask |= (1<<(signum-1));
79     return 0;
80 }
81
// 系统调用中断处理程序中真正的信号处理程序 (在 kernel/system_call.s, 119 行)。
// 该段代码的主要作用是将信号的处理句柄插入到用户程序堆栈中, 并在本系统调用结束
// 返回后立刻执行信号句柄程序, 然后继续执行用户的程序。
82 void do_signal(long signr, long eax, long ebx, long ecx, long edx,
83               long fs, long es, long ds,
84               long eip, long cs, long eflags,
85               unsigned long * esp, long ss)
86 {
87     unsigned long sa_handler;
88     long old_eip=eip;
89     struct sigaction * sa = current->sigaction + signr - 1; //current->sigaction[signum-1]。
90     int longs;
91     unsigned long * tmp_esp;
92
93     sa_handler = (unsigned long) sa->sa_handler;
// 如果信号句柄为 SIG_IGN(忽略), 则返回; 如果句柄为 SIG_DFL(默认处理), 则如果信号是
// SIGCHLD 则返回, 否则终止进程的执行
94     if (sa_handler==1)
95         return;
96     if (!sa_handler) {
97         if (signr==SIGCHLD)
98             return;
99         else
100             do_exit(1<<(signr-1)); // 为什么以信号位图为参数? 不为什么!?!? ☹
// 这里应该是 do_exit(1<<(signr))。
101     }
// 如果该信号句柄只需使用一次, 则将该句柄置空(该信号句柄已经保存在 sa_handler 指针中)。
102     if (sa->sa_flags & SA_ONESHOT)
103         sa->sa_handler = NULL;
// 下面这段代码将信号处理句柄插入到用户堆栈中, 同时也将 sa_restorer, signr, 进程屏蔽码(如果
// SA_NOMASK 没置位), eax, ecx, edx 作为参数以及原调用系统调用的程序返回指针及标志寄存器值
// 压入堆栈。因此在本次系统调用中断(0x80)返回用户程序时会首先执行用户的信号句柄程序, 然后
// 再继续执行用户程序。

// 将用户调用系统调用的代码指针 eip 指向该信号处理句柄。
104     *(&eip) = sa_handler;
// 如果允许信号自己的处理句柄收到信号自己, 则也需要将进程的阻塞码压入堆栈。
// 注意, 这里 longs 的结果应该选择 (7*4):(8*4), 因为堆栈是以 4 字节为单位操作的。
105     longs = (sa->sa_flags & SA_NOMASK)?7:8;
// 将原调用程序的用户堆栈指针向下扩展 7 (或 8) 个长字 (用来存放调用信号句柄的参数等),

```

```

// 并检查内存使用情况（例如如果内存超界则分配新页等）。
106      *(&esp) -= longs;
107      verify_area(esp, longs*4);
// 在用户堆栈中从下到上存放 sa_restorer, 信号 signr, 屏蔽码 blocked(如果 SA_NOMASK 置位),
// eax, ecx, edx, eflags 和用户程序原代码指针。
108      tmp_esp=esp;
109      put_fs_long((long) sa->sa_restorer, tmp_esp++);
110      put_fs_long(signr, tmp_esp++);
111      if (!(sa->sa_flags & SA_NOMASK))
112          put_fs_long(current->blocked, tmp_esp++);
113      put_fs_long(eax, tmp_esp++);
114      put_fs_long(ecx, tmp_esp++);
115      put_fs_long(edx, tmp_esp++);
116      put_fs_long(eflags, tmp_esp++);
117      put_fs_long(old_eip, tmp_esp++);
118      current->blocked |= sa->sa_mask; // 进程阻塞码(屏蔽码)添上 sa_mask 中的码位。
119 }
120

```

## 5.9.3 其它信息

### 5.9.3.1 进程信号说明

进程中的信号是用于进程之间通信的一种简单消息，通常是下表中的一个标号数值，并且不携带任何其它的信息。例如当一个子进程终止或结束时，就会产生一个标号为 17 的 SIGCHLD 信号发送给父进程，以通知父进程有关子进程的当前状态。

关于一个进程如何处理收到的信号，一般有两种做法：一是程序的进程不去处理，此时该信号会由系统相应的默认信号处理程序进行处理；第二种做法是进程使用自己的信号处理程序来处理信号。

表 5 - 4 进程信号

标号	名称	说明	默认操作
1	SIGHUP	(Hangup) 当你不再控制终端时内核会产生该信号，或者当你关闭 Xterm 或断开 modem。由于后台程序没有控制的终端，因而它们常用 SIGUP 来发出需要重新读取其配置文件的信号。	(Abort) 挂断控制终端或进程。
2	SIGINT	(Interrupt) 来自键盘的终端。通常终端驱动程序会将其与^C绑定。	(Abort) 终止程序。
3	SIGQUIT	(Quit) 来自键盘的终端。通常终端驱动程序会将其与^绑定。	(Dump) 程序被终止并产生 dump core 文件。
4	SIGILL	(Illegal Instruction) 程序出错或者执行了一个非法的操作指令。	(Dump) 程序被终止并产生 dump core 文件。
5	SIGTRAP	(Breakpoint/Trace Trap) 调试用，跟踪断点。	
6	SIGABRT	(Abort) 放弃执行，异常结束。	(Dump) 程序被终止并产生 dump core 文件。
6	SIGIOT	(IO Trap) 同 SIGABRT	(Dump) 程序被终止并产生 dump core 文件。
7	SIGUNUSED	(Unused) 没有使用。	
8	SIGFPE	(Floating Point Exception) 浮点异常。	(Dump) 程序被终止并产生

			dump core 文件。
9	SIGKILL	(Kill) 程序被终止。该信号不能被捕获或者被忽略。想立刻终止一个进程，就发送信号 9。注意程序将没有任何机会做清理工作。	(Abort) 程序被终止。
10	SIGUSR1	(User defined Signal 1) 用户定义的信号。	(Abort) 进程被终止。
11	SIGSEGV	(Segmentation Violation) 当程序引用无效的内存时会产生此信号。比如：寻址没有映射的内存；寻址未许可的内存。	(Dump) 程序被终止并产生 dump core 文件。
12	SIGUSR2	(User defined Signal 2) 保留给用户程序用于 IPC 或其它目的。	(Abort) 进程被终止。
13	SIGPIPE	(Pipe) 当程序向一个套接字或管道写时由于没有读者而产生该信号。	(Abort) 进程被终止。
14	SIGALRM	(Alarm) 该信号会在用户调用 alarm 系统调用所设置的延迟秒数到后产生。该信号常用判别于系统调用超时。	(Abort) 进程被终止。
15	SIGTERM	(Terminate) 用于和善地要求一个程序终止。它是 kill 的默认信号。与 SIGKILL 不同，该信号能被捕获，这样就能在退出运行前做清理工作。	(Abort) 进程被终止。
16	SIGSTKFLT	(Stack fault on coprocessor) 协处理器堆栈错误。	(Abort) 进程被终止。
17	SIGCHLD	(Child) 父进程发出。停止或终止子进程。可改变其含义挪作它用。	(Ignore) 子进程停止或结束。
18	SIGCONT	(Continue) 该信号致使被 SIGSTOP 停止的进程恢复运行。可以被捕获。	(Continue) 恢复进程的执行。
19	SIGSTOP	(Stop) 停止进程的运行。该信号不可被捕获或忽略。	(Stop) 停止进程运行。
20	SIGTSTP	(Terminal Stop) 向终端发送停止键序列。该信号可以被捕获或忽略。	(Stop) 停止进程运行。
21	SIGTTIN	(TTY Input on Background) 后台进程试图从一个不再被控制的终端上读取数据，此时该进程将被停止，直到收到 SIGCONT 信号。该信号可以被捕获或忽略。	(Stop) 停止进程运行。
22	SIGTTOU	(TTY Output on Background) 后台进程试图向一个不再被控制的终端上输出数据，此时该进程将被停止，直到收到 SIGCONT 信号。该信号可被捕获或忽略。	(Stop) 停止进程运行。

## 5.10 exit.c 程序

### 5.10.1 功能描述

该程序主要描述了进程（任务）终止和退出的处理事宜。主要包含进程释放、会话（进程组）终止和程序退出处理函数以及杀死进程、终止进程、挂起进程等系统调用函数。还包括进程信号发送函数 `send_sig()` 和通知父进程子进程终止的函数 `tell_father()`。

释放进程的函数 `release()` 主要根据指定的任务数据结构（任务描述符）指针，在任务数组中删除指定的进程指针、释放相关内存页并立刻让内核重新调度任务的运行。

进程组终止函数 `kill_session()` 通过向会话号与当前进程相同的进程发送挂断进程的信号。

系统调用 `sys_kill()` 用于向进程发送任何指定的信号。根据参数 `pid`（进程标识号）的数值的不同，

该系统调用会向不同的进程或进程组发送信号。程序注释中已经列出了各种不同情况的处理方式。

程序退出处理函数 `do_exit()` 是在系统调用中断处理程序中被调用。它首先会释放当前进程的代码段和数据段所占的内存页面，然后向子进程发送终止信号 `SIGCHLD`。接着关闭当前进程打开的所有文件、释放使用的终端设备、协处理器设备，若当前进程是进程组的领头进程，则还需要终止所有相关进程。随后把当前进程置为僵死状态，设置退出码，并向其父进程发送子进程终止信号。最后让内核重新调度任务的运行。

系统调用 `waitpid()` 用于挂起当前进程，直到 `pid` 指定的子进程退出（终止）或者收到要求终止该进程的信号，或者是需要调用一个信号句柄（信号处理程序）。如果 `pid` 所指的子进程早已退出（已成所谓的僵死进程），则本调用将立刻返回。子进程使用的所有资源将释放。该函数的具体操作也要根据其参数进行不同的处理。详见代码中的相关注释。

## 5.10.2 代码注释

程序 5-8 linux/kernel/exit.c

