第2章 Linux 内核体系结构

本章首先概要介绍了 Linux 内核的编制模式和体系结构，然后详细描述了 Linux 内核源代码目录中组织形式以及子目录中各个代码文件的主要功能以及基本调用的层次关系。接下来就直接切入正题，从内核源文件 Linux/目录下的第一个文件 Makefile 开始，对每一行代码进行详细注释说明。本章内容可以看作是对内核源代码的总结概述，可以作为阅读后续章节的参考信息。

一个完整可用的操作系统主要由 4 部分组成：硬件、操作系统内核、操作系统服务和用户应用程序，见图 2-1 所示。用户应用程序是指那些字处理程序、Internet 浏览器程序或用户自行编制的各种应用程序；操作系统服务程序是指那些向用户提供的服务被看作是操作系统部分功能的程序。在 Linux 操作系统上，这些程序包括 X 窗口系统、shell 命令解释系统以及那些内核编程接口等系统程序；操作系统内核程序即是本书所感兴趣的部分，它主要用于对硬件资源的抽象和访问调度。

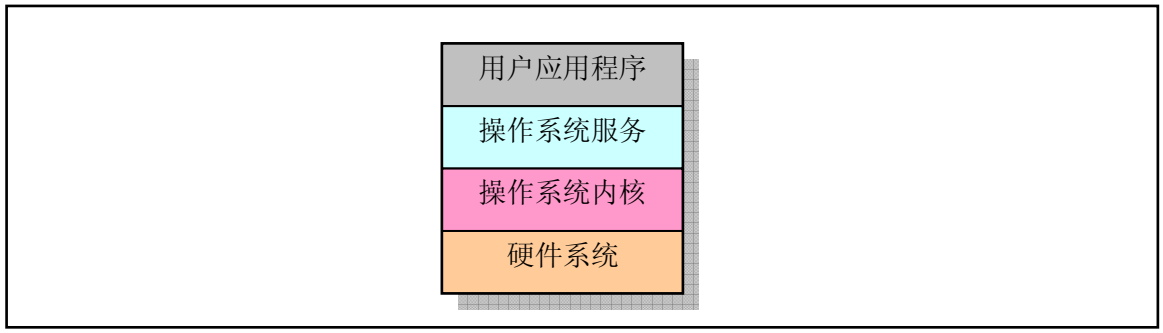

图 2-1 操作系统组成部分。

Linux 内核的主要用途就是为了与计算机硬件进行交互，实现对硬件部件的编程控制和接口操作，调度对硬件资源的访问，并为计算机上的用户程序提供一个高级的执行环境和对硬件的虚拟接口。
在本章内容中，我们首先基于 Linux 0.11 版的内核源代码，简明地描述 Linux 内核的基本体系结构、主要构成模块。然后对源代码中出现的几个重要数据结构进行说明。最后描述了构建 Linux 0.11 内核编译实验环境的方法。

2.1 Linux 内核模式

目前，操作系统内核的结构模式主要可分为整体式的单内核模式和层次式的微内核模式。而本书所注释的 Linux 0.11 内核，则是采用了单内核模式。单内核模式的主要优点是内核代码结构紧凑、执行速度快，不足之处主要是层次结构性不强。

在单内核模式的系统中，操作系统所提供服务的流程为：应用主程序使用指定的参数值执行系统调用指令(int x80)，使 CPU 从用户态（User Mode）切换到核心态（Kernel Model），然后操作系统根据具体的参数值调用特定的系统调用服务程序，而这些服务程序则根据需要再底层的一些支持函数以完成特定的功能。在完成了应用程序所要求的服务后，操作系统又从核心态切换回用户态，返回到应用程序中继续执行后面的指令。因此概要地讲，单内核模式的内核也可粗略地分为三个层次：调用服务的主程序层、

执行系统调用的服务层和支持系统调用的底层函数。见图 2-2 所示。

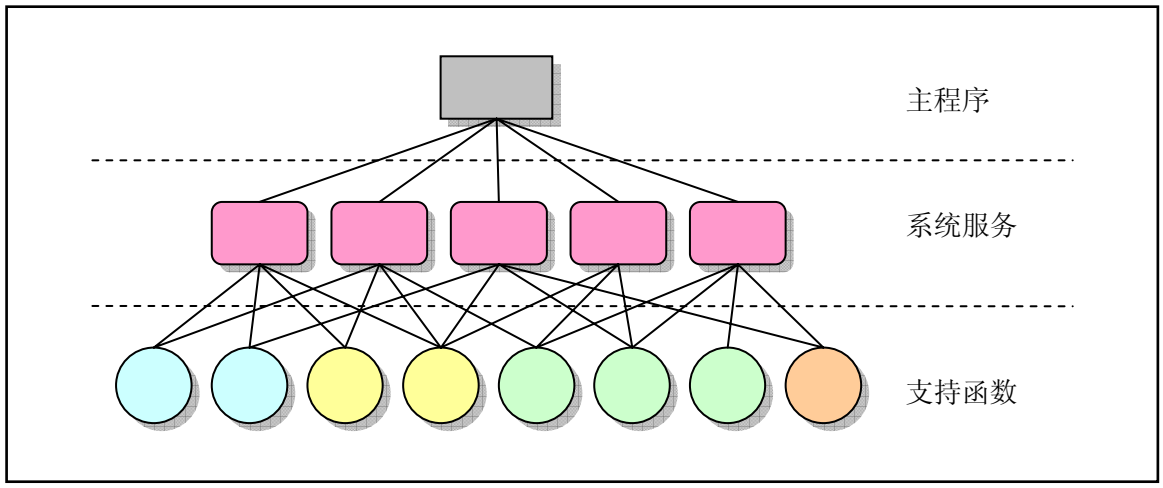

图 2-2 单内核模式的简单结构模型

2.2 Linux 内核系统体系结构

Linux 内核主要由 5 个模块构成，它们分别是：进程调度模块、内存管理模块、文件系统模块、进程间通信模块和网络接口模块。

进程调度模块用来负责控制进程对 CPU 资源的使用。所采取的调度策略是各进程能够公平合理地访问 CPU，同时保证内核能及时地执行硬件操作。内存管理模块用于确保所有进程能够安全地共享机器主内存区，同时，内存管理模块还支持虚拟内存管理方式，使得 Linux 支持进程使用比实际内存空间更多的内存容量。并可以利用文件系统把暂时不用的内存数据块会被交换到外部存储设备上去，当需要时再交换回来。文件系统模块用于支持对外部设备的驱动和存储。虚拟文件系统模块通过向所有的外部存储设备提供一个通用的文件接口，隐藏了各种硬件设备的不同细节。从而提供并支持与其它操作系统兼容的多种文件系统格式。进程间通信模块子系统用于支持多种进程间的信息交换方式。网络接口模块提供对多种网络通信标准的访问并支持许多网络硬件。

这几个模块之间的依赖关系见图 2-3 所示。其中的连线代表它们之间的依赖关系，虚线和虚框部分表示 Linux 0.11 中还未实现的部分（从 Linux 0.95 版才开始逐步实现虚拟文件系统，而网络接口的支持到 0.96 版才有）。

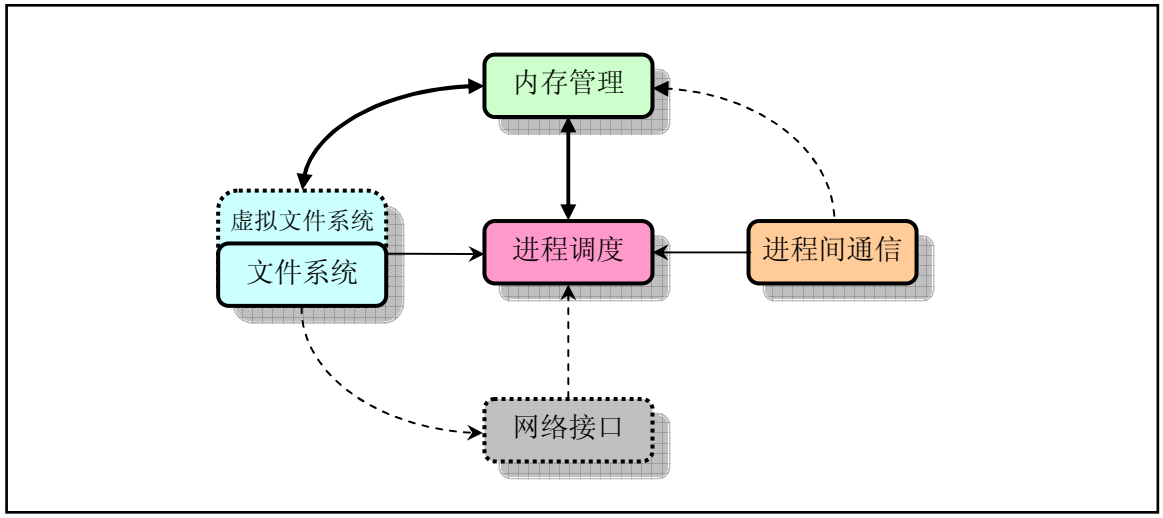

图 2-3Linux 内核系统模块结构及相互依赖关系

由图可以看出，所有的模块都与进程调度模块存在依赖关系。因为它们都需要依靠进程调度程序来挂起（暂停）或重新运行它们的进程。通常，一个模块会在等待硬件操作期间被挂起，而在操作完成后才可继续运行。例如，当一个进程试图将一数据块写到软盘上去时，软盘驱动程序就可能在启动软盘旋转期间将该进程置为挂起等待状态，而在软盘进入到正常转速后再使得该进程能继续运行。另外 3 个模块也是由于类似的原因而与进程调度模块存在依赖关系。

其它几个依赖关系有些不太明显，但同样也很重要。进程调度子系统需要使用内存管理器来调整一特定进程所使用的物理内存空间。进程间通信子系统则需要依靠内存管理器来支持共享内存通信机制。这种通信机制允许两个进程访问内存的同一个区域以进行进程间信息的交换。虚拟文件系统也会使用网络接口来支持网络文件系统（NFS），同样也能使用内存管理子系统来提供内存虚拟盘（ramdisk）设备。而内存管理子系统也会使用文件系统来支持内存数据块的交换操作。

若从单内核模式结构模型出发，我们还可以根据 Linux 0.11 内核源代码的结构将内核主要模块绘制成图 2-4 所示的框图结构。

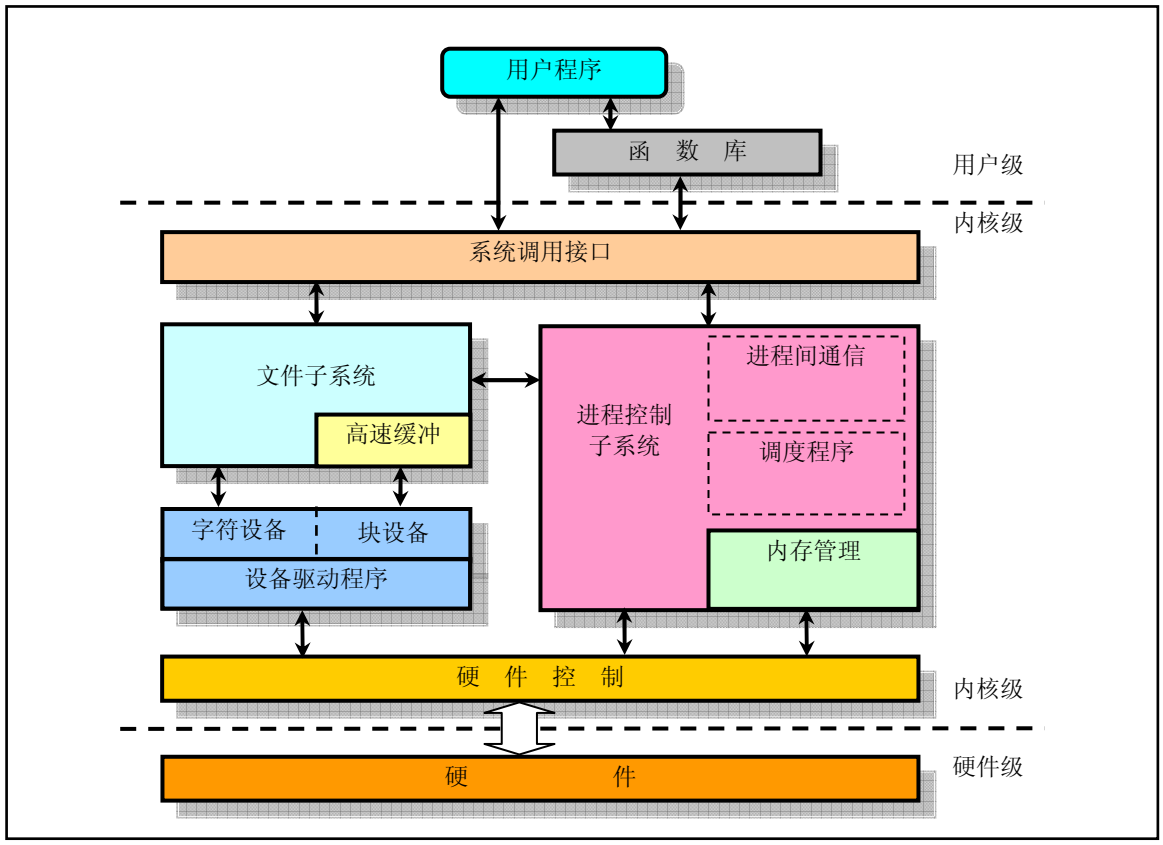

图 2-4 内核结构框图

其中内核级中的几个方框，除了硬件控制方框以外，其它粗线方框分别对应内核源代码的目录组织结构。

除了这些图中已经给出的依赖关系以外，所有这些模块还会依赖于内核中的通用资源。这些资源包括内核所有子系统都会调用的内存分配和收回函数、打印警告或出错信息函数以及一些系统调试函数。

2.3 中断机制

在使用 80X86 组成的 PC 机中，采用了两片 8259A 可编程中断控制芯片。每片可以管理 8 个中断源。通过多片的级联方式，能构成最多管理 64 个中断向量的系统。在 PC/AT 系列兼容机中，使用了两片 8259A 芯片，共可管理 15 级中断向量。其级连示意图见图 2-5 所示。其中从芯片的 INT 引脚连接到主芯片的 IR2 引脚上。主 8259A 芯片的端口基地址是 0x20，从芯片是 0xA0。

