第4章 初始化程序(init)

4.1 概述

在内核源代码的 init/目录中只有一个 main.c 文件。系统在执行完 boot/目录中的 head.s 程序后就会将执行权交给 main.c。该程序虽然不长，但却包括了内核初始化的所有工作。因此在阅读该程序的代码时需要参照很多其它程序中的初始化部分。如果能完全理解这里调用的所有程序，那么看完这章内容后你应该对 Linux 内核有了大致的了解。

从这一章开始，我们将接触大量的 C 程序代码，因此读者最好具有一定的 C 语言知识。最好的一本参考书还是 Brian W. Kernighan 和 Dennis M. Ritchie 编著的《C 程序设计语言》，对该书第五章关于指针和数组的理解，可以说是弄懂 C 语言的关键。

在注释 C 语言程序时，为了与程序中原有的注释相区别，我们使用'//'作为注释语句的开始。有关原有注释的翻译则采用与其一样的注释标志。对于程序中包含的头文件（*.h），仅作概要含义的解释，具体详细注释内容将在注释相应头文件的章节中给出。

4.2 main.c 程序

4.2.1 功能描述

main.c 程序首先利用前面 setup.s 程序取得的系统参数设置系统的根文件设备号以及一些内存全局变量。这些内存变量指明了主内存的开始地址、系统所拥有的内存容量和作为高速缓冲区内存的末端地址。如果还定义了虚拟盘（RAMDISK），则主内存将适当减少。整个内存的映像示意图见图 4-1 所示。

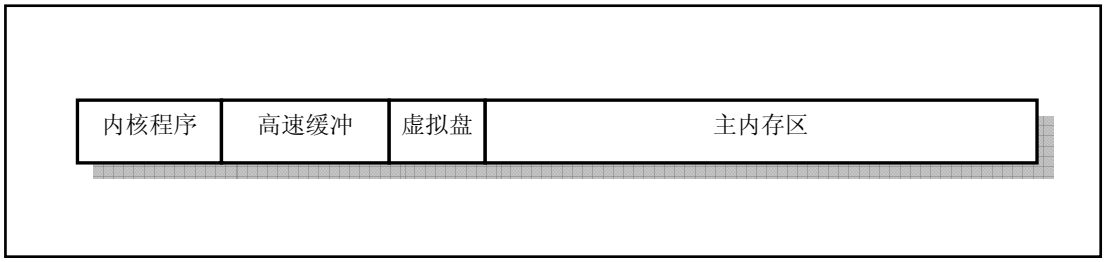

图 4-1 系统中内存功能划分示意图。

图中，高速缓冲部分还要扣除被显存和 ROM BIOS 占用的部分。高速缓冲区是用于磁盘等块设备临时存放数据的地方，以 1K（1024）字节为一个数据块单位。主内存区域的内存是由内存管理模块 mm 通过分页机制进行管理分配，以 4K 字节为一个内存页单位。内核程序可以自由访问高速缓冲中的数据，但需要通过 mm 才能使用分配到的内存页面。

然后，内核进行所有方面的硬件初始化工作。包括陷阱门、块设备、字符设备和 tty，包括人工设置

第一个任务（task 0）。待所有初始化工作完成后就设置中断允许标志以开启中断，main()也切换到了任务 0 中运行。在阅读这些初始化子程序时，最好是跟着被调用的程序深入进去看，如果实在看不下去了，就暂时先放一放，继续看下一个初始化调用。在有些理解之后再继续研究没有看完的地方。

在整个内核完成初始化后，内核将执行权切换到了用户模式（任务 0），也即 CPU 从 0 特权级切换到了第 3 特权级。此时 main.c 的主程序就工作在任务 0 中。然后系统第一次调用进程创建函数 fork()，创建出一个用于运行 init()的子进程。系统整个初始化过程见图 4-2 所示。