```

1  /*
2   * linux/kernel/exit.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  #include <errno.h>           // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)
8  #include <signal.h>         // 信号头文件。定义信号符号常量，信号结构以及信号操作函数原型。
9  #include <sys/wait.h>       // 等待调用头文件。定义系统调用 wait() 和 waitpid() 及相关常数符号。
10
11 #include <linux/sched.h>     // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                               // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
12 #include <linux/kernel.h>   // 内核头文件。含有一些内核常用函数的原形定义。
13 #include <linux/tty.h>      // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
14 #include <asm/segment.h>    // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
15
16 int sys_pause(void);        // 把进程置为睡眠状态的系统调用。(kernel/sched.c, 144 行)
17 int sys_close(int fd);      // 关闭指定文件的系统调用。(fs/open.c, 192 行)
18
19 // 释放指定进程占用的任务槽及其任务数据结构所占用的内存。
20 // 参数 p 是任务数据结构的指针。该函数在后面的 sys_kill() 和 sys_waitpid() 中被调用。
21 // 扫描任务指针数组表 task[] 以寻找指定的任务。如果找到，则首先清空该任务槽，然后释放
22 // 该任务数据结构所占用的内存页面，最后执行调度函数并在返回时立即退出。如果在任务数组
23 // 表中没有找到指定任务对应的项，则内核 panic()。
19 void release(struct task_struct * p)
20 {
21     int i;
22
23     if (!p)                // 如果进程数据结构指针是 NULL，则什么也不做，退出。
24         return;
25     for (i=1 ; i<NR_TASKS ; i++)    // 扫描任务数组，寻找指定任务。
26         if (task[i]==p) {
27             task[i]=NULL;          // 置空该任务项并释放相关内存页。
28             free_page((long)p);
29             schedule();            // 重新调度（似乎没有必要）。
30             return;

```



```

31     }
32     panic("trying to release non-existent task"); // 指定任务若不存在则死机。
33 }
34
35 // 向指定任务(*p)发送信号(sig), 权限为 priv。
36 // 参数: sig -- 信号值;
37 //        p -- 指定任务的指针;
38 //        priv -- 强制发送信号的标志。即不需要考虑进程用户属性或级别而能发送信号的权利。
39 // 该函数首先在判断参数的正确性, 然后判断条件是否满足。如果满足就向指定进程发送信号 sig
40 // 并退出, 否则返回未许可错误号。
41 static inline int send_sig(long sig, struct task_struct * p, int priv)
42 {
43     // 若信号不正确或任务指针为空则出错退出。
44     if (!p || sig<1 || sig>32)
45         return -EINVAL;
46     // 如果强制发送标志置位, 或者当前进程的有效用户标识符(euid)就是指定进程的 euid(也即是自己),
47     // 或者当前进程是超级用户, 则在进程位图中添加该信号, 否则出错退出。
48     // 其中 suser() 定义为(current->euid==0), 用于判断是否是超级用户。
49     if (priv || (current->euid==p->euid) || suser())
50         p->signal |= (1<<(sig-1));
51     else
52         return -EPERM;
53     return 0;
54 }
55
56 // 终止会话(session)。
57 // 进程会话的概念请参见第 4 章中有关进程组和会话的说明。
58 static void kill_session(void)
59 {
60     struct task_struct **p = NR_TASKS + task; // 指针*p 首先指向任务数组最末端。
61
62     // 扫描任务指针数组, 对于所有的任务(除任务 0 以外), 如果其会话号 session 等于当前进程的
63     // 会话号就向它发送挂断进程信号 SIGHUP。
64     while (--p > &FIRST_TASK) {
65         if (*p && (*p)->session == current->session)
66             (*p)->signal |= 1<<(SIGHUP-1); // 发送挂断进程信号。
67     }
68 }
69
70 /*
71  * XXX need to check permissions needed to send signals to process
72  * groups, etc. etc. kill() permissions semantics are tricky!
73  */
74 /*
75  * 为了向进程组等发送信号, XXX 需要检查许可。kill() 的许可机制非常巧妙!
76  */
77 // 系统调用 kill() 可用于向任何进程或进程组发送任何信号, 而并非只是杀死进程。
78 // 参数 pid 是进程号; sig 是需要发送的信号。
79 // 如果 pid 值>0, 则信号被发送给进程号是 pid 的进程。
80 // 如果 pid=0, 那么信号就会被发送给当前进程的进程组中的所有进程。
81 // 如果 pid=-1, 则信号 sig 就会发送给除第一个进程外的所有进程。
82 // 如果 pid < -1, 则信号 sig 将发送给进程组-pid 的所有进程。
83 // 如果信号 sig 为 0, 则不发送信号, 但仍会进行错误检查。如果成功则返回 0。

```



```

// 该函数扫描任务数组表，并根据 pid 的值对满足条件的进程发送指定的信号 sig。若 pid 等于 0，
// 表明当前进程是进程组组长，因此需要向所有组内的进程强制发送信号 sig。
60 int sys_kill(int pid, int sig)
61 {
62     struct task_struct **p = NR_TASKS + task;
63     int err, retval = 0;
64
65     if (!pid) while (--p > &FIRST_TASK) {
66         if (*p && (*p)->pgrp == current->pid)
67             if (err=send_sig(sig, *p, 1))          // 强制发送信号。
68                 retval = err;
69     } else if (pid>0) while (--p > &FIRST_TASK) {
70         if (*p && (*p)->pid == pid)
71             if (err=send_sig(sig, *p, 0))
72                 retval = err;
73     } else if (pid == -1) while (--p > &FIRST_TASK)
74         if (err = send_sig(sig, *p, 0))
75             retval = err;
76     else while (--p > &FIRST_TASK)
77         if (*p && (*p)->pgrp == -pid)
78             if (err = send_sig(sig, *p, 0))
79                 retval = err;
80     return retval;
81 }
82
//// 通知父进程 -- 向进程 pid 发送信号 SIGCHLD：默认情况下子进程将停止或终止。
// 如果没有找到父进程，则自己释放。但根据 POSIX.1 要求，若父进程已先行终止，则子进程应该
// 被初始进程 1 收容。
83 static void tell_father(int pid)
84 {
85     int i;
86
87     if (pid)
88         // 扫描进程数组表寻找指定进程 pid，并向其发送子进程将停止或终止信号 SIGCHLD。
89         for (i=0; i<NR_TASKS; i++) {
90             if (!task[i])
91                 continue;
92             if (task[i]->pid != pid)
93                 continue;
94             task[i]->signal |= (1<<(SIGCHLD-1));
95             return;
96         }
97     /* if we don't find any fathers, we just release ourselves */
98     /* This is not really OK. Must change it to make father 1 */
99     /* 如果没有找到父进程，则进程就自己释放。这样做并不好，必须改成由进程 1 充当其父进程。*/
100     printk("BAD BAD - no father found\n\r");
101     release(current);          // 如果没有找到父进程，则自己释放。
102 }
103
//// 程序退出处理程序。在下面 137 行处的系统调用 sys_exit() 中被调用。
102 int do_exit(long code)      // code 是错误码。
103 {
104     int i;

```

```

105 // 释放当前进程代码段和数据段所占的内存页 (free_page_tables() 在 mm/memory.c, 105 行)。
106     free_page_tables(get_base(current->ldt[1]), get_limit(0x0f));
107     free_page_tables(get_base(current->ldt[2]), get_limit(0x17));
108 // 如果当前进程有子进程, 就将子进程的 father 置为 1(其父进程改为进程 1 - init)。如果该子进程
109 // 已经处于僵死(ZOMBIE)状态, 则向进程 1 发送子进程终止信号 SIGCHLD。
110     for (i=0 ; i<NR_TASKS ; i++)
111         if (task[i] && task[i]->father == current->pid) {
112             task[i]->father = 1;
113             if (task[i]->state == TASK_ZOMBIE)
114                 /* assumption task[1] is always init */ /* 这里假设 task[1] 肯定是进程 init */
115                 (void) send_sig(SIGCHLD, task[1], 1);
116         }
117 // 关闭当前进程打开着的所有文件。
118     for (i=0 ; i<NR_OPEN ; i++)
119         if (current->filp[i])
120             sys_close(i);
121 // 对当前进程的工作目录 pwd、根目录 root 以及程序的 i 节点进行同步操作, 并分别置空(释放)。
122     iput(current->pwd);
123     current->pwd=NULL;
124     iput(current->root);
125     current->root=NULL;
126     iput(current->executable);
127     current->executable=NULL;
128 // 如果当前进程是会话头领(leader)进程并且其有控制终端, 则释放该终端。
129     if (current->leader && current->tty >= 0)
130         tty_table[current->tty].pgrp = 0;
131 // 如果当前进程上次使用过协处理器, 则将 last_task_used_math 置空。
132     if (last_task_used_math == current)
133         last_task_used_math = NULL;
134 // 如果当前进程是 leader 进程, 则终止该会话的所有相关进程。
135     if (current->leader)
136         kill_session();
137 // 把当前进程置为僵死状态, 表明当前进程已经释放了资源。并保存将由父进程读取的退出码。
138     current->state = TASK_ZOMBIE;
139     current->exit_code = code;
140 // 通知父进程, 也即向父进程发送信号 SIGCHLD -- 子进程将停止或终止。
141     tell_father(current->father);
142     schedule(); // 重新调度进程运行, 以让父进程处理僵死进程其它的善后事宜。
143     return (-1); /* just to suppress warnings */
144 }
145
146 // 系统调用 exit()。终止进程。
147 int sys_exit(int error_code)
148 {
149     return do_exit((error_code&0xff)<<8);
150 }
151
152 // 系统调用 waitpid()。挂起当前进程, 直到 pid 指定的子进程退出(终止)或者收到要求终止
153 // 该进程的信号, 或者是需要调用一个信号句柄(信号处理程序)。如果 pid 所指的子进程早已
154 // 退出(已成所谓的僵死进程), 则本调用将立刻返回。子进程使用的所有资源将释放。
155 // 如果 pid > 0, 表示等待进程号等于 pid 的子进程。
156 // 如果 pid = 0, 表示等待进程组号等于当前进程组号的任何子进程。

```