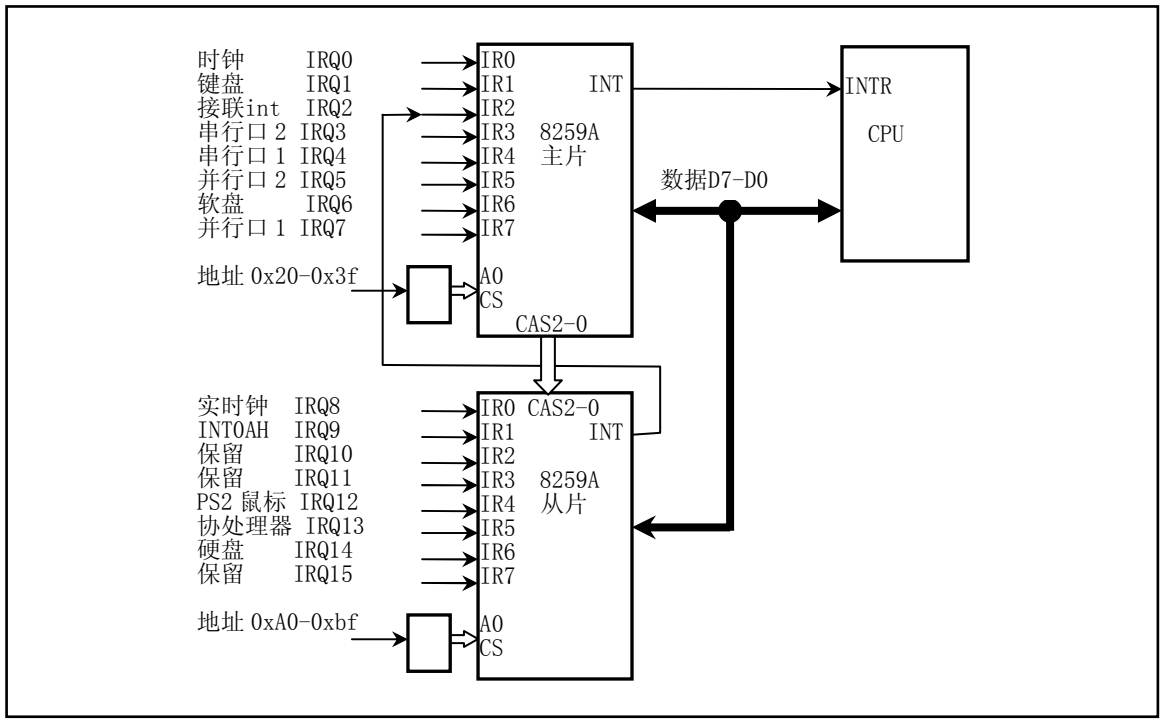

图 2-5 PC/AT 微机级连式 8259 控制系统

在总线控制器控制下，8259A 芯片可以处于编程状态和操作状态。编程状态是 CPU 使用 IN 或 OUT 指令对 8259A 芯片进行初始化编程的状态。一旦完成了初始化编程，芯片即进入操作状态，此时芯片即可随时响应外部设备提出的中断请求（IRQ0 – IRQ15）。通过中断判优选择，芯片将选中当前最高优先级的中断请求作为中断服务对象，并通过 CPU 引脚 INT 通知 CPU 外中断请求的到来，CPU 响应后，芯片从数据总线 D7-D0 将编程设定的当前服务对象的中断号送出，CPU 由此获取对应的中断向量值，并执行中断服务程序。

对于 Linux 内核来说，中断信号通常分为两类：硬件中断和软件中断(异常)。每个中断是由 0-255 之间的一个数字来标识。对于中断 int0--int31(0x00--0x1f)，每个中断的功能由 Intel 公司固定设定或保留用, 属于软件中断，但 Intel 公司称之为异常。因为这些中断是在 CPU 执行指令时探测到异常情况而引起的。通常还可分为故障(Fault)和陷阱(traps)两类。中断 int32--int255 (0x20--0xff)可以由用户自己设定。在 Linux 系统中，则将 int32--int47(0x20--0x2f)对应于 8259A 中断控制芯片发出的硬件中断请求信号 IRQ0-IRQ15，并把程序编程发出的系统调用(system_call)中断设置为 int128(0x80)。

2.4 系统定时

在 Linux 0.11 内核中，PC 机的可编程定时芯片 Intel 8253 被设置成每隔 10 毫秒就发出一个时钟中断（IRQ0）信号。这个时间节拍就是系统运行的脉搏，我们称之为 1 个系统滴答。因此每经过 1 个滴答就会调用一次时钟中断处理程序（timer_interrupt）。该处理程序主要用来通过 jiffies 变量来累计自系统启动以来经过的时钟滴答数。每当发生一次时钟中断该值就增 1。然后从被中断程序的段选择符中取得当前特权级 CPL 作为参数调用 do_timer()函数。

do_timer()函数则根据特权级对当前进程运行时间作累计。如果 CPL=0，则表示进程是运行在内核态时被中断，因此把进程的内核运行时间统计值 stime 增 1，否则把进程用户态运行时间统计值增 1。如果程序添加过定时器，则对定时器链表进行处理。若某个定时器时间到（递减后等于 0），则调用该定时器的处理函数。然后对当前进程运行时间进行处理，把当前进程运行时间片减 1。如果此时当前进程时间

片并还大于 0，表示其时间片还没有用完，于是就退出 do_timer()继续运行当前进程。如果此时进程时间片已经递减为 0，表示该进程已经用完了此次使用 CPU 的时间片，于是程序就会根据被中断程序的级别来确定进一步处理的方法。若被中断的当前进程是工作在用户态的（特权级别大于 0），则 do_timer()就会调用调度程序 schedule()切换到其它进程去运行。如果被中断的当前进程工作在内核态，也即在内核程序中运行时被中断，则 do_timer()会立刻退出。因此这样的处理方式决定了 Linux 系统在内核态运行时不会被调度程序切换。内核态程序是不可抢占的，但当处于用户态程序中运行时则是可以被抢占的。

2.5 Linux 进程控制

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。利用分时技术，在 Linux 操作系统上同时可以运行多个进程。分时技术的基本原理是把 CPU 的运行时间划分成一个个规定长度的时间片，让每个进程在一个时间片内运行。当进程的时间片用完时系统就利用调度程序切换到另一个进程去运行。因此实际上对于具有单个 CPU 的机器来说某一时刻只能运行一个进程。但由于每个进程运行的时间片很短（例如 15 个系统滴答=150 毫秒），所以表面看来好象所有进程在同时运行着。

对于 Linux 0.11 内核来讲，系统最多可有 64 个进程同时存在。除了第一个进程是“手工”建立以外，其余的都是进程使用系统调用 fork 创建的新进程，被创建的进程称为子进程（child process），创建者，则称为父进程（parent process）。内核程序使用进程标识号（process ID，pid）来标识每个进程。进程由可执行的指令代码、数据和堆栈区组成。进程中的代码和数据部分分别对应一个执行文件中的代码段、数据段。每个进程只能执行自己的代码和访问自己的数据及堆栈区。进程之间相互之间的通信需要通过系统调用来进行。对于只有一个 CPU 的系统，在某一时刻只能有一个进程正在运行。内核通过调度程序分时调度各个进程运行。

Linux 系统中，一个进程可以在内核态（kernel mode）或用户态（user mode）下执行，因此，Linux 内核堆栈和用户堆栈是分开的。用户堆栈用于进程在用户态下临时保存调用函数的参数、局部变量等数据。内核堆栈则含有内核程序执行函数调用时的信息。

2.5.1 任务数据结构

内核程序通过进程表对进程进行管理，每个进程在进程表中占有一项。在 Linux 系统中，进程表项是一个 task_struct 任务结构指针。任务数据结构定义在头文件 include/linux/sched.h 中。有些书上称其为进程控制块 PCB（Process Control Block）或进程描述符 PD（Processor Descriptor）。其中保存着用于控制和管理进程的所有信息。主要包括进程当前运行的状态信息、信号、进程号、父进程号、运行时间累计值、正在使用的文件和本任务的局部描述符以及任务状态段信息。该结构每个字段的具体含义如下所示。

```
struct task_struct {
        long state                      //任务的运行状态（-1 不可运行，0 可运行(就绪)，>0 已停止）。
        long counter                    // 任务运行时间计数(递减)（滴答数），运行时间片。
        long priority                   // 运行优先数。任务开始运行时 counter=priority，越大运行越长。
        long signal                     // 信号。是位图，每个比特位代表一种信号，信号值=位偏移值+1。
        struct sigaction sigaction[32]  // 信号执行属性结构，对应信号将要执行的操作和标志信息。
        long blocked                    // 进程信号屏蔽码（对应信号位图）。
        int exit_code                   // 任务执行停止的退出码，其父进程会取。
        unsigned long start_code        // 代码段地址。
        unsigned long end_code          // 代码长度（字节数）。
        unsigned long end_data          // 代码长度 + 数据长度（字节数）。
        unsigned long brk               // 总长度（字节数）。
        unsigned long start_stack       // 堆栈段地址。
```

```
    long pid                    // 进程标识号(进程号)。
    long father                 // 父进程号。
    long pgrp                   // 父进程组号。
    long session                // 会话号。
    long leader                 // 会话首领。
    unsigned short uid          // 用户标识号（用户 id）。
    unsigned short euid         // 有效用户 id。
    unsigned short suid         // 保存的用户 id。
    unsigned short gid          // 组标识号（组 id）。
    unsigned short egid         // 有效组 id。
    unsigned short sgid         // 保存的组 id。
    long alarm                  // 报警定时值（滴答数）。
    long utime                  // 用户态运行时间（滴答数）。
    long stime                  // 系统态运行时间（滴答数）。
    long cutime                 // 子进程用户态运行时间。
    long cstime                 // 子进程系统态运行时间。
    long start_time             // 进程开始运行时刻。
    unsigned short used_math    // 标志：是否使用了协处理器。
    int tty                     // 进程使用 tty 的子设备号。-1 表示没有使用。
    unsigned short umask        // 文件创建属性屏蔽位。
    struct m_inode * pwd        // 当前工作目录 i 节点结构。
    struct m_inode * root       // 根目录 i 节点结构。
    struct m_inode * executable // 执行文件 i 节点结构。
    unsigned long close_on_exec // 执行时关闭文件句柄位图标志。（参见 include/fcntl.h）
    struct file * filp[NR_OPEN] // 文件结构指针表，最多 32 项。表项号即是文件描述符的值。
    struct desc_struct ldt[3]  // 任务局部描述符表。0-空，1-代码段 cs，2-数据和堆栈段 ds&ss。
    struct tss_struct tss       // 进程的任务状态段信息结构。
};
```

当一个进程在执行时，CPU 的所有寄存器中的值、进程的状态以及堆栈中的内容被称为该进程的上下文。当内核需要切换（switch）至另一个进程时，它就需要保存当前进程的所有状态，也即保存当前进程的上下文，以便在再次执行该进程时，能够恢复到切换时的状态执行下去。在 Linux 中，当前进程上下文均保存在进程的任务数据结构中。在发生中断时，内核就在被中断进程的上下文中，在内核态下执行中断服务例程。但同时会保留所有需要用到的资源，以便中断服务结束时能恢复被中断进程的执行。

2.5.2 进程运行状态

一个进程在其生存期内，可处于一组不同的状态下，称为进程状态。见图 2-6 所示。进程状态保存在进程任务结构的 state 字段中。当进程正在等待系统中的资源而处于等待状态时，则称其处于睡眠等待状态。在 Linux 系统中，睡眠等待状态被分为可中断的和不可中断的等待状态。

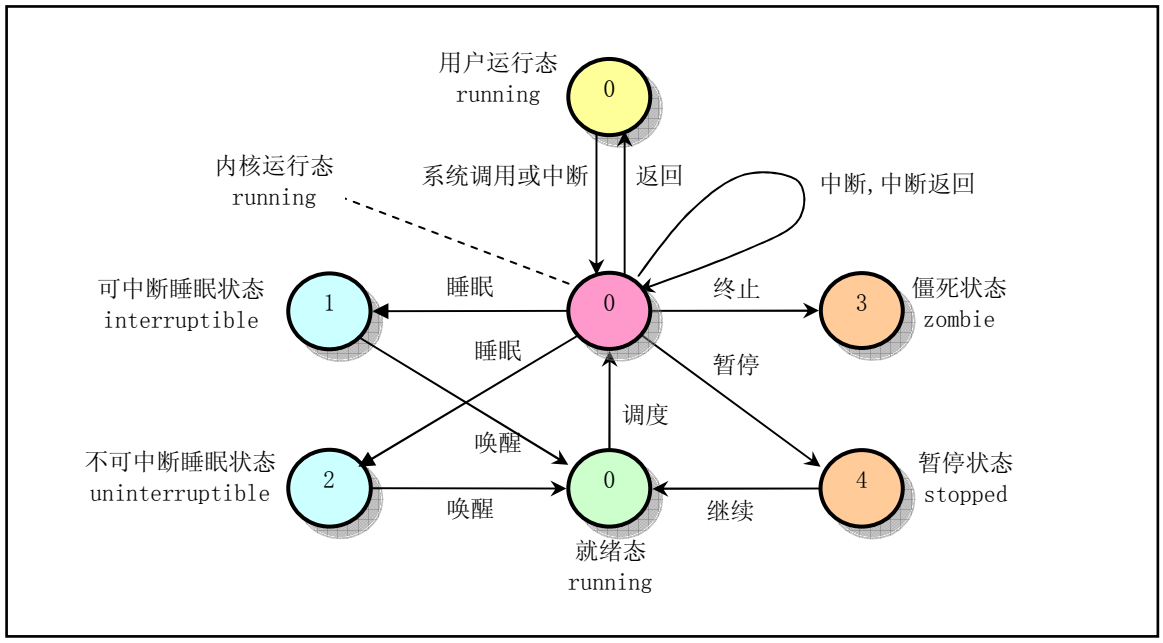

图 2-6 进程状态及转换关系

◆运行状态（TASK_RUNNING）

当进程正在被 CPU 执行，或已经准备就绪随时可由调度程序执行，则称该进程为处于运行状态（running）。进程可以在内核态运行，也可以在用户态运行。当系统资源已经可用时，进程就被唤醒而进入准备运行状态，该状态称为就绪态。这些状态（图中中间一列）在内核中表示方法相同，都被成为处于 TASK_RUNNING 状态。

◆可中断睡眠状态（TASK_INTERRUPTIBLE）

当进程处于可中断等待状态时，系统不会调度该进行执行。当系统产生一个中断或者释放了进程正在等待的资源，或者进程收到一个信号，都可以唤醒进程转换到就绪状态（运行状态）。

◆不可中断睡眠状态（TASK_UNINTERRUPTIBLE）

与可中断睡眠状态类似。但处于该状态的进程只有被使用 wake_up()函数明确唤醒时才能转换到可运行的就绪状态。

◆暂停状态（TASK_STOPPED）

当进程收到信号 SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU 时就会进入暂停状态。可向其发送 SIGCONT 信号让进程转换到可运行状态。在 Linux 0.11 中，还未实现对该状态的转换处理。处于该状态的进程将被作为进程终止来处理。

◆僵死状态（TASK_ZOMBIE）

当进程已停止运行，但其父进程还没有询问其状态时，则称该进程处于僵死状态。

当一个进程的运行时间片用完，系统就会使用调度程序强制切换到其它的进程去执行。另外，如果进程在内核态执行时需要等待系统的某个资源，此时该进程就会调用 sleep_on()或 sleep_on_interruptible()自愿地放弃 CPU 的使用权，而让调度程序去执行其它进程。进程则进入睡眠状态（TASK_UNINTERRUPTIBLE 或 TASK_INTERRUPTIBLE）。

只有当进程从“内核运行态”转移到“睡眠状态”时，内核才会进行进程切换操作。在内核态下运行的进程不能被其它进程抢占，而且一个进程不能改变另一个进程的状态。为了避免进程切换时造成内核数据错误，内核在执行临界区代码时会禁止一切中断。

2.5.3 进程初始化

在 boot/目录中引导程序把内核从磁盘上加载到内存中，并让系统进入保护模式下运行后，就开始执行系统初始化程序 init/main.c。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。在进程 1 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。