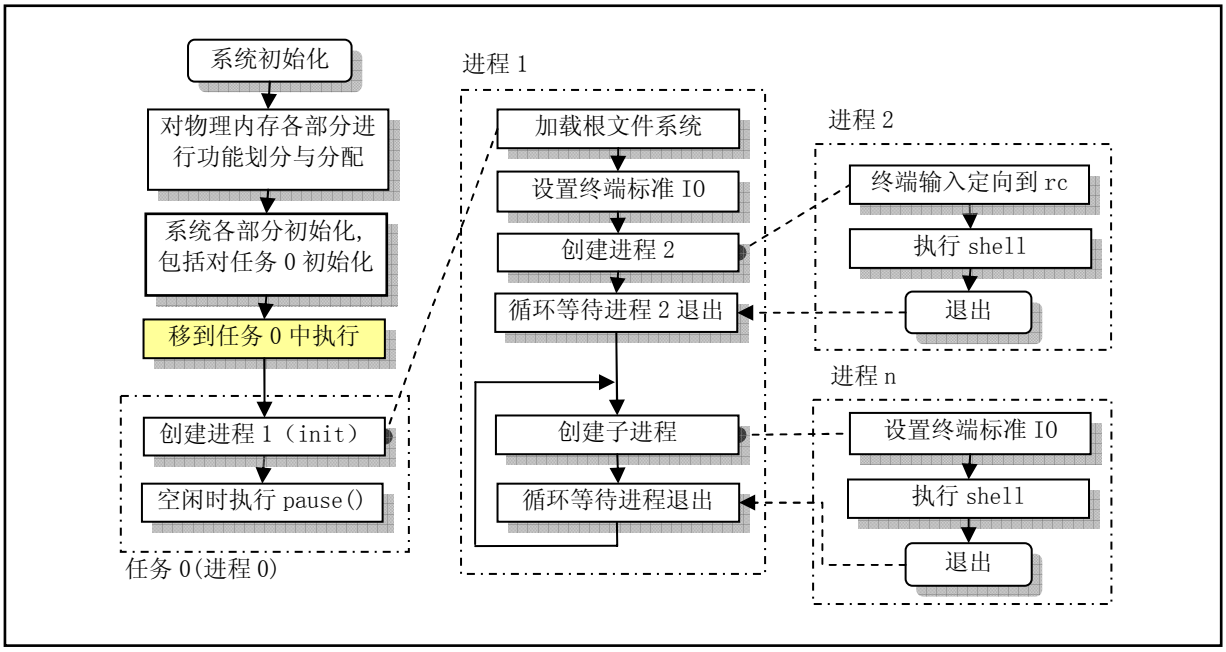

图 4-2 内核初始化程序流程示意图

该程序首先确定如何分配使用系统物理内存，然后调用内核各部分的初始化函数分别对内存管理、中断处理、块设备和字符设备、进程管理以及硬盘和软盘硬件进行初始化处理。在完成了这些操作之后，系统各部分已经处于可运行状态。此后程序把自己“手工”移动到任务 0（进程 0）中运行，并使用 fork()调用首次创建出进程 1（init 进程）。在 init 进程中程序将继续进行应用环境的初始化并执行 shell 登录程序。而原进程 0 则会在系统空闲时被调度执行，此时任务 0 仅执行 pause()系统调用，并又会调用调度函数。

在 init 进程中，如果终端环境建立成功，则会再生成一个子进程（进程 2），用于运行 shell 程序/bin/sh。若该子进程退出，则父进程进入一个死循环内，继续生成子进程，并在此子进程中再次执行 shell 程序/bin/sh，而父进程则继续等待。

由于创建新进程的过程是通过完全复制父进程代码段和数据段的方式实现的，因此在首次使用 fork()创建新进程 init 时，为了确保新进程用户态堆栈没有进程 0 的多余信息，要求进程 0 在创建首个新进程之前不要使用用户态堆栈，也即要求任务 0 不要调用函数。因此在 main.c 主程序移动到任务 0 执行后，任务 0 中的代码 fork()不能以函数形式进行调用。程序中实现的方法是采用 gcc 函数内嵌的形式来执行这个系统调用。参见下面程序第 23 行。

> 通过申明一个内嵌（inline）函数，可以让 gcc 把函数的代码集成到调用它的代码中。这会提高代码执行的速度，因为省去了函数调用的开销。另外，如果任何一个实际参数是一个常量，那么在编译时这些已知值就可能使得无需把内嵌函数的所有代码都包括进来而让代码也得到简化。

另外，任务 0 中的 pause()也需要使用函数内嵌形式来定义。如果调度程序首先执行新创建的子进程 init，那么 pause()采用函数调用形式不会有什么问题。但是内核调度程序执行父进程（进程 0）和子进程 init 的次序是随机的，在创建了 init 后有可能首先会调度进程 0 执行。因此 pause()也必须采用宏定义来实现。

对于 Linux 来说，所有任务都是在用户模式运行的，包括很多系统应用程序，如 shell 程序、网络子系统程序等。内核源代码 lib/目录下的库文件就是专门为这里新创建的进程提供支持函数的。

