```
1  /*
2   *  linux/init/main.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
   // 定义宏“__LIBRARY__”是为了包括定义在 unistd.h 中的内嵌汇编代码等信息。
7  #define __LIBRARY__
   // *.h 头文件所在的默认目录是 include/，则在代码中就不用明确指明其位置。如果不是 UNIX 的
   // 标准头文件，则需要指明所在的目录，并用双引号括住。unistd.h 是标准符号常数与类型文件。
   // 其中定义了各种符号常数和类型，并声明了各种函数。如果还定义了符号__LIBRARY__，则还会
   // 包含系统调用号和内嵌汇编代码 syscall0()等。
8  #include <unistd.h>
9  #include <time.h>    // 时间类型头文件。其中最主要定义了 tm 结构和一些有关时间的函数原形。
10
11 /*
12  * we need this inline - forking from kernel space will result
13  * in NO COPY ON WRITE (!!!), until an execve is executed. This
14  * is no problem, but for the stack. This is handled by not letting
15  * main() use the stack at all after fork(). Thus, no function
16  * calls - which means inline code for fork too, as otherwise we
17  * would use the stack upon exit from 'fork()'.
18  *
19  * Actually only pause and fork are needed inline, so that there
20  * won't be any messing with the stack from main(), but we define
21  * some others too.
22  */
   /*
    * 我们需要下面这些内嵌语句 - 从内核空间创建进程将导致没有写时复制(COPY ON WRITE)!!!
    * 直到执行一个 execve 调用。这对堆栈可能带来问题。处理方法是在 fork()调用后不让 main()
    * 使用任何堆栈。因此就不能有函数调用 - 这意味着 fork 也要使用内嵌的代码，否则我们在从
    * fork()退出时就要使用堆栈了。
    *
    * 实际上只有 pause 和 fork 需要使用内嵌方式，以保证从 main()中不会弄乱堆栈，但是我们同
    * 时还定义了其他一些函数。
    */
   // Linux 在内核空间创建进程时不使用写时复制技术（Copy on write）。main()在移动到用户
   // 模式（到任务 0）后执行内嵌方式的 fork()和 pause()，因此可保证不使用任务 0 的用户栈。
   // 在执行 moveto_user_mode()之后，本程序 main()就以任务 0 的身份在运行了。而任务 0 是所
   // 有将创建子进程的父进程。当它创建一个子进程时（init 进程），由于任务 1 代码属于内核
   // 空间，因此没有使用写时复制功能。此时任务 0 的用户栈就是任务 1 的用户栈，即它们共同
   // 使用一个栈空间。因此希望在 main.c 运行在任务 0 的环境下时不要有对堆栈的任何操作，以
   // 免弄乱堆栈。而在再次执行 fork()并执行过 execve()函数后，被加载程序已不属于内核空间，
   // 因此可以使用写时复制技术了。请参见 5.3 节“Linux 内核使用内存的方法”内容。

   // 下面_syscall0()是 unistd.h 中的内嵌宏代码。以嵌入汇编的形式调用 Linux 的系统调用中断
   // 0x80。该中断是所有系统调用的入口。该条语句实际上是 int fork()创建进程系统调用。可展
   // 开看之就会立刻明白。syscall0 名称中最后的 0 表示无参数，1 表示 1 个参数。
   // 参见 include/unistd.h，133 行。
23 static inline _syscall0(int,fork)
   // int pause()系统调用：暂停进程的执行，直到收到一个信号。
24 static inline _syscall0(int,pause)
```

```
   // int setup(void * BIOS)系统调用，仅用于 linux 初始化（仅在这个程序中被调用）。
25 static inline _syscall1(int,setup,void *,BIOS)
   // int sync()系统调用：更新文件系统。
26 static inline _syscall0(int,sync)
27
28 #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。
29 #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。
30 #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。
31 #include <asm/system.h>   // 系统头文件。以宏形式定义了许多有关设置或修改描述符/中断门
                             // 等的嵌入式汇编子程序。
32 #include <asm/io.h>       // io 头文件。以宏的嵌入汇编程序形式定义对 io 端口操作的函数。
33
34 #include <stddef.h>       // 标准定义头文件。定义了 NULL, offsetof(TYPE, MEMBER)。
