第 5 章

CHAPTER 5

ARM-Linux 内核

Linux 是一个一体化内核(monolithic kernel)系统。这里的“内核”是指一个提供硬件抽象层、磁盘及文件系统控制、多任务等功能的系统软件,一个内核不是一套完整的操作系统。一套建立在 Linux 内核之上的完整操作系统叫作 Linux 操作系统,或是 GNU/Linux。

Linux 操作系统的灵魂是 Linux 内核,内核为系统其他部分提供系统服务。ARM-Linux 内核是专门为适应 ARM 体系结构而设计的 Linux 内核,它负责实现整个系统的进程管理和调度、内存管理、文件管理、设备管理和网络管理等主要系统功能。

5.1 ARM-Linux 概述

5.1.1 GNU/Linux 操作系统的基本体系结构

如图 5-1 所示 GNU/Linux 操作系统的基本体系结构。从图 5-1 中可以看到,GNU/Linux 被分成了两个空间。

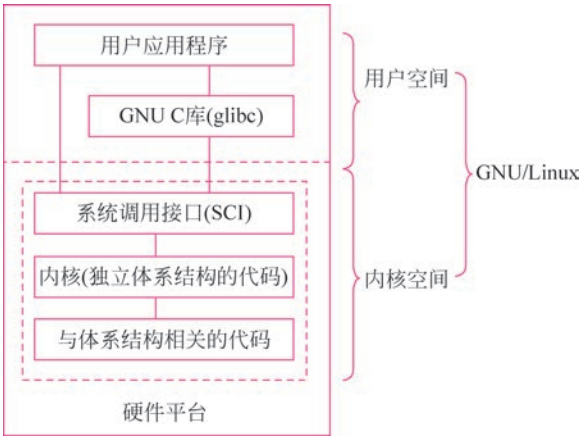

图 5-1 GNU/Linux 操作系统的基本体系结构

相对于操作系统其他部分,Linux 内核具有很高的安全级别和严格的保护机制。这种机制确保应用程序只能访问许可的资源,而不许可的资源是拒绝被访问的。因此系统设计者将内核和上层的应用程序进行抽象隔离,分别称之为内核空间和用户空间,如图 5-1 所示。

用户空间包括用户应用程序和 GNU C 库(glibc 库),负责执行用户应用程序。在该空

间中，一般的应用程序由 glibc 库间接调用系统调用接口而不是直接调用内核的系统调用接口去访问系统资源，这样做的主要理由是内核空间和用户空间的应用程序使用的是不同的保护地址空间。每个用户空间的进程都使用自己的虚拟地址空间，而内核则占用单独的地址空间。从面向对象的思想出发，glibc 库对内核的系统调用接口做了一层封装。

用户空间的下面是内核空间，Linux 内核空间可以进一步划分成 3 层。最上面是系统调用接口（System Call Interface，SCI），它是用户空间与内核空间的桥梁，用户空间的应用程序通过这个统一接口来访问系统中的硬件资源，通过此接口，所有的资源访问都是在内核的控制下执行的，以免用户程序对系统资源的越权访问，从而保障了系统的安全和稳定。从功能上看，系统调用接口实际上是一个非常有用的函数调用多路复用器和多路分解服务器。用户可以在./Linux/kernel 中找到系统调用接口的实现代码。系统调用接口之下是内核代码部分，实际可以更精确地定义为独立于体系结构的内核代码。这些代码是 Linux 支持的所有处理器体系结构所通用的。在这些代码之下是依赖于体系结构的代码，构成了通常称为板级支持包（Board Support Package，BSP）的部分。这些代码用作给定体系结构的处理器和特定于平台的代码，一般位于内核的 arch 目录（./Linux/arch 目录）和 drivers 目录中。arch 目录含有诸如 x86、IA64、ARM 等体系结构的支持。drivers 目录含有块设备、字符设备、网络设备等不同硬件驱动的支持。

5.1.2 ARM-Linux 内核版本及特点

据前所述，ARM-Linux 内核是基于 ARM 处理器的 Linux 内核，因而 ARM-Linux 内核版本变化与 Linux 内核版本变化保持同步。由于 Linux 标准内核是针对 x86 处理器架构设计的，并不能保证在其他架构（如 ARM）上能正常运行。因而嵌入式 Linux 系统内核（如 ARM-Linux 内核）往往在标准 Linux 基础上通过安装补丁实现，如 ARM-Linux 内核就是对 Linux 安装 rmk 补丁形成的，只有安装了这些补丁，内核才能顺利地移植到 ARM-Linux 上。当然也可以通过已经安装好补丁的内核源代码包实现。

在 2.6 版本之前，Linux 内核版本的命名格式为“A.B.C”。数字 A 是内核版本号，版本号只有在代码和内核的概念有重大改变的时候才会改变，历史上有两次变化：第一次是 1994 年的 1.0 版，第二次是 1996 年的 2.0 版。2011 年，3.0 版发布，但这次在内核的概念上并没有发生大的变化。数字 B 是内核主版本号，主版本号根据传统的奇-偶系统版本编号来分配：奇数为开发版，偶数为稳定版。数字 C 是内核次版本号，次版本号是无论在内核增加安全补丁、修复错误（bug）、实现新的特性或者驱动时都会改变。

2004 年 2.6 版本发布之后，内核开发者觉得基于更短的时间为发布周期更有益，所以大约七年的时间里，内核版本号的前两个数一直保持是 2.6，第三个数随着发布次数增加，发布周期大约是两三个月。考虑到对某个版本的错误（bug）和安全漏洞的修复，有时也会出现第四个数字。2011 年 5 月 29 日，设计者 Linus Torvalds 宣布为了纪念 Linux 发布 20 周年，在 2.6.39 版本发布之后，内核版本将升级到 3.0。Linux 继续使用在 2.6.0 版本引入的基于时间的发布规律，但是使用第二个数字——例如在 3.0 发布的几个月之后发布 3.1，同时当需要修复错误（bug）和安全漏洞的时候，增加一个数字（现在是第三个数）来表示，如 3.0.18。

在 Linux 内核官网上可以看到主要有 3 种类型的内核版本，如图 5-2 所示。

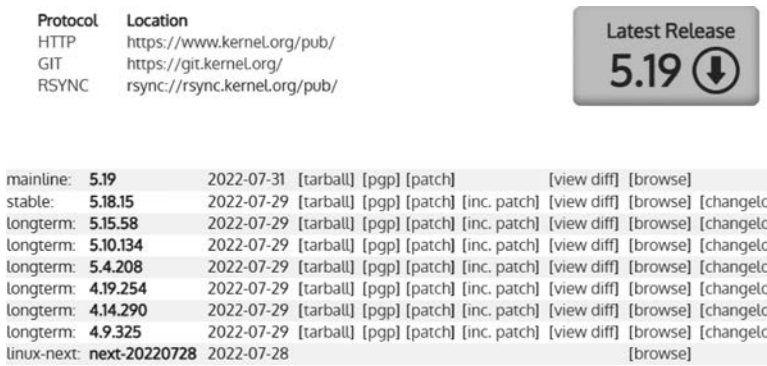

图 5-2 Linux 内核当前可支持版本一览

mainline 是主线版本，目前的主线版本号为 5.19。

stable 是稳定版，由 mainline 在时机成熟时发布，稳定版也会在相应版本号的主线上提供错误修复和安全补丁。

longterm 是长期支持版，目前还处在长期支持版的有 6 个版本的内核，等到不再支持长期支持版的内核时，也会标记 EOL(停止支持)。

操作系统内核主要可以分为两大体系结构：单内核和微内核。单内核中所有的部分都集中在一起，而且所有的部件在一起编译连接。这样做的好处比较明显，系统各部分直接沟通，系统响应速度高和 CPU 利用率好，而且实时性好。但是单内核的不足也显而易见，当系统较大时体积也较大，不符合嵌入式系统容量小、资源有限的特点。

微内核是将内核中的功能划分为独立的过程，每个过程被定义为一个服务器，不同的服务器保持独立并运行在各自的地址空间。这种体系结构在内核中只包含了一些基本的内核功能，如创建删除任务、任务调度、内存管理和中断处理等部分，而文件系统、网络协议栈等部分是在用户内存空间运行的。这种结构虽然执行效率不如单内核，但是大幅减小了内核体积、同时有利于系统的维护、升级和移植。

Linux 是一个内核运行在单独的内核地址空间的单内核，但是汲取了微内核的精华，如模块化设计、抢占式内核、支持内核线程及动态装载内核模块等特点。以 2.6 版本为例，其主要特点有：

(1) 支持动态加载内核模块机制。

(2) 支持对称多处理机制(SMP)。

(3) $O(1)$的调度算法。

(4) Linux 内核可抢占，Linux 内核具有允许在内核中运行的任务优先执行的能力。

(5) Linux 不区分线程和其他一般的进程，对内核来说，所有的进程都一样(仅部分共享资源)。

(6) Linux 提供具有设备类的面向对象的设备模块、热插拔事件，以及用户空间的设备文件系统。

5.1.3 ARM-Linux 内核的主要架构及功能

图 5-3 说明了 Linux 内核的主要架构。根据内核的核心功能，Linux 内核具有 5 个主要的子系统，分别负责进程管理、内存管理、虚拟文件系统、进程间通信和网络管理。

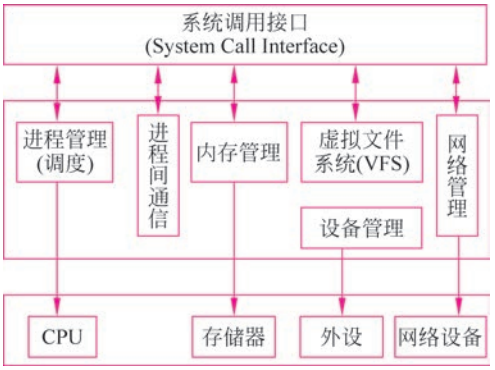

图 5-3　Linux 内核的主要架构

1. 进程管理

进程管理负责管理 CPU 资源，以便让各个进程能够以尽量公平的方式访问 CPU。进程管理负责进程的创建和销毁，并处理它们和外部世界之间的连接（输入/输出）。除此之外，控制进程如何共享调度器也是进程管理的一部分。概括来说，内核进程管理活动就是在单个或多个 CPU 上实现了多个进程的抽象。进程管理源代码可参考./Linux/kernel 目录。