4.2.2 代码注释

程序 4-1 linux/init/main.c

```
 1 /*
 2  *  linux/init/main.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #define __LIBRARY__    // 定义该变量是为了包括定义在 unistd.h 中的内嵌汇编代码等信息。
 8 #include <unistd.h>    // *.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。
                          // 标准符号常数与类型文件。定义了各种符号常数和类型，并申明了各种函数。
                          // 如果定义了__LIBRARY__，则还含系统调用号和内嵌汇编代码 syscall0()等。
 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(forking)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。处理的方法是在 fork()调用之后不让 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
     * 定义了其它一些函数。
     */
    // 本程序将会在移动到用户模式（切换到任务 0）后才执行 fork()，因此避免了在内核空间写时复制问
题。
    // 在执行了 moveto_user_mode()之后，本程序就以任务 0 的身份在运行了。而任务 0 是所有将创建的子
    // 进程的父进程。当创建第一个子进程时，任务 0 的堆栈也会被复制。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，从而也不会弄乱所有子进程的堆栈。
```

```
23 static inline _syscall0(int,fork)
   // 这是 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。
   // 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。
24 static inline _syscall0(int,pause)           // int pause()系统调用：暂停进程的执行，直到
                                               // 收到一个信号。
25 static inline _syscall1(int,setup,void *,BIOS)  // 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）。
26 static inline _syscall0(int,sync)               // int sync()系统调用：更新文件系统。
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。以宏的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
42 static char printbuf[1024];       // 静态字符串数组，用作内核显示信息的缓存。
43
44 extern int vsprintf();            // 送格式化输出到一字符串中（在 kernel/vsprintf.c，92 行）。
45 extern void init(void);           // 函数原形，初始化（在 168 行）。
46 extern void blk_dev_init(void);   // 块设备初始化子程序（kernel/blk_drv/ll_rw_blk.c,157 行）
47 extern void chr_dev_init(void);   // 字符设备初始化（kernel/chr_drv/tty_io.c, 347 行）
48 extern void hd_init(void);        // 硬盘初始化程序（kernel/blk_drv/hd.c, 343 行）
49 extern void floppy_init(void);    // 软驱初始化程序（kernel/blk_drv/floppy.c, 457 行）
50 extern void mem_init(long start, long end);     // 内存管理初始化（mm/memory.c, 399 行）
51 extern long rd_init(long mem_start, int length); // 虚拟盘初始化(kernel/blk_drv/ramdisk.c,52)
52 extern long kernel_mktime(struct tm * tm);      // 计算系统开机启动时间（秒）。
53 extern long startup_time;                       // 内核启动时间（开机时间）（秒）。
54
55 /*
56  * This is set up by the setup-routine at boot-time
57  */
   /*
    * 以下这些数据是由 setup.s 程序在引导时间设置的（参见第 3 章中表 3.2）。
    */
58 #define EXT_MEM_K (*(unsigned short *)0x90002)      // 1M 以后的扩展内存大小（KB）。
59 #define DRIVE_INFO (*(struct drive_info *)0x90080)  // 硬盘参数表基址。
60 #define ORIG_ROOT_DEV (*(unsigned short *)0x901FC)  // 根文件系统所在设备号。
61
62 /*
```

```
 * Yeah, yeah, it's ugly, but I cannot find how to do this correctly
 * and this seems to work. I anybody has more info on the real-time
 * clock I'd be interested. Most of this was trial and error, and some
 * bios-listing reading. Urghh.
 */
/*
 * 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，以及看了一些 bios 程序，呵！
 */

#define CMOS_READ(addr) ({ \          // 这段宏读取 CMOS 实时时钟信息。
outb_p(0x80|addr,0x70); \             // 0x70 是写端口号，0x80|addr 是要读取的 CMOS 内存地址。
inb_p(0x71); \                        // 0x71 是读端口号。
})

// 定义宏。将 BCD 码转换成二进制数值。
#define BCD_TO_BIN(val) ((val)=((val)&15) + ((val)>>4)*10)

// 该子程序取 CMOS 时钟，并设置开机时间→startup_time(秒)。参见后面 CMOS 内存列表。
static void time_init(void)
{
        struct tm time;               // 时间结构 tm 定义在 include/time.h 中。

// 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 的时间误差控制在 1 秒之内。
        do {
                time.tm_sec = CMOS_READ(0);       // 当前时间秒值（均是 BCD 码值）。
                time.tm_min = CMOS_READ(2);       // 当前分钟值。
                time.tm_hour = CMOS_READ(4);      // 当前小时值。
                time.tm_mday = CMOS_READ(7);      // 一月中的当天日期。
                time.tm_mon = CMOS_READ(8);       // 当前月份（1—12）。
                time.tm_year = CMOS_READ(9);      // 当前年份。
        } while (time.tm_sec != CMOS_READ(0));
        BCD_TO_BIN(time.tm_sec);                  // 转换成二进制数值。
        BCD_TO_BIN(time.tm_min);
        BCD_TO_BIN(time.tm_hour);
        BCD_TO_BIN(time.tm_mday);
        BCD_TO_BIN(time.tm_mon);
        BCD_TO_BIN(time.tm_year);
        time.tm_mon--;                            // tm_mon 中月份范围是 0—11。
// 调用 kernel/mktime.c 中函数，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。
        startup_time = kernel_mktime(&time);
}

static long memory_end = 0;           // 机器具有的物理内存容量（字节数）。
static long buffer_memory_end = 0;    // 高速缓冲区末端地址。
static long main_memory_start = 0;    // 主内存（将用于分页）开始的位置。

struct drive_info { char dummy[32]; } drive_info;  // 用于存放硬盘参数表信息。

void main(void)         /* This really IS void, no error here. */
{                       /* The startup routine assumes (well, ...) this */
```

```
                        /* 这里确实是 void，并没错。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。
106 /*
107  * Interrupts are still disabled. Do necessary setups, then
108  * enable them
109  */
    /*
     * 此时中断仍被禁止着，做完必要的设置后就将其开启。
     */
     // 下面这段代码用于保存：
     // 根设备号  ➔ROOT_DEV；  高速缓存末端地址➔buffer_memory_end；
     // 机器内存数➔memory_end；主内存开始地址  ➔main_memory_start；
110         ROOT_DEV = ORIG_ROOT_DEV;               // ROOT_DEV 定义在 fs/super.c,29 行。
111         drive_info = DRIVE_INFO;                // 复制 0x90080 处的硬盘参数表。
112         memory_end = (1<<20) + (EXT_MEM_K<<10); // 内存大小=1Mb 字节+扩展内存(k)*1024 字节。
113         memory_end &= 0xfffff000;               // 忽略不到 4Kb（1 页）的内存数。
114         if (memory_end > 16*1024*1024)          // 如果内存超过 16Mb，则按 16Mb 计。
115                 memory_end = 16*1024*1024;
116         if (memory_end > 12*1024*1024)          // 如果内存>12Mb，则设置缓冲区末端=4Mb
117                 buffer_memory_end = 4*1024*1024;
118         else if (memory_end > 6*1024*1024)      // 否则如果内存>6Mb，则设置缓冲区末端=2Mb