35 #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-个
                             // 类型(va_list)和三个宏(va_start, va_arg 和 va_end)，vsprintf、
                             // vprintf、vfprintf。
36 #include <unistd.h>
37 #include <fcntl.h>        // 文件控制头文件。用于文件及其描述符的操作控制常数符号的定义。
38 #include <sys/types.h>    // 类型头文件。定义了基本的系统数据类型。
39
40 #include <linux/fs.h>     // 文件系统头文件。定义文件表结构（file,buffer_head,m_inode 等）。
41                           // 其中有定义：extern int ROOT_DEV。
42 #include <string.h>       // 字符串头文件。主要定义了一些有关内存或字符串操作的嵌入函数。
43
44 static char printbuf[1024];           // 静态字符串数组，用作内核显示信息的缓存。
45
46 extern char *strcpy();
47 extern int vsprintf();                // 送格式化输出到一字符串中（vsprintf.c，92 行）。
48 extern void init(void);               // 函数原形，初始化（本程序 168 行）。
49 extern void blk_dev_init(void);       // 块设备初始化子程序（blk_drv/ll_rw_blk.c,157 行）
50 extern void chr_dev_init(void);       // 字符设备初始化（chr_drv/tty_io.c, 347 行）
51 extern void hd_init(void);            // 硬盘初始化程序（blk_drv/hd.c, 343 行）
52 extern void floppy_init(void);        // 软驱初始化程序（blk_drv/floppy.c, 457 行）
53 extern void mem_init(long start, long end);      // 内存管理初始化（mm/memory.c, 399 行）
54 extern long rd_init(long mem_start, int length); // 虚拟盘初始化(blk_drv/ramdisk.c,52)
55 extern long kernel_mktime(struct tm * tm);       // 计算系统开机启动时间（秒）。
56
   // 内核专用 sprintf()函数。该函数用于产生格式化信息并输出到指定缓冲区 str 中。参数'*fmt'
   // 指定输出将采用的格式，参见标准 C 语言书籍。该子程序正好是 vsprintf 如何使用的一个简单
   // 例子。函数使用 vsprintf()将格式化字符串放入 str 缓冲区，参见第 179 行上的 printf()函数。
57 static int sprintf(char * str, const char *fmt, ...)
58 {
59         va_list args;
60         int i;
61
62         va_start(args, fmt);
63         i = vsprintf(str, fmt, args);
64         va_end(args);
65         return i;
66 }
67
```

```
68 /*
69  * This is set up by the setup-routine at boot-time
70  */
   /*
    * 以下这些数据是在内核引导期间由 setup.s 程序设置的。
    */
   // 下面三行分别将指定的线性地址强行转换为给定数据类型的指针，并获取指针所指内容。由于
   // 内核代码段被映射到从物理地址零开始的地方，因此这些线性地址正好也是对应的物理地址。
   // 这些指定地址处内存值的含义请参见第 6 章的表 6-2（setup 程序读取并保存的参数）。
   // drive_info 结构请参见下面第 125 行。
71 #define EXT_MEM_K (*(unsigned short *)0x90002)      // 1MB 以后的扩展内存大小（KB）。
72 #define CON_ROWS ((*(unsigned short *)0x9000e) & 0xff)  // 选定的控制台屏幕行、列数。
73 #define CON_COLS (((*(unsigned short *)0x9000e) & 0xff00) >> 8)
74 #define DRIVE_INFO (*(struct drive_info *)0x90080)  // 硬盘参数表 32 字节内容。
75 #define ORIG_ROOT_DEV (*(unsigned short *)0x901FC)  // 根文件系统所在设备号。
76 #define ORIG_SWAP_DEV (*(unsigned short *)0x901FA)  // 交换文件所在设备号。
77
78 /*
79  * Yeah, yeah, it's ugly, but I cannot find how to do this correctly
80  * and this seems to work. I anybody has more info on the real-time
81  * clock I'd be interested. Most of this was trial and error, and some
82  * bios-listing reading. Urghh.