“移动到任务 0 中执行”这个过程由宏 move_to_user_mode（include/asm/system.h）完成。它把 main.c 程序执行流从内核态（特权级 0）移动到了用户态（特权级 3）的任务 0 中继续运行。在移动之前，系统在对调度程序的初始化过程（sched_init()）中，首先对任务 0 的运行环境进行了设置。这包括人工预先设置好任务 0 数据结构各字段的值（include/linux/sched.h）、在全局描述符表中添入任务 0 的任务状态段（TSS）描述符和局部描述符表（LDT）的段描述符，并把它们分别加载到任务寄存器 tr 和局部描述符表寄存器 ldtr 中。

这里需要强调的是，内核初始化是一个特殊过程，内核初始化代码也即是任务 0 的代码。从任务 0 数据结构中设置的初始数据可知，任务 0 的代码段和数据段的基址是 0、段限长是 640KB。而内核代码段和数据段的基址是 0、段限长是 16MB，因此任务 0 的代码段和数据段分别包含在内核代码段和数据段中。内核初始化程序 main.c 也即是任务 0 中的代码，只是在移动到任务 0 之前系统正以内核态特权级 0 运行着 main.c 程序。宏 move_to_user_mode 的功能就是把运行特权级从内核态的 0 级变换到用户态的 3 级，但是仍然继续执行原来的代码指令流。

在移动到任务 0 的过程中，宏 move_to_user_mode 使用了中断返回指令造成特权级改变的方法。该方法的主要思想是在堆栈中构筑中断返回指令需要的内容，把返回地址的段选择符设置成任务 0 代码段选择符，其特权级为 3。此后执行中断返回指令 iret 时将导致系统 CPU 从特权级 0 跳转到外层的特权级 3 上运行。参见图 2-7 所示的特权级发生变化时中断返回堆栈结构示意图。

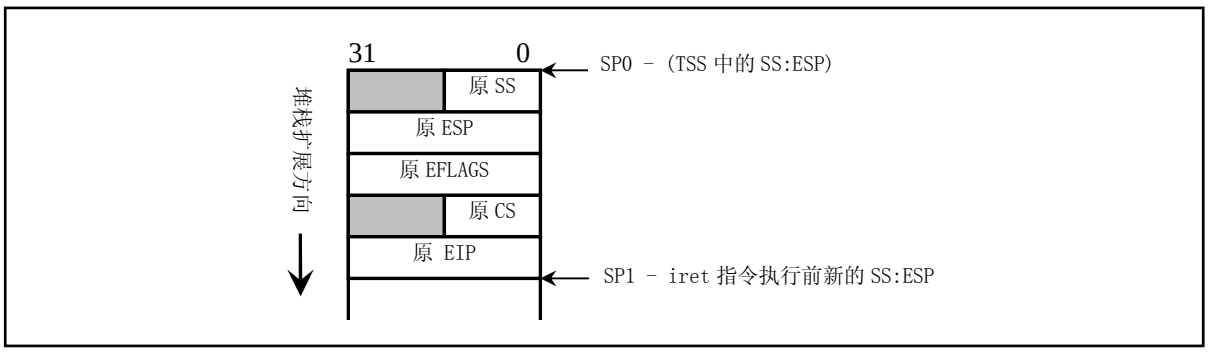

图 2-7 特权级发生变化时中断返回堆栈结构示意图

宏 move_to_user_mode 首先往内核堆栈中压入任务 0 数据段选择符和内核堆栈指针。然后压入标志寄存器内容。最后压入任务 0 代码段选择符和执行中断返回后需要执行的下一条指令的偏移位置。该偏移位置是 iret 后的一条指令处。

当执行 iret 指令时，CPU 把返回地址送入 CS:EIP 中，同时弹出堆栈中标志寄存器内容。由于 CPU 判断出目的代码段的特权级是 3，与当前内核态的 0 级不同。于是 CPU 会把堆栈中的堆栈段选择符和堆栈指针弹出到 SS:ESP 中。由于特权级发上了变化，段寄存器 DS、ES、FS 和 GS 的值变得无效，此时 CPU 会把这些段寄存器清零。因此在执行了 iret 指令后需要重新加载这些段寄存器。此后，系统就开始

以特权级 3 运行在任务 0 的代码上。所使用的用户态堆栈还是原来在移动之前使用的堆栈。而其内核态堆栈则被指定为其任务数据结构所在页面的顶端开始（PAGE_SIZE + (long)&init_task）。由于以后在创建新进程时，需要复制任务 0 的任务数据结构，包括其用户堆栈指针，因此要求任务 0 的用户态堆栈在创建任务 1（进程 1）之前保持“干净”状态。

2.5.4 创建新进程

Linux 系统中创建新进程使用 fork()系统调用。所有进程都是通过复制进程 0 而得到的，都是进程 0 的子进程。

在创建新进程的过程中，系统首先在任务数组中找出一个还没有被任何进程使用的空项（空槽）。如果系统已经有 64 个进程在运行，则 fork()系统调用会因为任务数组表中没有可用空项而出错返回。然后系统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。为了防止这个还未处理完成的新建进程被调度函数执行，此时应该立刻将新进程状态置为不可中断的等待状态（TASK_UNINTERRUPTIBLE）。

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，并设置初始运行时间片值为 15 个系统滴答数（150 毫秒）。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，则应将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，其中基地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。

2.5.5 进程调度

由前面描述可知，Linux 进程是抢占式的。被抢占的进程仍然处于 TASK_RUNNING 状态，只是暂时没有被 CPU 运行。进程的抢占发生在进程处于用户态执行阶段，在内核态执行时是不能被抢占的。

为了能让进程有效地使用系统资源，又能使进程有较快的响应时间，就需要对进程的切换调度采用一定的调度策略。在 Linux 0.11 中采用了基于优先级排队的调度策略。

调度程序

schedule()函数首先扫描任务数组。通过比较每个就绪态（TASK_RUNNING）任务的运行时间递减滴答计数 counter 的值来确定当前哪个进程运行的时间最少。哪一个的值大，就表示运行时间还不长，于是就选中该进程，并使用任务切换宏函数切换到该进程运行。

如果此时所有处于 TASK_RUNNING 状态进程的时间片都已经用完，系统就会根据每个进程的优先权值 priority，对系统中所有进程（包括正在睡眠的进程）重新计算每个任务需要运行的时间片值 counter。计算的公式是：

$$counter = \frac{counter}{2} + priority$$

然后 schdeule()函数重新扫描任务数组中所有处于 TASK_RUNNING 状态，重复上述过程，直到选择出一个进程为止。最后调用 switch_to()执行实际的进程切换操作。

如果此时没有其它进程可运行，系统就会选择进程 0 运行。对于 Linux 0.11 来说，进程 0 会调用 pause()把自己置为可中断的睡眠状态并再次调用 schedule()。不过在调度进程运行时，schedule()并不在意进程 0

处于什么状态。只要系统空闲就调度进程 0 运行。

进程切换

执行实际进程切换的任务由 switch_to()宏定义的一段汇编代码完成。在进行切换之前，switch_to()首先检查要切换到的进程是否就是当前进程，如果是则什么也不做，直接退出。否则就首先把内核全局变量 current 置为新任务的指针，然后长跳转到新任务的任务状态段 TSS 组成的地址处，造成 CPU 执行任务切换操作。此时 CPU 会把其所有寄存器的状态保存到当前任务寄存器 TR 中 TSS 段选择符所指向的当前进程任务数据结构的 tss 结构中，然后把新任务状态段选择符所指向的新任务数据结构中 tss 结构中的寄存器信息恢复到 CPU 中，系统就正式开始运行新切换的任务了。这个过程可参见图 2-8 所示。

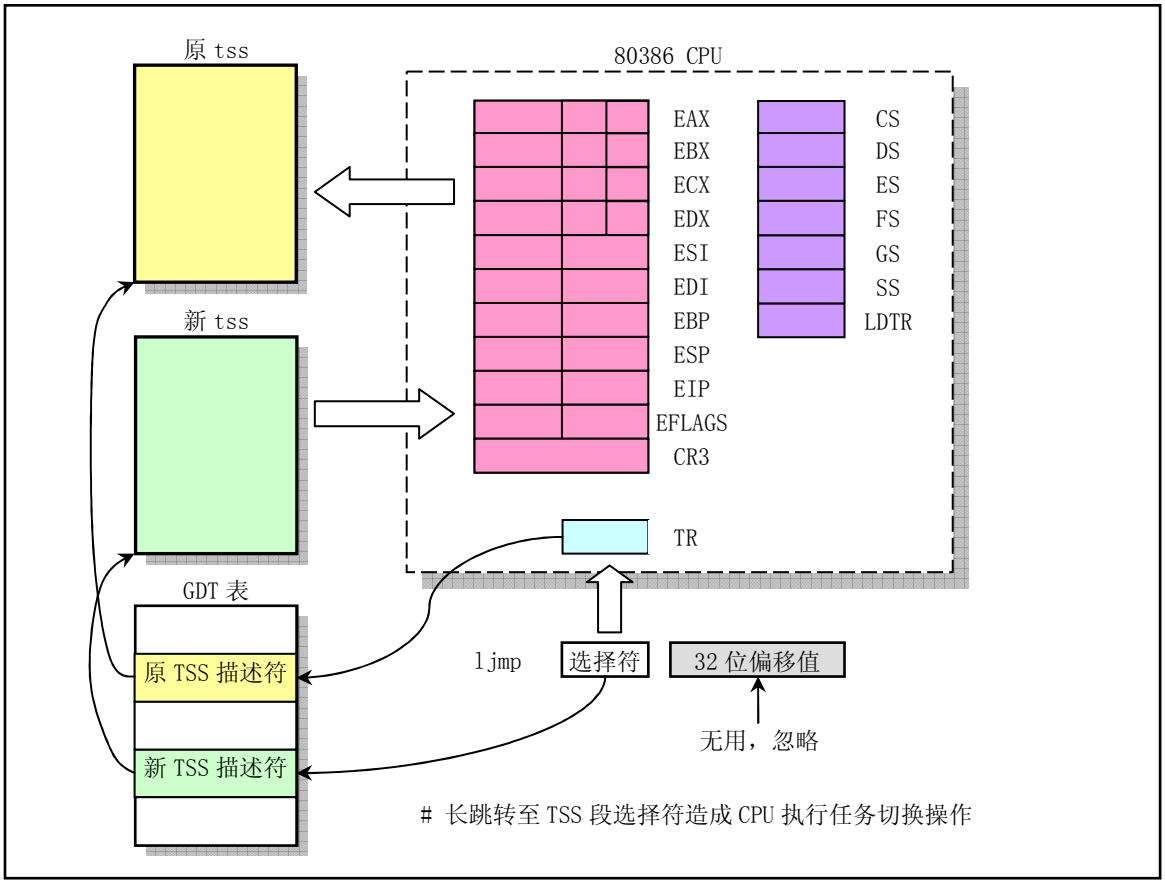

图 2-8 任务切换操作示意图

2.5.6 终止进程

当一个进程结束了运行或在半途中终止了运行，那么内核就需要释放该进程所占用的系统资源。这包括进程运行时打开的文件、申请的内存等。

当一个用户程序调用 exit()系统调用时，就会执行内核函数 do_exit()。该函数会首先释放进程代码段和数据段占用的内存页面，关闭进程打开着的所有文件，对进程使用的当前工作目录、根目录和运行程序的 i 节点进行同步操作。如果进程有子进程，则让 init 进程作为其所有子进程的父进程。如果进程是一个会话头进程并且有控制终端，则释放控制终端，并向属于该会话的所有进程发送挂断信号 SIGHUP，这通常会终止该会话中的所有进程。然后把进程状态置为僵死状态 TASK_ZOMBIE。并向其原父进程发送 SIGCHLD 信号，通知其某个子进程已经终止。最后 do_exit()调用调度函数去执行其它进程。由此可

见在进程被终止时，它的任务数据结构仍然保留着。因为其父进程还需要使用其中的信息。

在子进程在执行期间，父进程通常使用 wait()或 waitpid()函数等待其某个子进程终止。当等待的子进程被终止并处于僵死状态时，父进程就会把子进程运行所使用的时间累加到自己进程中。最终释放已终止子进程任务数据结构所占用的内存页面，并置空子进程在任务数组中占用的指针项。

2.6 Linux 内核对内存的使用方法

在 Linux 0.11 内核中，为了有效地使用机器中的物理内存，内存被划分成几个功能区域，见下图 2-9 所示。

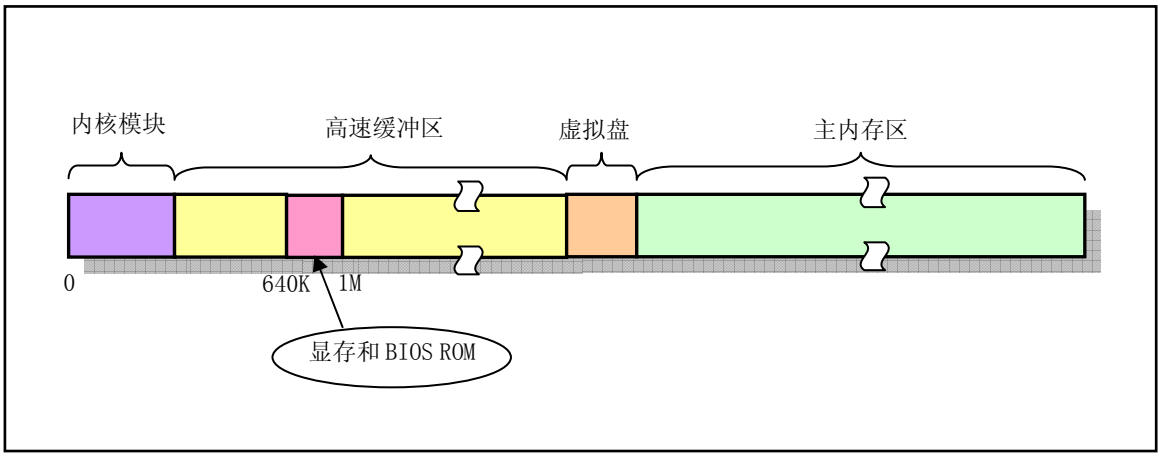

图 2-9 物理内存使用的功能分布图

其中，Linux 内核程序占据在物理内存的开始部分，接下来是用于供硬盘或软盘等块设备使用的高速缓冲区部分。当一个进程需要读取块设备中的数据时，系统会首先将数据读到高速缓冲区中；当有数据需要写到块设备上去时，系统也是先将数据放到高速缓冲区中，然后由块设备驱动程序写到设备上。最后部分是供所有程序可以随时申请使用的主内存区部分。内核程序在使用主内存区时，也同样要首先向内核的内存管理模块提出申请，在申请成功后方能使用。对于含有 RAM 虚拟盘的系统，主内存区头部还要划去一部分，供虚拟盘存放数据。

由于计算机系统中所含的实际物理内存容量是有限的，因此 CPU 中通常都提供了内存管理机制对系统中的内存进行有效的管理。在 Intel CPU 中，提供了两种内存管理（变换）系统：内存分段系统（Segmentation System）和分页系统（Paging System）。而分页管理系统是可选择的，由系统程序员通过编程来确定是否采用。为了能有效地使用这些物理内存，Linux 系统同时采用了 Intel CPU 的内存分段和分页管理机制。

在 Linux 0.11 内核中，在进行地址映射时，我们需要首先分清 3 种地址以及它们之间的变换概念：a. 程序（进程）的逻辑地址；b. CPU 的线性地址；c. 实际物理内存地址。

逻辑地址（Logical Address）是指有程序产生的与段相关的偏移地址部分。在 Intel 保护模式下即是指程序执行代码段限长内的偏移地址（假定代码段、数据段完全一样）。应用程序员仅需与逻辑地址打交道，而分段和分页机制对他来说是完全透明的，仅由系统编程人员涉及。