2. 内存管理

Linux 内核所管理的另外一个重要资源是内存。内存管理策略是决定系统性能好坏的一个关键因素。内核在有限的可用资源之上为每个进程都创建了一个虚拟空间。内存管理的源代码可以在./Linux/mm 中找到。

3. 虚拟文件系统

文件系统在 Linux 内核中具有十分重要的地位，用于对外设的驱动和存储，隐藏了各种硬件的具体细节。Linux 引入了虚拟文件系统（Virtual File System，VFS），为用户提供了统一、抽象的文件系统界面，以支持越来越繁杂的具体的文件系统。Linux 内核将不同功能的外部设备（例如 Disk 设备（硬盘、磁盘、NAND Flash、Nor Flash 等）、输入/输出设备、显示设备等）抽象为可以通过统一的文件操作接口来访问的形式。Linux 中的绝大部分对象都可以视为文件并进行相关操作。

4. 进程间通信

不同进程之间的通信是操作系统的基本功能之一。Linux 内核通过支持 POSIX 规范中标准的 IPC（Inter Process Communication，进程间通信）机制和其他许多广泛使用的 IPC 机制实现进程间通信。IPC 不管理任何硬件，它主要负责 Linux 系统中进程之间的通信。比如 UNIX 中最常见的管道、信号量、消息队列和共享内存等。另外，信号（signal）也常被用来作为进程间的通信手段。Linux 内核支持 POSIX 规范的信号及信号处理并广泛应用。

5. 网络管理

网络管理提供了各种网络标准的存取和各种网络硬件的支持，负责管理系统的网络设备，并实现多种多样的网络标准。网络接口可以分为网络设备驱动程序和网络协议。

这 5 个子系统相互依赖，缺一不可，但是相对而言进程管理处于比较重要的地位，其他子系统的挂起和恢复进程的运行都必须依靠进程调度子系统的参与。当然，其他子系统的地位也非常重要：调度程序的初始化及执行过程中需要内存管理模块分配其内存地址空间

并进行处理；进程间通信需要内存管理实现进程间的内存共享；而内存管理利用虚拟文件系统支持数据交换，交换进程(swapd)定期由调度程序调度；虚拟文件系统需要使用网络接口实现网络文件系统，而且使用内存管理子系统实现内存设备管理，同时虚拟文件系统实现了内存管理中内存的交换。

除了这些依赖关系外，内核中的所有子系统还依赖于一些共同的资源。这些资源包括所有子系统都用到的过程。例如，分配和释放内存空间的过程，打印警告或错误信息的过程，还有系统的调试例程等。

5.1.4 Linux 内核源代码目录结构


视频讲解

为了实现 Linux 内核的基本功能，Linux 内核源代码的各个目录也大致与此相对应，其组成如下：

arch 目录包括了所有和体系结构相关的核心代码。它下面的每一个子目录都代表一种 Linux 支持的体系结构，例如，ARM 就是 ARM CPU 及与之相兼容体系结构的子目录。

include 目录包括编译核心所需要的大部分头文件，例如，与平台无关的头文件在 include/Linux 子目录下。

init 目录包含核心的初始化代码，需要注意的是，该代码不是系统的引导代码。

mm 目录包含了所有的内存管理代码。与具体硬件体系结构相关的内存管理代码位于 arch/ * /mm 目录下。

drivers 目录中是系统中所有的设备驱动程序。它又进一步划分成几类设备驱动，如字符设备、块设备等。每一种设备驱动均有对应的子目录。

ipc 目录包含了核心进程间的通信代码。

modules 目录存放了已建好的、可动态加载的模块。

fs 目录存放 Linux 支持的文件系统代码。不同的文件系统有不同的子目录对应，如 jffs2 文件系统对应的就是 jffs2 子目录。

kernel 目录存放内核管理的核心代码。另外，与处理器结构相关的代码都放在 arch/ * /kernel 目录下。

net 目录中是核心的网络代码。

lib 目录包含了核心的库代码，但是与处理器结构相关的库代码被放在 arch/ * /lib/目录下。

scripts 目录包含用于配置核心的脚本文件。

documentation 目录下是一些文档，是对目录作用的具体说明。

5.2 ARM-Linux 进程管理

进程是处于执行期的程序以及它所管理的资源的总称，这些资源包括如打开的文件、挂起的信号、进程状态、地址空间等。程序并不是进程，实际上两个或多个进程不仅有可能执行同一程序，而且有可能共享地址空间等资源。

进程管理是 Linux 内核中最重要的子系统，它主要提供对 CPU 的访问控制。由于计算机中的 CPU 资源是有限的，而众多的应用程序都要使用 CPU 资源，所以需要“进程调度子

系统”对 CPU 进行调度管理。进程调度子系统包括 4 个子模块，如图 5-4 所示，它们的功能如下：

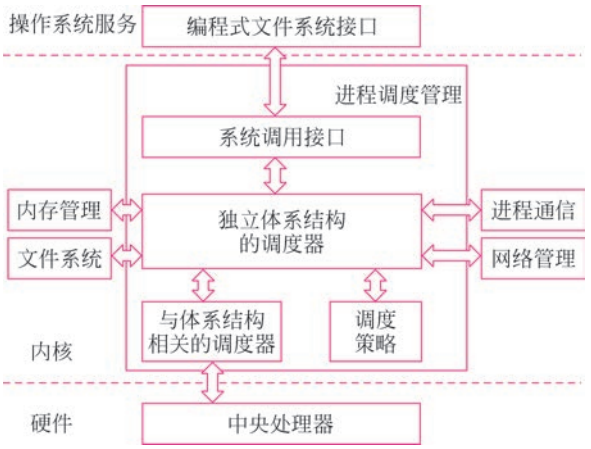

图 5-4　Linux 进程调度子系统基本架构

（1）调度策略（Scheduling Policy）模块。该模块实现进程调度的策略，它决定哪个（或者哪几个）进程将拥有 CPU 资源。

（2）与体系结构相关的调度器（Architecture-specific Schedulers）模块。该模块涉及体系结构相关的部分，用于将对不同 CPU 的控制抽象为统一的接口。这些控制功能主要在 suspend 和 resume 进程时使用，包含 CPU 的寄存器访问、汇编指令操作等。

（3）独立体系结构的调度器（Architecture-independent Scheduler）模块。该模块涉及体系结构无关的部分，会和调度策略模块沟通，决定接下来要执行哪个进程，然后通过与体系结构相关的调度器模块指定的进程予以实现。

（4）系统调用接口（System Call Interface）。进程调度子系统通过系统调用接口将需要提供给用户空间的接口开放出去，同时屏蔽掉不需要用户空间程序关心的细节。

5.2.1　进程的表示和切换

Linux 内核通过一个被称为进程描述符的 task_struct 结构体（也叫进程控制块）来管理进程，这个结构体记录了进程的最基本的信息，它的所有域按功能可以分为状态信息、链接信息、各种标志符、进程间通信信息、时间和定时器信息、调度信息、文件系统信息、虚拟内存信息、处理器环境信息等。进程描述符中不仅包含了许多描述进程属性的字段，而且包含了一系列指向其他数据结构的指针。内核将每个进程的描述符放在一个叫作任务队列的双向循环链表中，它定义在./include/Linux/sched.h 文件中。

```
struct task_struct {
     volatile long state;          /* 进程状态, -1 unrunnable,0 runnable,>0 stopped */
     void *stack;
     atomic_t usage;
     unsigned int flags;           /* 每个进程的标志 */
     unsigned int ptrace;
#ifdef CONFIG_SMP
```

```
        struct task_struct * wake_entry;
        int on_cpu;
#endif
        int on_rq;
        int prio,static_prio,normal_prio;     /* 优先级和静态优先级 */
        unsigned int rt_priority;
        const struct sched_class * sched_class;
        struct sched_entity se;
        struct sched_rt_entity rt;
...
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define __TASK_STOPPED          4
#define __TASK_TRACED           8
#define EXIT_ZOMBIE             16
#define EXIT_DEAD               32
#define TASK_DEAD               64
#define TASK_WAKEKILL           128
#define TASK_WAKING             256
...
```

系统中的每个进程都必然处于以上所列进程状态中的一种。这里对进程状态给予说明。

TASK_RUNNING 表示进程要么正在执行,要么正要准备执行。

TASK_INTERRUPTIBLE 表示进程被阻塞(睡眠),直到某个条件变为真。条件一旦达成,进程的状态就被设置为 TASK_RUNNING。

TASK_UNINTERRUPTIBLE 的意义与 TASK_INTERRUPTIBLE 基本类似,除了不能通过接收一个信号来唤醒以外。

__ TASK_STOPPED 表示进程被停止执行。

__ TASK_TRACED 表示进程被 debugger 等进程监视。

TASK-WAKEKILL 该状态表示当进程收到致命错误信号时唤醒进程。

TASK_WAKING 状态说明该任务正在唤醒,其他唤醒操作均会失败,都被置为 TASK_DEAD 状态。

TASK_DEAD 表示一个进程在退出时,state 字段都被置于该状态。

EXIT_ZOMBIE 表示进程的执行被终止,但是其父进程还没有使用 wait()等系统调用来获知它的终止信息。

EXIT_DEAD 表示进程的最终状态,进程在系统中被删除时将进入该状态。

EXIT_ZOMBIE 和 EXIT_DEAD 也可以存放在 exit_state 成员中。

调度程序负责选择下一个要运行的进程,它在可运行态进程之间分配有限的处理器时间资源,使系统资源最大限度地发挥作用,实现多进程并发执行的效果。进程状态的切换过程如图 5-5 所示。