83  */
   /*
    * 是啊，是啊，下面这段程序很差劲，但我不知道如何正确地实现，而且好象
    * 它还能运行。如果有关于实时时钟更多的资料，那我很感兴趣。这些都是试
    * 探出来的，另外还看了一些 bios 程序，呵！
    */
84
   // 这段宏读取 CMOS 实时时钟信息。outb_p 和 inb_p 是 include/asm/io.h 中定义的端口输入输出宏。
85 #define CMOS_READ(addr) ({ \
86 outb_p(0x80|addr,0x70); \       // 0x70 是写地址端口号，0x80|addr 是要读取的 CMOS 内存地址。
87 inb_p(0x71); \                  // 0x71 是读数据端口号。
88 })
89
   // 定义宏。将 BCD 码转换成二进制数值。BCD 码利用半个字节（4 比特）表示一个 10 进制数，因此
   // 一个字节表示 2 个 10 进制数。(val)&15 取 BCD 表示的 10 进制个位数，而 (val)>>4 取 BCD 表示
   // 的 10 进制十位数，再乘以 10。因此最后两者相加就是一个字节 BCD 码的实际二进制数值。
90 #define BCD_TO_BIN(val) ((val)=((val)&15) + ((val)>>4)*10)
91
   // 该函数取 CMOS 实时钟信息作为开机时间，并保存到全局变量 startup_time(秒)中。参见后面
   // CMOS 内存列表说明。其中调用的函数 kernel_mktime()用于计算从 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（kernel/mktime.c 41 行）。
92 static void time_init(void)
93 {
94         struct tm time;                         // 时间结构 tm 定义在 include/time.h 中。
95
   // CMOS 的访问速度很慢。为了减小时间误差，在读取了下面循环中所有数值后，若此时 CMOS 中
   // 秒值发生了变化，那么就重新读取所有值。这样内核就能把与 CMOS 时间误差控制在 1 秒之内。
96         do {
97                 time.tm_sec = CMOS_READ(0);         // 当前时间秒值（均是 BCD 码值）。
98                 time.tm_min = CMOS_READ(2);         // 当前分钟值。
99                 time.tm_hour = CMOS_READ(4);        // 当前小时值。
```

```
100                 time.tm_mday = CMOS_READ(7);        // 一月中的当天日期。
101                 time.tm_mon = CMOS_READ(8);         // 当前月份（1—12）。
102                 time.tm_year = CMOS_READ(9);        // 当前年份。
103         } while (time.tm_sec != CMOS_READ(0));
104         BCD_TO_BIN(time.tm_sec);                    // 转换成二进制数值。
105         BCD_TO_BIN(time.tm_min);
106         BCD_TO_BIN(time.tm_hour);
107         BCD_TO_BIN(time.tm_mday);
108         BCD_TO_BIN(time.tm_mon);
109         BCD_TO_BIN(time.tm_year);
110         time.tm_mon--;                              // tm_mon 中月份范围是 0—11。
111         startup_time = kernel_mktime(&time);        // 计算开机时间。kernel/mktime.c 41 行。
112 }
113
    // 下面定义一些局部变量。
114 static long memory_end = 0;                         // 机器具有的物理内存容量（字节数）。
115 static long buffer_memory_end = 0;                  // 高速缓冲区末端地址。
116 static long main_memory_start = 0;                  // 主内存（将用于分页）开始的位置。
117 static char term[32];                               // 终端设置字符串（环境参数）。
118
    // 读取并执行/etc/rc 文件时所使用的命令行参数和环境参数。
119 static char * argv_rc[] = { "/bin/sh", NULL };      // 调用执行程序时参数的字符串数组。
120 static char * envp_rc[] = { "HOME=/", NULL ,NULL }; // 调用执行程序时的环境字符串数组。
121
    // 运行登录 shell 时所使用的命令行参数和环境参数。
    // 第 122 行中 argv[0]中的字符“-”是传递给 shell 程序 sh 的一个标志。通过识别该标志，
    // sh 程序会作为登录 shell 执行。其执行过程与在 shell 提示符下执行 sh 不一样。
122 static char * argv[] = { "-/bin/sh",NULL };       // 同上。
123 static char * envp[] = { "HOME=/usr/root", NULL, NULL };
124
125 struct drive_info { char dummy[32]; } drive_info;  // 用于存放硬盘参数表信息。
126
    // 内核初始化主程序。初始化结束后将以任务 0（idle 任务即空闲任务）的身份运行。
    // 英文注释含义是“这里确实是 void，没错。在 startup 程序(head.s)中就是这样假设的”。参见
    // head.s 程序第 136 行开始的几行代码。
127 void main(void)            /* This really IS void, no error here. */
128 {                          /* The startup routine assumes (well, ...) this */
129 /*
130  * Interrupts are still disabled. Do necessary setups, then
131  * enable them
132  */
    /*
     * 此时中断仍被禁止着，做完必要的设置后就将其开启。
     */