119                 buffer_memory_end = 2*1024*1024;
120         else
121                 buffer_memory_end = 1*1024*1024;// 否则则设置缓冲区末端=1Mb
122         main_memory_start = buffer_memory_end;  // 主内存起始位置=缓冲区末端；
    // 如果定义了内存虚拟盘，则初始化虚拟盘。此时主内存将减少。参见 kernel/blk_drv/ramdisk.c。
123 #ifdef RAMDISK
124         main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
125 #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☺。
126         mem_init(main_memory_start,memory_end);
127         trap_init();            // 陷阱门（硬件中断向量）初始化。（kernel/traps.c，181）
128         blk_dev_init();         // 块设备初始化。     （kernel/blk_drv/ll_rw_blk.c，157）
129         chr_dev_init();         // 字符设备初始化。   （kernel/chr_drv/tty_io.c，347）
130         tty_init();             // tty 初始化。       （kernel/chr_drv/tty_io.c，105）
131         time_init();            // 设置开机启动时间➔startup_time（见 76 行）。
132         sched_init();           // 调度程序初始化(加载了任务 0 的 tr,ldtr)(kernel/sched.c,385)
133         buffer_init(buffer_memory_end); // 缓冲管理初始化，建内存链表等。（fs/buffer.c，348）
134         hd_init();              // 硬盘初始化。       （kernel/blk_drv/hd.c，343）
135         floppy_init();          // 软驱初始化。       （kernel/blk_drv/floppy.c，457）
136         sti();                  // 所有初始化工作都做完了，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。
137         move_to_user_mode();    // 移到用户模式下执行。（include/asm/system.h，第 1 行）
138         if (!fork()) {          /* we count on this going ok */
139                 init();         // 在新建的子进程（任务 1）中执行。
140         }
    // 下面代码开始以任务 0 的身份运行。
141 /*
142  *   NOTE!!   For any other task 'pause()' would mean we have to get a
143  * signal to awaken, but task0 is the sole exception (see 'schedule()')
144  * as task 0 gets activated at every idle moment (when no other tasks
145  * can run). For task0 'pause()' just means we go check if some other
```

```
146  * task can run, and if not we return here.
147  */
     /* 注意!! 对于任何其它的任务，'pause()'将意味着我们必须等待收到一个信号才会返
     * 回就绪运行态，但任务 0（task0）是唯一的例外情况（参见'schedule()'），因为任务 0 在
     * 任何空闲时间里都会被激活（当没有其它任务在运行时），因此对于任务 0'pause()'仅意味着
     * 我们返回来查看是否有其它任务可以运行，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它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。
148          for(;;) pause();
149  }
150
151  static int printf(const char *fmt, ...)
     // 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，参见各种标准 C 语言书籍。该子程序正好是 vsprintf 如何使用的一个例子。
     // 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。
152  {
153          va_list args;
154          int i;
155
156          va_start(args, fmt);
157          write(1,printbuf,i=vsprintf(printbuf, fmt, args));
158          va_end(args);
159          return i;
160  }
161
162  static char * argv_rc[] = { "/bin/sh", NULL };     // 调用执行程序时参数的字符串数组。
163  static char * envp_rc[] = { "HOME=/", NULL };      // 调用执行程序时的环境字符串数组。
164
165  static char * argv[] = { "-/bin/sh",NULL };        // 同上。
166  static char * envp[] = { "HOME=/usr/root", NULL };
     // 上面 165 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 不太一样。
167
     // 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，然后加载该程序并执行之。
168  void init(void)
169  {
170          int pid,i;
171
     // 这是一个系统调用。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。该函数是用 25 行上的宏定义的，对应函数是 sys_setup()，在 kernel/blk_drv/hd.c,71
行。
172          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。这里把它以读和写的方式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。
173          (void) open("/dev/tty0",O_RDWR,0);
174          (void) dup(0);                    // 复制句柄，产生句柄 1 号 -- stdout 标准输出设备。
```

```
175         (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。
176         printf("%d buffers = %d bytes buffer space\n\r",NR_BUFFERS,
177                 NR_BUFFERS*BLOCK_SIZE);
178         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 180-184 句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。所带参数和环境变量分别由 argv_rc 和 envp_rc 数组
    // 给出。关于 execve()请参见 fs/exec.c 程序，182 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。
179         if (!(pid=fork())) {
180                 close(0);
181                 if (open("/etc/rc",O_RDONLY,0))
182                         _exit(1);               // 如果打开文件失败，则退出(lib/_exit.c,10)。
183                 execve("/bin/sh",argv_rc,envp_rc);      // 替换成/bin/sh 程序并执行。
184                 _exit(2);                               // 若 execve()执行失败则退出。
185         }
    // 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。
186         if (pid>0)
187                 while (pid != wait(&i))
188                         /* 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 165-167 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。
189         while (1) {
190                 if ((pid=fork())<0) {
191                         printf("Fork failed in init\r\n");
192                         continue;
193                 }
194                 if (!pid) {                             // 新的子进程。
195                         close(0);close(1);close(2);
196                         setsid();                       // 创建一新的会话期，见后面说明。
197                         (void) open("/dev/tty0",O_RDWR,0);
198                         (void) dup(0);
199                         (void) dup(0);
200                         _exit(execve("/bin/sh",argv,envp));
201                 }
202                 while (1)
203                         if (pid == wait(&i))
204                                 break;
205                 printf("\n\rchild %d died with code %04x\n\r",pid,i);
206                 sync();                                 // 同步操作，刷新缓冲区。
207         }
208         _exit(0);       /* NOTE! _exit, not exit() */   /* 注意！是_exit()，不是 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。
209 }
210
```