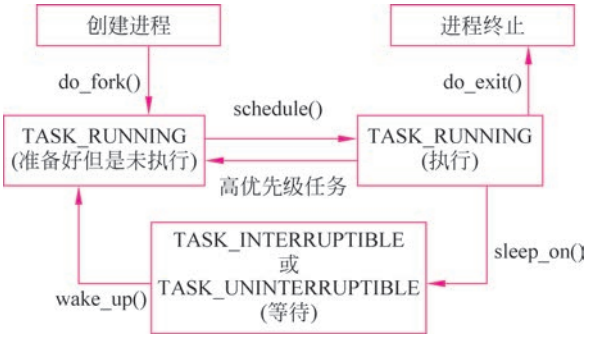

图 5-5　进程状态的切换

5.2.2　进程、线程和内核线程

在 Linux 内核中，内核是采用进程、线程和内核线程统一管理的方法实现进程管理的。内核将进程、线程和内核线程一视同仁，即内核使用唯一的数据结构 task_struct 来分别表示它们。内核使用相同的调度算法对这三者进行调度。并且内核也使用同一个函数 do_fork()来分别创建这 3 种执行线程(thread of execution)。执行线程通常是指任何正在执行的代码实例，比如一个内核线程、一个中断处理程序或一个进入内核的进程。Linux 内核的这种处理方法简洁方便，并且内核在统一处理这三者之余保留了它们本身所具有的特性。

本节首先介绍进程、线程和内核线程的概念，然后结合进程、线程和内核线程的特性分析进程在内核中的功能。

进程是系统资源分配的基本单位，线程是程序独立运行的基本单位。线程有时候也被称作小型进程，这是因为多个线程之间是可以共享资源的，而且多个线程之间的切换所花费的代价远比进程低。在用户态下，使用最广泛的线程操作接口即为 POSIX 线程接口，即 pthread。通过这组接口可以进行线程的创建以及多线程之间的并发控制等。

如果内核要对线程进行调度，那么线程必须像进程那样在内核中对应一个数据结构。进程在内核中有相应的进程描述符，即 task_struct 结构。事实上，从 Linux 内核的角度而言，并不存在线程这个概念。内核对线程并没有设立特别的数据结构，而是与进程一样使用 task_struct 结构进行描述。也就是说，线程在内核中也是以一个进程的形式存在的，只不过它比较特殊，它和同类的进程共享某些资源，比如进程地址空间、进程的信号、打开的文件等。这类特殊的进程称为轻量级进程(light weight process)。

按照这种线程机制的定义，每个用户态的线程都和内核中的一个轻量级进程相对应。多个轻量级进程之间共享资源，从而体现了多线程之间资源共享的特性。同时这些轻量级进程跟普通进程一样由内核进行独立调度，从而实现了多个进程之间的并发执行。

在内核中还有一种特殊的线程，称为内核线程(kernel thread)。由于在内核中对进程和线程不做区分，因此也可以将其称为内核进程。内核线程在内核中也是通过 task_struct 结构来表示的。

内核线程和普通进程一样也是内核调度的实体，但是有着明显的不同。首先，内核线程永远都运行在内核态，而不同进程既可以运行在用户态也可以运行在内核态。从地址空间的使用角度来讲，内核线程只能使用大于 3GB 的地址空间，而普通进程则可以使用整个

4GB 的地址空间。其次，内核线程只能调用内核函数无法使用用户空间的函数，而普通进程必须通过系统调用才能使用内核函数。

5.2.3 进程描述符 task_struct 的几个特殊字段

上述 3 种执行线程在内核中都使用统一的数据结构 task_struct 来表示。这里简单介绍进程描述符中几个比较特殊的字段，它们分别指向代表进程所拥有的资源的数据结构。

(1) mm 字段：指向 mm_struct 结构的指针，该类型用来描述进程整个的虚拟地址空间。其数据结构如下所示：

```
struct mm_struct * mm, * active_mm;
# ifdef CONFIG_COMPAT_BRK
    unsigned brk_randomized:1;
# endif
# if defined(SPLIT_RSS_COUNTING)
    struct task_rss_stat    rss_stat;
# endif
```

(2) fs 字段：指向 fs_struct 结构的指针，该字段用来描述进程所在文件系统的根目录和当前进程所在的目录信息。

(3) files 字段：指向 files_struct 结构的指针，该字段用来描述当前进程所打开文件的信息。

(4) signal 字段：指向 signal_struct 结构(信号描述符)的指针，该字段用来描述进程所能处理的信号。其数据结构如下：

```
/ * signal handlers * /
    struct signal_struct * signal;
    struct sighand_struct * sighand;
    sigset_t blocked,real_blocked;
    sigset_t saved_sigmask; / * 如果 set_restore_sigmask 被使用,则存储该值 * /
    struct sigpending pending;
    unsigned long sas_ss_sp;
    size_t sas_ss_size;
int ( * notifier)(void * priv);
    void * notifier_data;
    sigset_t * notifier_mask;
```

对于普通进程来说，上述字段分别指向具体的数据结构以表示该进程所拥有的资源。对应每个线程而言，内核通过轻量级进程与其进行关联。轻量级进程之所轻量，是因为它与其他进程共享上述所提及的进程资源。比如进程 A 创建了线程 B，则 B 线程会在内核中对应一个轻量级进程。这个轻量级进程对应一个进程描述符，而且 B 线程的进程描述符中的某些代表资源指针会和 A 进程中对应的字段指向同一个数据结构，这样就实现了多线程之间的资源共享。

内核线程只运行在内核态，并不需要像普通进程那样的独立地址空间。因此内核线程的进程描述符中的 mm 指针即为 NULL。

5.2.4 do_fork()函数

进程、线程以及内核线程都有对应的创建函数，不过这三者所对应的创建函数最终在内核中都是由 do_fork()进行创建的，具体的调用关系如图 5-6 所示。

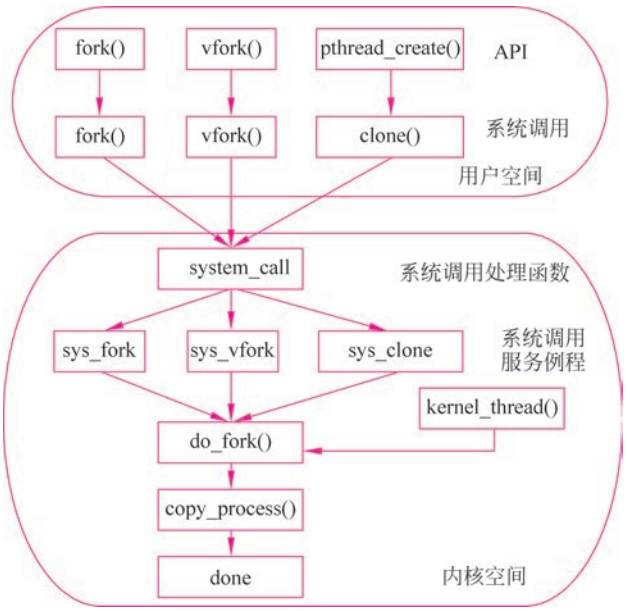

图 5-6 do_fork()函数对于进程、线程以及内核线程的应用

从图 5-6 中可以看出，内核中创建进程的核心函数即为 do_fork()，该函数的原型如下：

```
long do_fork(unsigned long clone_flags,
                    unsigned long stack_start,
                    struct pt_regs *regs,
                    unsigned long stack_size,
                    int __user *parent_tidptr,
                    int __user *child_tidptr)
```

该函数的参数的功能说明如下。

- clone_flags：代表进程各种特性的标志。低字节指定子进程结束时发送给父进程的信号代码，一般为 SIGCHLD 信号，剩余 3 字节是若干标志或运算的结果。
- stack_start：子进程用户态堆栈的指针，该参数会被赋值给子进程的 esp 寄存器。
- regs：指向通用寄存器值的指针，当进程从用户态切换到内核态时通用寄存器中的值会被保存到内核态堆栈中。
- stack_size：未被使用，默认值为 0。
- parent_tidptr：该子进程的父进程用户态变量的地址，仅当 CLONE_PARENT_SETTID 被设置时有效。
- child_tidptr：该子进程用户态变量的地址，仅当 CLONE_CHILD_SETTID 被设置时有效。

既然进程、线程和内核线程在内核中都是通过 do_fork()完成创建的，那么 do_fork()如

何体现其功能的多样性？其实，clone_flags 参数在这里起到了关键作用，通过选取不同的标志，从而保证了 do_fork()函数实现多角色——创建进程、线程和内核线程——功能的实现。clone_flags 参数可取的标志很多，下面只介绍其中几个主要的标志。

- CLONE_VIM：子进程共享父进程内存描述符和所有的页表。
- CLONE_FS：子进程共享父进程所在文件系统的根目录和当前工作目录。
- CLONE_FILES：子进程共享父进程打开的文件。
- CLONE_SIGHAND：子进程共享父进程的信号处理程序、阻塞信号和挂起的信号。使用该标志必须同时设置 CLONE_VM 标志。

如果创建子进程时设置了上述标志，那么子进程会共享这些标志所代表的父进程资源。

5.2.5 进程的创建


视频讲解

在用户态程序中，可以通过接口函数 fork()、vfork()和 clone()创建进程，这 3 个函数在库中分别对应同名的系统调用。系统调用函数通过 128 号软中断进入内核后，会调用相应的系统调用服务例程。这 3 个函数对应的服务例程分别是 sys_fork()、sys_vfork()和 sys_clone()。

```
int sys_fork(struct pt_regs *regs)
  {
          return do_fork(SIGCHLD,regs->sp,regs,0,NULL,NULL);
  }
  int sys_vfork(struct pt_regs *regs)
  {
          return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD,regs->sp,regs,0,NULL,NULL);
  }
  long
  sys_clone(unsigned long clone_flags,unsigned long newsp,
  void __user *parent_tid,void __user *child_tid,struct pt_regs *regs)
  {
          if (!newsp)
                  newsp = regs->sp;
          return do_fork(clone_flags,newsp,regs,0,parent_tid,child_tid);
  }
```