```

// 如果 pid < -1, 表示等待进程组号等于 pid 绝对值的任何子进程。
// [ 如果 pid = -1, 表示等待任何子进程。]
// 若 options = WUNTRACED, 表示如果子进程是停止的, 也马上返回 (无须跟踪)。
// 若 options = WNOHANG, 表示如果没有子进程退出或终止就马上返回。
// 如果返回状态指针 stat_addr 不为空, 则就将状态信息保存到那里。
142 int sys_waitpid(pid_t pid, unsigned long * stat_addr, int options)
143 {
144     int flag, code;
145     struct task_struct ** p;
146
147     verify_area(stat_addr, 4);
148 repeat:
149     flag=0;
    // 从任务数组末端开始扫描所有任务, 跳过空项、本进程项以及非当前进程的子进程项。
150     for(p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
151         if (!*p || *p == current)           // 跳过空项和本进程项。
152             continue;
153         if ((*p)->father != current->pid)    // 如果不是当前进程的子进程则跳过。
154             continue;
    // 此时扫描选择到的进程 p 肯定是当前进程的子进程。
    // 如果等待的子进程号 pid>0, 但与被扫描子进程 p 的 pid 不相等, 说明它是当前进程另外的子进程,
    // 于是跳过该进程, 接着扫描下一个进程。
155         if (pid>0) {
156             if ((*p)->pid != pid)
157                 continue;
    // 否则, 如果指定等待进程的 pid=0, 表示正在等待进程组号等于当前进程组号的任何子进程。如果
    // 此时被扫描进程 p 的进程组号与当前进程的组号不等, 则跳过。
158         } else if (!pid) {
159             if ((*p)->pgrp != current->pgrp)
160                 continue;
    // 否则, 如果指定的 pid<-1, 表示正在等待进程组号等于 pid 绝对值的任何子进程。如果此时被扫描
    // 进程 p 的组号与 pid 的绝对值不等, 则跳过。
161         } else if (pid != -1) {
162             if ((*p)->pgrp != -pid)
163                 continue;
164         }
    // 如果前 3 个对 pid 的判断都不符合, 则表示当前进程正在等待其任何子进程, 也即 pid=-1 的情况。
    // 此时所选择到的进程 p 正是所等待的子进程。接下来根据这个子进程 p 所处的状态来处理。
165     switch ((*p)->state) {
    // 子进程 p 处于停止状态时, 如果此时 WUNTRACED 标志没有置位, 表示程序无须立刻返回, 于是继续
    // 扫描处理其它进程。如果 WUNTRACED 置位, 则把状态信息 0x7f 放入 *stat_addr, 并立刻返回子进程
    // 号 pid。这里 0x7f 表示的返回状态使 WIFSTOPPED() 宏为真。参见 include/sys/wait.h, 14 行。
166     case TASK_STOPPED:
167         if (!(options & WUNTRACED))
168             continue;
169         put_fs_long(0x7f, stat_addr);
170         return (*p)->pid;
    // 如果子进程 p 处于僵死状态, 则首先把它在用户态和内核态运行的时间分别累计到当前进程(父进程)
    // 中, 然后取出子进程的 pid 和退出码, 并释放该子进程。最后返回子进程的退出码和 pid。
171     case TASK_ZOMBIE:
172         current->cutime += (*p)->utime;
173         current->cstime += (*p)->stime;
174         flag = (*p)->pid;           // 临时保存子进程 pid。

```

```

175         code = (*p)->exit_code;           // 取子进程的退出码。
176         release(*p);                         // 释放该子进程。
177         put\_fs\_long(code, stat_addr);       // 置状态信息为退出码值。
178         return flag;                       // 退出，返回子进程的 pid.
// 如果这个子进程 p 的状态既不是停止也不是僵死，那么就置 flag=1。表示找到过一个符合要求的
// 子进程，但是它处于运行态或睡眠态。
179         default:
180             flag=1;
181             continue;
182     }
183 }
// 在上面对任务数组扫描结束后，如果 flag 被置位，说明有符合等待要求的子进程并没有处于退出
// 或僵死状态。如果此时已设置 WNOHANG 选项（表示若没有子进程处于退出或终止态就立刻返回），
// 就立刻返回 0，退出。否则把当前进程置为可中断等待状态并重新执行调度。
184     if (flag) {
185         if (options & WNOHANG)             // 若 options = WNOHANG，则立刻返回。
186             return 0;
187         current->state=TASK\_INTERRUPTIBLE; // 置当前进程为可中断等待状态。
188         schedule();                     // 重新调度。
// 当又开始执行本进程时，如果本进程没有收到除 SIGCHLD 以外的信号，则还是重复处理。否则，
// 返回出错码并退出。
189         if (!(current->signal &= ~(1<<(SIGCHLD-1))))
190             goto repeat;
191         else
192             return -EINTR;             // 退出，返回出错码。
193     }
// 若没有找到符合要求的子进程，则返回出错码。
194     return -ECHILD;
195 }
196
197
198

```

## 5.11 fork.c 程序

### 5.11.1 功能描述

`fork()` 系统调用用于创建子进程。Linux 中所有进程都是进程 0 (任务 0) 的子进程。该程序是 `sys_fork()` (在 `kernel/system_call.s` 中定义) 系统调用的辅助处理函数集，给出了 `sys_fork()` 系统调用中使用的两个 C 语言函数：`find_empty_process()` 和 `copy_process()`。还包括进程内存区域验证与内存分配函数 `verify_area()`。

`copy_process()` 用于创建并复制进程的代码段和数据段以及环境。在进程复制过程中，主要牵涉到进程数据结构中信息的设置。系统首先为新建进程在主内存区中申请一页内存来存放其任务数据结构信息，并复制当前进程任务数据结构中的所有内容作为新进程任务数据结构的模板。

随后对复制的任务数据结构进行修改。把当前进程设置为新进程的父进程，清除信号位图并复位新进程各统计值。接着根据当前进程设置任务状态段 (TSS) 中各寄存器的值。由于创建进程时新进程返回值应为 0，所以需要设置 `tss.eax = 0`。新建进程内核态堆栈指针 `tss.esp0` 被设置成新进程任务数据结构所在内存页面的顶端，而堆栈段 `tss.ss0` 被设置成内核数据段选择符。`tss.ldt` 被设置为局部表描述符在 GDT

中的索引值。如果当前进程使用了协处理器，把还需要把协处理器的完整状态保存到新进程的 `tss.i387` 结构中。

此后系统设置新任务的代码和数据段基址、限长并复制当前进程内存分页管理的页表。如果父进程中有文件是打开的，则将对对应文件的打开次数增 1。接着在 GDT 中设置新任务的 TSS 和 LDT 描述符项，其中基址信息指向新进程任务结构中的 tss 和 ldt。最后再将新任务设置成可运行状态并返回新进程号。

图 5-10 是内存验证函数 `verify_area()` 中验证内存的起始位置和范围的调整示意图。因为内存写验证函数 `write_verify()` 需要以内存页面为单位（4096 字节）进行操作，因此在调用 `write_verify()` 之前，需要把验证的起始位置调整为页面起始位置，同时对验证范围作相应调整。

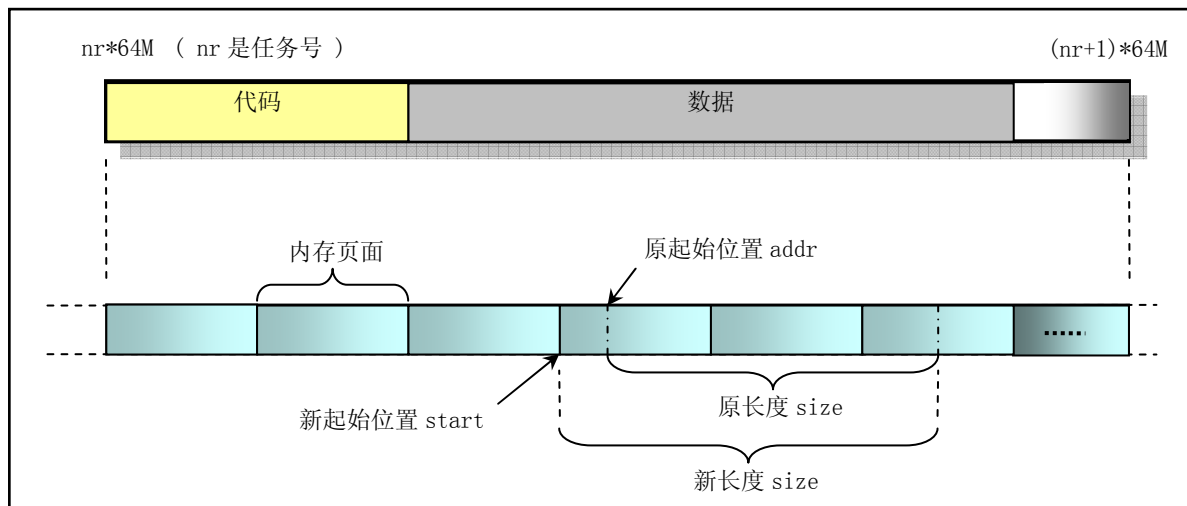


图 5-10 内存验证范围和起始位置的调整

## 5.11.2 代码注释

程序 5-9 linux/kernel/fork.c

```

1  /*
2  *  linux/kernel/fork.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *  'fork.c' contains the help-routines for the 'fork' system call
9  *  (see also system_call.s), and some misc functions ('verify_area').
10 *  Fork is rather simple, once you get the hang of it, but the memory
11 *  management can be a bitch. See 'mm/mm.c': 'copy_page_tables()'
12 */
13 /*
14 *  'fork.c' 中含有系统调用'fork'的辅助子程序（参见 system_call.s），以及一些其它函数
15 *  （'verify_area'）。一旦你了解了 fork，就会发现它是非常简单的，但内存管理却有些难度。
16 *  参见' mm/mm.c' 中的' copy_page_tables()'。
17 */
18 #include <errno.h>          // 错误号头文件。包含系统中各种出错号。（Linus 从 minix 中引进的）。
19
20 #include <linux/sched.h>    // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，

```

```

// 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
16 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
17 #include <asm/segment.h> // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
18 #include <asm/system.h> // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
19
20 extern void write_verify(unsigned long address);
21
22 long last_pid=0; // 最新进程号，其值由 get_empty_process() 生成。
23
//// 进程空间区域写前验证函数。
// 对当前进程地址从 addr 到 addr + size 这一段空间以页为单位执行写操作前的检测操作。
// 由于检测判断是以页面为单位进行操作，因此程序首先需要找出 addr 所在页面开始地址 start，
// 然后 start 加上进程数据段基址，使这个 start 变换成 CPU 4G 线性空间中的地址。最后循环
// 调用 write_verify() 对指定大小的内存空间进行写前验证。若页面是只读的，则执行共享检验
// 和复制页面操作（写时复制）。
24 void verify_area(void * addr,int size)
25 {
26     unsigned long start;
27
28     start = (unsigned long) addr;
// 将起始地址 start 调整为其所在页的左边界开始位置，同时相应地调整验证区域大小。
// 下句中的 start & 0xfff 用来获得指定起始位置 addr（也即 start）在所在页面中的偏移值，
// 原验证范围 size 加上这个偏移值即扩展成以 addr 所在页面起始位置开始的范围值。因此在 30 行
// 上也需要把验证开始位置 start 调整成页面边界值。参见前面的图“内存验证范围的调整”。
29     size += start & 0xfff;
30     start &= 0xfffff000;
// 下面把 start 加上进程数据段在线性地址空间中的起始基址，变成系统整个线性空间中的地址位置。
// 对于 0.11 内核，其数据段和代码段在线性地址空间中的基址和限长均相同。
31     start += get_base(current->ldt[2]);
32     while (size>0) {
33         size -= 4096;
// 写页面验证。若页面不可写，则复制页面。（mm/memory.c，261 行）
34         write_verify(start);
35         start += 4096;
36     }
37 }
38
// 设置新任务的代码和数据段基址、限长并复制页表。
// nr 为新任务号；p 是新任务数据结构的指针。
39 int copy_mem(int nr,struct task_struct * p)
40 {
41     unsigned long old_data_base,new_data_base,data_limit;
42     unsigned long old_code_base,new_code_base,code_limit;
43
// 取当前进程局部描述符表中描述符项的段限长（字节数）。
44     code_limit=get_limit(0x0f); // 取局部描述符表中代码段描述符项中段限长。
45     data_limit=get_limit(0x17); // 取局部描述符表中数据段描述符项中段限长。
// 取当前进程代码段和数据段在线性地址空间中的基地址。
46     old_code_base = get_base(current->ldt[1]); // 取原代码段基址。
47     old_data_base = get_base(current->ldt[2]); // 取原数据段基址。
48     if (old_data_base != old_code_base) // 0.11 版不支持代码和数据段分立的情况。
49         panic("We don't support separate I&D");
50     if (data_limit < code_limit) // 如果数据段长度 < 代码段长度也不对。

```