4.2.3 其它信息

4.2.3.1 CMOS 信息

PC 机的 CMOS（complementary metal oxide semiconductor 互补金属氧化物半导体）内存实际上是由电池供电的 64 或 128 字节 RAM 内存块，是系统时钟芯片的一部分。有些机器还有更大的内存容量。

该 64 字节的 CMOS 原先在 IBM PC-XT 机器上用于保存时钟和日期信息，存放的格式是 BCD 码。由于这些信息仅用去 14 字节，剩余的字节就用来存放一些系统配置数据了。

CMOS 的地址空间是在基本地址空间之外的。因此其中不包括可执行的代码。它需要使用在端口 70h,71h 使用 IN 和 OUT 指令来访问。为了读取指定偏移位置的字节，首先需要使用 OUT 向端口 70h 发送指定字节的偏移值，然后使用 IN 指令从 71h 端口读取指定的字节信息。

这段程序中（行 70）把欲读取的字节地址或上了一个 80h 值是没有必要的。因为那时的 CMOS 内存容量还没有超过 128 字节，因此或上 80h 的操作是没有任何作用的。之所以会有这样的操作是因为当时 Linus 手头缺乏有关 CMOS 方面的资料，CMOS 中时钟和日期的偏移地址都是他逐步实验出来的，也许在他实验中将偏移地址或上 80h（并且还修改了其它地方）后正好取得了所有正确的结果，因此他的代码中也就有了这步不必要的操作。不过从 1.0 版本之后，该操作就被去除了(可参见 1.0 版内核程序 drivers/block/hd.c 第 42 行起的代码)。表 4–1 是 CMOS 内存信息的一张简表。

表 4－1 CMOS 64 字节信息简表

地址偏移值	内容说明	地址偏移值	内容说明
0x00	当前秒值 (实时钟)	0x11	保留
0x01	报警秒值	0x12	硬盘驱动器类型
0x02	当前分钟 (实时钟)	0x13	保留
0x03	报警分钟值	0x14	设备字节
0x04	当前小时值 (实时钟)	0x15	基本内存 (低字节)
0x05	报警小时值	0x16	基本内存 (高字节)
0x06	一周中的当前天 (实时钟)	0x17	扩展内存 (低字节)
0x07	一月中的当日日期 (实时钟)	0x18	扩展内存 (高字节)
0x08	当前月份 (实时钟)	0x19-0x2d	保留
0x09	当前年份 (实时钟)	0x2e	校验和 (低字节)
0x0a	RTC 状态寄存器 A	0x2f	校验和 (高字节)
0x0b	RTC 状态寄存器 B	0x30	1Mb 以上的扩展内存 (低字节)
0x0c	RTC 状态寄存器 C	0x31	1Mb 以上的扩展内存 (高字节)
0x0d	RTC 状态寄存器 D	0x32	当前所处世纪值
0x0e	POST 诊断状态字节	0x33	信息标志
0x0f	停机状态字节	0x34-0x3f	保留
0x10	磁盘驱动器类型		