由上述系统调用服务例程的源代码可以发现，3 个系统服务例程内部都调用了 do_fork()，主要差别在于第一个参数所传的值不同。这也正好导致由这 3 个进程创建函数所创建的进程有不同的特性。下面予以简单说明。

(1) fork()。由于 do_fork()中 clone_flags 参数除了子进程结束时返回给父进程的 SIGCHLD 信号外并无其他特性标志，因此由 fork()创建的进程不会共享父进程的任何资源。子进程会完全复制父进程的资源，也就是说，父子进程相对独立。不过由于写时复制技术(Copy On Write)的引入，子进程可以只读父进程的物理页，只有当父进程或者子进程去写某个物理页时，内核此时才会将这个页的内容复制到一个新的物理页，并把这个新的物理页分配给正在写的进程。

(2) vfork()。在 do_fork()中的 clone_flags 使用了 CLONE_VFORK 和 CLONE_VM 两个标志。CLONE_VFORK 标志使得子进程先于父进程执行，父进程会阻塞到子进程结

束或执行新的程序。CLONE_VM 标志使得子进程可以共享父进程的内存地址空间(父进程的页表项除外)。在引入写时复制技术前,vfork()适用子进程形成后立即执行 execv()的情形。因此,vfork()现如今已经没有特别的使用之处,因为写时复制技术完全可以取代它创建进程时所带来的高效性。

(3) clone()。这里 clone 通常用于创建轻量级进程。通过传递不同的标志可以对父子进程之间数据的共享和复制做精确的控制,一般 flags 的取值为 CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND。由上述标志可以看到,轻量级进程通常共享父进程的内存地址空间、父进程所在文件系统的根目录以及工作目录信息、父进程当前打开的文件以及父进程所拥有的信号处理函数。


视频讲解

5.2.6 线程和内核线程的创建

每个线程在内核中对应一个轻量级进程,两者的关联是通过线程库完成的。因此通过 pthread_create()创建的线程最终在内核中是通过 clone()完成创建的,而 clone()最终调用 do_fork()。

一个新内核线程的创建是通过在现有的内核线程中使用 kernel_thread()而创建的,其本质也是向 do_fork()提供特定的 flags 标志而创建的。

```
Int kernel_thread(int ( * fn)(void * ),void  * arg,unsigned long flags)
{
return do_fork(flags|CLONE_VM|CLONE_UNTRACED,0,&regs,0,NULL,NULL);
 }
```

从上面的组合 flags 标志可以看出,新的内核线程至少会共享父内核线程的内存地址空间。这样做其实是为了避免赋值调用线程的页表,因为内核线程无论如何都不会访问用户地址空间。CLONE_UNTRACED 标志保证内核线程不会被任何进程所跟踪。

5.2.7 进程的执行——exec 函数族

fork()函数是用于创建一个子进程,该子进程几乎复制了父进程的所有内容。但是这个新创建的进程是如何执行的呢? 在 Linux 中使用 exec 函数族来解决这个问题,exec 函数族提供了一个在进程中启动另一个程序执行的方法。它可以根据指定的文件名或目录名找到可执行文件,并用它来取代原调用进程的数据段、代码段和堆栈段,在执行完之后,除了进程号外,原调用进程的内容其他全部被新的进程替换了。

在 Linux 中使用 exec 函数族主要有两种情况。

(1) 当进程认为自己不能再为系统和用户做出任何贡献时,就可以调用 exec 函数族中的任意一个函数让自己重生。

(2) 如果一个进程希望执行另一个程序,那么它就可以调用 fork()函数新建一个进程,然后调用 exec 函数族中的任意一个函数,这样看起来就像通过执行应用程序而产生了一个新进程。

相对来说第二种情况非常普遍。实际上,在 Linux 中并没有 exec()函数,而是有 6 个以 exec 开头的函数,表 5-1 列举了 exec 函数族的 6 个成员函数的语法。

表 5-1 exec 函数族成员函数语法

所需头文件	#include <unistd.h>
函数原型	int execl(const char *path,const char *arg,...)
	int execv(const char *path,char *const argv[])
	int execle(const char *path,const char *arg,...,char *const envp[])
	int execve(const char *path,char *const argv[],char *const envp[])
	int execlp(const char *file,const char *arg,...)
	int execvp(const char *file,char *const argv[])
函数返回值	-1：出错

事实上，这 6 个函数中真正的系统调用只有 execve()，其他 5 个都是库函数，它们最终都会调用 execve()。这里简要介绍 execve()的执行流程。

(1) 打开可执行文件，获取该文件的 file 结构。

(2) 获取参数区长度，将存放参数的页面清零。

(3) 对 Linux_binprm 结构的其他项进行初始化。这里的 Linux_binprm 结构用来读取并存储运行可执行文件的必要信息。

5.2.8 进程的终止

当进程终结时，内核必须释放它所占有的资源，并告知其父进程。进程的终止可以通过以下 3 个事件驱动：正常的进程结束、信号和 exit()函数的调用。进程的终结最终都要通过 do_exit()来完成(Linux/kernel/exit.c 中)。进程终结后，与进程相关的所有资源都要被释放，进程不可运行并处于 TASK_ZOMBIE 状态，此时进程存在的唯一目的就是向父进程提供信息。当父进程检索到信息后，或者通知内核该信息是无关信息后，进程所持有的剩余内存被释放。

这里 exit()函数所需的头文件为#include <stdlib.h>，函数原型是：

```
void exit(int status)
```

其中，status 是一个整型的参数，可以利用这个参数传递进程结束时的状态。一般来说，0 表示正常结束。其他的数值表示出现了错误，进程非正常结束。在实际编程时，可以用 wait()系统调用接收子进程的返回值，从而针对不同的情况进行不同的处理。

下面简要介绍 do_exit()的执行过程。

(1) 将 task_struct 中的标志成员设置 PF_EXITING，表明该进程正在被删除，释放当前进程占用的 mm_struct，如果没有其他进程使用，即没有被共享，则彻底释放它们。

(2) 如果进程排队等候 IPC 信号，则离开队列。

(3) 分别将文件描述符、文件系统数据、进程名字空间的引用计数递减。如果这些引用计数的数值降为 0，则表示没有进程在使用这些资源，可以释放。

(4) 向父进程发送信号：将当前进程的子进程的父进程重新设置为线程组中的其他线程或者 init 进程，并将进程状态设成 TASK_ZOMBIE。

(5) 切换到其他进程，处于 TASK_ZOMBIE 状态的进程不会再被调用。此时进程占用的资源就是内核堆栈、thread_info 结构、task_struct 结构，而进程存在的唯一目的就是向它

的父进程提供信息。父进程检索到信息或者通知内核那是无关的信息后，由进程所持有的剩余内存被释放，归还给系统使用。

5.2.9 进程的调度

由于进程、线程和内核线程使用统一数据结构来表示，因此内核对这三者并不作区分，也不会为其中某一个设立单独的调度算法。内核将这三者一视同仁，进行统一的调度。

1. Linux 调度时机

Linux 进程调度分为主动调度和被动调度两种方式。

主动调度随时都可以进行，内核可以通过 schedule()启动一次调度，当然也可以将进程状态设置为 TASK_INTERRUPTIBLE、TASK_UNINTERRUPTIBLE，暂时放弃运行而进入睡眠，用户空间也可以通过 pause()达到同样的目的。如果为这种暂时的睡眠加上时间限制，内核态有 schedule_timeout，用户态有 nanosleep()用于此目的。注意，内核中这种主动放弃是不可见的，而是隐藏在每一个可能受阻的系统调用中，如 open()、read()、select()等。被动调度发生在系统调用返回的前、中断异常处理返回前或者用户态处理软中断返回前。

从 Linux 2.6 内核后，Linux 实现了抢占式内核，即处于内核态的进程也可能被调度出去。比如一个进程正运行在内核态，此时一个中断发生使另一个高优先级进程就绪，在中断处理程序结束之后，Linux 2.6 内核之前的版本会恢复原进程的运行，直到该进程退出内核态才会引发调度程序；而 Linux 2.6 抢占式内核，在处理完中断后，会立即引发调度，切换到高优先级进程。为支持内核代码可抢占，在 Linux 2.6 内核中通过采用禁止抢占的自旋锁(spin_unlock_mutex)来保护临界区。在释放自旋锁时，同样会引发调度检查。而对那些长期持锁或禁止抢占的代码片段插入了抢占点，此时检查调度需求，以避免发生不合理的延迟。而在检查过程中，只要新的进程不需要持有该锁，调度进程很可能就会中止当前的进程来让另外一个进程运行。

2. 进程调度的一般原理

调度程序运行时，要在所有可运行的进程中选择最值得运行的进程。选择进程的依据主要有进程的调度策略(policy)、静态优先级(priority)、动态优先级(counter)以及实时优先级(rt-priority)4 个部分。policy 是进程的调度策略，用来区分实时进程和普通进程，Linux 从整体上区分为实时进程和普通进程，二者调度算法不同，实时进程优先于普通进程运行。进程依照优先级的高低被依次调用，实时优先级级别最高。

counter 是实际意义上的进程动态优先级，它是进程剩余的时间片，起始值就是 priority 的值。从某种意义上讲，所有位于当前队列的任务都将被执行并且都将被移到“过期”队列中(实时进程则例外，交互性强的进程也可能例外)。当这种情况发生时，队列就会被进行切换，原来的“过期”队列成为当前队列，而空的当前队列也就变成了过期队列。

在 Linux 中，用函数 goodness()综合 4 项依据及其他因素，赋予各影响因素权重(weight)，调度程序以优先级的高低作为选择进程的依据。

3. Linux O(1)调度

内核实现了一种新型的调度算法，不管有多少个线程在竞争 CPU，这种算法都可以在固定时间内进行操作。这种算法就称为 $O(1)$调度程序，这个名字就表示它调度多个线程所