```

51         panic("Bad data_limit");
// 创建中新进程在线性地址空间中的基址等于 64MB * 其任务号。
52         new_data_base = new_code_base = nr * 0x4000000; // 新基址=任务号*64Mb(任务大小)。
53         p->start_code = new_code_base;
// 设置新进程局部描述符表中段描述符中的基址。
54         set_base(p->ldt[1], new_code_base); // 设置代码段描述符中基址域。
55         set_base(p->ldt[2], new_data_base); // 设置数据段描述符中基址域。
// 设置新进程的页目录表项和页表项。即把新进程的线性地址内存页对应到实际物理地址内存页面上。
56         if (copy_page_tables(old_data_base, new_data_base, data_limit)) { // 复制代码和数据段。
57             free_page_tables(new_data_base, data_limit); // 如果出错则释放申请的内存。
58             return -ENOMEM;
59         }
60         return 0;
61     }
62
63 /*
64  * Ok, this is the main fork-routine. It copies the system process
65  * information (task[nr]) and sets up the necessary registers. It
66  * also copies the data segment in it's entirety.
67  */
/*
* OK, 下面是主要的 fork 子程序。它复制系统进程信息(task[n])并且设置必要的寄存器。
* 它还整个地复制数据段。
*/
// 复制进程。
// 其中参数 nr 是调用 find_empty_process() 分配的任务数组项号。none 是 system_call.s 中调用
// sys_call_table 时压入堆栈的返回地址。
68 int copy_process(int nr, long ebp, long edi, long esi, long gs, long none,
69                 long ebx, long ecx, long edx,
70                 long fs, long es, long ds,
71                 long eip, long cs, long eflags, long esp, long ss)
72 {
73     struct task_struct *p;
74     int i;
75     struct file *f;
76
77     p = (struct task_struct *) get_free_page(); // 为新任务数据结构分配内存。
78     if (!p) // 如果内存分配出错, 则返回出错码并退出。
79         return -EAGAIN;
80     task[nr] = p; // 将新任务结构指针放入任务数组中。
// 其中 nr 为任务号, 由前面 find_empty_process() 返回。
81     *p = *current; /* NOTE! this doesn't copy the supervisor stack */
/* 注意! 这样做不会复制超级用户的堆栈 */ (只复制当前进程内容)。
82     p->state = TASK_UNINTERRUPTIBLE; // 将新进程的状态先置为不可中断等待状态。
83     p->pid = last_pid; // 新进程号。由前面调用 find_empty_process() 得到。
84     p->father = current->pid; // 设置父进程号。
85     p->counter = p->priority;
86     p->signal = 0; // 信号位图置 0。
87     p->alarm = 0; // 报警定时值(滴答数)。
88     p->leader = 0; /* process leadership doesn't inherit */
/* 进程的领导力是不能继承的 */
89     p->utime = p->stime = 0; // 初始化用户态时间和核心态时间。
90     p->cutime = p->cstime = 0; // 初始化子进程用户态和核心态时间。

```



```

91     p->start_time = jiffies;    // 当前滴答数时间。
// 以下设置任务状态段 TSS 所需的数据（参见列表后说明）。
92     p->tss.back_link = 0;
// 由于是给任务结构 p 分配了 1 页新内存，所以此时 esp0 正好指向该页顶端。ss0:esp0 用于作为程序
// 在内核态执行时的堆栈。
93     p->tss.esp0 = PAGE_SIZE + (long) p; // 内核态堆栈指针。
94     p->tss.ss0 = 0x10;                // 堆栈段选择符（与内核数据段相同）。
95     p->tss.eip = eip;                  // 指令代码指针。
96     p->tss.eflags = eflags;            // 标志寄存器。
97     p->tss.eax = 0;                    // 这是当 fork() 返回时，新进程会返回 0 的原因所在。
98     p->tss.ecx = ecx;
99     p->tss.edx = edx;
100    p->tss.ebx = ebx;
101    p->tss.esp = esp;                  // 新进程完全复制了父进程的堆栈内容。因此要求 task0
102    p->tss.ebp = ebp;                  // 的堆栈比较“干净”。
103    p->tss.esi = esi;
104    p->tss.edi = edi;
105    p->tss.es = es & 0xffff;           // 段寄存器仅 16 位有效。
106    p->tss.cs = cs & 0xffff;
107    p->tss.ss = ss & 0xffff;
108    p->tss.ds = ds & 0xffff;
109    p->tss.fs = fs & 0xffff;
110    p->tss.gs = gs & 0xffff;
111    p->tss.ldt = LDT(nr); // 设置新任务的局部描述符表的选择符（LDT 描述符在 GDT 中）。
112    p->tss.trace_bitmap = 0x80000000; //（高 16 位有效）。
// 如果当前任务使用了协处理器，就保存其上下文。汇编指令 clts 用于清除控制寄存器 CR0 中的任务
// 已交换（TS）标志。每当发生任务切换，CPU 都会设置该标志。该标志用于管理协处理器：如果
// 该标志置位，那么每个 ESC 指令都会被捕获。如果协处理器存在标志也同时置位的话那么就会捕获
// WAIT 指令。因此，如果任务切换发生在一个 ESC 指令开始执行之后，则协处理器中的内容就可能需
// 要在执行新的 ESC 指令之前保存起来。错误处理句柄会保存协处理器的内容并复位 TS 标志。
// 指令 fnsave 用于把协处理器的所有状态保存到目的操作数指定的内存区域中（tss.i387）。
113    if (last_task_used_math == current)
114        __asm__ ("clts ; fnsave %0"::"m" (p->tss.i387));
// 设置新任务的代码和数据段基址、限长并复制页表。如果出错（返回值不是 0），则复位任务数组中
// 相应项并释放为该新任务分配的内存页。
115    if (copy_mem(nr,p)) {                // 返回不为 0 表示出错。
116        task[nr] = NULL;
117        free_page((long) p);
118        return -EAGAIN;
119    }
// 如果父进程中有文件是打开的，则将对对应文件的打开次数增 1。
120    for (i=0; i<NR_OPEN;i++)
121        if (f=p->filp[i])
122            f->f_count++;
// 将当前进程（父进程）的 pwd, root 和 executable 引用次数均增 1。
123    if (current->pwd)
124        current->pwd->i_count++;
125    if (current->root)
126        current->root->i_count++;
127    if (current->executable)
128        current->executable->i_count++;
// 在 GDT 中设置新任务的 TSS 和 LDT 描述符项，数据从 task 结构中取。
// 在任务切换时，任务寄存器 tr 由 CPU 自动加载。

```