线性地址（Linear Address）是逻辑地址到物理地址变换之间的中间层。程序代码会产生逻辑地址，或者说是段中的偏移地址，加上相应段的基地址就生成了一个线性地址。如果启用了分页机制，那么线性地址可以再经变换以产生一个物理地址。若没有启用分页机制，那么线性地址直接就是物理地址。Intel 80386 的线性地址空间容量为 4G。

物理地址（Physical Address）是指出现在 CPU 外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果地址。如果启用了分页机制，那么线性地址会使用页目录和页表中的项变换成物理地址。如果没有启用分页机制，那么线性地址就直接成为物理地址了。

虚拟内存（Virtual Memory）是指计算机呈现出要比实际拥有的内存大得多的内存量。因此它允许程序员编制并运行比实际系统拥有的内存大得多的程序。这使得许多大型项目也能够在具有有限内存资源的系统上实现。一个很恰当的比喻是：你不需要很长的轨道就可以让一列火车从上海开到北京。你只需要足够长的铁轨（比如说 3 公里）就可以完成这个任务。采取的方法是把后面的铁轨立刻铺到火车的前面，只要你的操作足够快并能满足要求，列车就能象在一条完整的轨道上运行。这也就是虚拟内存管理需要完成的任务。在 Linux 0.11 内核中，给每个程序（进程）都划分了总容量为 64MB 的虚拟内存空间。因此程序的逻辑地址范围是 0x0000000 到 0x4000000。

有时我们也把逻辑地址称为虚拟地址。因为与虚拟内存空间的概念类似，逻辑地址也是与实际物理内存容量无关的。

在内存分段系统中，一个程序的逻辑地址是通过分段机制自动地映射（变换）到中间层的线性地址上。每次对内存的引用都是对内存段中内存的引用。当一个程序引用一个内存地址时，通过把相应的段基址加到程序员看得见的逻辑地址上就形成了一个对应的线性地址。此时若没有启用分页机制，则该线性地址就被送到 CPU 的外部地址总线上，用于直接寻址对应的物理内存。

若采用了分页机制，则此时线性地址只是一个中间结果，还需要使用分页机制进行变换，再最终映射到实际物理内存地址上。与分段机制类似，分页机制允许我们重新定向（变换）每次内存引用，以适应我们的特殊要求。使用分页机制最普遍的场合是当系统内存实际上被分成很多凌乱的块时,它可以建立一个大而连续的内存空间的映象，好让程序不用操心和管理这些分散的内存块。分页机制增强了分段机制的性能。页地址变换是建立在段变换基础之上的。任何分页机制的保护措施并不会取代段变换的保护措施而只是进行更进一步的检查操作。

因此，CPU 进行地址变换（映射）的主要目的是为了解决虚拟内存空间到物理内存空间的映射问题。虚拟内存空间的含义是指一种利用二级或外部存储空间，使程序能不受实际物理内存量限制而使用内存的一种方法。通常虚拟内存空间要比实际物理内存量大得多。

那么虚拟内存空间管理是怎样实现的呢？原理与上述列车运行的比喻类似。首先，当一个程序需要使用一块不存在的内存时（也即在内存页表项中已标出相应内存页面不在内存中），CPU 就需要一种方法来得知这个情况。这是通过 80386 的页错误异常中断来实现的。当一个进程引用一个不存在页面中的内存地址时，就会触发 CPU 产生页出错异常中断，并把引起中断的线性地址放到 CR2 控制寄存器中。因此处理该中断的过程就可以知道发生页异常的确切地址，从而可以把进程要求的页面从二级存储空间（比如硬盘上）加载到物理内存中。如果此时物理内存已经被全部占用，那么可以借助二级存储空间的一部分作为交换缓冲区（Swapper）把内存中暂时不使用的页面交换到二级缓冲区中，然后把要求的页面调入内存中。这也就是内存管理的缺页加载机制，在 Linux 0.11 内核中是在程序 mm/memory.c 中实现。

Intel CPU 使用段（Segment）的概念来对程序进行寻址。每个段定义了内存中的某个区域以及访问的优先级等信息。而每个程序都可有若干个内存段组成。程序的逻辑地址（或称为虚拟地址）即是用于寻址这些段和段中具体地址位置。在 Linux 0.11 中，程序逻辑地址到线性地址的变换过程使用了 CPU 的全局段描述符表 GDT 和局部段描述符表 LDT。由 GDT 映射的地址空间称为全局地址空间，由 LDT 映射的地址空间则称为局部地址空间，而这两者构成了虚拟地址的空间。具体的使用方式见图 2-10 所示。

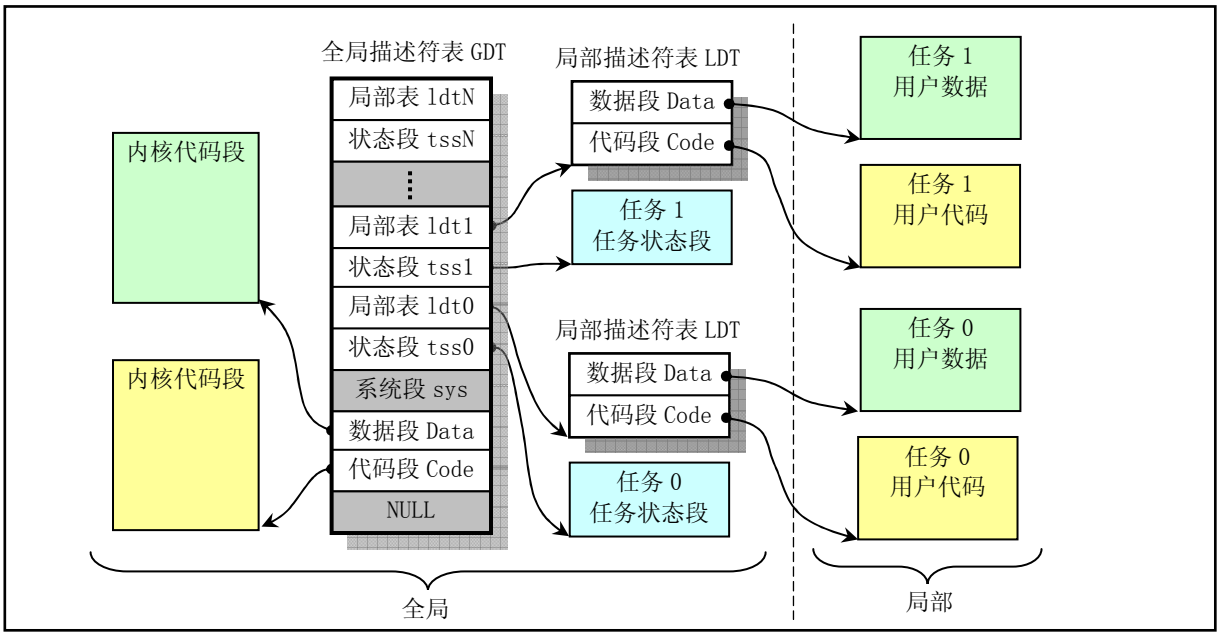

图 2-10 Linux 系统中虚拟地址空间分配图

图中画出了具有两个任务时的情况。对于中断描述符表 idt，它是保存在内核代码段中的。由于在 Linux 0.11 内核中，内核和各任务的代码段和数据段都分别被映射到线性地址空间中相同基址处，且段限长也一样，因此内核和任务的代码段和数据段都分别是重叠的。另外，Linux 0.11 内核中没有使用系统段描述符。

内存分页管理的基本原理是将整个主内存区域划分成 4096 字节为一页的内存页面。程序申请使用内存时，就以内存页为单位进行分配。

在使用这种内存分页管理方法时，每个执行中的进程（任务）可以使用比实际内存容量大得多的连续地址空间。对于 Intel 80386 系统，其 CPU 可以提供多达 4G 的线性地址空间。对于 Linux 0.11 内核，系统设置全局描述符表 GDT 中的段描述符项数最大为 256，其中 2 项空闲、2 项系统使用，每个进程使用两项。因此，此时系统可以最多容纳(256-4)/2 + 1=127 个任务，并且虚拟地址范围是 ((256-4)/2)* 64MB 约等于 8G。但 0.11 内核中人工定义最大任务数 NR_TASKS = 64 个，每个进程虚拟地址范围是 64M，并且各个进程的虚拟地址起始位置是(任务号-1)*64MB。因此所使用的虚拟地址空间范围是 64MB*64 =4G，见图 2-11 所示。4G 正好与 CPU 的线性地址空间范围或物理地址空间范围相同，因此在 0.11 内核中比较容易混淆三种地址概念。

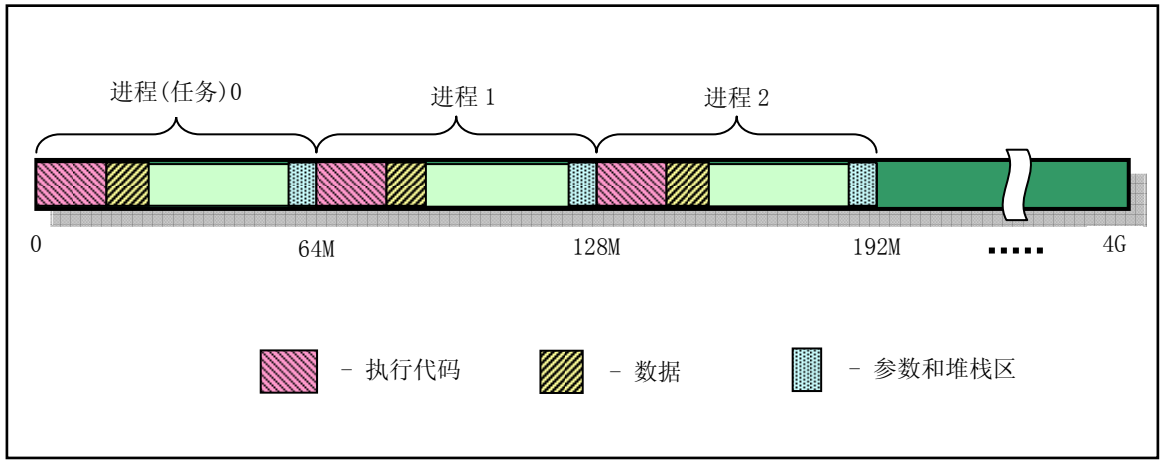

图 2-11 Linux 0.11 线性地址空间的使用示意图

进程的虚拟地址需要首先通过其局部段描述符变换为 CPU 整个线性地址空间中的地址，然后再使用页目录表 PDT（一级页表）和页表 PT（二级页表）映射到实际物理地址页上。因此两种变换不能混淆。

为了使用实际物理内存，每个进程的线性地址通过二级内存页表动态地映射到主内存区域的不同内存页上。因此每个进程最大可用的虚拟内存空间是 64MB。每个进程的逻辑地址通过加上任务号*64M，即可转换为线性地址。不过在注释中，我们通常将进程中的地址简单地称为线性地址。

有关内存分页管理的详细信息，请参见第 10 章开始部分的有关说明，或参见附录。

从 Linux 内核 0.99 版以后，对内存空间的使用方式发生了变化。每个进程可以单独享用整个 4G 的地址空间范围。由于篇幅所限，这里对此不再说明。

2.7 Linux 系统中堆栈的使用方法

本节内容概要描述了 Linux 内核从开机引导到系统正常运行过程中对堆栈的使用方式。这部分内容的说明与内核代码关系比较密切，可以先跳过。在开始阅读相应代码时再回来仔细研究。

Linux 0.11 系统中共使用了四种堆栈。一种是系统初始化时临时使用的堆栈；一种是供内核程序自己使用的堆栈（内核堆栈），只有一个，位于系统地址空间固定的位置，也是后来任务 0 的用户态堆栈；另一种是每个任务通过系统调用，执行内核程序时使用的堆栈，我们称之为任务的内核态堆栈，每个任务都有自己独立的内核态堆栈；最后一种是任务在用户态执行的堆栈，位于任务（进程）地址空间的末端。下面分别对它们进行说明。

2.7.1 初始化阶段

开机初始化时(bootsect.s，setup.s)

当 bootsect 代码被 ROM BIOS 引导加载到物理内存 0x7c00 处时，并没有设置堆栈段，当然程序也没有使用堆栈。直到 bootsect 被移动到 0x9000:0 处时，才把堆栈段寄存器 SS 设置为 0x9000，堆栈指针 esp 寄存器设置为 0xff00，也即堆栈顶端在 0x9000:0xff00 处，参见 boot/bootsect.s 第 61、62 行。setup.s 程序中也沿用了 bootsect 中设置的堆栈段。这就是系统初始化时临时使用的堆栈。

进入保护模式时(head.s)

从 head.s 程序起，系统开始正式在保护模式下运行。此时堆栈段被设置为内核数据段（0x10），堆栈指针 esp 设置成指向 user_stack 数组的顶端（参见 head.s，第 31 行），保留了 1 页内存（4K）作为堆栈使用。user_stack 数组定义在 sched.c 的 67--72 行，共含有 1024 个长字。它在物理内存中的位置可参见下图 2-12 所示。此时该堆栈是内核程序自己使用的堆栈。

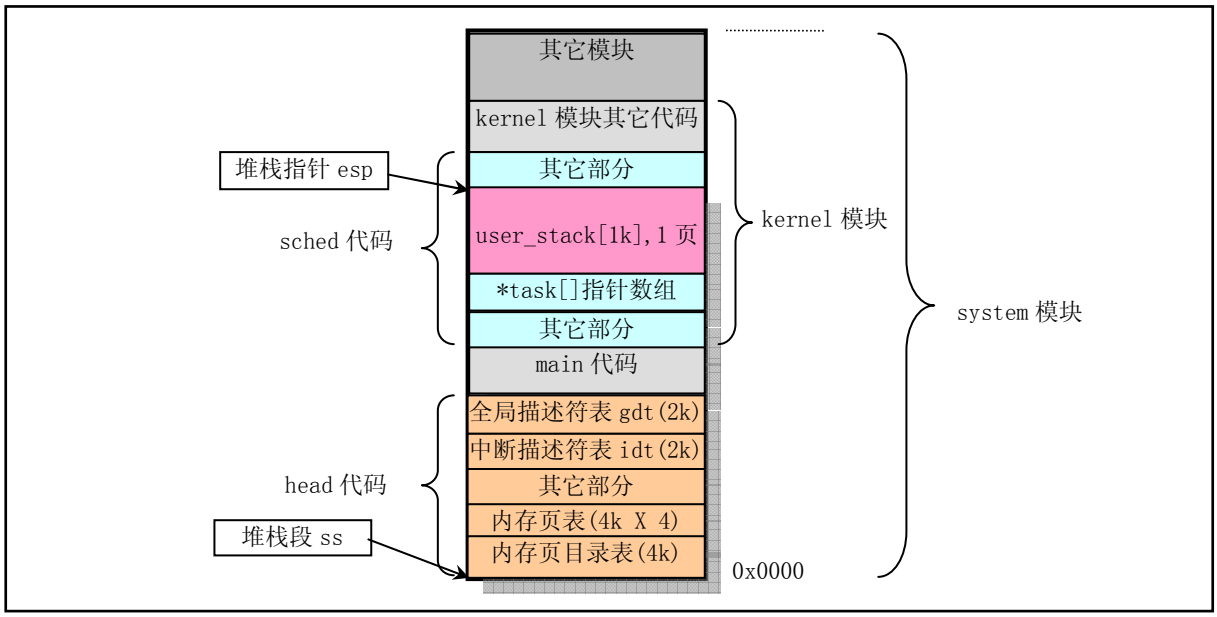

图 2-12 刚进入保护模式时内核使用的堆栈示意图

初始化时(main.c)