    // 首先保存根文件系统设备号和交换文件设备号，并根据 setup.s 程序中获取的信息设置控制台
    // 终端屏幕行、列数环境变量 TERM，并用其设置初始 init 进程中执行 etc/rc 文件和 shell 程序
    // 使用的环境变量，以及复制内存 0x90080 处的硬盘参数表。
    // 其中 ROOT_DEV 已在前面包含进的 include/linux/fs.h 文件第 206 行上被声明为 extern int，
    // 而 SWAP_DEV 在 include/linux/mm.h 文件内也作了相同声明。这里 mm.h 文件并没有显式地列在
    // 本程序前部，因为前面包含进的 include/linux/sched.h 文件中已经含有它。
133         ROOT_DEV = ORIG_ROOT_DEV;               // ROOT_DEV 定义在 fs/super.c，29 行。
134         SWAP_DEV = ORIG_SWAP_DEV;               // SWAP_DEV 定义在 mm/swap.c，36 行。
135         sprintf(term, "TERM=con%dx%d", CON_COLS, CON_ROWS);
```

```
136         envp[1] = term;
137         envp_rc[1] = term;
138         drive_info = DRIVE_INFO;                // 复制内存 0x90080 处的硬盘参数表。

    // 接着根据机器物理内存容量设置高速缓冲区和主内存区的位置和范围。
    // 高速缓存末端地址➔buffer_memory_end；机器内存容量➔memory_end；
    // 主内存开始地址  ➔main_memory_start；
139         memory_end = (1<<20) + (EXT_MEM_K<<10); // 内存大小=1Mb + 扩展内存(k)*1024 字节。
140         memory_end &= 0xfffff000;               // 忽略不到 4Kb（1 页）的内存数。
141         if (memory_end > 16*1024*1024)          // 如果内存量超过 16Mb，则按 16Mb 计。
142                 memory_end = 16*1024*1024;
143         if (memory_end > 12*1024*1024)          // 如果内存>12Mb，则设置缓冲区末端=4Mb
144                 buffer_memory_end = 4*1024*1024;
145         else if (memory_end > 6*1024*1024)      // 否则若内存>6Mb，则设置缓冲区末端=2Mb
146                 buffer_memory_end = 2*1024*1024;
147         else
148                 buffer_memory_end = 1*1024*1024; // 否则则设置缓冲区末端=1Mb
149         main_memory_start = buffer_memory_end;  // 主内存起始位置 = 缓冲区末端。

    // 如果在 Makefile 文件中定义了内存虚拟盘符号 RAMDISK，则初始化虚拟盘。此时主内存将减少。
    //参见 kernel/blk_drv/ramdisk.c。
150 #ifdef RAMDISK
151         main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
152 #endif
    // 以下是内核进行所有方面的初始化工作。阅读时最好跟着调用的程序深入进去看，若实在看
    // 不下去了，就先放一放，继续看下一个初始化调用 -- 这是经验之谈☺。
153         mem_init(main_memory_start,memory_end);  // 主内存区初始化。（mm/memory.c，399）
154         trap_init();            // 陷阱门（硬件中断向量）初始化。（kernel/traps.c，181）
155         blk_dev_init();         // 块设备初始化。      （blk_drv/ll_rw_blk.c，157）
156         chr_dev_init();         // 字符设备初始化。    （chr_drv/tty_io.c，347）
157         tty_init();             // tty 初始化。        （chr_drv/tty_io.c，406）
158         time_init();            // 设置开机启动时间。（见第 92 行）
159         sched_init();           // 调度程序初始化（加载任务 0 的 tr,ldtr）（kernel/sched.c，385）
160         buffer_init(buffer_memory_end); // 缓冲管理初始化，建内存链表等。（fs/buffer.c，348）
161         hd_init();              // 硬盘初始化。        （blk_drv/hd.c，343）
162         floppy_init();          // 软驱初始化。        （blk_drv/floppy.c，457）
163         sti();                  // 所有初始化工作都做完了，于是开启中断。

    // 下面过程通过在堆栈中设置的参数，利用中断返回指令启动任务 0 执行。
164         move_to_user_mode();    // 移到用户模式下执行。（include/asm/system.h，第 1 行）
165         if (!fork()) {          /* we count on this going ok */
166                 init();         // 在新建的子进程（任务 1 即 init 进程）中执行。
167         }

    // 下面代码开始以任务 0 的身份运行。
168 /*
169  *  NOTE!!  For any other task 'pause()' would mean we have to get a
170  * signal to awaken, but task0 is the sole exception (see 'schedule()')
171  * as task 0 gets activated at every idle moment (when no other tasks
172  * can run). For task0 'pause()' just means we go check if some other
173  * task can run, and if not we return here.