```

129     set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY, &(p->tss));
130     set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY, &(p->ldt));
131     p->state = TASK_RUNNING;          /* do this last, just in case */
                                      /* 最后再将新任务设置成可运行状态，以防万一 */
132     return last_pid;                  // 返回新进程号（与任务号是不同的）。
133 }
134
// 为新进程取得不重复的进程号 last_pid，并返回在任务数组中的任务号(数组 index)。
135 int find_empty_process(void)
136 {
137     int i;
138
139     repeat:
// 如果 last_pid 增 1 后超出其正数表示范围，则重新从 1 开始使用 pid 号。
140         if ((++last_pid)<0) last_pid=1;
// 在任务数组中搜索刚设置的 pid 号是否已经被任何任务使用。如果是则重新获得一个 pid 号。
141         for(i=0 ; i<NR_TASKS ; i++)
142             if (task[i] && task[i]->pid == last_pid) goto repeat;
// 在任务数组中为新任务寻找一个空闲项，并返回项号。last_pid 是一个全局变量，不用返回。
143         for(i=1 ; i<NR_TASKS ; i++)    // 任务 0 排除在外。
144             if (!task[i])
145                 return i;
// 如果任务数组中 64 个项已经被全部占用，则返回出错码。
146         return -EAGAIN;
147 }
148

```

## 5.11.3 其它信息

### 5.11.3.1 任务状态段（TSS）信息

下面图 5-11 是任务状态段 TSS（Task State Segment）的内容。对它的说明请参见附录。

31	23	15	7	0	
I/O 映射图基地址(MAP BASE)		0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			64
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		局部描述符表(LDT)的选择符			60
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		GS			5C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		FS			58
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		DS			54
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		SS			50
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		CS			4C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		ES			48
EDI					44
ESI					40
EBP					3C
ESP					38
EBX					34
EDX					30

ECX		2C
EAX		28
EFLAGS		24
指令指针(EIP)		20
页目录基地址寄存器 CR3 (PDBR)		1C
0000000000000000	SS2	18
ESP2		14
0000000000000000	SS1	10
ESP1		0C
0000000000000000	SS0	08
ESP0		04
0000000000000000	前一执行任务 TSS 的描述符	00

图 5-11 任务状态段 TSS 中的信息。

CPU 管理任务需要的所有信息被存储于一个特殊类型的段中，任务状态段(task state segment - TSS)。图中显示出执行 80386 任务的 TSS 格式。

TSS 中的字段可以分为两类：

1. CPU 在进行任务切换时更新的动态信息集。这些字段有：

- o 通用寄存器 (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI);
- o 段寄存器 (ES, CS, SS, DS, FS, GS);
- o 标志寄存器 (EFLAGS);
- o 指令指针 (EIP);
- o 前一个执行任务的 TSS 的选择符 (仅当返回时才更新)。

2. CPU 读取但不会更改的静态信息集。这些字段有：

- o 任务的 LDT 的选择符;
- o 含有任务页目录基地址的寄存器 (PDBR);
- o 特权级 0-2 的堆栈指针;
- o 当任务进行切换时导致 CPU 产生一个调试(debug)异常的 T-比特位 (调试跟踪位);
- o I/O 比特位图基地址 (其长度上限就是 TSS 的长度上限, 在 TSS 描述符中说明)。

任务状态段可以存放在线性空间的任何地方。与其它各类段相似，任务状态段也是由描述符来定义的。当前正在执行任务的 TSS 是由任务寄存器 (TR) 来指示的。指令 LTR 和 STR 用来修改和读取任务寄存器中的选择符 (任务寄存器的可见部分)。

I/O 比特位图中的每 1 比特对应 1 个 I/O 端口。比如端口 41 的比特位就是 I/O 位图基地址+5，位偏移 1 处。在保护模式中，当遇到 1 个 I/O 指令时(IN, INS, OUT, OUTS)，CPU 首先就会检查当前特权级是否小于标志寄存器的 IOPL，如果这个条件满足，就执行该 I/O 操作。如果不满足，那么 CPU 就会检查 TSS 中的 I/O 比特位图。如果相应比特位是置位的，就会产生一般保护性异常，否则就会执行该 I/O 操作。

在 Linux 0.11 中，图中 SS0:ESP0 用于存放任务在内核态运行时的堆栈指针。SS1:ESP1 和 SS2:ESP2 分别对应运行于特权级 1 和 2 时使用的堆栈指针，这两个特权级在 Linux 中没有使用。而任务工作于用户态时堆栈指针则保存在 SS:ESP 寄存器中。由上所述可知，每当任务进入内核态执行时，其内核态堆栈指针初始位置不变，均为任务数据结构所在页面的顶端位置处。

## 5.12 sys.c 程序

### 5.12.1 功能描述

sys.c 程序主要包含有很多系统调用功能的实现函数。其中，若返回值为-ENOSYS，则表示本版的 Linux 还没有实现该功能，可以参考目前的代码来了解它们的实现方法。所有系统调用的功能说明请参见头文件 include/linux/sys.h。

该程序中含有很多有关进程 ID、进程组 ID、用户 ID、用户组 ID、实际用户 ID、有效用户 ID 以及会话 ID (session) 等的操作函数。下面首先对这些 ID 作一说明。

一个用户有用户 ID (uid) 和用户组 ID (gid)。这两个 ID 是 passwd 文件中对该用户设置的 ID，通常被称为实际用户 ID (ruid) 和实际组 ID (rgid)。而在每个文件的 i 节点信息中都保存着宿主的用户 ID 和组 ID，它们指明了文件拥有者和所属用户组。主要用于访问或执行文件时的权限判别操作。另外，在一个进程的任务数据结构中，为了实现不同功能而保存了 3 种用户 ID 和组 ID。见表 5-5 所示。

表 5-5 与进程相关的用户 ID 和组 ID

类别	用户 ID	组 ID
进程的	uid - 用户 ID。指明拥有该进程的用户。	gid - 组 ID。指明拥有该进程的用户组。
有效的	eid - 有效用户 ID。指明访问文件的权限。	egid - 有效组 ID。指明访问文件的权限。
保存的	suid - 保存的用户 ID。当执行文件的设置用户 ID 标志 (set-user-ID) 置位时，suid 中保存着执行文件的 uid。否则 suid 等于进程的 eid。	sgid - 保存的组 ID。当执行文件的设置组 ID 标志 (set-group-ID) 置位时，sgid 中保存着执行文件的 gid。否则 sgid 等于进程的 egid。

进程的 uid 和 gid 分别就是进程拥有者的用户 ID 和组 ID，也即实际用户 ID(ruid)和实际组 ID(rgid)。超级用户可以使用 set\_uid()和 set\_gid()对它们进行修改。有效用户 ID 和有效组 ID 用于进程访问文件时的许可权判断。

保存的用户 ID (suid) 和保存的组 ID (sgid) 用于进程访问设置了 set-user-ID 或 set-group-ID 标志的文件。当执行一个程序时，进程的 eid 通常就是实际用户 ID，egid 通常就是实际组 ID。因此进程只能访问进程的有效用户、有效用户组规定的文件或其它允许访问的文件。但是如果一个文件的 set-user-ID 标志置位时，那么进程的有效用户 ID 就会被设置成该文件宿主的用户 ID，因此进程就可以访问设置了这种标志的受限文件，同时该文件宿主的用户 ID 被保存在 suid 中。同理，文件的 set-group-ID 标志也有类似的作用并作相同的处理。

例如，如果一个程序的宿主是超级用户，但该程序设置了 set-user-ID 标志，那么当该程序被一个进程运行时，则该进程的有效用户 ID (eid) 就会被设置成超级用户的 ID (0)。于是这个进程就拥有了超级用户的权限。一个实际例子就是 Linux 系统的 passwd 命令。该命令是一个设置了 set-user-Id 的程序，因此允许用户修改自己的口令。因为该程序需要把用户的新口令写入/etc/passwd 文件中，而该文件只有超级用户才有写权限，因此 passwd 程序就需要使用 set-user-ID 标志。

另外，进程也有标识自己属性的进程 ID (pid)、所属进程组的进程组 ID (pgrp 或 pgid) 和所属会话的会话 ID (session)。这 3 个 ID 用于表明进程与进程之间的关系，与用户 ID 和组 ID 无关。

### 5.12.2 代码注释

程序 5-10 linux/kernel/sys.c 程序

1 /\*

```

2  * linux/kernel/sys.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 #include <errno.h>          // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
8
9 #include <linux/sched.h>    // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
10                             // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
11 #include <linux/tty.h>      // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
12 #include <linux/kernel.h>   // 内核头文件。含有一些内核常用函数的原形定义。
13 #include <asm/segment.h>    // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
14 #include <sys/times.h>      // 定义了进程中运行时间的结构 tms 以及 times() 函数原型。
15 #include <sys/utsname.h>    // 系统名称结构头文件。
16
17 // 返回日期和时间。
18 int sys_ftime()
19 {
20     return -ENOSYS;
21 }
22
23 //
24 int sys_break()
25 {
26     return -ENOSYS;
27 }
28
29 // 用于当前进程对子进程进行调试(debugging)。
30 int sys_ptrace()
31 {
32     return -ENOSYS;
33 }
34
35 // 改变并打印终端行设置。
36 int sys_stty()
37 {
38     return -ENOSYS;
39 }
40
41 // 取终端行设置信息。
42 int sys_gtty()
43 {
44     return -ENOSYS;
45 }
46
47 // 修改文件名。
48 int sys_rename()
49 {
50     return -ENOSYS;
51 }
52
53 //
54 int sys_prof()

```

```

47 {
48     return -ENOSYS;
49 }
50
// 设置当前任务的实际以及/或者有效组 ID (gid)。如果任务没有超级用户特权，
// 那么只能互换其实际组 ID 和有效组 ID。如果任务具有超级用户特权，就能任意设置有效的和实际
// 的组 ID。保留的 gid (saved gid) 被设置成与有效 gid 同值。
51 int sys_setregid(int rgid, int egid)
52 {
53     if (rgid>0) {
54         if ((current->gid == rgid) ||
55             suser())
56             current->gid = rgid;
57         else
58             return(-EPERM);
59     }
60     if (egid>0) {
61         if ((current->gid == egid) ||
62             (current->egid == egid) ||
63             (current->sgid == egid) ||
64             suser())
65             current->egid = egid;
66         else
67             return(-EPERM);
68     }
69     return 0;
70 }
71
// 设置进程组号(gid)。如果任务没有超级用户特权，它可以使用 setgid()将其有效 gid
// (effective gid) 设置为成其保留 gid(saved gid)或其实际 gid(real gid)。如果任务有
// 超级用户特权，则实际 gid、有效 gid 和保留 gid 都被设置成参数指定的 gid。
72 int sys_setgid(int gid)
73 {
74     return(sys_setregid(gid, gid));
75 }
76
// 打开或关闭进程计帐功能。
77 int sys_acct()
78 {
79     return -ENOSYS;
80 }
81
// 映射任意物理内存到进程的虚拟地址空间。
82 int sys_phys()
83 {
84     return -ENOSYS;
85 }
86
87 int sys_lock()
88 {
89     return -ENOSYS;
90 }
91

```

```

92 int sys_mpx()
93 {
94     return -ENOSYS;
95 }
96
97 int sys_ulimit()
98 {
99     return -ENOSYS;
100 }
101
// 返回从 1970 年 1 月 1 日 00:00:00 GMT 开始计时的时间值（秒）。如果 tloc 不为 null，则时间值
// 也存储在那里。
102 int sys_time(long * tloc)
103 {
104     int i;
105
106     i = CURRENT_TIME;
107     if (tloc) {
108         verify_area(tloc, 4);    // 验证内存容量是否够（这里是 4 字节）。
109         put_fs_long(i, (unsigned long *)tloc);    // 也放入用户数据段 tloc 处。
110     }
111     return i;
112 }
113
114 /*
115  * Unprivileged users may change the real user id to the effective uid
116  * or vice versa.
117  */
118 /*
119  * 无特权的用户可以见实际用户标识符(real uid)改成有效用户标识符(effective uid)，反之亦然。
120  */
121 // 设置任务的实际以及/或者有效用户 ID (uid)。如果任务没有超级用户特权，那么只能互换其
122 // 实际用户 ID 和有效用户 ID。如果任务具有超级用户特权，就能任意设置有效的和实际的用户 ID。
123 // 保留的 uid (saved uid) 被设置成与有效 uid 同值。
124 int sys_setreuid(int ruid, int euid)
125 {
126     int old_ruid = current->ruid;
127
128     if (ruid > 0) {
129         if ((current->euid == ruid) ||
130             (old_ruid == ruid) ||
131             suser())
132             current->ruid = ruid;
133         else
134             return(-EPERM);
135     }
136     if (euid > 0) {
137         if ((old_ruid == euid) ||
138             (current->euid == euid) ||
139             suser())
140             current->euid = euid;
141         else {
142             current->euid = old_ruid;
143         }
144     }
145 }

```