使用的时间和调度一个线程所使用的时间是相同的。Linux 2.6 实现 $O(1)$调度，每个 CPU 都有两个进程队列，采用优先级为基础的调度策略。内核为每个进程计算出一个反映其运行“资格”的权值，然后挑选优先级最高的进程投入运行。在运行过程中，当前进程的资格随时间而递减，从而在下一次调度的时候原来资格较低的进程可能就有资格运行了。到所有进程的资格都为零时，就重新计算。

schedule()函数是完成进程调度的主要函数，并完成进程切换的工作。schedule()用于确定最高优先级进程的代码非常快捷高效，其性能的好坏对系统性能有着直接影响，它在/kernel/sched.c 中的定义如下：

```
asmlinkage void __sched schedule(void)
{
    struct task_struct *prev, *next;
    unsigned long *switch_count;
    struct rq *rq;
    int cpu;
need_resched:
    preempt_disable();
    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    rcu_sched_qs(cpu);
    prev = rq->curr;
    switch_count = &prev->nivcsw;
    release_kernel_lock(prev);
```

在上述代码中可以发现 schedule()函数中的两个重要变量：prev 指向当前正在使用 CPU 的进程，next 指向下一个将要使用 CPU 的进程。进程调度的一个很大的任务就是找到 next。

schedule 的主要工作可以分为两步。

(1) 找到 next。

- schedule()检查 prev 的状态。如果不是可运行状态，而且它没有在内核态被抢占，就应该从运行队列删除 prev 进程。不过，如果它是非阻塞挂起信号，而且状态为 TASH_INTERRUPTIBLE，那么函数就把该进程状态设置为 TASK_RUNNING，并将它插入运行队列。这个操作与把处理器分配给 prev 是不同的，它只是给 prev 一次选中执行的机会。在内核抢占的情况下，该步骤不会被执行。
- 检查本地运行队列中是否有进程。如果没有则在其他 CPU 的运行队列中迁移一部分进程过来。如果在单 CPU 系统或在其他 CPU 的运行队列中迁移进程失败，那么 next 只能选择 swapper 进程，然后马上跳去 switch_tasks 执行进程切换。
- 若本地运行队列中有进程，但没有活动进程队列为空集，也就是说，运行队列中的进程都在过期进程队列中，那么这时把活动进程队列改为过期进程队列，把原过期进程队列改为活动进程队列。空集用于接收过期进程。
- 在活动进程队列中搜索一个可运行进程。首先，schedule()搜索活动进程队列的集合位掩码的第一个非 0 位。当对应的优先级链表不为空时，就把位掩码的相应位置 1。因此，第一个非 0 位下标对应包含最佳运行进程的链表。随后，返回该链表的第一个进程。值得一提的是，在 Linux 2.6 下这一步能在很短的固定时间内完成。这时 next 就被找到了。

- 检查 next 是否是实时进程以及是否从 TASK_INTERRUPTIBLE 或 TASK_STOPPED 状态中被唤醒。如果这两个条件都满足，则重新计算其动态优先级。然后将 next 从原来的优先级撤离插入到新的优先级中。也就是说，实时进程是不会改变其优先级的。

(2) 切换进程。

找到 next 后，就可以实施进程切换了。

- 把 next 的进程描述符第一部分字段的内容装入硬件高速缓存。
- 清除 prev 的 TIF_NEED_RESCHED 的标志。
- 设置 prev 的进程切换时刻。
- 重新计算并设置 prev 的平均睡眠时间。
- 如果 prev ！ ＝ next，则切换 prev 和 next 硬件上下文。

这时，CPU 已经开始执行 next 进程了。

5.3 ARM-Linux 内存管理


视频讲解

5.3.1 ARM-Linux 内存管理概述


视频讲解

内存管理是 Linux 内核中最重要的子系统之一，它主要提供对内存资源的访问控制机制。这种机制主要涵盖了如下功能。

- 内存的分配和回收。内存管理记录每个内存单元的使用状态，为运行进程的程序段和数据段等需求分配内存空间，并在不需要时回收它们。
- 地址转换。当程序写入内存执行时，如果程序中编译时生成的地址（逻辑地址）与写入内存的实际地址（物理地址）不一致，则要把逻辑地主转换成物理地址。这种地址转换通常是由内存管理单元（Memory Management Unit，MMU）完成的。
- 内存扩充。由于计算机资源的迅猛发展，内存容量在不断变大。同时，当物理内存容量不足时，操作系统需要在不改变物理内存的情况下通过对外存的借用实现内存容量的扩充。最常见的方法包括虚拟存储、覆盖和交换等。
- 内存的共享与保护。所谓内存共享是指多个进程能共同访问内存中的同一段内存单元。内存保护是指防止内存中的各程序在执行中相互干扰，并保证对内存中信息访问的正确性。

Linux 系统会在硬件物理内存和进程所使用的内存（称作虚拟内存）之间建立一种映射关系，这种映射是以进程为单位的，因而不同的进程可以使用相同的虚拟内存，而这些相同的虚拟内存，可以映射到不同的物理内存上。

内存管理子系统包括 3 个子模块，其结构如图 5-7 所示。

(1) 与体系结构相关管理器子模块，涉及体系结构相关部分，提供用于访问硬件 Memory 的虚拟接口。

(2) 独立体系结构管理器子模块，涉及体系结构无关部分，提供所有的内存管理机制，包括以进程为单位的内存映射（memory mapping）、虚拟内存的交换技术（Swapping）等。

(3) 系统调用接口子模块。通过该接口，向用户空间程序应用程序提供内存的分配、释放，文件的映射等功能。

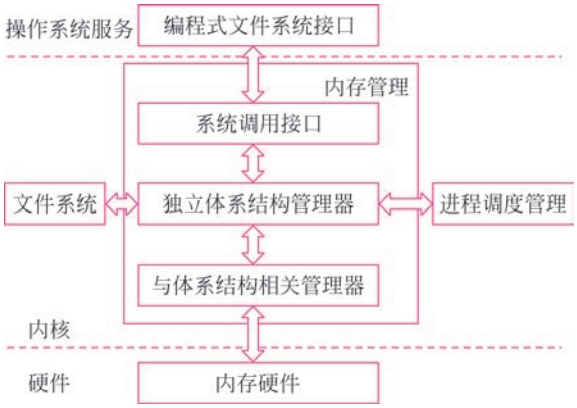

图 5-7 内存管理主要子系统架构

ARM-Linux 内核的内存管理功能是采用请求调页式的虚拟存储技术实现的。ARM-Linux 内核根据内存的当前使用情况动态换进换出进程页，通过外存上的交换空间存放换出页。内存与外存之间的相互交换信息是以页为单位进行的，这样的管理方法具有良好的灵活性，并具有很高的内存利用率。

5.3.2 ARM-Linux 虚拟存储空间及分布


视频讲解

32 位的 ARM 处理器具有 4GB 大小的虚拟地址容量，即每个进程的最大虚拟地址空间为 4GB，如图 5-8 所示。ARM-Linux 内核处于高端的 1GB 空间处，而低端的 3GB 属于用户空间，被用户程序所使用。所以在系统空间，即在内核中，虚拟地址与物理地址在数值上是相同的，至于用户空间的地址映射是动态的，根据需要分配物理内存，并且建立起具体进程的虚拟地址与所分配的物理内存间的映射。值得注意的是，系统空间的一部分不是映射到物理内存，而是映射到一些 I/O 设备，包括寄存器和一些小块的存储器。

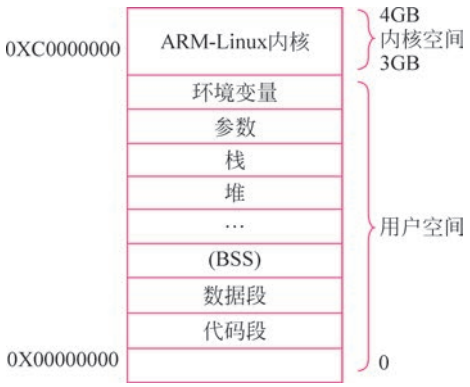

图 5-8 Linux 进程的虚拟内存空间及其组成(32 位平台)

这里简单说明进程对应的内存空间中所包含的 5 种不同的数据区。

- 代码段：代码段是用来存放可执行文件的操作指令，也就是说，它是可执行程序在内存中的镜像。代码段需要防止在运行时被非法修改，所以只准许读取操作，而不允许写入(修改)操作。
- 数据段：数据段用来存放可执行文件中已初始化全局变量，换句话说，就是存放程序静态分配的变量和全局变量。
- BSS 段：BSS 段包含了程序中未初始化的全局变量，在内存中 BSS 段全部置零。
- 堆(heap)：堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用 malloc()等函数分配内存时，新分配的内存就被动态添加到堆上(堆被扩张)。当利用 free()等函数释放内存时，被释放的内存从堆中被剔除(堆被缩减)。

- 栈：栈是用户存放程序临时创建的局部变量，也就是函数括号“{}”中定义的变量（但不包括 static 声明的变量，static 意味着在数据段中存放变量）。除此以外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的先进先出特点，所以栈特别适合用来保存/恢复调用现场。从这个意义上讲，堆栈也被看成一个寄存、交换临时数据的内存区。