174  */
    /* 注意!! 对于任何其他的任务，'pause()'将意味着我们必须等待收到一个信号
```

```
     * 才会返回就绪态，但任务 0（task0）是唯一例外情况（参见'schedule()'），
     * 因为任务 0 在任何空闲时间里都会被激活（当没有其他任务在运行时），因此
     * 对于任务 0'pause()'仅意味着我们返回来查看是否有其他任务可以运行，如果
     * 没有的话我们就回到这里，一直循环执行'pause()'。
     */
    // pause()系统调用（kernel/sched.c,144）会把任务 0 转换成可中断等待状态，再执行调度函数。
    // 但是调度函数只要发现系统中没有其他任务可以运行时就会切换到任务 0，而不依赖于任务 0 的
    // 状态。
175         for(;;)
176                 __asm__("int $0x80"::"a" (__NR_pause):"ax");  // 即执行系统调用 pause()。
177 }
178
    // 下面函数产生格式化信息并输出到标准输出设备 stdout(1)，这里是指屏幕上显示。参数'*fmt'
    // 指定输出将采用的格式，参见标准 C 语言书籍。该子程序正好是 vsprintf 如何使用的一个简单
    // 例子。该程序使用 vsprintf()将格式化的字符串放入 printbuf 缓冲区，然后用 write()将缓冲
    // 区的内容输出到标准设备（1--stdout）。vsprintf()函数的实现见 kernel/vsprintf.c。
179 static int printf(const char *fmt, ...)
180 {
181         va_list args;
182         int i;
183
184         va_start(args, fmt);
185         write(1,printbuf,i=vsprintf(printbuf, fmt, args));
186         va_end(args);
187         return i;
188 }
189


    // 在 main()中已经进行了系统初始化，包括内存管理、各种硬件设备和驱动程序。init()函数
    // 运行在任务 0 第 1 次创建的子进程（任务 1）中。它首先对第一个将要执行的程序（shell）
    // 的环境进行初始化，然后以登录 shell 方式加载该程序并执行之。
190 void init(void)
191 {
192         int pid,i;
193
    // setup() 是一个系统调用。用于读取硬盘参数包括分区表信息并加载虚拟盘（若存在的话）和
    // 安装根文件系统设备。该函数用 25 行上的宏定义，对应函数是 sys_setup()，在块设备子目录
    // kernel/blk_drv/hd.c，74 行。
194         setup((void *) &drive_info);

    // 下面以读写访问方式打开设备“/dev/tty0”，它对应终端控制台。由于这是第一次打开文件
    // 操作，因此产生的文件句柄号（文件描述符）肯定是 0。该句柄是 UNIX 类操作系统默认的控
    // 制台标准输入句柄 stdin。这里再把它以读和写的方式分别打开是为了复制产生标准输出（写）
    // 句柄 stdout 和标准出错输出句柄 stderr。函数前面的“(void)”前缀用于表示强制函数无需
    // 返回值。
195         (void) open("/dev/tty1",O_RDWR,0);
196         (void) dup(0);                  // 复制句柄，产生句柄 1 号--stdout 标准输出设备。
197         (void) dup(0);                  // 复制句柄，产生句柄 2 号--stderr 标准出错输出设备。

    // 下面打印缓冲区块数和总字节数，每块 1024 字节，以及主内存区空闲内存字节数。
198         printf("%d buffers = %d bytes buffer space\n\r",NR_BUFFERS,
199                 NR_BUFFERS*BLOCK_SIZE);
200         printf("Free mem: %d bytes\n\r",memory_end-main_memory_start);
```

```
    // 下面 fork()用于创建一个子进程（任务 2）。对于被创建的子进程，fork()将返回 0 值，对于
    // 原进程（父进程）则返回子进程的进程号 pid。所以第 202--206 行是子进程执行的内容。该子
    // 进程关闭了句柄 0（stdin）、以只读方式打开/etc/rc 文件，并使用 execve()函数将进程自身
    // 替换成 /bin/sh 程序（即 shell 程序），然后执行 /bin/sh 程序。所携带的参数和环境变量分
    // 别由 argv_rc 和 envp_rc 数组给出。关闭句柄 0 并立刻打开 /etc/rc 文件的作用是把标准输入