```

137         return(-EPERM);
138     }
139 }
140 return 0;
141 }
142
143 // 设置任务用户号(uid)。如果任务没有超级用户特权，它可以使用 setuid() 将其有效 uid
144 // (effective uid) 设置成其保留 uid(saved uid) 或其实际 uid(real uid)。如果任务有
145 // 超级用户特权，则实际 uid、有效 uid 和保留 uid 都被设置成参数指定的 uid。
146 int sys_setuid(int uid)
147 {
148     return(sys_setreuid(uid, uid));
149 }
150
151 // 设置系统时间和日期。参数 tptr 是从 1970 年 1 月 1 日 00:00:00 GMT 开始计时的时间值（秒）。
152 // 调用进程必须具有超级用户权限。
153 int sys_stime(long * tptr)
154 {
155     if (!suser()) // 如果不是超级用户则出错返回（许可）。
156         return -EPERM;
157     startup_time = get_fs_long((unsigned long *)tptr) - jiffies/HZ;
158     return 0;
159 }
160
161 // 获取当前任务时间。tms 结构中包括用户时间、系统时间、子进程用户时间、子进程系统时间。
162 int sys_times(struct tms * tbuf)
163 {
164     if (tbuf) {
165         verify_area(tbuf, sizeof *tbuf);
166         put_fs_long(current->utime, (unsigned long *)&tbuf->tms_utime);
167         put_fs_long(current->stime, (unsigned long *)&tbuf->tms_stime);
168         put_fs_long(current->cutime, (unsigned long *)&tbuf->tms_cutime);
169         put_fs_long(current->cstime, (unsigned long *)&tbuf->tms_cstime);
170     }
171     return jiffies;
172 }
173
174 // 当参数 end_data_seg 数值合理，并且系统确实有足够的内存，而且进程没有超越其最大数据段大小
175 // 时，该函数设置数据段末尾为 end_data_seg 指定的值。该值必须大于代码结尾并且要小于堆栈
176 // 结尾 16KB。返回值是数据段的新结尾值（如果返回值与要求值不同，则表明有错发生）。
177 // 该函数并不被用户直接调用，而由 libc 库函数进行包装，并且返回值也不一样。
178 int sys_brk(unsigned long end_data_seg)
179 {
180     if (end_data_seg >= current->end_code && // 如果参数>代码结尾，并且
181         end_data_seg < current->start_stack - 16384) // 小于堆栈-16KB，
182         current->brk = end_data_seg; // 则设置新数据段结尾值。
183     return current->brk; // 返回进程当前的数据段结尾值。
184 }
185
186 /*
187  * This needs some heave checking ...
188  * I just haven't get the stomach for it. I also don't fully
189  * understand sessions/pgrp etc. Let somebody who does explain it.

```

```

180 */
181 /*
182  * 下面代码需要某些严格的检查...
183  * 我只是没有胃口来做这些。我也不完全明白 sessions/pgrp 等。还是让了解它们的人来做吧。
184  */
185 // 设置进程的进程组 ID 为 pgid。
186 // 如果参数 pid=0，则使用当前进程号。如果 pgid 为 0，则使用参数 pid 指定的进程的组 ID 作为
187 // pgid。如果该函数用于将进程从一个进程组移到另一个进程组，则这两个进程组必须属于同一个
188 // 会话(session)。在这种情况下，参数 pgid 指定了要加入的现有进程组 ID，此时该组的会话 ID
189 // 必须与将要加入进程的相同(193 行)。
190 int sys_setpgid(int pid, int pgid)
191 {
192     int i;
193
194     if (!pid) // 如果参数 pid=0，则使用当前进程号。
195         pid = current->pid;
196     if (!pgid) // 如果 pgid 为 0，则使用当前进程 pid 作为 pgid。
197         pgid = current->pid; // [??这里与 POSIX 的描述有出入]
198     for (i=0 ; i<NR_TASKS ; i++) // 扫描任务数组，查找指定进程号的任务。
199         if (task[i] && task[i]->pid==pid) {
200             if (task[i]->leader) // 如果该任务已经是首领，则出错返回。
201                 return -EPERM;
202             if (task[i]->session != current->session) // 如果该任务的会话 ID
203                 return -EPERM; // 与当前进程的不同，则出错返回。
204             task[i]->pgrp = pgid; // 设置该任务的 pgrp。
205             return 0;
206         }
207     return -ESRCH;
208 }
209
210 // 返回当前进程的组号。与 getpgid(0)等同。
211 int sys_getpgrp(void)
212 {
213     return current->pgrp;
214 }
215
216 // 创建一个会话(session) (即设置其 leader=1)，并且设置其会话号=其组号=其进程号。
217 // setsid -- SET Session ID.
218 int sys_setsid(void)
219 {
220     if (current->leader && !suser()) // 如果当前进程已是会话首领并且不是超级用户
221         return -EPERM; // 则出错返回。
222     current->leader = 1; // 设置当前进程为新会话首领。
223     current->session = current->pgrp = current->pid; // 设置本进程 session = pid。
224     current->tty = -1; // 表示当前进程没有控制终端。
225     return current->pgrp; // 返回会话 ID。
226 }
227
228 // 获取系统信息。其中 utsname 结构包含 5 个字段，分别是：本版本操作系统的名称、网络节点名称、
229 // 当前发行级别、版本级别和硬件类型名称。
230 int sys_uname(struct utsname * name)
231 {
232     static struct utsname thisname = { // 这里给出了结构中的信息，这种编码肯定会改变。

```

```

219         "linux .0", "nodename", "release ", "version ", "machine "
220     };
221     int i;
222
223     if (!name) return -ERROR;           // 如果存放信息的缓冲区指针为空则出错返回。
224     verify_area(name, sizeof *name);    // 验证缓冲区大小是否超限（超出已分配的内存等）。
225     for(i=0;i<sizeof *name;i++)         // 将 utsname 中的信息逐字节复制到用户缓冲区中。
226         put_fs_byte(((char *) &thisname)[i], i+(char *) name);
227     return 0;
228 }
229
// 设置当前进程创建文件属性屏蔽码为 mask & 0777。并返回原屏蔽码。
230 int sys_umask(int mask)
231 {
232     int old = current->umask;
233
234     current->umask = mask & 0777;
235     return (old);
236 }
237

```

## 5.13 vsprintf.c 程序

### 5.13.1 功能描述

主要包括 vsprintf() 函数，用于对参数产生格式化的输出。由于该函数是 C 函数库中的标准函数，基本没有涉及内核工作原理，因此可以跳过。直接阅读代码后对该函数的使用说明。

### 5.13.2 代码注释

程序 5-11 linux/kernel/vsprintf.c

```

1  /*
2   *  linux/kernel/vsprintf.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /* vsprintf.c -- Lars Wirzenius & Linus Torvalds. */
8  /*
9   * Wirzenius wrote this portably, Torvalds fucked it up :-)
10  */
// Lars Wirzenius 是 Linus 的好友，在 Helsinki 大学时曾同处一间办公室。在 1991 年夏季开发 Linux
// 时，Linus 当时对 C 语言还不是很熟悉，还不会使用可变参数列表函数功能。因此 Lars Wirzenius
// 就为他编写了这段用于内核显示信息的代码。他后来(1998 年)承认在这段代码中有一个 bug，直到
// 1994 年才有人发现，并予以纠正。这个 bug 是在使用*作为输出域宽度时，忘记递增指针跳过这个星
// 号了。在本代码中这个 bug 还仍然存在（130 行）。 他的个人主页是 http://liw.iki.fi/liw/
11
12 #include <stdarg.h>           // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了一个
                                // 类型(va_list)和三个宏(va_start, va_arg 和 va_end)，用于
                                // vsprintf、vprintf、vfprintf 函数。

```

```

13 #include <string.h>          // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
14
15 /* we use this so that we can do without the ctype library */
16 /* 我们使用下面的定义，这样我们就可以不使用 ctype 库了 */
17 #define is_digit(c)          ((c) >= '0' && (c) <= '9') // 判断字符是否是数字字符。
18
19 // 该函数将字符数字串转换成整数。输入是数字串指针的指针，返回是结果数值。另外指针将前移。
20 static int skip_atoi(const char **s)
21 {
22     int i=0;
23     while (is_digit(**s))
24         i = i*10 + *((*s)++) - '0';
25     return i;
26 }
27
28 // 这里定义转换类型的各种符号常数。
29 #define ZEROPAD 1             /* pad with zero */           /* 填充零 */
30 #define SIGN 2               /* unsigned/signed long */         /* 无符号/符号长整数 */
31 #define PLUS 4               /* show plus */                   /* 显示加 */
32 #define SPACE 8             /* space if plus */               /* 如是加，则置空格 */
33 #define LEFT 16             /* left justified */              /* 左调整 */
34 #define SPECIAL 32          /* 0x */                         /* 0x */
35 #define SMALL 64            /* use 'abcdef' instead of 'ABCDEF' */ /* 使用小写字母 */
36
37 // 除操作。输入：n 为被除数，base 为除数；结果：n 为商，函数返回值为余数。
38 // 参见 4.5.3 节有关嵌入汇编的信息。
39 #define do_div(n,base) ({ \
40     int __res; \
41     __asm__("divl %4\" : \"=a\" (n), \"=d\" (__res): \"\" (n), \"1\" (0), \"r\" (base)); \
42     __res; })
43
44 // 将整数转换为指定进制的字符串。
45 // 输入：num-整数；base-进制；size-字符串长度；precision-数字长度(精度)；type-类型选项。
46 // 输出：str 字符串指针。
47 static char * number(char * str, int num, int base, int size, int precision
48 ,int type)
49 {
50     char c, sign, tmp[36];
51     const char *digits="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
52     int i;
53
54     // 如果类型 type 指出用小写字母，则定义小写字母集。
55     // 如果类型指出要左调整（靠左边界），则屏蔽类型中的填零标志。
56     // 如果进制基数小于 2 或大于 36，则退出处理，也即本程序只能处理基数在 2-32 之间的数。
57     if (type&SMALL) digits="0123456789abcdefghijklmnopqrstuvwxyz";
58     if (type&LEFT) type &= ~ZEROPAD;
59     if (base<2 || base>36)
60         return 0;
61
62     // 如果类型指出要填零，则置字符变量 c='0'（也即''），否则 c 等于空格字符。
63     // 如果类型指出是带符号数并且数值 num 小于 0，则置符号变量 sign=负号，并使 num 取绝对值。
64     // 否则如果类型指出是加号，则置 sign=加号，否则若类型带空格标志则 sign=空格，否则置 0。
65     c = (type & ZEROPAD) ? '0' : ' ';

```

```

52     if (type&SIGN && num<0) {
53         sign='-';
54         num = -num;
55     } else
56         sign=(type&PLUS) ? '+' : ((type&SPACE) ? ' ' : 0);
// 若带符号，则宽度值减1。若类型指出是特殊转换，则对于十六进制宽度再减少2位(用于0x)，
// 对于八进制宽度减1(用于八进制转换结果前放一个零)。
57     if (sign) size--;
58     if (type&SPECIAL)
59         if (base==16) size -= 2;
60         else if (base==8) size--;
// 如果数值 num 为 0，则临时字符串='0'；否则根据给定的基数将数值 num 转换成字符形式。
61     i=0;
62     if (num==0)
63         tmp[i++]='';
64     else while (num!=0)
65         tmp[i++]=digits[do_div(num, base)];
// 若数值字符个数大于精度值，则精度值扩展为数字个数。
// 宽度值 size 减去用于存放数值字符的个数。
66     if (i>precision) precision=i;
67     size -= precision;