4.2.3.2 调用 fork()创建新进程

fork 是一个系统调用函数。该系统调用复制当前进程，并在进程表中创建一个与原进程(被称为父进程）几乎完全一样的新表项，并执行同样的代码，但该新进程（这里被称为子进程）拥有自己的数据空间和环境参数。

在父进程中，调用 fork()返回的是子进程的进程标识号 PID，而在子进程中 fork()返回的将是 0 值，这样，虽然此时还是在同样一程序中执行，但已开始叉开，各自执行自己的那段代码。如果 fork()调用失败，则会返回小于 0 的值。如示意图 4-3 所示。

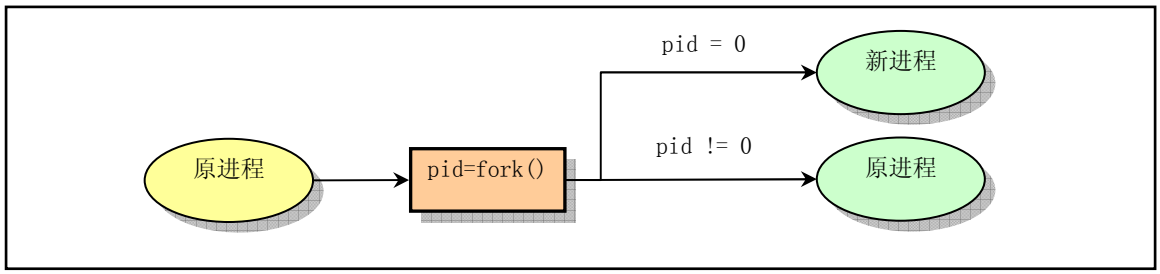

图 4-3 调用 fork()创建新进程

init 程序即是用 fork()调用的返回值来区分和执行不同的代码段的。上面代码中第 179 和 194 行是子进程的判断并开始子进程代码块的执行（利用 execve()系统调用执行其它程序，这里执行的是 sh），第 186 和 202 行是父进程执行的代码块。

4.2.3.3 关于会话期(session)的概念

在第 2 章我们说过，程序是一个可执行的文件，而进程（process）是一个执行中的程序实例。在内核中，每个进程都使用一个不同的大于零的正整数来标识，称为进程标识号 pid(Porcess ID)。而一个进程可以通过 fork()调用创建一个或多个子进程，这些进程就可以构成一个进程组。例如，对于下面在 shell 命令行上键入的一个管道命令，

```
[plinux root]# cat main.c | grep for | more
```

其中的每个命令：cat、grep 和 more 就都属于一个进程组。

进程组是一个或多个进程的集合。与进程类似，每个进程组都有一个唯一的进程组标识号 gid（Group ID）。进程组 gid 也是一个正整数。每一个进程组有一个称为组长的进程，组长进程就是其进程号 pid 等于进程组 gid 的进程。一个进程可以通过调用 setpgid()来参加一个现有的进程组或者创建一个新的进程组。进程组的概念有很多用途，但其中最常见的是我们在终端上向前台执行程序发出终止信号（通常是按 Ctrl-C 组合键），同时终止整个进程组中的所有进程。例如，如果我们向上述管道命令发出终止信号，则三个命令将同时终止执行。

而会话期（Session，或称为会话）则是一个或多个进程组的集合。通常情况下，用户登录后所执行的所有程序都属于一个会话期，而其登录 shell 则是会话期首进程（Session leader）。当我们退出登录（logout）时，所有属于我们这个会话期的进程都将被终止。这也是会话期概念的主要用途之一。setsid()函数就是用于建立一个新的会话期。通常该函数由环境初始化程序进行调用，见下节说明。进程、进程组和会话期之间的关系见图 4-4 所示。