在 main.c 中，在执行 move_to_user_mode()代码之前，系统一直使用上述堆栈。而在执行过 move_to_user_mode()之后，main.c 的代码被“切换”成任务 0 中执行。通过执行 fork()系统调用，main.c 中的 init()将在任务 1 中执行，并使用任务 1 的堆栈。而 main()本身则在被“切换”成为任务 0 后，仍然继续使用上述内核程序自己的堆栈作为任务 0 的用户态堆栈。关于任务 0 所使用堆栈的详细描述见后面说明。

2.7.2 任务的堆栈

每个任务都有两个堆栈，分别用于用户态和内核态程序的执行，并且分别称为用户态堆栈和内核态堆栈。这两个堆栈之间的主要区别在于任务的内核态堆栈很小，所保存的数据量最多不能超过（4096 – 任务数据结构）个字节，大约为 3K 字节。而任务的用户态堆栈却可以在用户的 64MB 空间内延伸。

在用户态运行时

每个任务（除了任务 0）有自己的 64MB 地址空间。当一个任务（进程）刚被创建时，它的用户态堆栈指针被设置在其地址空间的末端（64MB 顶端），而其内核态堆栈则被设置成位于其任务数据结构所在页面的末端。应用程序在用户态下运行时就一直使用这个堆栈。堆栈实际使用的物理内存则由 CPU 分页机制确定。由于 Linux 实现了写时复制功能（Copy on Write），因此在进程被创建后，若该进程及其父进程没有使用堆栈，则两者共享同一堆栈对应的物理内存页面。

在内核态运行时

每个任务有其自己的内核态堆栈，与每个任务的任务数据结构（task_struct）放在同一页面内。这是在建立新任务时，fork()程序在任务 tss 段的内核级堆栈字段(tss.esp0 和 tss.ss0)中设置的，参见 kernel/fork.c，93 行：

```
    p->tss.esp0 = PAGE_SIZE + (long)p;
    p->tss.ss0 = 0x10;
```

其中 p 是新任务的任务数据结构指针，tss 是任务状态段结构。内核为新任务申请内存用作保存其 task_struct 结构数据，而 tss 结构（段）是 task_struct 中的一个字段。该任务的内核堆栈段值 tss.ss0 也被

设置成为 0x10（即内核数据段），而 tss.esp0 则指向保存 task_struct 结构页面的末端。见图 2-13 所示。

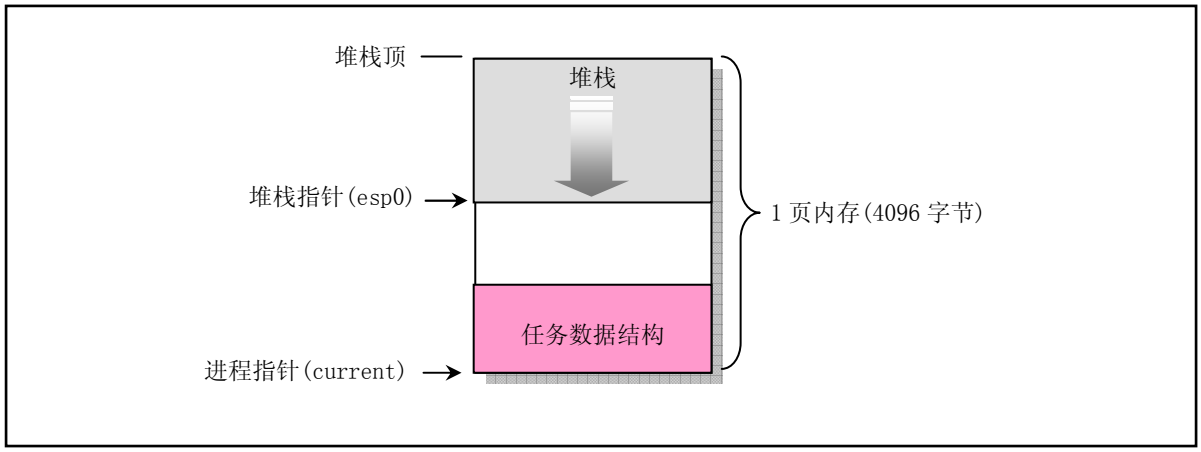

图 2-13 进程的内核态堆栈示意图

为什么通过内存管理程序从主内存区分配得来的用于保存任务数据结构的一页内存也能被设置成内核数据段中的数据呢，也即 tss.ss0 为什么能被设置成 0x10 呢？这要从内核代码段的长度范围来说明。在 head.s 程序的末端，分别设置了内核代码段和数据段的描述符。其中段的长度被设置成了 16MB。这个长度值是 Linux 0.11 内核所能支持的最大物理内存长度（参见 head.s，110 行开始的注释）。因此，内核代码可以寻址到整个物理内存范围中的任何位置，当然也包括主内存区。到 Linux 0.98 版后内核段的限长被修改成了 1GB。

每当任务执行内核程序而需要使用其内核栈时，CPU 就会利用 TSS 结构把它的内核态堆栈设置成由这两个值构成。在任务切换时，老任务的内核栈指针(esp0)不会被保存。对 CPU 来讲，这两个值是只读的。因此每当一个任务进入内核态执行时，其内核态堆栈总是空的。

任务 0 的堆栈

任务 0 的堆栈比较特殊，需要特别予以说明。

任务 0 的代码段和数据段相同，段基地址都是从 0 开始，限长也都是 640KB。这个地址范围也就是内核代码和基本数据所在的地方。在执行了 move_to_user_mode()之后，它的内核态堆栈位于其任务数据结构所在页面的末端，而它的用户态堆栈就是前面进入保护模式后所使用的堆栈，也即 sched.c 的 user_stack 数组的位置。任务 0 的内核态堆栈是在其人工设置的初始化任务数据结构中指定的，而它的用户态堆栈是在执行 movie_to_user_mode()时，在模拟 iret 返回之前的堆栈中设置的。在该堆栈中，esp 仍然是 user_stack 中原来的位置，而 ss 被设置成 0x17，也即用户态局部表中的数据段，也即从内存地址 0 开始并且限长为 640KB 的段。参见图 2-7 所示。

2.7.3 任务内核态堆栈与用户态堆栈之间的切换

任务调用系统调用时就会进入内核，执行内核代码。此时内核代码就会使用该任务的内核态堆栈进行操作。当进入内核程序时，由于优先级别发生了改变（从用户态转到内核态），用户态堆栈的堆栈段和堆栈指针以及 eflags 会被保存在任务的内核态堆栈中。而在执行 iret 退出内核程序返回到用户程序时，将恢复用户态的堆栈和 eflags。这个过程见图 2-14 所示。

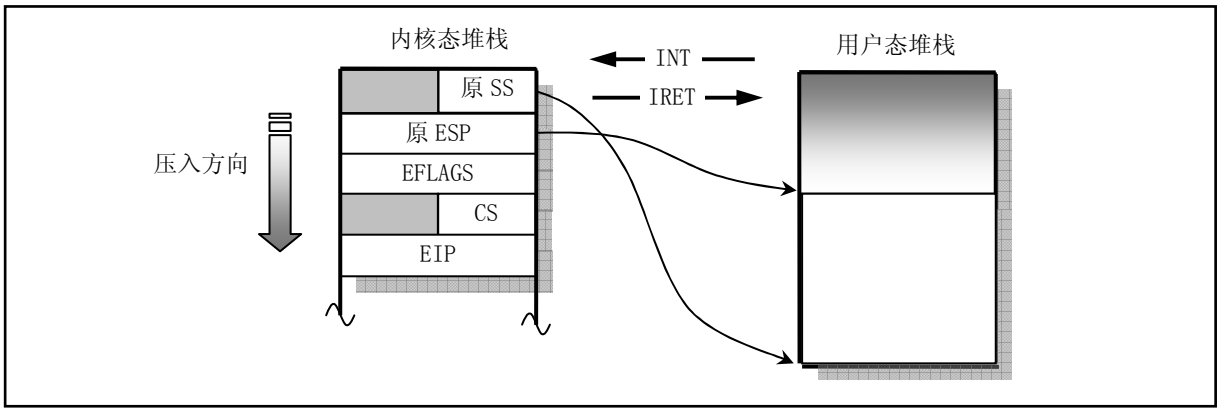

图 2-14 内核态和用户态堆栈的切换

2.8 Linux 内核源代码的目录结构

由于 Linux 内核是一种单内核模式的系统，因此，内核中所有的程序几乎都有紧密的联系，它们之间的依赖和调用关系非常密切。所以在阅读一个源代码文件时往往需要参阅其它相关的文件。因此有必要在开始阅读内核源代码之前，先熟悉一下源代码文件的目录结构和安排。

这里我们首先列出 Linux 内核完整的源代码目录，包括其中的子目录。然后逐一介绍各个目录中所包含程序的主要功能，使得整个内核源代码的安排形式能在我们的头脑中建立起一个大概的框架，以便于下一章开始的源代码阅读工作。

当我们使用 tar 命令将 linux-0.11.tar.gz 解开时，内核源代码文件被放到了 linux/目录中。其中的目录结构见图 2-15 所示：

```
linux
├── boot            系统引导汇编程序
├── fs              文件系统
├── include         头文件(*.h)
│   ├── asm         与 CPU 体系结构相关的部分
│   ├── linux       Linux 内核专用部分
│   └── sys         系统数据结构部分
├── init            内核初始化程序
├── kernel          内核进程调度、信号处理、系统调用等程序
│   ├── blk_drv     块设备驱动程序
│   ├── chr_drv     字符设备驱动程序
│   └── math        数学协处理器仿真处理程序
├── lib             内核库函数
├── mm              内存管理程序
└── tools           生成内核 Image 文件的工具程序
```

图 2-15 Linux 内核源代码目录结构

该内核版本的源代码目录中含有 14 个子目录，总共包括 102 个代码文件。下面逐个对这些子目录中的内容进行描述。

2.8.1 内核主目录 linux

linux 目录是源代码的主目录，在该主目录中除了包括所有的 14 个子目录以外，还含有唯一的一个 Makefile 文件。该文件是编译辅助工具软件 make 的参数配置文件。make 工具软件的主要用途是通过识

别哪些文件已被修改过，从而自动地决定在一个含有多个源程序文件的程序系统中哪些文件需要被重新编译。因此，make 工具软件是程序项目的管理软件。

linux 目录下的这个 Makefile 文件还嵌套地调用了所有子目录中包含的 Makefile 文件，这样，当 linux 目录（包括子目录）下的任何文件被修改过时，make 都会对其进行重新编译。因此为了编译整个内核所有的源代码文件，只要在 linux 目录下运行一次 make 软件即可。

2.8.2 引导启动程序目录 boot

boot 目录中含有 3 个汇编语言文件，是内核源代码文件中最先被编译的程序。这 3 个程序完成的主要功能是当计算机加电时引导内核启动，将内核代码加载到内存中，并做一些进入 32 位保护运行方式前的系统初始化工作。其中 bootsect.s 和 setup.s 程序需要使用 as86 软件来编译，使用的是 as86 的汇编语言格式（与微软的类似），而 head.s 需要用 GNU as 来编译，使用的是 AT&T 格式的汇编语言。这两种汇编语言在下一章的代码注释里以及代码列表后面的说明中会有简单的介绍。

bootsect.s 程序是磁盘引导块程序，编译后会驻留在磁盘的第一个扇区中（引导扇区，0 磁道（柱面），0 磁头，第 1 个扇区）。在 PC 机加电 ROM BIOS 自检后，将被 BIOS 加载到内存 0x7C00 处进行执行。

setup.s 程序主要用于读取机器的硬件配置参数，并把内核模块 system 移动到适当的内存位置处。

head.s 程序会被编译连接在 system 模块的最前部分，主要进行硬件设备的探测设置和内存管理页面的初始设置工作。

2.8.3 文件系统目录 fs

Linux 0.11 内核的文件系统采用了 1.0 版的 MINIX 文件系统，这是由于 Linux 是在 MINIX 系统上开发的，采用 MINIX 文件系统便于进行交叉编译，并且可以从 MINIX 中加载 Linux 分区。虽然使用的是 MINIX 文件系统，但 Linux 对其处理方式与 MINIX 系统不同。主要的区别在于 MINIX 对文件系统采用单线程处理方式，而 Linux 则采用了多线程方式。由于采用了多线程处理方式，Linux 程序就必须处理多线程带来的竞争条件、死锁等问题，因此 Linux 文件系统代码要比 MINIX 系统的复杂得多。为了避免竞争条件的发生，Linux 系统对资源分配进行了严格地检查，并且在内核模式下运行时，如果任务没有主动睡眠（调用 sleep()），就不让内核切换任务。

fs/目录是文件系统实现程序的目录，共包含 17 个 C 语言程序。这些程序之间的主要引用关系见图 2-16 所示图中每个方框代表一个文件，从上到下按基本按引用关系放置。其中各文件名均略去了后缀.c，虚框中是的程序文件不属于文件系统，带箭头的线条表示引用关系，粗线条表示有相互引用关系。

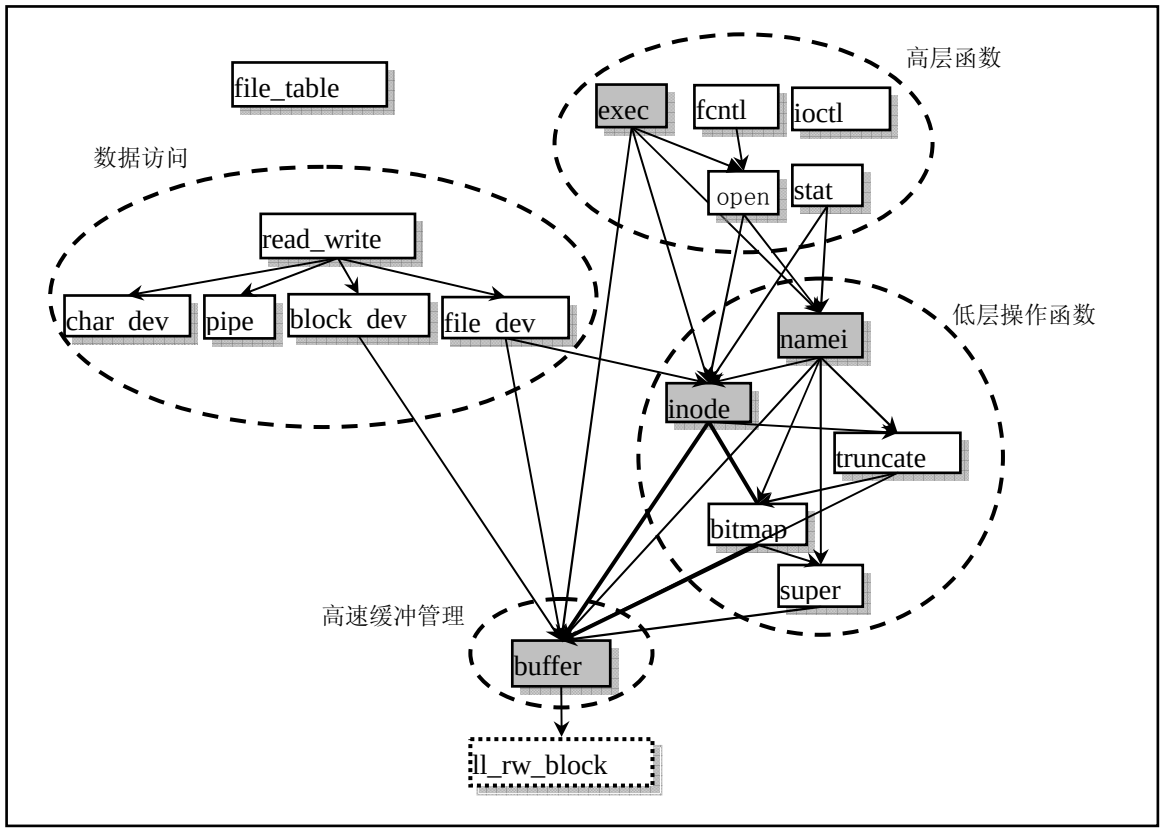

图 2-16 fs 目录中各程序中函数之间的引用关系。

由图可以看出，该目录中的程序可以划分成四个部分：高速缓冲区管理、低层文件操作、文件数据访问和文件高层函数，在对本目录中文件进行注释说明时，我们也将分成这四个部分来描述。