// 从这里真正开始形成所需要的转换结果，并暂时放在字符串 str 中。
// 若类型中没有填零(ZEROPAD)和左靠齐(左调整)标志，则在 str 中首先
// 填放剩余宽度值指出的空格数。若需带符号位，则存入符号。
68     if (!(type&(ZEROPAD+LEFT)))
69         while(size-->0)
70             *str++ = ' ';
71     if (sign)
72         *str++ = sign;
// 若类型指出是特殊转换，则对于八进制转换结果头一位放置一个'0'；而对于十六进制则存放'0x'。
73     if (type&SPECIAL)
74         if (base==8)
75             *str++ = '';
76         else if (base==16) {
77             *str++ = '';
78             *str++ = digits[33]; // 'X' 或 'x'
79         }
// 若类型中没有左调整(左靠齐)标志，则在剩余宽度中存放 c 字符('0' 或空格)，见 51 行。
80     if (!(type&LEFT))
81         while(size-->0)
82             *str++ = c;
// 此时 i 存有数值 num 的数字个数。若数字个数小于精度值，则 str 中放入(精度值-i)个'0'。
83     while(i<precision--)
84         *str++ = '';
// 将转数值换好的数字字符填入 str 中。共 i 个。
85     while(i-->0)
86         *str++ = tmp[i];
// 若宽度值仍大于零，则表示类型标志中有左靠齐标志标志。则在剩余宽度中放入空格。
87     while(size-->0)
88         *str++ = ' ';
89     return str; // 返回转换好的字符串。
90 }

```

```

91 // 下面函数是送格式化输出到字符串中。
92 // 为了能在内核中使用格式化的输出，Linux 在内核实现了该 C 标准函数。
93 // 其中参数 fmt 是格式字符串；args 是个数变化的值；buf 是输出字符串缓冲区。
94 // 请参见本代码列表后的有关格式转换字符的介绍。
95 int vsprintf(char *buf, const char *fmt, va_list args)
96 {
97     int len;
98     int i;
99     char *str;          // 用于存放转换过程中的字符串。
100    char *s;
101    int *ip;
102
103    int flags;           /* flags to number() */
104                        /* number() 函数使用的标志 */
105    int field_width;     /* width of output field */
106                        /* 输出字段宽度 */
107    int precision;       /* min. # of digits for integers; max
108                        number of chars for from string */
109                        /* min. 整数数字个数; max. 字符串中字符个数 */
110    int qualifier;       /* 'h', 'l', or 'L' for integer fields */
111                        /* 'h', 'l', 或 'L' 用于整数字段 */
112    // 首先将字符指针指向 buf，然后扫描格式字符串，对各个格式转换指示进行相应的处理。
113    for (str=buf ; *fmt ; ++fmt) {
114        // 格式转换指示字符串均以 '%' 开始，这里从 fmt 格式字符串中扫描 '%'，寻找格式转换字符串的开始。
115        // 不是格式指示的一般字符均被依次存入 str。
116        if (*fmt != '%') {
117            *str++ = *fmt;
118            continue;
119        }
120
121        // 下面取得格式指示字符串中的标志域，并将标志常量放入 flags 变量中。
122        /* process flags */
123        flags = 0;
124        repeat:
125            ++fmt;          /* this also skips first '%' */
126            switch (*fmt) {
127                case '-': flags |= LEFT; goto repeat;    // 左靠齐调整。
128                case '+': flags |= PLUS; goto repeat;   // 放加号。
129                case ' ': flags |= SPACE; goto repeat;  // 放空格。
130                case '#': flags |= SPECIAL; goto repeat; // 是特殊转换。
131                case '0': flags |= ZEROPAD; goto repeat; // 要填零(即'0')。
132            }
133
134        // 取当前参数字段宽度域值，放入 field_width 变量中。如果宽度域中是数值则直接取其为宽度值。
135        // 如果宽度域中是字符 '*'，表示下一个参数指定宽度。因此调用 va_arg 取宽度值。若此时宽度值
136        // 小于 0，则该负数表示其带有标志域 '-' 标志（左靠齐），因此还需在标志变量中添入该标志，并
137        // 将字段宽度值取其绝对值。
138        /* get field width */
139        field_width = -1;
140        if (is_digit(*fmt))
141            field_width = skip_atoi(&fmt);
142        else if (*fmt == '*') {

```

```

130      /* it's the next argument */    // 这里有个 bug, 应插入++fmt;
131      field_width = va\_arg(args, int);
132      if (field_width < 0) {
133          field_width = -field_width;
134          flags |= LEFT;
135      }
136  }
137
138  // 下面这段代码, 取格式转换串的精度域, 并放入 precision 变量中。精度域开始的标志是'.'。
139  // 其处理过程与上面宽度域的类似。如果精度域中是数值则直接取其为精度值。如果精度域中是
140  // 字符 '*', 表示下一个参数指定精度。因此调用 va_arg 取精度值。若此时宽度值小于 0, 则
141  // 将字段精度值取为其绝对值。
142
143      /* get the precision */
144      precision = -1;
145      if (*fmt == '.') {
146          ++fmt;
147          if (is\_digit(*fmt))
148              precision = skip\_atoi(&fmt);
149          else if (*fmt == '*') {
150              /* it's the next argument */
151              precision = va\_arg(args, int);
152          }
153          if (precision < 0)
154              precision = 0;
155      }
156
157  // 下面这段代码分析长度修饰符, 并将其存入 qualifier 变量。(h, l, L 的含义参见列表后的说明)。
158
159      /* get the conversion qualifier */
160      qualifier = -1;
161      if (*fmt == 'h' || *fmt == 'l' || *fmt == 'L') {
162          qualifier = *fmt;
163          ++fmt;
164      }
165
166  // 下面分析转换指示符。
167
168      switch (*fmt) {
169          // 如果转换指示符是'c', 则表示对应参数应是字符。此时如果标志域表明不是左靠齐, 则该字段前面
170          // 放入宽度域值-1 个空格字符, 然后再放入参数字符。如果宽度域还大于 0, 则表示为左靠齐, 则在
171          // 参数字符后面添加宽度值-1 个空格字符。
172          case 'c':
173              if (!(flags & LEFT))
174                  while (--field_width > 0)
175                      *str++ = ' ';
176              *str++ = (unsigned char) va\_arg(args, int);
177              while (--field_width > 0)
178                  *str++ = ' ';
179              break;
180
181          // 如果转换指示符是's', 则表示对应参数是字符串。首先取参数字符串的长度, 若其超过了精度域值,
182          // 则扩展精度域=字符串长度。此时如果标志域表明不是左靠齐, 则该字段前放入(宽度值-字符串长度)
183          // 个空格字符。然后再放入参数字符串。如果宽度域还大于 0, 则表示为左靠齐, 则在参数字符串后面
184          // 添加(宽度值-字符串长度)个空格字符。
185          case 's':

```



```

170         s = va\_arg(args, char *);
171         len = strlen(s);
172         if (precision < 0)
173             precision = len;
174         else if (len > precision)
175             len = precision;
176
177         if (!(flags & LEFT))
178             while (len < field_width--)
179                 *str++ = ' ';
180         for (i = 0; i < len; ++i)
181             *str++ = *s++;
182         while (len < field_width--)
183             *str++ = ' ';
184         break;
185
186     // 如果格式转换符是'o'，表示需将对应的参数转换成八进制数的字符串。调用 number() 函数处理。
187     case 'o':
188         str = number(str, va\_arg(args, unsigned long), 8,
189                     field_width, precision, flags);
190         break;
191
192     // 如果格式转换符是'p'，表示对应参数的一个指针类型。此时若该参数没有设置宽度域，则默认宽度
193     // 为 8，并且需要添零。然后调用 number() 函数进行处理。
194     case 'p':
195         if (field_width == -1) {
196             field_width = 8;
197             flags |= ZEROPAD;
198         }
199         str = number(str,
200                     (unsigned long) va\_arg(args, void *), 16,
201                     field_width, precision, flags);
202         break;
203
204     // 若格式转换指示是'x'或'X'，则表示对应参数需要打印成十六进制数输出。'x'表示用小写字母表示。
205     case 'x':
206         flags |= SMALL;
207     case 'X':
208         str = number(str, va\_arg(args, unsigned long), 16,
209                     field_width, precision, flags);
210         break;
211
212     // 如果格式转换字符是'd','i'或'u'，则表示对应参数是整数，'d','i'代表符号整数，因此需要加上
213     // 带符号标志。'u'代表无符号整数。
214     case 'd':
215     case 'i':
216         flags |= SIGN;
217     case 'u':
218         str = number(str, va\_arg(args, unsigned long), 10,
219                     field_width, precision, flags);
220         break;

```