视频讲解

5.3.3 进程空间描述

1. 关键数据结构描述

一个进程的虚拟地址空间主要由两个数据结来描述：一个是最高层次的 mm_struct；另一个是较高层次的 vm_area_structs。最高层次的 mm_struct 结构描述了一个进程的整个虚拟地址空间。每个进程只有一个 mm_struct 结构，在每个进程的 task_struct 结构中，有一个指向该进程的 mm_struct 结构的指针，每个进程与用户相关的各种信息都存放在 mm_struct 结构体中，其中包括本进程的页目录表的地址，本进程的用户区的组成情况等重要信息。可以说，mm_struct 结构是对整个用户空间的描述。

mm_struct 用来描述一个进程的整个虚拟地址空间，在./include/Linux/mm_types.h 中描述如下：

```
struct mm_struct {
     struct vm_area_struct * mmap;                    //指向虚拟区间(VMA)链表
struct rb_root mm_rb;                                 //指向 red_black 树
     struct vm_area_struct * mmap_cache;              //指向最近找到的虚拟区间
#ifdef CONFIG_MMU
     unsigned long ( * get_unmapped_area) (struct file * filp,
                unsigned long addr, unsigned long len,
                unsigned long pgoff, unsigned long flags);
     void ( * unmap_area) (struct mm_struct * mm, unsigned long addr);
#endif
     unsigned long mmap_base;
     unsigned long task_size;
     unsigned long cached_hole_size;
     unsigned long free_area_cache;
     pgd_t * pgd;                                     //指向进程的页目录
     atomic_t mm_users;
     int map_count;
     spinlock_t page_table_lock;                      //保护任务页表和 mm->rss
struct rw_semaphore mmap_sem;
     struct list_head mmlist;                         //所有活动的(active)mm 链表
     unsigned long hiwater_rss;
     unsigned long hiwater_vm;
     unsigned long total_vm, locked_vm, shared_vm, exec_vm;
     unsigned long stack_vm, reserved_vm, def_flags, nr_ptes;
     unsigned long start_code, end_code, start_data, end_data;
                                                      // start_code 代码段起始地址,
                                                      // end_code 代码段结束地址,
                                                      // start_data 数据段起始地址,
                                                      // start_end 数据段结束地址
     unsigned long start_brk, brk, start_stack;       // start_brk 和 brk 记录有关堆的信息
                                                      // start_brk 是用户虚拟地址空间初
                                                      // 始化时,堆的结束地址,brk 是当前
                                                      // 堆的结束地址,start_stack 是栈的
                                                      // 起始地址
```

```
    unsigned long arg_start, arg_end, env_start, env_end;  // arg_start 参数段的起始地址,
                                                           // arg_end 参数段的结束地址,
                                                           // env_start 环境段的起始地址,
                                                           // env_end 环境段的结束地址
    unsigned long saved_auxv[AT_VECTOR_SIZE];
    struct mm_rss_stat rss_stat;
    struct Linux_binfmt * binfmt;
    cpumask_var_t cpu_vm_mask_var;
    mm_context_t context;                                  // Architecture - specific
                                                           // MM context,是与平台相关的结构
    unsigned int faultstamp;
    unsigned int token_priority;
    unsigned int last_interval;
         atomic_t oom_disable_count;
    unsigned long flags;
    };
```

Linux 内核中对应进程内存区域的数据结构是 vm_area_struct，内核将每个内存区域作为一个单独的内存对象管理，相应的操作也一致。每个进程的用户区是由一组 vm_area_struct 结构体组成的链表来描述的。用户区的每个段（如代码段、数据段和栈等）都由一个 vm_area_struct 结构体描述，其中包含了本段的起始虚拟地址和结束虚拟地址，也包含了当发生缺页异常时如何找到本段在外存上的相应内容（如通过 nopage()函数）。

vm_area_struct 是描述进程地址空间的基本管理单元，如上所述，vm_area_struct 结构是以链表形式链接，不过为了方便查找，内核又以红黑树（red_black tree）的形式组织内存区域，以便降低搜索耗时。值得注意的是，并存的两种组织形式并非冗余：链表用于需要遍历全部节点的时候用，而红黑树适用于在地址空间中定位特定内存区域的时候。内核为了内存区域上的各种不同操作都能获得高性能，所以同时使用了这两种数据结构。

图 5-9 反映了进程地址空间的管理模型。

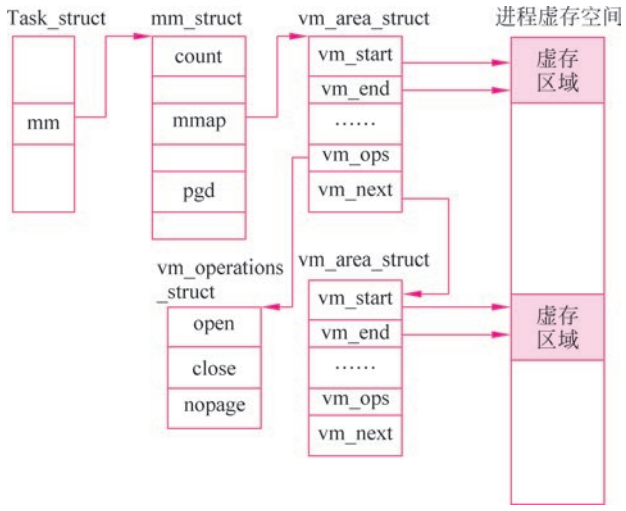

图 5-9 Linux 进程地址空间的管理模型

图 5-9 中的内存映射（mmap）是 Linux 操作系统的一个很大特色，它可以将系统内存映射到一个文件（设备）上，以便通过访问文件内容来达到访问内存的目的。这样做的最大好

处是提高了内存访问速度，并且可以利用文件系统的接口编程（设备在 Linux 中作为特殊文件处理）访问内存，降低了开发难度。许多设备驱动程序便是利用内存映射功能将用户空间的一段地址关联到设备内存上，无论何时，只要内存在分配的地址范围内进行读/写，实际上就是对设备内存的访问。同时对设备文件的访问等同于对内存区域的访问，也就是说，通过文件操作接口可以访问内存。vm_area_struct 结构体描述如下：

```
struct vm_area_struct {
      struct mm_struct * vm_mm;
      unsigned long vm_start;
      unsigned long vm_end;
      struct vm_area_struct * vm_next, * vm_prev;
      pgprot_t vm_page_prot;
      unsigned long vm_flags;
      struct rb_node vm_rb;
      union {
            struct {
                  struct list_head list;
                  void * parent;
                  struct vm_area_struct * head;
            } vm_set;
            struct raw_prio_tree_node prio_tree_node;
      } shared;
```

2. Linux 的分页模型

由于分段机制和 Intel 处理器相关联，在其他的硬件系统中，可能并不支持分段式内存管理，因此在 Linux 中，操作系统使用分页的方式管理内存。在 Linux 2.6 中，Linux 采用了通用的四级页表结构，4 种页表分别称为：页全局目录、页上级目录、页中间目录、页表。

为了实现跨平台运行 Linux 的目标（如在 ARM 平台上），设计者提供了一系列转换宏使得 Linux 内核可以访问特定进程的页表。该系列转换宏实现逻辑页表和物理页表在逻辑上的一致。这样内核就无须知道页表入口的结构和排列方式。采用这种方法后，在使用不同级数页表的处理器架构中，Linux 就可以使用相同的页表操作代码了。

分页机制将整个线性地址空间及整个物理内存看成由许多大小相同的存储块组成的，并将这些块作为页（虚拟空间分页后每个单位称为页）或页帧（物理内存分页后每个单位称为页帧）进行管理。当不考虑内存访问权限时，线性地址空间的任何一页理论上可以映射为物理地址空间中的任何一个页帧。Linux 内核的分页方式是一般以 4KB 单位划分页，并且保证页地址边界对齐，即每一页的起始地址都应被 4K 整除。在 4KB 的页单位下，32 位机的整个虚拟空间就被划分成了 220 个页。操作系统按页为每个进程分配虚拟地址范围，理论上根据程序需要最大可使用 4GB 的虚拟内存。但由于操作系统需要保护内核进程内存，所以将内核进程虚拟内存和用户进程虚拟内存分离，前者可用空间为 1GB 虚拟内存，后者为 3GB 虚拟内存。

创建进程 fork()、程序载入 execve()、映射文件 mmap()、动态内存分配 malloc()/brk() 等进程相关操作都需要分配内存给进程。而此时进程申请和获得的内存实际为虚拟内存，获得的是虚拟地址。值得注意的是，进程对内存区域的分配最终都会归结到 do_mmap()函数上来（brk 调用被单独以系统调用实现，不用 do_mmap()函数）。同样，释放一个内存区

域应使用函数 do_ummap()，它会销毁对应的内存区域。

由于进程所能直接操作的地址都是虚拟地址，所以当进程需要内存时，从内核获得的仅仅是虚拟的内存区域，而不是实际的物理地址。进程并没有获得物理内存(物理页面)，而只是对一个新的线性地址区间的使用权。实际的物理内存只有当进程实际访问新获取的虚拟地址时，才会由“请求页机制”产生“缺页”异常，从而进入分配实际页面的例程。这个过程可以借助 nopage()函数完成。当访问的进程虚拟内存并未真正分配页面时，便调用该函数来分配实际的物理页，并为该页建立页表项的功能。

这种“缺页”异常是虚拟内存机制赖以存在的基本保证——它会告诉内核去真正为进程分配物理页，并建立对应的页表，然后虚拟地址才真正地映射到了系统的物理内存中。当然，如果页被换出到外存，也会产生缺页异常，不用再建立页表了。这种请求页机制利用了内存访问的“局部性原理”，请求页带来的好处是减少了空闲内存，提高了系统的吞吐率。

5.3.4 物理内存管理(页管理)


视频讲解

Linux 内核管理物理内存是通过分页机制实现的，它将整个内存划分成无数个固定大小的页，从而分配和回收内存的基本单位便是内存页了。在此前提下，系统可以拼凑出所需要的任意内存供进程使用。但实际上系统使用内存时还是倾向于分配连续的内存块，因为分配连续内存时，页表不需要更改，因此能降低 TLB(页地址块表)的刷新率(频繁刷新会在很大程度上降低访问速度)。

鉴于上述需求，内核分配物理页面时为了尽量减少不连续情况，采用了“伙伴”(buddy)算法来管理空闲页面。Linux 系统采用伙伴算法管理系统页框的分配和回收，该算法对不同的管理区使用单独的伙伴系统管理。伙伴算法把内存中的所有页框按照大小分成 10 组不同大小的页块，每块分别包含 1、2、4、……、512 个页框。每种不同的页块都通过一个 free_area_struct 结构体来管理。系统将 10 个 free_area_struct 结构体组成一个 free_area[]数组。