对于文件系统，我们可以将它看成是内存高速缓冲区的扩展部分。所有对文件系统中数据的访问，都需要首先读取到高速缓冲区中。本目录中的程序主要用来管理高速缓冲区中缓冲块的使用分配和块设备上的文件系统。管理高速缓冲区的程序是 buffer.c，而其它程序则主要都是用于文件系统管理。

在 file_table.c 文件中，目前仅定义了一个文件句柄（描述符）结构数组。ioctl.c 文件将引用 kernel/chr_drv/tty.c 中的函数，实现字符设备的 io 控制功能。exec.c 程序主要包含一个执行程序函数 do_execve()，它是所有 exec()函数簇中的主要函数。fcntl.c 程序用于实现文件 i/o 控制的系统调用函数。read_write.c 程序用于实现文件读/写和定位三个系统调用函数。stat.c 程序中实现了两个获取文件状态的系统调用函数。open.c 程序主要包含实现修改文件属性和创建与关闭文件的系统调用函数。

char_dev.c 主要包含字符设备读写函数 rw_char()。pipe.c 程序中包含管道读写函数和创建管道的系统调用。file_dev.c 程序中包含基于 i 节点和描述符结构的文件读写函数。namei.c 程序主要包括文件系统中目录名和文件名的操作函数和系统调用函数。block_dev.c 程序包含块数据读和写函数。inode.c 程序中包含针对文件系统 i 节点操作的函数。truncate.c 程序用于在删除文件时释放文件所占用的设备数据空间。bitmap.c 程序用于处理文件系统中 i 节点和逻辑数据块的位图。super.c 程序中包含对文件系统超级块的处理函数。buffer.c 程序主要用于对内存高速缓冲区进行处理。虚框中的 ll_rw_block 是块设备的底层读函数，它并不在 fs 目录中，而是 kernel/blk_drv/ll_rw_block.c 中的块设备读写驱动函数。放在这里只是让我们清楚的看到，文件系统对于块设备中数据的读写，都需要通过高速缓冲区与块设备的驱动程序（ll_rw_block()）来操作来进行，文件系统程序集本身并不直接与块设备的驱动程序打交道。

在对程序进行注释过程中，我们将另外给出这些文件中各个主要函数之间的调用层次关系。

2.8.4 头文件主目录 include

头文件目录中总共有 32 个.h 头文件。其中主目录下有 13 个，asm 子目录中有 4 个，linux 子目录中有 10 个，sys 子目录中有 5 个。这些头文件各自的功能见如下简述，具体的作用和所包含的信息请参见对头文件的注释一章。

<a.out.h>　　a.out 头文件，定义了 a.out 执行文件格式和一些宏。
<const.h>　　常数符号头文件，目前仅定义了 i 节点中 i_mode 字段的各标志位。
<ctype.h>　　字符类型头文件。定义了一些有关字符类型判断和转换的宏。
<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)。
<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。
<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。
<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），用于 vsprintf、vprintf、vfprintf 函数。
<stddef.h>　　标准定义头文件。定义了 NULL, offsetof(TYPE, MEMBER)。
<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。
<termios.h>　　终端输入输出函数头文件。主要定义控制异步通信口的终端接口。
<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。
<unistd.h>　　Linux 标准头文件。定义了各种符号常数和类型，并申明了各种函数。如定义了__LIBRARY__，则还包括系统调用号和内嵌汇编_syscall0()等。
<utime.h>　　用户时间头文件。定义了访问和修改时间结构以及 utime()原型。

体系结构相关头文件子目录 include/asm

这些头文件主要定义了一些与 CPU 体系结构密切相关的数据结构、宏函数和变量。共 4 个文件。

<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。
<asm/memory.h>　　内存拷贝头文件。含有 memcpy()嵌入式汇编宏函数。
<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。
<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。

Linux 内核专用头文件子目录 include/linux

<linux/config.h>　内核配置头文件。定义键盘语言和硬盘类型（HD_TYPE）可选项。
<linux/fdreg.h>　　软驱头文件。含有软盘控制器参数的一些定义。
<linux/fs.h>　　文件系统头文件。定义文件表结构（file,buffer_head,m_inode 等）。
<linux/hdreg.h>　　硬盘参数头文件。定义访问硬盘寄存器端口，状态码，分区表等信息。
<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。
<linux/kernel.h>　内核头文件。含有一些内核常用函数的原形定义。
<linux/mm.h>　　内存管理头文件。含有页面大小定义和一些页面释放函数原型。
<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。
<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。
<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。

系统专用数据结构子目录 include/sys

<sys/stat.h>　　文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。
<sys/times.h>　　定义了进程中运行时间结构 tms 以及 times()函数原型。
<sys/types.h>　　类型头文件。定义了基本的系统数据类型。
<sys/utsname.h>　系统名称结构头文件。
<sys/wait.h>　　等待调用头文件。定义系统调用 wait()核 waitpid()及相关常数符号。

2.8.5 内核初始化程序目录 init

该目录中仅包含一个文件 main.c。用于执行内核所有的初始化工作，然后移到用户模式创建新进程，并在控制台设备上运行 shell 程序。

程序首先根据机器内存的多少对缓冲区内存容量进行分配，如果还设置了要使用虚拟盘，则在缓冲区内存后面也为它留下空间。之后就进行所有硬件的初始化工作，包括人工创建第一个任务（task 0），并设置了中断允许标志。在执行从核心态移到用户态之后，系统第一次调用创建进程函数 fork()，创建出一个用于运行 init()的进程，在该子进程中，系统将进行控制台环境设置，并且在生成一个子进程用来运行 shell 程序。

2.8.6 内核程序主目录 kernel

linux/kernel 目录中共包含 12 个代码文件和一个 Makefile 文件，另外还有 3 个子目录。所有处理任务的程序都保存在 kernel/目录中，其中包括象 fork、exit、调度程序以及一些系统调用程序等。还包括处理中断异常和陷阱的处理过程。子目录中包括了低层的设备驱动程序，如 get_hd_block 和 tty_write 等。由于这些文件中代码之间调用关系复杂，因此这里就不详细列出各文件之间的引用关系图，但仍然可以进行大概分类，见图 2-17 所示。

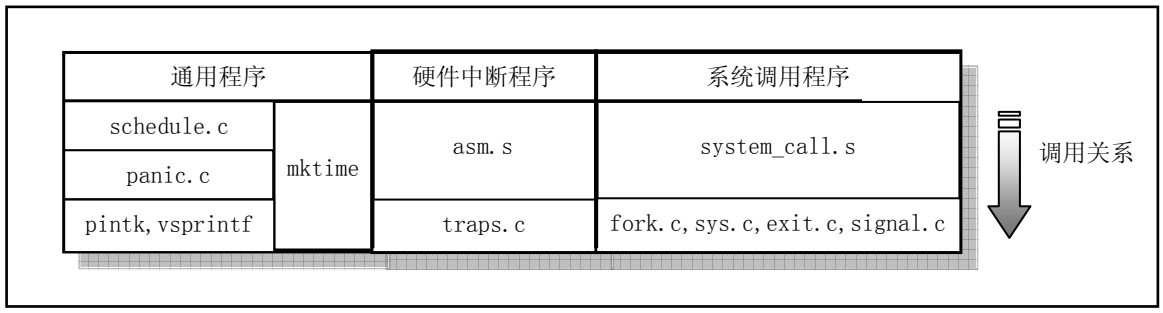

图 2-17 各文件的调用层次关系

asm.s 程序是用于处理系统硬件异常所引起的中断，对各硬件异常的实际处理程序则是在 traps.c 文件中，在各个中断处理过程中，将分别调用 traps.c 中相应的 C 语言处理函数。

exit.c 程序主要包括用于处理进程终止的系统调用。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。

fork.c 程序给出了 sys_fork()系统调用中使用了两个 C 语言函数：find_empty_process()和 copy_process()。

mktime.c 程序包含一个内核使用的时间函数 mktime()，用于计算从 1970 年 1 月 1 日 0 时起到开机当日的秒数，作为开机秒时间。仅在 init/main.c 中被调用一次。

panic.程序包含一个显示内核出错信息并停机的函数 panic()。

printk.c 程序包含一个内核专用信息显示函数 printk()。

sched.c 程序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。另外还有几个与定时相关的软盘操作函数。

signal.c 程序中包括了有关信号处理的 4 个系统调用以及一个在对应的中断处理程序中处理信号的函数 do_signal()。

sys.c 程序包括很多系统调用函数，其中有些还没有实现。

system_call.s 程序实现了 Linux 系统调用（int 0x80）的接口处理过程，实际的处理过程则包含在各系统调用相应的 C 语言处理函数中，这些处理函数分布在整个 Linux 内核代码中。

vsprintf.c 程序实现了现在已经归入标准库函数中的字符串格式化函数。

块设备驱动程序子目录 kernel/blk_drv

通常情况下，用户是通过文件系统来访问设备的，因此设备驱动程序为文件系统实现了调用接口。在使用块设备时，由于其数据吞吐量大，为了能够高效率地使用块设备上的数据，在用户进程与块设备之间使用了高速缓冲机制。在访问块设备上的数据时，系统首先以数据块的形式把块设备上的数据读入到高速缓冲区中，然后再提供给用户。blk_drv 子目录共包含 4 个 c 文件和 1 个头文件。头文件 blk.h 由于是块设备程序专用的，所以与 C 文件放在一起。这几个文件之间的大致关系，见图 2-18 所示。

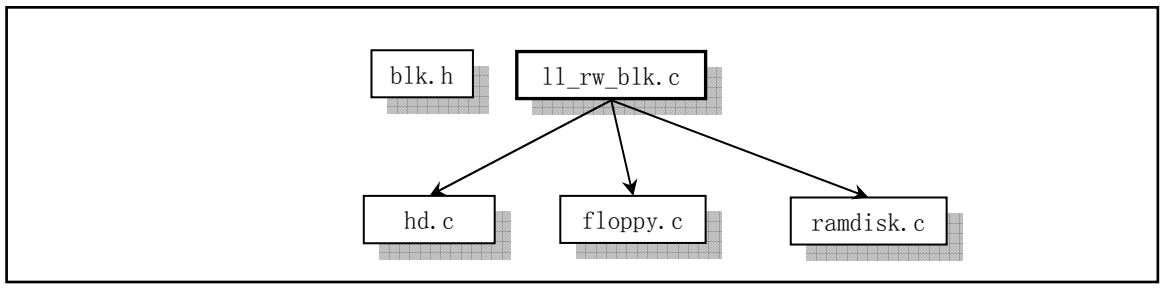

图 2-18 blk_drv 目录中文件的层次关系。

blk.h 中定义了 3 个 C 程序中共用的块设备结构和数据块请求结构。hd.c 程序主要实现对硬盘数据块进行读/写的底层驱动函数，主要是 do_hd__request()函数；floppy.c 程序中主要实现了对软盘数据块的读/写驱动函数，主要是 do_fd_request()函数。ll_rw_blk.c 中程序实现了低层块设备数据读/写函数 ll_rw_block()，内核中所有其它程序都是通过该函数对块设备进行数据读写操作。你将看到该函数在许多访问块设备数据的地方被调用，尤其是在高速缓冲区处理文件 fs/buffer.c 中。

字符设备驱动程序子目录 kernel/chr_drv

字符设备程序子目录共含有 4 个 C 语言程序和 2 个汇编程序文件。这些文件实现了对串行端口 rs-232、串行终端、键盘和控制台终端设备的驱动。图 2-19 是这些文件之间的大致调用层次关系。

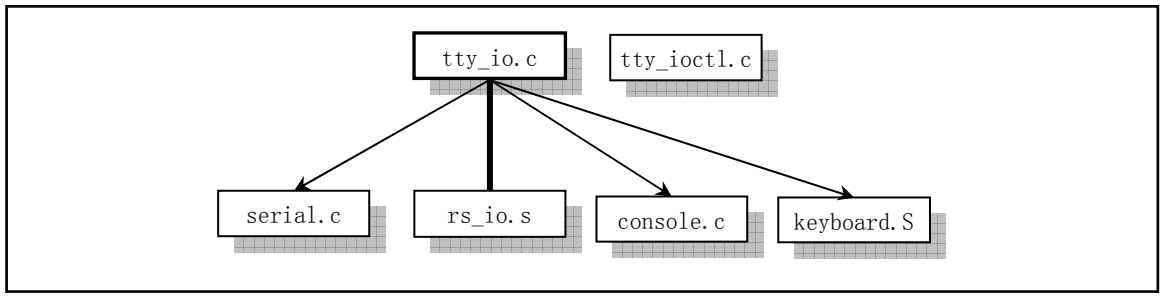

图 2-19 字符设备程序之间的关系示意图

tty_io.c 程序中包含 tty 字符设备读函数 tty_read()和写函数 tty_write()，为文件系统提供了上层访问接口。另外还包括在串行中断处理过程中调用的 C 函数 do_tty_interrupt()，该函数将会在中断类型为读字符的处理中被调用。

console.c 文件主要包含控制台初始化程序和控制台写函数 con_write()，用于被 tty 设备调用。还包含对显示器和键盘中断的初始化设置程序 con_init()。

rs_io.s 汇编程序用于实现两个串行接口的中断处理程序。该中断处理程序会根据从中断标识寄存器（端口 0x3fa 或 0x2fa）中取得的 4 种中断类型分别进行处理，并在处理中断类型为读字符的代码中调用 do_tty_interrupt()。

serial.c 用于对异步串行通信芯片 UART 进行初始化操作，并设置两个通信端口的中断向量。另外还

包括 tty 用于往串口输出的 rs_write()函数。

tty_ioctl.c 程序实现了 tty 的 io 控制接口函数 tty_ioctl()以及对 termio(s)终端 io 结构的读写函数，并会在实现系统调用 sys_ioctl()的 fs/ioctl.c 程序中被调用。

keyboard.S 程序主要实现了键盘中断处理过程 keyboard_interrupt。

协处理器仿真和操作程序子目录 kernel/math

该子目录中目前仅有一个 C 程序 math_emulate.c。其中的 math_emulate()函数是中断 int7 的中断处理程序调用的 C 函数。当机器中没有数学协处理器，而 CPU 却又执行了协处理器的指令时，就会引发该中断。因此，使用该中断就可以用软件来仿真协处理器的功能。本书所讨论的内核版本还没有包含有关协处理器的仿真代码。本程序中只是打印一条出错信息，并向用户程序发送一个协处理器错误信号 SIGFPE。