```

// 若格式转换指示符是'n'，则表示要把到目前为止转换输出的字符数保存到对应参数指针指定的位置
中。
// 首先利用 va_arg() 取得该参数指针，然后将已经转换好的字符数存入该指针所指的位置。
216         case 'n':
217             ip = va_arg(args, int *);
218             *ip = (str - buf);
219             break;
220
// 若格式转换符不是'%', 则表示格式字符串有错，直接将一个 '%' 写入输出串中。
// 如果格式转换符的位置处还有字符，则也直接将该字符写入输出串中，并返回到 107 行继续处理
// 格式字符串。否则表示已经处理到格式字符串的结尾处，则退出循环。
221         default:
222             if (*fmt != '%')
223                 *str++ = '%';
224             if (*fmt)
225                 *str++ = *fmt;
226             else
227                 --fmt;
228             break;
229     }
230 }
231 *str = '\0';           // 最后在转换好的字符串结尾处添上 null。
232 return str - buf;      // 返回转换好的字符串长度值。
233 }
234

```

## 5.13.3 其它信息

### 5.13.3.1 vsprintf()的格式字符串

int [vsprintf](#)(char \*[buf](#), const char \*fmt, [va\\_list](#) args)

vsprintf()函数是 printf()系列函数之一。这些函数都产生格式化的输出：接受确定输出格式的格式字符串 fmt，用格式字符串对个数变化的参数进行格式化，产生格式化的输出。

printf 直接把输出送到标准输出句柄 stdout。cprintf 把输出送到控制台。fprintf 把输出送到文件句柄。printf 前带'v'字符的(例如 vfprintf)表示参数是从 va\_arg 数组的 va\_list args 中接受。printf 前面带's'字符则表示把输出送到以 null 结尾的字符串 buf 中（此时用户应确保 buf 有足够的空间存放字符串）。下面详细说明格式字符串的使用方法。

#### 1. 格式字符串

printf 系列函数中的格式字符串用于控制函数转换方式、格式化和输出其参数。对于每个格式，必须有对应的参数，参数过多将被忽略。格式字符串中含有两类成份，一种是将被直接复制到输出中的简单字符；另一种是用于对对应参数进行格式化的转换指示字符串。

#### 2. 格式指示字符串

格式指示串的形式如下：

%[flags][width][.prec][h|L][type]

每一个转换指示串均需要以百分号(%)开始。其中

[flags] 是可选的标志字符序列；

[width]	是可选的的宽度指示符;
[.prec]	是可选的精度(precision)指示符;
[h l L]	是可选的输入长度修饰符;
[type]	是转换类型字符(或称为转换指示符)。

flags 控制输出对齐方式、数值符号、小数点、尾零、二进制、八进制或十六进制等, 参见上面列表 27-33 行的注释。标志字符及其含义如下:

# 表示需要将相应参数转换为“特殊形式”。对于八进制(o), 则转换后的字符串的首位必须是一个零。对于十六进制(x 或 X), 则转换后的字符串需以'0x'或'0X'开头。对于 e,E,f,F,g 以及 G, 则即使没有小数位, 转换结果也将总是有一个小数点。对于 g 或 G, 后拖的零也不会删除。

0 转换结果应该是附零的。对于 d,i,o,u,x,X,e,E,f,g 和 G, 转换结果的左边将用零填空而不是用空格。如果同时出现 0 和-标志, 则 0 标志将被忽略。对于数值转换, 如果给出了精度域, 0 标志也被忽略。

- 转换后的结果在相应字段边界内将作左调整(靠左)。(默认是作右调整--靠右)。n 转换例外, 转换结果将在右面填空格。

' ' 表示带符号转换产生的一个正数结果前应该留一个空格。

+ 表示在一个符号转换结果之前总需要放置一个符号(+或-)。对于默认情况, 只有负数使用负号。

width 指定了输出字符串宽度, 即指定了字段的最小宽度值。如果被转换的结果要比指定的宽度小, 则在其左边(或者右边, 如果给出了左调整标志)需要填充空格或零(由 flags 标志确定)的个数等。除了使用数值来指定宽度域以外, 也可以使用'\*'来指出字段的宽度由下一个整型参数给出。当转换值宽度大于 width 指定的宽度时, 在任何情况下小宽度值都不会截断结果。字段宽度会扩充以包含完整结果。

precision 是说明输出数字起码的个数。对于 d,I,o,u,x 和 X 转换, 精度值指出了起码出现数字的个数。对于 e,E,f 和 F, 该值指出在小数点之后出现的数字的个数。对于 g 或 G, 指出最大有效数字个数。对于 s 或 S 转换, 精度值说明输出字符串的最大字符数。

长度修饰指示符说明了整型数转换后的输出类型形式。下面叙述中‘整型数转换’代表 d,i,o,u,x 或 X 转换。

hh 说明后面的整型数转换对应于一个带符号字符或无符号字符参数。

h 说明后面的整型数转换对应于一个带符号整数或无符号短整数参数。

l 说明后面的整型数转换对应于一个长整数或无符号长整数参数。

ll 说明后面的整型数转换对应于一个长长整数或无符号长长整数参数。

L 说明 e,E,f,F,g 或 G 转换结果对应于一个长双精度参数。

type 是说明接受的输入参数类型和输出的格式。各个转换指示符的含义如下:

d,I 整型参数将被转换为带符号整数。如果有精度(precision)的话, 则给出了需要输出的最少数字个数。如果被转换的值数字个数较少, 就会在其左边添零。默认的精度值是 1。

o,u,x,X 会将无符号的整数转换为无符号八进制(o)、无符号十进制(u)或者是无符号十六进制(x 或 X)表示方式输出。x 表示要使用小写字母(abcdef)来表示十六进制数, X 表示用大写字母(ABCDEF)表示十六进制数。如果存在精度域的话, 说明需要输出的最少数字个数。如果被转换的值数字个数较少, 就会在其左边添零。默认的精度值是 1。

e,E 这两个转换字符用于经四舍五入将参数转换成[-]d.ddde±dd 的形式。小数点之后的数字个

数等于精度。如果没有精度域，就取默认值 6。如果精度是 0，则没有小数出现。E 表示用大写字母 E 来表示指数。指数部分总是用 2 位数字表示。如果数值为 0，那么指数就是 00。

**f,F** 这两个转换字符用于经四舍五入将参数转换成[-]ddd.ddd 的形式。小数点之后的数字个数等于精度。如果没有精度域，就取默认值 6。如果精度是 0，则没有小数出现。如果有小数点，那么后面起码会有 1 位数字。

**g,G** 这两个转换字符将参数转换为 f 或 e 的格式（如果是 G，则是 F 或 E 格式）。精度值指定了整数的个数。如果没有精度域，则其默认值为 6。如果精度为 0，则作为 1 来对待。如果转换时指数小于 -4 或大于等于精度，则采用 e 格式。小数部分后拖的零将被删除。仅当起码有一位小数时才会出现小数点。

**c** 参数将被转换成无符号字符并输出转换结果。

**s** 要求输入为指向字符串的指针，并且该字符串要以 null 结尾。如果有精度域，则只输出精度所要求的字符个数，并且字符串无须以 null 结尾。

**p** 以指针形式输出十六进制数。

**n** 用于把到目前为止转换输出的字符个数保存到由对应输入指针指定的位置中。不对参数进行转换。

**%** 输出一个百分号%，不进行转换。也即此时整个转换指示为%%。

### 5.13.4 与当前版本的区别

由于该文件也属于库函数，所以从 1.2 版内核开始就直接使用库中的函数了。也即删除了该文件。

## 5.14 printk.c 程序

### 5.14.1 功能描述

printk()是内核中使用的打印（显示）函数，功能与 C 标准函数库中的 print()相同。重新编写这么一个函数的原因是在内核中不能使用专用于用户模式的 fs 段寄存器，需要首先保存它。printk()函数首先使用 svprintf()对参数进行格式化处理，然后在保存了 fs 段寄存器的情况下调用 tty\_write()进行信息的打印显示。

### 5.14.2 代码注释

程序 5-12 linux/kernel/printk.c

```

1  /*
2   *  linux/kernel/printk.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   *  When in kernel-mode, we cannot use printf, as fs is liable to
9   *  point to 'interesting' things. Make a printf with fs-saving, and
10  *  all is well.
11  */
12
13  /*
14   *  当处于内核模式时，我们不能使用 printf，因为寄存器 fs 指向其它不感兴趣的地方。
15   *  自己编制一个 printf 并在使用前保存 fs，一切就解决了。
16  */

```

```

12 #include <stdarg.h>          // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了一个
                               // 类型(va_list)和三个宏(va_start, va_arg 和 va_end), 用于
                               // vsprintf、vprintf、vfprintf 函数。
13 #include <stddef.h>          // 标准定义头文件。定义了 NULL, offsetof(TYPE, MEMBER)。
14
15 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
16
17 static char buf[1024];
18
19 // 下面该函数 vsprintf() 在 linux/kernel/vsprintf.c 中 92 行开始。
20 extern int vsprintf(char * buf, const char * fmt, va_list args);
21
22 // 内核使用的显示函数。
23 int printk(const char *fmt, ...)
24 {
25     va_list args;                // va_list 实际上是一个字符指针类型。
26     int i;
27
28     va_start(args, fmt);          // 参数处理开始函数。在 (include/stdarg.h, 13)
29     i=vsprintf(buf, fmt, args);    // 使用格式串 fmt 将参数列表 args 输出到 buf 中。
30                                   // 返回值 i 等于输出字符串的长度。
31     va_end(args);                // 参数处理结束函数。
32     __asm__ ("push %%fs\n\t"      // 保存 fs。
33             "push %%ds\n\t"
34             "pop %%fs\n\t"        // 令 fs = ds。
35             "pushl %0\n\t"        // 将字符串长度压入堆栈(这三个入栈是调用参数)。
36             "pushl $_buf\n\t"     // 将 buf 的地址压入堆栈。
37             "pushl $0\n\t"        // 将数值 0 压入堆栈。是通道号 channel。
38             "call _tty_write\n\t" // 调用 tty_write 函数。(kernel/chr_drv/tty_io.c, 290)。
39             "addl $8, %%esp\n\t"  // 跳过 (丢弃) 两个入栈参数(buf, channel)。
40             "popl %0\n\t"        // 弹出字符串长度值, 作为返回值。
41             "pop %%fs"           // 恢复原 fs 寄存器。
42             :: "r" (i): "ax", "cx", "dx"); // 通知编译器, 寄存器 ax, cx, dx 值可能已经改变。
43     return i;                    // 返回字符串长度。
44 }

```

## 5.15 panic.c 程序

### 5.15.1 功能描述

当内核程序出错时, 则调用函数 panic(), 显示错误信息并使系统进入死循环。在内核程序的许多地方, 若出现严重出错时就要调用到该函数。在很多情况下, 调用 panic() 函数是一种简明的处理方法。这样做很好地遵循了 UNIX “尽量简明” 的原则。

panic 是“惊慌, 恐慌”的意思。在 Douglas Adams 的小说《Hitch hikers Guide to the Galaxy》(《银河徒步旅行者指南》) 中, 书中最有名的一句话就是 “Don't Panic! ”。该系列小说是 linux 骇客最常阅读的一类书籍。

## 5.15.2 代码注释

程序 5-13 linux/kernel/panic.c

---

```

1  /*
2   *  linux/kernel/panic.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   *  This function is used through-out the kernel (includeinh mm and fs)
9   *  to indicate a major problem.
10  */
11 /*
12  *  该函数在整个内核中使用（包括在 头文件*.h，内存管理程序 mm 和文件系统 fs 中），
13  *  用以指出主要的出错问题。
14  */
15 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
16 #include <linux/sched.h> // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
17 // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
18
19 void sys_sync(void); /* it's really int */ /* 实际上是整型 int (fs/buffer.c, 44) */
20
21 // 该函数用来显示内核中出现的重大错误信息，并运行文件系统同步函数，然后进入死循环 — 死机。
22 // 如果当前进程是任务 0 的话，还说明是交换任务出错，并且还没有运行文件系统同步函数。
23 volatile void panic(const char * s)
24 {
25     printk("Kernel panic: %s\n\r", s);
26     if (current == task[0])
27         printk("In swapper task - not syncing\n\r");
28     else
29         sys_sync();
30     for(;;);
31 }
32

```

---

## 5.16 本章小结

linux/kernel 目录下的 12 个代码文件给出了内核中最为重要的一些机制的实现，主要包括系统调用、进程调度、进程复制以及进程的终止处理四部分。