其核心数据结构如下：

```
typedef struct free __ area __ struct
  {
struct list __ head free __ list.
unsigned long * map.
} free __ area __ t.
```

当向内核请求分配一定数目的页框时，若所请求的页框数目不是 2 的幂次方，则按稍微大于此数目的 2 的幂次方在页块链表中查找空闲页块，如果对应的页块链表中没有空闲页块，则在更大的页块链表中查找。当分配的页块中有多余的页框时，伙伴系统将根据多余的页框大小插入到对应的空闲页块链表中。向伙伴系统释放页框时，伙伴系统会将页框插入到对应的页框链表中，并且检查新插入的页框能否和原有的页块组合构成一个更大的页块，如果有两个块的大小相同且这两个块的物理地址连续，则合并成一个新页块并加入到对应的页块链表中，迭代此过程直到不能合并为止，这样可以极大限度地减少内存碎片。

ARM-Linux 内核中分配空闲页面的基本函数是 get_free_page/get_free_pages()，它们或是分配单页或是分配指定的页面(2、4、8、……、512 页)。值得注意的是，get_free_page()

是在内核中分配内存，不同于 malloc 函数在用户空间中分配方法。malloc()函数利用堆动态分配，实际上是调用 brk()系统调用，该调用的作用是扩大或缩小进程堆空间（它会修改进程的 brk 域）。如果现有的内存区域不够容纳堆空间，则会以页面大小的倍数为单位扩张或收缩对应的内存区域，但 brk 值并非以页面大小为倍数修改，而是按实际请求修改。因此，malloc()在用户空间分配内存时可以以字节为单位分配，但内核在内部仍然会以页为单位分配。

需要注意的是，物理页在系统中由页结构 struct_page 描述，系统中所有的页面都存储在数组 mem_map[]中，可以通过该数组找到系统中的每一页（空闲或非空闲）。而其中的空闲页面则可由上述提到的以伙伴关系组织的空闲页链表（free_area[MAX_ORDER]）来索引。图 5-10 显示了内核空间物理页分配技术。

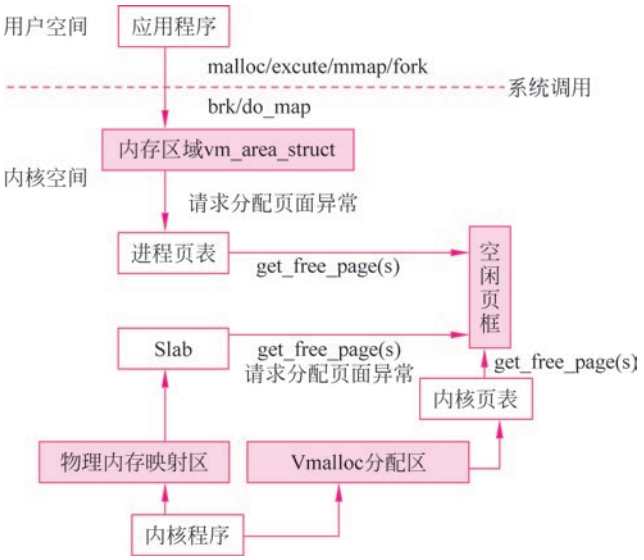

图 5-10　内核空间物理页分配技术

5.3.5　基于 slab 分配器的管理技术

伙伴算法采用页面作为分配内存的基本单位，虽然有利于解决外部碎片问题，但却只适合大块内存的请求，而且伙伴算法的充分条件要求较高并且容易产生内存浪费。由于内核自身最常使用的内存往往是很小（远远小于一页）的内存块——比如存放文件描述符、进程描述符、虚拟内存区域描述符等行为所需的内存都不足一页。这些用来存放描述符的内存相比页面是差距非常大的。一个整页中可以聚集多个小块内存。而且这些小块内存块一样频繁地被生成或者销毁。

为了满足内核对这种小内存块的需要，Linux 系统采用了一种被称为 slab 分配器（slab allocator）的技术。slab 并非脱离伙伴关系而独立存在的一种内存分配方式，slab 仍然是建立在页面基础之上的。slab 分配器主要的功能就是对频繁分配和释放的小对象提供高效的内存管理。它的核心思想是实现一个缓存池，分配对象的时候从缓存池中取，释放对象的时候再放入缓存池。slab 分配器是基于对象类型进行内存管理的，每种对象被划分为一类，例如，索引节点对象是一类，进程描述符又是一类等等。每当需要申请一个特定的对象时，就

从相应的类中分配一个空白的对象出去。当这个对象被使用完毕时，就重新“插入”到相应的类中(其实并不存在插入的动作，仅仅是将该对象重新标记为空闲而已)。下面是 slab 的结构体定义。

```
struct slab {
    union {
        struct {
            struct list_head list;
            unsigned long colouroff;
            void * s_mem;
            unsigned int inuse;
            kmem_bufctl_t free;
            unsigned short nodeid;
        };
        struct slab_rcu __ slab_cover_slab_rcu;
    };
};
```

与传统的内存管理模式相比，slab 缓存分配器有很多优点。首先，内核通常依赖于对小对象的分配，它们会在系统生命周期内进行无数次分配，slab 缓存分配器通过对类似大小的对象进行缓存，可以大大减少内存碎片。同时，slab 缓存分配器还支持通用对象的初始化，从而避免了为同一目标而对一个对象重复进行初始化。事实上，内核中常用的 kmalloc()函数(类似于用户态的 malloc())就使用了 slab 缓存分配器来进行可能的优化。

slab 缓存分配器不仅仅只用来存放内核专用的结构体，它还被用来处理内核对小块内存的请求。一般来说，内核程序中对小于一页的小块内存的请求才通过 slab 缓存分配器提供的接口 kmalloc()来完成(虽然它可分配 32～131072B 的内存)。从内核内存分配的角度来讲，kmalloc()可被看成是 get_free_page(s)的一个有效补充，内存分配粒度更灵活了。

关于 kmalloc()与 kfree()的具体实现，可参考内核源程序中的 include/Linux/slab.h 文件。如果希望分配大一点的内存空间，那么内核会利用一个更好的面向页的机制。分配页的相关函数有以下 3 个，这 3 个函数定义在 mm/page_alloc.c 文件中。

- get_zeroed_page(unsigned int gfp_mask)函数的作用是申请一个新的页，初始化该页的值为零，并返回页的指针。
- __get_free_page(unsigned int flags)函数与 get_zeroed_page 类似，但是它不初始化页的值为零。
- __get_free_pages(unsigned int flags，unsigned int order)函数类似__get_free_page，但是它可以申请多个页，并且返回的是第一个页的指针。

5.3.6 内核非连续内存分配

伙伴关系也好、slab 技术也好，从内存管理理论角度而言目的基本是一致的，它们都是为了防止“分片”，分片又分为外部分片和内部分片。所谓内部分片，是系统为了满足一小段内存区连续的需要，不得不分配了一大区域连续内存给它，从而造成了空间浪费。外部分片是指系统虽有足够的内存，但是分散的碎片，无法满足对大块“连续内存”的需求。无论哪种分片都是系统有效利用内存的障碍。由前面的介绍可知，slab 分配器使得一个页面内包含

的众多小块内存可独立被分配使用，避免了内部分片，减少了空闲内存。伙伴关系把内存块按大小分组管理，一定程度上减轻了外部分片的危害，但并未彻底消除。

所以避免外部分片的最终解决思路还是落到了如何利用不连续的内存块组合成“看起来很大的内存块”——这里的情况类似于用户空间分配虚拟内存，内存在逻辑上连续，其实会映射到并不一定连续的物理内存上。Linux 内核借用了这个技术，允许内核程序在内核地址空间中分配虚拟地址，同样也利用页表（内核页表）将虚拟地址映射到分散的内存页上。以此完美地解决了内核内存使用中的外部分片问题。内核提供 vmalloc()函数分配内核虚拟内存，该函数不同于 kmalloc()，它可以分配较 kmalloc()大得多的内存空间（可远大于128KB，但必须是页大小的倍数），但相比 kmalloc()来说，vmalloc()需要对内核虚拟地址进行重映射，必须更新内核页表，因此分配效率上相对较低。

与用户进程相似，内核也有一个名为 init_mm 的 mm_strcut 结构来描述内核地址空间，其中页表项 pdg＝swapper_pg_dir 包含了系统内核空间的映射关系。因此，vmalloc()分配内核虚拟地址必须更新内核页表，而 kmalloc()或 get_free_page()由于分配的连续内存，所以不需要更新内核页表。

vmalloc()分配的内核虚拟内存与 kmalloc()/get_free_page()分配的内核虚拟内存位于不同的区间，不会重叠。因为内核虚拟空间被分区管理，各司其职。进程用户空间地址分布从 0 到 3GB（其到 PAGE_OFFSET），从 3GB 到 vmalloc_start 这段地址是物理内存映射区域（该区域中包含了内核镜像、物理页面表 mem_map 等等）。

vmalloc()函数的相关原型包含在 include/Linux/vmalloc.h 头文件中。

主要函数说明如下：

（1）void * vmalloc(unsigned long size)——该函数的作用是申请 size 大小的虚拟内存空间，发生错误时返回 0，成功时返回一个指向大小为 size 的线性地址空间的指针。

（2）void vfree(void * addr)——该函数的作用是释放一块由 vmalloc()函数申请的内存，释放内存的基地址为 addr。

（3）void * vmap(struct page ** pages，unsigned int count，unsigned long flags，pgport_t prot)——该函数的作用是映射一个数组（其内容为页）到连续的虚拟空间中。第一个参数 pages 为指向页数组的指针。第二个参数 count 为要映射页的个数。第三个参数 flags 为传递 vm_area-> flags 值。第四个参数 prot 为映射时页保护。