2.8.7 内核库函数目录 lib

内核库函数用于为内核初始化程序 init/main.c 运行在用户态的进程（进程 0、1）提供调用支持。它与普通静态库的实现方法完全一样。读者可从中了解一般 libc 函数库的基本组成原理。在 lib/目录中共有 12 个 C 语言文件，除了一个由 tytso 编制的 malloc.c 程序较长以外，其它的程序很短，有的只有一二行代码，实现了一些系统调用的接口函数。

这些文件中主要包括有退出函数_exit()、关闭文件函数 close(fd)、复制文件描述符函数 dup()、文件打开函数 open()、写文件函数 write()、执行程序函数 execve()、内存分配函数 malloc()、等待子进程状态函数 wait()、创建会话系统调用 setsid()以及在 include/string.h 中实现的所有字符串操作函数。

2.8.8 内存管理程序目录 mm

该目录包括 2 个代码文件。主要用于管理程序对主内存区的使用，实现了进程逻辑地址到线性地址以及线性地址到主内存区中物理内存地址的映射，通过内存的分页管理机制，在进程的虚拟内存页与主内存区的物理内存页之间建立了对应关系。

Linux 内核对内存的处理使用了分页和分段两种方式。首先是将 386 的 4G 虚拟地址空间分割成 64 个段，每个段 64MB。所有内核程序占用其中第一个段，并且物理地址与该段线性地址相同。然后每个任务分配一个段使用。分页机制用于把指定的物理内存页面映射到段内，检测 fork 创建的任何重复的拷贝，并执行写时复制机制。

page.s 文件包括内存页面异常中断（int 14）处理程序，主要用于处理程序由于缺页而引起的页异常中断和访问非法地址而引起的页保护。

memory.c 程序包括对内存进行初始化的函数 mem_init()，由 page.s 的内存处理中断过程调用的 do_no_page()和 do_wp_page()函数。在创建新进程而执行复制进程操作时，即使用该文件中的内存处理函数来分配管理内存空间。

2.8.9 编译内核工具程序目录 tools

该目录下的 build.c 程序用于将 Linux 各个目录中被分别编译生成的目标代码连接合并成一个可运行的内核映象文件 image。其具体的功能可参见下一章内容。

2.9 内核系统与用户程序的关系

在 Linux 系统中，内核为应用程序提供了两方面的接口。其一是系统调用接口（在第 5 章中说明），也即中断调用 int 0x80；另一方面是通过内核库函数（在第 12 章中说明）与内核进行信息交流。内核库函数是基本 C 函数库 libc 的组成部分。许多系统调用是作为基本 C 语言函数库的一部分实现的。

系统调用主要是提供给系统软件直接使用或用于库函数的实现。而一般用户开发的程序则是通过调

用象 libc 等库中的函数来访问内核资源。通过调用这些库中的程序，应用程序代码能够完成各种常用工作，例如，打开和关闭对文件或设备的访问、进行科学计算、出错处理以及访问组和用户标识号 ID 等系统信息。

系统调用是内核与外界接口的最高层。在内核中，每个系统调用都有一个序列号（在 include/linux/unistd.h 头文件中定义），并常以宏的形式实现。应用程序不应该直接使用系统调用，因为这样的话，程序的移植性就不好了。因此目前 Linux 标准库 LSB（Linux Standard Base）和许多其它标准都不允许应用程序直接访问系统调用宏。系统调用的有关文档可参见 Linux 操作系统的在线手册的第 2 部分。

库函数一般包括 C 语言没有提供的执行高级功能的用户级函数，例如输入/输出和字符串处理函数。某些库函数只是系统调用的增强功能版。例如，标准 I/O 库函数 fopen 和 fclose 提供了与系统调用 open 和 close 类似的功能，但却是在更高的层次上。在这种情况下，系统调用通常能提供比库函数略微好一些的性能，但是库函数却能提供更多的功能，而且更具检错能力。系统提供的库函数有关文档可参见操作系统的在线手册第 3 部分。

2.10 linux/Makefile 文件

从本节起，我们开始对内核源代码文件进行注释。首先注释 linux 目录下遇到的第一个文件 Makefile。后续章节将按照这里类似的描述结构进行注释。

2.10.1 功能描述

Makefile 文件相当于程序编译过程中的批处理文件。是工具程序 make 运行时的输入数据文件。只要在含有 Makefile 的当前目录中键入 make 命令，它就会依据 Makefile 文件中的设置对源程序或目标代码文件进行编译、连接或进行安装等活动。

make 工具程序能自动地确定一个大程序系统中那些程序文件需要被重新编译，并发出命令对这些程序文件进行编译。在使用 make 之前，需要编写 Makefile 信息文件，该文件描述了整个程序包中各程序之间的关系，并针对每个需要更新的文件给出具体的控制命令。通常，执行程序是根据其目标文件进行更新的，而这些目标文件则是由编译程序创建的。一旦编写好一个合适的 Makefile 文件，那么在你每次修改过程序系统中的某些源代码文件后，执行 make 命令就能进行所有必要的重新编译工作。make 程序是使用 Makefile 数据文件和代码文件的最后修改时间(last-modification time)来确定那些文件需要进行更新，对于每一个需要更新的文件它会根据 Makefile 中的信息发出相应的命令。在 Makefile 文件中，开头为'#'的行是注释行。文件开头部分的'='赋值语句定义了一些参数或命令的缩写。

这个 Makefile 文件的主要作用是指示 make 程序最终使用独立编译连接成的 tools/目录中的 build 执行程序将所有内核编译代码连接和合并成一个可运行的内核映像文件 image。具体是对 boot/中的 bootsect.s、setup.s 使用 8086 汇编器进行编译，分别生成各自的执行模块。再对源代码中的其它所有程序使用 GNU 的编译器 gcc/gas 进行编译，并连接成模块 system。再用 build 工具将这三块组合成一个内核映象文件 image. 基本编译连接/组合结构如图 2-20 所示。

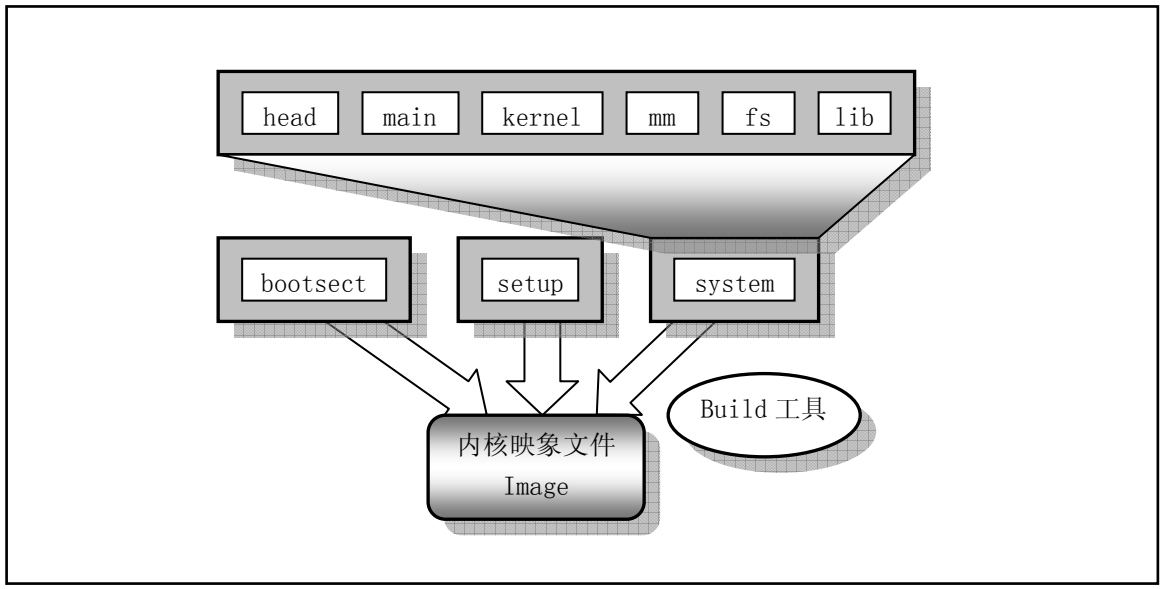

图 2-20 内核编译连接/组合结构

2.10.2 代码注释

程序 2-1 linux/Makefile 文件

```
 1 #
 2 # if you want the ram-disk device, define this to be the # 如果你要使用 RAM 盘设备的话，就
 3 # size in blocks.                                       # 定义块的大小。
 4 #
 5 RAMDISK = #-DRAMDISK=512
 6
 7 AS86    =as86 -0 -a        # 8086 汇编编译器和连接器，见列表后的介绍。后带的参数含义分别
 8 LD86    =ld86 -0           # 是：-0 生成 8086 目标程序；-a 生成与 gas 和 gld 部分兼容的代码。
 9
10 AS      =gas               # GNU 汇编编译器和连接器，见列表后的介绍。
11 LD      =gld
12 LDFLAGS =-s -x -M          # GNU 连接器 gld 运行时用到的选项。含义是：-s 输出文件中省略所
                              # 有的符号信息；-x 删除所有局部符号；-M 表示需要在标准输出设备
                              # (显示器)上打印连接映象(link map)，是指由连接程序产生的一种
                              # 内存地址映象，其中列出了程序段装入到内存中的位置信息。具体
                              # 来讲有如下信息：
                              # • 目标文件及符号信息映射到内存中的位置；
                              # • 公共符号如何放置；
                              # • 连接中包含的所有文件成员及其引用的符号。
13 CC      =gcc $(RAMDISK)    # gcc 是 GNU C 程序编译器。对于 UNIX 类的脚本(script)程序而言，
                              # 在引用定义的标识符时，需在前面加上$符号并用括号括住标识符。
14 CFLAGS  =-Wall -O -fstrength-reduce -fomit-frame-pointer \
15 -fcombine-regs -mstring-insns  # gcc 的选项。前一行最后的'\'符号表示下一行是续行。
                                  # 选项含义为：-Wall 打印所有警告信息；-O 对代码进行优化；
                                  # -fstrength-reduce 优化循环语句；-mstring-insns 是
                                  # Linus 在学习 gcc 编译器时为 gcc 增加的选项，用于 gcc-1.40
                                  # 在复制结构等操作时使用 386 CPU 的字符串指令，可以去掉。
16 CPP     =cpp -nostdinc -Iinclude  # cpp 是 gcc 的前(预)处理程序。-nostdinc -Iinclude 的含
                                     # 义是不要搜索标准的头文件目录中的文件，而是使用-I
                                     # 选项指定的目录或者是在当前目录里搜索头文件。
```

```
17
18 #
19 # ROOT_DEV specifies the default root-device when making the image.
20 # This can be either FLOPPY, /dev/xxxx or empty, in which case the
21 # default of /dev/hd6 is used by 'build'.
22 #
23 ROOT_DEV=/dev/hd6   # ROOT_DEV 指定在创建内核映像(image)文件时所使用的默认根文件系统所
                      # 在的设备，这可以是软盘(FLOPPY)、/dev/xxxx 或者干脆空着，空着时
                      # build 程序（在 tools/目录中）就使用默认值/dev/hd6。
24
25 ARCHIVES=kernel/kernel.o mm/mm.o fs/fs.o  # kernel 目录、mm 目录和 fs 目录所产生的目标代
                                            # 码文件。为了方便引用在这里将它们用
                                            # ARCHIVES（归档文件）标识符表示。
26 DRIVERS =kernel/blk_drv/blk_drv.a kernel/chr_drv/chr_drv.a  # 块和字符设备库文件。.a 表
                                # 示该文件是个归档文件，也即包含有许多可执行二进制代码子程
                                # 序集合的库文件，通常是用 GNU 的 ar 程序生成。ar 是 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。
27 MATH    =kernel/math/math.a        # 数学运算库文件。
28 LIBS    =lib/lib.a                 # 由 lib/目录中的文件所编译生成的通用库文件。
29
30 .c.s:                      # make 老式的隐式后缀规则。该行指示 make 利用下面的命令将所有的
                              # .c 文件编译生成.s 汇编程序。':'表示下面是该规则的命令。
31         $(CC) $(CFLAGS) \
32         -nostdinc -Iinclude -S -o $*.s $<  # 指使 gcc 采用前面 CFLAGS 所指定的选项以及
                              # 仅使用 include/目录中的头文件，在适当地编译后不进行汇编就
                              # 停止（-S），从而产生与输入的各个 C 文件对应的汇编语言形式的
                              # 代码文件。默认情况下所产生的汇编程序文件是原 C 文件名去掉.c
                              # 而加上.s 后缀。-o 表示其后是输出文件的形式。其中$*.s（或$@）
                              # 是自动目标变量，$<代表第一个先决条件，这里即是符合条件
                              # *.c 的文件。
33 .s.o:                         # 表示将所有.s 汇编程序文件编译成.o 目标文件。下一条是实
                                 # 现该操作的具体命令。
34         $(AS) -c -o $*.o $<  # 使用 gas 编译器将汇编程序编译成.o 目标文件。-c 表示只编译
                                 # 或汇编，但不进行连接操作。
35 .c.o:                                    # 类似上面，*.c 文件-→*.o 目标文件。
36         $(CC) $(CFLAGS) \
37         -nostdinc -Iinclude -c -o $*.o $<  # 使用 gcc 将 C 语言文件编译成目标文件但不连接。
38
39 all:     Image            # all 表示创建 Makefile 所知的最顶层的目标。这里即是 image 文件。
40
41 Image: boot/bootsect boot/setup tools/system tools/build  # 说明目标（Image 文件）是由
                            # 分号后面的 4 个元素产生，分别是 boot/目录中的 bootsect 和
                            # setup 文件、tools/目录中的 system 和 build 文件。
42         tools/build boot/bootsect boot/setup tools/system $(ROOT_DEV) > Image
43         sync             # 这两行是执行的命令。第一行表示使用 tools 目录下的 build 工具
                            # 程序（下面会说明如何生成）将 bootsect、setup 和 system 文件
                            # 以$(ROOT_DEV)为根文件系统设备组装成内核映像文件 Image。
                            # 第二行的 sync 同步命令是迫使缓冲块数据立即写盘并更新超级块。
44
45 disk: Image                                  # 表示 disk 这个目标要由 Image 产生。
46         dd bs=8192 if=Image of=/dev/PS0  # dd 为 UNIX 标准命令：复制一个文件，根据选项
                                        # 进行转换和格式化。bs=表示一次读/写的字节数。
                                        # if=表示输入的文件，of=表示输出到的文件。
```

```
                                        # 这里/dev/PS0 是指第一个软盘驱动器(设备文件)。
47
48 tools/build: tools/build.c           # 由 tools 目录下的 build.c 程序生成执行程序 build。
49         $(CC) $(CFLAGS) \
50         -o tools/build tools/build.c  # 编译生成执行程序 build 的命令。
51
52 boot/head.o: boot/head.s              # 利用上面给出的.s.o 规则生成 head.o 目标文件。
53
54 tools/system:   boot/head.o init/main.o \
55                 $(ARCHIVES) $(DRIVERS) $(MATH) $(LIBS)  # 表示 tools 目录中的 system 文件
                                                          # 要由分号右边所列的元素生成。
56         $(LD) $(LDFLAGS) boot/head.o init/main.o \
57         $(ARCHIVES) \
58         $(DRIVERS) \
59         $(MATH) \
60         $(LIBS) \
61         -o tools/system > System.map   # 生成 system 的命令。最后的 > System.map 表示
                                          # gld 需要将连接映象重定向存放在 System.map 文件中。
                                          # 关于 System.map 文件的用途参见注释后的说明。
62
63 kernel/math/math.a:                   # 数学协处理函数文件 math.a 由下一行上的命令实现。
64         (cd kernel/math; make)        # 进入 kernel/math/目录；运行 make 工具程序。
                                         # 下面从 66--82 行的含义与此处的类似。
65
66 kernel/blk_drv/blk_drv.a:            # 块设备函数文件 blk_drv.a
67         (cd kernel/blk_drv; make)
68
69 kernel/chr_drv/chr_drv.a:            # 字符设备函数文件 chr_drv.a
70         (cd kernel/chr_drv; make)
71
72 kernel/kernel.o:                     # 内核目标模块 kernel.o
73         (cd kernel; make)
74
75 mm/mm.o:                             # 内存管理模块 mm.o
76         (cd mm; make)
77
78 fs/fs.o:                             # 文件系统目标模块 fs.o
79         (cd fs; make)
80
81 lib/lib.a:                           # 库函数 lib.a
82         (cd lib; make)
83
84 boot/setup: boot/setup.s                        # 这里开始的三行是使用 8086 汇编和连接器
85         $(AS86) -o boot/setup.o boot/setup.s    # 对 setup.s 文件进行编译生成 setup 文件。
86         $(LD86) -s -o boot/setup boot/setup.o   # -s 选项表示要去除目标文件中的符号信息。
87
88 boot/bootsect:  boot/bootsect.s                         # 同上。生成 bootsect.o 磁盘引导块。
89         $(AS86) -o boot/bootsect.o boot/bootsect.s
90         $(LD86) -s -o boot/bootsect boot/bootsect.o
91
92 tmp.s:  boot/bootsect.s tools/system   # 从 92--95 这四行的作用是在 bootsect.s 程序开头添加
      # 一行有关 system 文件长度信息。方法是首先生成含有“SYSSIZE = system 文件实际长度”
      # 一行信息的 tmp.s 文件，然后将 bootsect.s 文件添加在其后。取得 system 长度的方法是：
```

```
        # 首先利用命令 ls 对 system 文件进行长列表显示，用 grep 命令取得列表行上文件字节数字段
        # 信息，并定向保存在 tmp.s 临时文件中。cut 命令用于剪切字符串，tr 用于去除行尾的回车符。
        # 其中：(实际长度 + 15)/16 用于获得用‘节’表示的长度信息。1 节 = 16 字节。
 93         (echo -n "SYSSIZE = ";ls -l tools/system | grep system \
 94                 | cut -c25-31 | tr '\012' ' '; echo "+ 15 ) / 16") > tmp.s
 95         cat boot/bootsect.s >> tmp.s
 96 a
 97 clean:  # 当执行'make clean'时，就会执行 98--103 行上的命令，去除所有编译连接生成的文件。
          # '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。
 98         rm -f Image System.map tmp_make core boot/bootsect boot/setup
 99         rm -f init/*.o tools/system tools/build boot/*.o
100         (cd mm;make clean)       # 进入 mm/目录；执行该目录 Makefile 文件中的 clean 规则。
101         (cd fs;make clean)
102         (cd kernel;make clean)
103         (cd lib;make clean)
104 a
105 backup: clean   # 该规则将首先执行上面的 clean 规则，然后对 linux/目录进行压缩，生成
                  # backup.Z 压缩文件。'cd .. '表示退到 linux/的上一级（父）目录；
                  # 'tar cf - linux'表示对 linux/目录执行 tar 归档程序。-cf 表示需要创建
                  # 新的归档文件 '| compress -'表示将 tar 程序的执行通过管道操作('|')
                  # 传递给压缩程序 compress，并将压缩程序的输出存成 backup.Z 文件。
106         (cd .. ; tar cf - linux | compress - > backup.Z)
107         sync    # 迫使缓冲块数据立即写盘并更新磁盘超级块。
108
109 dep:
    # 该目标或规则用于各文件之间的依赖关系。创建的这些依赖关系是为了给 make 用来确定是否需要要
    # 重建一个目标对象的。比如当某个头文件被改动过后，make 就通过生成的依赖关系，重新编译与该
    # 头文件有关的所有*.c 文件。具体方法如下：
    # 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
    # 文件中'### Dependencies'行后面的所有行（下面从 118 开始的行），并生成 tmp_make
    # 临时文件（也即 110 行的作用）。然后对 init/目录下的每一个 C 文件（其实只有一个文件
    # main.c）执行 gcc 预处理操作，-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。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。
    # 111 行中的$$i 实际上是$($i)的意思。这里$i 是这句前面的 shell 变量的值。
    # 然后把预处理结果都添加到临时文件 tmp_make 中，然后将该临时文件复制成新的 Makefile 文件。
110         sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
111         (for i in init/*.c;do echo -n "init/";$(CPP) -M $$i;done) >> tmp_make
112         cp tmp_make Makefile
113         (cd fs; make dep)        # 对 fs/目录下的 Makefile 文件也作同样的处理。
114         (cd kernel; make dep)
115         (cd mm; make dep)
116
117 ### Dependencies:
118 init/main.o : init/main.c include/unistd.h include/sys/stat.h \
119   include/sys/types.h include/sys/times.h include/sys/utsname.h \
120   include/utime.h include/time.h include/linux/tty.h include/termios.h \
121   include/linux/sched.h include/linux/head.h include/linux/fs.h \
122   include/linux/mm.h include/signal.h include/asm/system.h include/asm/io.h \
123   include/stddef.h include/stdarg.h include/fcntl.h
```