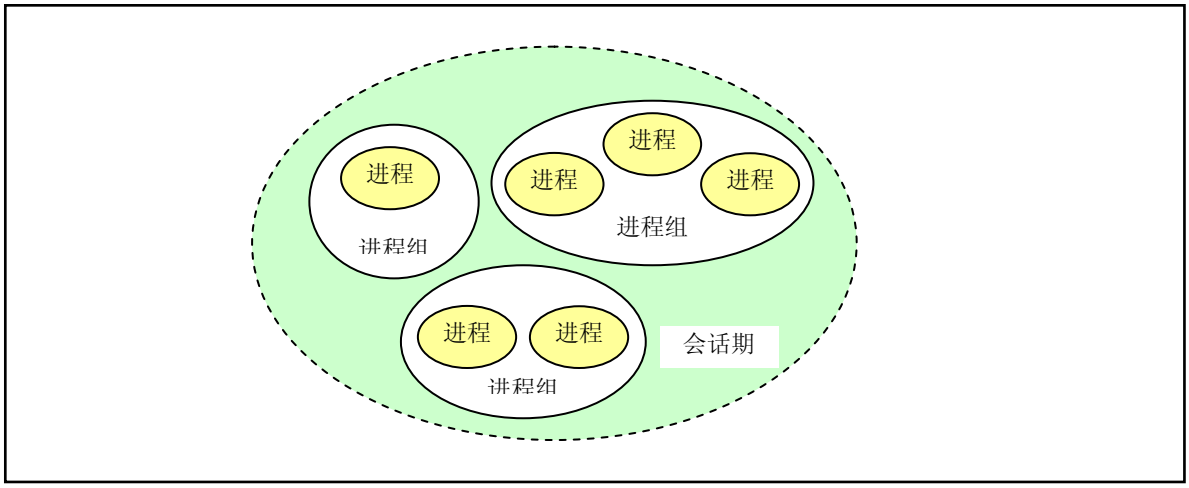

图 4-4 进程、进程组和会话期之间的关系

4.3 环境初始化工作

在内核系统初始化完毕之后，系统还需要根据具体配置执行进一步的环境初始化工作，才能真正具备一个常用系统所具备的一些工作环境。在前面的第 183 行和 200 行上，init()函数直接开始执行了命令解释程序（shell 程序）/bin/sh，而在实际可用的系统中却并非如此。为了能具有登录系统的处理和多人同时使用系统的能力，通常的系统是在这里或类似地方，执行系统环境初始化程序 init.c，而此程序会根据系统/etc/目录中配置文件的设置信息，对系统中支持的每个终端设备创建子进程，并在子进程中运行终端初始化设置程序 agetty（统称 getty 程序），getty 程序则会在终端上显示用户登录提示信息“login:”。当用户键入了用户名后，getty 替换去执行 login 程序。login 程序在验证了用户输入口令的正确性以后，最终调用 shell 程序，并进入 shell 交互工作界面。它们之间的执行关系见图 4-5 所示。

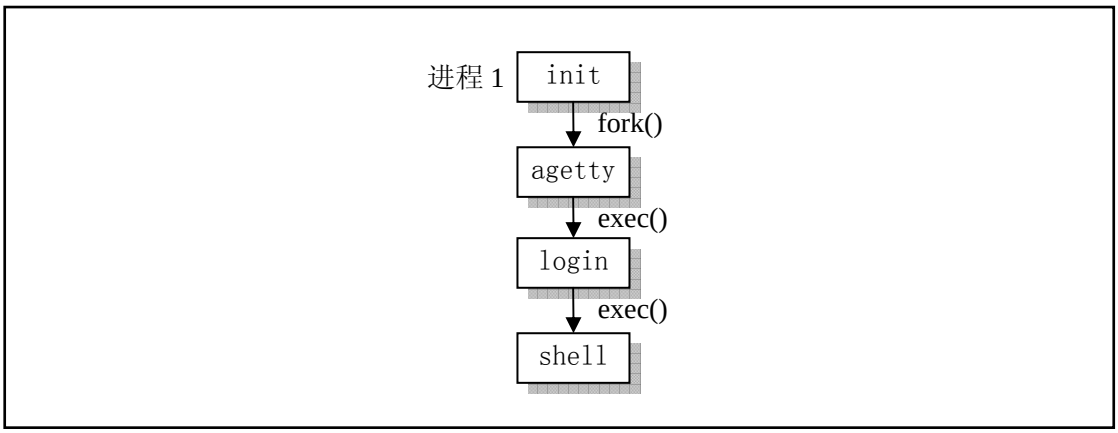

图 4-5 有关环境初始化的程序

虽然这几个程序（init, getty, login, shell）并不属于内核范畴，但对这几个程序的作用有一些基本了解会促进对内核为什么提供那么多功能的理解。

init 进程的主要任务是根据/etc/rc 文件中设置的信息，执行其中设置的命令，然后根据/etc/inittab 文件中的信息，为每一个允许登录的终端设备使用 fork()创建一个子进程，并在每个新创建的子进程中运

行 agetty[3]（getty）程序。而 init 进程则调用 wait()，进入等待子进程结束状态。每当它的一个子进程结束退出，它就会根据 wait()返回的 pid 号知道是哪个对应终端的子进程结束了，因此就会为相应终端设备再创建一个新的子进程，并在该子进程中重新执行 agetty 程序。这样，每个被允许的终端设备都始终有一个对应的进程为其等待处理。

在正常的操作下，init 确定 agetty 正在工作着以允许用户登录，并且收取孤立进程。孤立进程是指那些其父辈进程已结束的进程；在 Linux 中所有的进程必须属于单棵进程树，所以孤立进程必须被收取。当系统关闭时，init 负责杀死所有其它的进程，卸载所有的文件系统以及停止处理器的工作，以及任何它被配置成要做的工作。

getty 程序的主要任务是设置终端类型、属性、速度和线路规程。它打开并初始化一个 tty 端口，显示提示信息，并等待用户键入用户名。该程序只能由超级用户执行。通常，若/etc/issue 文本文件存在，则 getty 会首先显示其中的文本信息，然后显示登录提示信息（例如：plinux login: ），读取用户键入的登录名，并执行 login 程序。

login 程序则主要用于要求登录用户输入密码。根据用户输入的用户名，它从口令文件 passwd 中取得对应用户的登录项，然后调用 getpass()以显示”password:”提示信息，读取用户键入的密码，然后使用加密算法对键入的密码进行加密处理，并与口令文件中该用户项中 pw_passwd 字段作比较。如果用户几次键入的密码均无效，则 login 程序会以出错码 1 退出执行，表示此次登录过程失败。此时父进程（进程 init）的 wait()会返回该退出进程的 pid，因此会根据记录下来的信息再次创建一个子进程，并在该子进程中针对该终端设备再次执行 agetty 程序，重复上述过程。