    // stdin 重定向到 /etc/rc 文件。这样 shell 程序/bin/sh 就可以运行 rc 文件中设置的命令。由
    // 于这里 sh 的运行方式是非交互式的，因此在执行完 rc 文件中的命令后就会立刻退出，进程 2
    // 也随之结束。关于 execve()函数说明请参见 fs/exec.c 程序，207 行。
    // 函数_exit()退出时的出错码 1 - 操作未许可；2 -- 文件或目录不存在。
201         if (!(pid=fork())) {
202                 close(0);
203                 if (open("/etc/rc",O_RDONLY,0))
204                         _exit(1);                       // 若打开文件失败，则退出(lib/_exit.c,10)。
205                 execve("/bin/sh",argv_rc,envp_rc);          // 替换成/bin/sh 程序并执行。
206                 _exit(2);                                   // 若 execve()执行失败则退出。
207         }

    // 下面还是父进程（1）执行的语句。wait()等待子进程停止或终止，返回值应是子进程的进程号
    // (pid)。这三句的作用是父进程等待子进程的结束。&i 是存放返回状态信息的位置。如果 wait()
    // 返回值不等于子进程号，则继续等待。
208         if (pid>0)
209                 while (pid != wait(&i))
210                         /* nothing */;         /* 空循环 */

    // 如果执行到这里，说明刚创建的子进程的执行已停止或终止了。下面循环中首先再创建一个子
    // 进程，如果出错，则显示“初始化程序创建子进程失败”信息并继续执行。对于所创建的子进
    // 程将关闭所有以前还遗留的句柄(stdin, stdout, stderr)，新创建一个会话并设置进程组号，
    // 然后重新打开 /dev/tty0 作为 stdin，并复制成 stdout 和 stderr。再次执行系统解释程序
    // /bin/sh。但这次执行所选用的参数和环境数组另选了一套（见上面 122--123 行）。然后父进
    // 程再次运行 wait()等待。如果子进程又停止了执行，则在标准输出上显示出错信息“子进程
    // pid 停止了运行，返回码是 i”，然后继续重试下去…，形成“大”死循环。
211         while (1) {
212                 if ((pid=fork())<0) {
213                         printf("Fork failed in init\r\n");
214                         continue;
215                 }
216                 if (!pid) {                             // 新的子进程。
217                         close(0);close(1);close(2);
218                         setsid();                       // 创建一新的会话期，见后面说明。
219                         (void) open("/dev/tty1",O_RDWR,0);
220                         (void) dup(0);
221                         (void) dup(0);
222                         _exit(execve("/bin/sh",argv,envp));
223                 }
224                 while (1)
225                         if (pid == wait(&i))
226                                 break;
227                 printf("\n\rchild %d died with code %04x\n\r",pid,i);
228                 sync();                                 // 同步操作，刷新缓冲区。
229         }
230         _exit(0);       /* NOTE! _exit, not exit() */  /*注意！是_exit()，非 exit()*/
    // _exit()和 exit()都用于正常终止一个函数。但_exit()直接是一个 sys_exit 系统调用，而
```

```
    // exit()则通常是普通函数库中的一个函数。它会先执行一些清除操作，例如调用执行各终止
    // 处理程序、关闭所有标准 IO 等，然后调用 sys_exit。
231 }
232
```