2.10.3 其它信息

Makefile 简介

makefile 文件是 make 工具程序的配置文件。Make 工具程序的主要用途是能自动地决定一个含有很多源程序文件的大型程序中哪个文件需要被重新编译。makefile 的使用比较复杂，这里只是根据上面的 makefile 文件作些简单的介绍。详细说明请参考 GNU make 使用手册。

为了使用 make 程序，你就需要 makefile 文件来告诉 make 要做些什么工作。通常，makefile 文件会告诉 make 如何编译和连接一个文件。当明确指出时，makefile 还可以告诉 make 运行各种命令（例如，作为清理操作而删除某些文件）。

make 的执行过程分为两个不同的阶段。在第一个阶段，它读取所有的 makefile 文件以及包含的 makefile 文件等，记录所有的变量及其值、隐式的或显式的规则，并构造出所有目标对象及其先决条件的一幅全景图。在第二阶段期间，make 就使用这些内部结构来确定哪个目标对象需要被重建，并且使用相应的规则来操作。

当 make 重新编译程序时，每个修改过的 C 代码文件必须被重新编译。如果一个头文件被修改过了，那么为了确保正确，每一个包含该头文件的 C 代码程序都将被重新编译。每次编译操作都产生一个与源程序对应的目标文件(object file)。最终，如果任何源代码文件被编译过了，那么所有的目标文件不管是刚编译完的还是以前就编译好的必须连接在一起以生成新的可执行文件。

简单的 makefile 文件含有一些规则，这些规则具有如下的形式：

```
目标(target)... : 先决条件(prerequisites)...
                  命令(command)
                  ...
                  ...
```

其中'目标'对象通常是程序生成的一个文件的名称；例如是一个可执行文件或目标文件。目标也可以是所要采取活动的名字，比如'清除'('clean')。'先决条件'是一个或多个文件名，是用作产生目标的输入条件。通常一个目标依赖几个文件。而'命令'是 make 需要执行的操作。一个规则可以有多个命令，每一个命令自成一行。请注意，你需要在每个命令行之前键入一个制表符！这是粗心者常常忽略的地方。

如果一个先决条件通过目录搜寻而在另外一个目录中被找到，这并不会改变规则的命令；它们将被如期执行。因此，你必须小心地设置命令，使得命令能够在 make 发现先决条件的目录中找到需要的先决条件。这就需要通过使用自动变量来做到。自动变量是一种在命令行上根据具体情况能被自动替换的变量。自动变量的值是基于目标对象及其先决条件而在命令执行前设置的。例如，’$^’的值表示规则的所有先决条件，包括它们所处目录的名称；’$<’的值表示规则中的第一个先决条件；’$@’表示目标对象；另外还有一些自动变量这里就不提了。

有时，先决条件还常包含头文件，而这些头文件并不愿在命令中说明。此时自动变量’$<’正是第一个先决条件。例如：

```
foo.o : foo.c defs.h hack.h
        cc -c $(CFLAGS) $< -o $@
```

其中的’$<’就会被自动地替换成 foo.c，而$@则会被替换为 foo.o

为了让 make 能使用习惯用法来更新一个目标对象，你可以不指定命令，写一个不带命令的规则或者不写规则。此时 make 程序将会根据源程序文件的类型（程序的后缀）来判断要使用哪个隐式规则。

后缀规则是为 make 程序定义隐式规则的老式方法。（现在这种规则已经不用了，取而代之的是使用

更通用更清晰的模式匹配规则)。下面例子就是一种双后缀规则。双后缀规则是用一对后缀定义的：源后缀和目标后缀。相应的隐式先决条件是通过使用文件名中的源后缀替换目标后缀后得到。因此，此时下面的'$<'值是*.c 文件名。而正条 make 规则的含义是将*.c 程序编译成*.s 代码。

```
.c.s:
        $(CC) $(CFLAGS) \
        -nostdinc -Iinclude -S -o $*.s $<
```

通常命令是属于一个具有先决条件的规则，并在任何先决条件改变时用于生成一个目标(target)文件。然而，为目标而指定命令的规则也并不一定要有先决条件。例如，与目标'clean'相关的含有删除(delete)命令的规则并不需要有先决条件。此时，一个规则说明了如何以及何时来重新制作某些文件，而这些文件是特定规则的目标。make 根据先决条件来执行命令以创建或更新目标。一个规则也可以说明如何及何时执行一个操作。

一个 makefile 文件也可以含有除规则以外的其它文字，但一个简单的 makefile 文件只需要含有适当的规则。规则可能看上去要比上面示出的模板复杂得多，但基本上都是符合的。

makefile 文件最后生成的依赖关系是用于让 make 来确定是否需要重建一个目标对象。比如当某个头文件被改动过后，make 就通过这些依赖关系，重新编译与该头文件有关的所有*.c 文件。

as86,ld86 简介

as86 和 ld86 是由 Bruce Evans 编写的 Intel 8086 汇编编译程序和连接程序。它完全是一个 8086 的汇编编译器，但却可以为 386 处理器编制 32 位的代码。Linux 使用它仅仅是为了创建 16 位的启动扇区(bootsector)代码和 setup 二进制执行代码。该编译器的语法与 GNU 的汇编编译器的语法是不兼容的，但近似于 Intel 的汇编语言语法（如操作数的次序相反等)。

Bruce Evans 是 minix 操作系统 32 位版本的主要编制者，他与 Linux 的创始人 Linus Torvalds 是很好的朋友。Linus 本人也从 Bruce Evans 那里学到了不少有关 UNIX 类操作系统的知识，minix 操作系统的不足之处也是两个好朋友互相探讨得出的结果，这激发了 Linus 在 Intel 386 体系结构上开发一个全新概念的操作系统，因此 Linux 操作系统的诞生与 Bruce Evans 也有着密切的关系。

有关这个编译器和连接器的源代码可以从 FTP 服务器 ftp.funet.fi 上或从我的网站(www.oldlinux.org)上下载。

这两个程序的使用方法和选项如下：

as 的使用方法和选项：

```
as [-03agjuw] [-b [bin]] [-lm [list]] [-n name] [-o obj] [-s sym] src
```

默认设置（除了以下默认值以外，其它选项默认为关闭或无；若没有明确说明 a 标志，则不会有输出)：
-03　　32 位输出；
list　　在标准输出上显示；
name　　源文件的基本名称（也即不包括“.“后的扩展名）；
选项含义：
-0　从 16 比特代码段开始；
-3　从 32 比特代码段开始；
-a　开启与 as、ld 的部分兼容性选项；
-b　产生二进制文件，后面可以跟文件名；
-g　在目标文件中仅存入全局符号；
-j　使所有跳转语句均为长跳转；
-l　产生列表文件，后面可以跟随列表文件名；

```
-m  在列表中扩展宏定义；
-n  后面跟随模块名称（取代源文件名称放入目标文件中）；
-o  产生目标文件，后跟目标文件名；
-s  产生符号文件，后跟符号文件名；
-u  将未定义符号作为输入的未指定段的符号；
-w  不显示警告信息；
```

ld 连接器的使用语法和选项：

```
对于生成 Minix a.out 格式的版本：
ld [-03Mims[-]]  [-T textaddr] [-llib_extension] [-o outfile] infile...

对于生成 GNU-Minix 的 a.out 格式的版本：
ld [-03Mimrs[-]] [-T textaddr] [-llib_extension] [-o outfile] infile...

默认设置(除了以下默认值以外，其它选项默认为关闭或无)：
-03      32 位输出；
outfile      a.out 格式输出；

-0  产生具有 16 比特魔数的头结构，并且对-lx 选项使用 i86 子目录；
-3  产生具有 32 比特魔数的头结构，并且对-lx 选项使用 i386 子目录；
-M  在标准输出设备上显示已链接的符号；
-T  后面跟随文本基地址（使用适合于 strtoul 的格式)；
-i  分离的指令与数据段（I&D）输出；
-lx 将库/local/lib/subdir/libx.a 加入链接的文件列表中；
-m  在标准输出设备上显示已链接的模块；
-o  指定输出文件名，后跟输出文件名；
-r  产生适合于进一步重定位的输出；
-s  在目标文件中删除所有符号。
```

System.map 文件

System.map 文件用于存放内核符号表信息。符号表是所有符号及其对应地址的一个列表。随着每次内核的编译，就会产生一个新的对应 System.map 文件。当内核运行出错时，通过 System.map 文件中的符号表解析，就可以查到一个地址值对应的变量名，或反之。

利用 System.map 符号表文件，在内核或相关程序出错时，就可以获得我们比较容易识别的信息。符号表的样例如下所示：

```
c03441a0 B dmi_broken
c03441a4 B is_sony_vaio_laptop
c03441c0 b dmi_ident
c0344200 b pci_bios_present
c0344204 b pirq_table
```

可以看出名称为 dmi_broken 的变量位于内核地址 c03441a0 处。

System.map 位于使用它的软件(例如内核日志记录后台程序 klogd)能够寻找到的地方。在系统启动时，如果没有以一个参数的形式为 klogd 给出 System.map 的位置，则 klogd 将会在三个地方搜寻 System.map。依次为：

```
/boot/System.map
/System.map
/usr/src/linux/System.map
```

尽管内核本身实际上不使用 System.map，但其它程序，象 klogd，lsof，ps 以及其它许多软件，象 dosemu，都需要有一个正确的 System.map 文件。利用该文件，这些程序就可以根据已知的内存地址查找出对应的内核变量名称，便于对内核的调试工作。

2.11 本章小结

本章概述了 Linux 早期操作系统的内核模式和体系结构。给出了 Linux 0.11 内核源代码的目录结构形式，并详细地介绍了各个子目录中代码文件的基本功能和层次关系。然后介绍了在 RedHat 9 系统下编译 Linux 0.11 内核时，对代码需要进行修改的地方。最后从 Linux 内核主目录下的 makefile 文件着手，开始对内核源代码进行注释。