如果用户键入的密码正确，则 login 就会把当前工作目录（Currend Work Directory）修改成口令文件中指定的该用户的起始工作目录。并把对该终端设备的访问权限修改成用户读/写和组写，设置进程的组 ID。然后利用所得到的信息初始化环境变量信息，例如起始目录（HOME=）、使用的 shell 程序（SHELL=）、用户名（USER=和 LOGNAME=）和系统执行程序的默认路径序列（PATH=）。接着显示/etc/motd 文件（message-of-the-day）中的文本信息，并检查并显示该用户是否有邮件的信息。最后 login 程序改变成登录用户的用户 ID 并执行口令文件中该用户项中指定的 shell 程序，如 bash 或 csh 等。

如果口令文件/etc/passwd 中该用户项中没有指定使用哪个 shell 程序，系统则会使用默认的/bin/sh 程序。如果口令文件中也没有为该用户指定用户起始目录的话，系统就会使用默认的根目录/。有关 login 程序的一些执行选项和特殊访问限制的说明，请参见 Linux 系统中的在线手册页（man 8 login）。

shell 程序是一个复杂的命令行解释程序，是当用户登录系统进行交互操作时执行的程序。它是用户与计算机进行交互操作的地方。它获取用户输入的信息，然后执行命令。用户可以在终端上向 shell 直接进行交互输入，也可以使用 shell 脚本文件向 shell 解释程序输入。

在登录过程中 login 开始执行 shell 时，所带参数 argv[0]的第一个字符是’-’，表示该 shell 是作为一个登录 shell 被执行。此时该 shell 程序会根据该字符，执行某些与登录过程相应的操作。登录 shell 会首先从/etc/profile 文件以及.profile 文件（若存在的话）读取命令并执行。如果在进入 shell 时设置了 ENV 环境变量，或者在登录 shell 的.profile 文件中设置了该变量，则 shell 下一步会从该变量命名的文件中读去命令并执行。因此用户应该把每次登录时都要执行的命令放在.profile 文件中，而把每次运行 shell 都要执行的命令放在 ENV 变量指定的文件中。设置 ENV 环境变量的方法是把下列语句放在你起始目录的.profile 文件中。

4.4 本章小结

对于 0.11 版内核，通过上面代码分析可知，只要根文件系统是一个 MINIX 文件系统，并且其中只要包含文件/etc/rc、/bin/sh、/dev/* 以及一些目录/etc/、/dev/、/bin/、/home/、/home/root/ 就可以构成一

[3] agetty - alternative Linux getty。

个最简单的根文件系统，让 Linux 运行起来。

从这里开始，对于后续章节的阅读，可以将 main.c 程序作为一条主线进行，并不需要按章节顺序阅读。若读者对内存分页管理机制不了解，则建议首先阅读第 10 章内存管理的内容。

为了能比较顺利地理解以下各章内容，作者强力希望读者此时能再次复习 32 位保护模式运行的机制，详细阅读一下附录中所提供的有关内容，或者参考 Intel 80x86 的有关书籍，把保护模式下的运行机制彻底弄清楚，然后再继续阅读。

如果您按章节顺序顺利地阅读到这里，那么您对 Linux 系统内核的初始化过程应该已经有了大致的了解。但您可能还会提出这样的问题：“在生成了一系列进程之后，系统是如何分时运行这些进程或者说如何调度这些进程运行的呢？也即‘轮子’是怎样转起来的呢？”。答案并不复杂：内核是通过执行 sched.c 程序中的调度函数 schedule()和 system_call.s 中的定时时钟中断过程_timer_interrupt 来操作的。内核设定每 10 毫秒发出一次时钟中断，并在该中断过程中，通过调用 do_timer()函数检查所有进程的当前执行情况来确定进程的下一步状态。

对于进程在执行过程中由于想用的资源暂时缺乏而临时需要等待一会时，它就会在系统调用中通过 sleep_on()类函数间接地调用 schedule()函数，将 CPU 的使用权自愿地移交给别的进程使用。至于系统接下来会运行哪个进程，则完全由 schedule()根据所有进程的当前状态和优先权决定。对于一直在可运行状态的进程，当时钟中断过程判断出它运行的时间片已被用完时，就会在 do_timer()中执行进程切换操作，该进程的 CPU 使用权就会被不情愿地剥夺，让给别的进程使用。

调度函数 schedule()和时钟中断过程即是下一章中的主题之一。