（4）void vunmap(void * addr)——该函数的作用是释放由 vmap 映射的虚拟内存，释放从 addr 地址开始的连续虚拟区域。

5.3.7 页面回收简述

有页面分配，就会有页面回收。页面回收的方法大体上可分为两种：

一是主动释放。就像用户程序通过 free()函数释放曾经通过 malloc()函数分配的内存一样，页面的使用者明确知道页面的使用时机。前面介绍的伙伴算法和 slab 分配器机制，一般都是由内核程序主动释放的。对于直接从伙伴系统分配的页面，这是由使用者使用 free_pages()之类的函数主动释放的，页面释放后被直接放归伙伴系统。从 slab 中分配的对象（使用 kmem_cache_alloc()函数），也是由使用者主动释放的（使用 kmem_cache_free()函数）。

二是通过 Linux 内核提供的页框回收算法(PFRA)进行回收。页面的使用者一般将页面当作某种缓存,以提高系统的运行效率。缓存一直存在固然好,但是如果缓存没有了也不会造成什么错误,仅仅是效率受影响而已。页面的使用者不需要知道这些缓存页面什么时候最好被保留,什么时候最好被回收,这些都交由 PFRA 来负责。

简单来说,PFRA 要做的事就是回收可以被回收的页面。PFRA 的使用策略是主要在内核线程中周期性地被调用运行,或者当系统已经页面紧缺,试图分配页面的内核执行流程因得不到需要的页面而同步地调用 PFRA。内核非连续内存分配方式一般是由 PFRA 来进行回收,也可以通过类似删除文件、进程退出这样的过程来同步回收。

5.4 ARM-Linux 模块

自 Linux 1.2 版本之后 Linux 引进了模块这一重要特性,该特性提供内核可在运行时进行扩展的功能。可装载模块(Loadable Kernel Module,LKM)也被称为模块,即可在内核运行时加载到内核的一组目标代码(并非一个完整的可执行程序)。这样做的最明显好处就是在重构和使用可装载模块时并不需要重新编译内核。

LKM 最重要的功能包括内核模块在操作系统中的加载和卸载两部分。内核模块是一些在启动操作系统内核时如有需要可以载入内核执行的代码块,这些代码块在不需要时由操作系统卸载。模块扩展了操作系统的内核功能但不需要重新编译内核和启动系统。需要注意的是,如果只是认为可装载模块就是外部模块或者认为在模块与内核通信时模块是位于内核的外部的,那么这在 Linux 下均是错误的。当模块被装载到内核后,可装载模块已是内核的一部分。

5.4.1 LKM 的编写和编译


视频讲解

1. 内核模块的基本结构

一个内核模块至少包含两个函数,模块被加载时执行的初始化函数 init_module()和模块被卸载时执行的结束函数 cleanup_module()。在 Linux 2.6 中,两个函数可以起任意的名字,通过宏 module_init()和 module_exit()实现。唯一需要注意的地方是函数必须在宏的使用前定义。例如:

```
static int __init hello_init(void){}
static void __exit hello_exit(void ){}
module_init(hello_init);
module_exit(hello_exit);
```

这里声明函数为 static 的目的是使函数在文件以外不可见,宏__init 的作用是在完成初始化后收回该函数占用的内存,宏__exit 用于模块被编译进内核时忽略结束函数。这两个宏只针对模块被编译进内核的情况,而对动态加载模块是无效的。这是因为编译进内核的模块是没有清理结束工作的,而动态加载模块却需要自己完成这些工作。

2. 内核模块的编译

内核模块编译时需要提供一个 Makefile 文件来隐藏底层大量的复杂操作,使用户通过 make 命令就可以完成编译的任务。下面列举一个简单的编译 hello.c 的 Makefile 文件。

```
obj- m += hello.ko
KDIR := /lib/modules/$(Shell uname -r)/build
PWD := $(Shell pwd)
default:
$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
```

编译后获得可加载的模块文件 hello.ko。

5.4.2 LKM 版本差异比较

LKM 可装载模块虽然在设备驱动程序的编写和扩充内核功能中扮演着非常重要的角色，但它仍有许多不足的地方，其中最大的缺陷就是 LKM 对于内核版本的依赖性过强，每一个 LKM 都是靠内核提供的函数和数据结构组织起来的。当这些内核函数和数据结构因为内核版本变化而发生变动时，原先的 LKM 不经过修改就可能无法正常运行。如可装载模块在 Linux 2.6 与 Linux 2.4 之间就存在巨大差异，其最大区别就是模块装载过程变化：在 Linux 2.6 中可装载模块是在内核中完成连接的。其他一些变化大致包括：

- 模块的后缀及装载工具的变化。对于使用模块的授权用户而言，模块最直观的改变应是模块文件扩展名由原先的.o(即 object)变成了.ko(即 kernel object)。同时，在 Linux 2.6 中，模块使用了新的装卸载工具集 module-init-tools(工具 insmod 和 rmmod 被重新设计)。模块的构建过程改变巨大，在 Linux 2.6 中代码先被编译成.o 文件，再从.o 文件生成.ko 文件，构建过程会生成如.mod.c、.mod.o 等文件。
- 模块信息的附加过程的变化。在 Linux 2.6 中，模块的信息在构建时完成了附加过程，这与 Linux 2.4 不同，先前模块信息的附加是在模块装载到内核时进行的(在 Linux 2.4 时，这一过程由工具 insmod 完成)。
- 模块的标记选项的变化。在 Linux 2.6 中，针对管理模块的选项做了一些调整，如取消了 can_unload 标记(用于标记模块的使用状态)，添加了 CONFIG_MODULE_UNLOAD 标记(用于标记禁止模块卸载)等；还修改了一些接口函数，如模块的引用计数。

发展到 Linux 2.6 后，内核中越来越多的功能被模块化。这是由于可装载模块相对内核有着易维护、易调试的特点。由于模块一般是在真正需要时才被加载，因而 LKM 可装载模块还为内核节省了内存空间。根据模块的功能作用不同，可装载模块还可分三大类型：设备驱动模块、文件系统模块以及系统调用模块。值得注意的是，虽然可装载模块是从用户空间加载到内核空间的，但是并非用户空间的程序。

5.4.3 模块的加载与卸载

1. 模块的加载

模块的加载一般有两种方法：第一种是使用 insmod 命令加载，另一种是当内核发现需要加载某个模块时，请求内核后台进程 kmod 加载适当的模块。当内核需要加载模块时，kmod 被唤醒并执行 modprobe，同时传递需加载模块的名字作为参数。modprobe 像 insmod 一样将模块加载进内核，不同的是在模块被加载时应查看它是否涉及当前没有定义在内核中的任何符号。如果有，则在当前模块路径的其他模块中查找。如果找到，那么它们

也会被加载到内核中。但在这种情况下使用 insmod，会以“未解析符号”信息结束。

关于模块加载，可以用图 5-11 来简要说明。

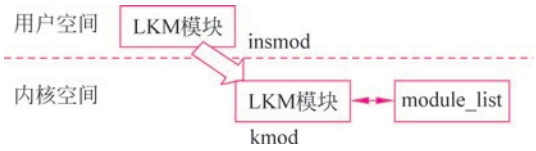

图 5-11 LKM 模块的加载

insmod 程序必须找到要求加载的内核模块，这些内核模块是已链接的目标文件。与其他文件不同的是，它们被链接成可重定位映像，这里的重定位映像首先强调的是映像没有被链接到特定地址上。insmod 将执行一个特权级系统调用来查找内核的输出符号，这些符号都以符号名和数值形式（如地址值）成对保存。内核输出符号表被保存在内核维护的模块链表的第一个 module 结构中。只有特殊符号才被添加，并且在内核编译与链接时确定。insmod 将模块读入虚拟内存并通过使用内核输出符号来修改其未解析的内核函数和资源的引用地址。这些工作通过由 insmod 程序直接将符号的地址写入模块中相应地址来进行。

当 insmod 修改完模块对内核输出符号的引用后，它将再次使用特权级系统调用申请足够的空间容纳新模块。内核将为其分配一个新的 module 结构以及足够的内核内存来保存新模块，并将其插入到内核模块链表的尾部，最后将新模块标记为 UNINITIALIZED。insmod 将模块复制到已分配空间中，如果为它分配的内核内存已用完，则再次申请，模块被多次加载必然处于不同的地址。

另外，此重定位工作包括使用适当地址来修改模块映像。如果新模块也希望将其符号输出到系统中，那么 insmod 将为其构造输出符号映像表。每个内核模块必须包含模块初始化和结束函数，所以为了避免冲突，它们的符号被设计成不输出，但是 insmod 必须知道这些地址，这样可以将它们传递给内核。在所有这些工作完成以后，insmod 将调用初始化代码并执行一个特权级系统调用将模块的初始化和结束函数地址传递给内核。

当将一个新模块加载到内核中时，内核必须更新其符号表并修改那些被新模块使用的老模块。那些依赖于其他模块的模块必须在其符号表尾部维护一个引用链表并在其 module 数据结构中指向它。

2. 模块的卸载

可以使用 rmmod 命令删除模块，这里有个特殊情况是，请求加载模块在其使用计数为 0 时会自动被系统删除。模块卸载可以用图 5-12 来描述。

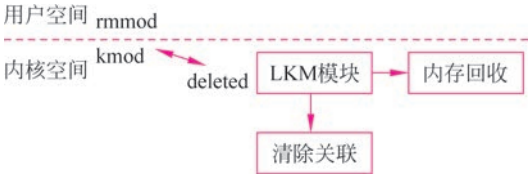

图 5-12 LKM 模块的卸载

内核中其他部分还在使用的模块不能被卸载。例如，若系统中安装了多个 VFAT 文件系统，则不能卸载 VFAT 模块。执行 lsmod 将看到每个模块的引用计数。模块的引用计数被保存在其映像的第一个常数字中，这个字还包含 autoclean 和 visited 标志。如果模块被

标记成 autoclean，则内核知道此模块可以自动卸载。visited 标志表示此模块正被一个或多个文件系统部分使用，只要有其他部分使用此模块则这个标志被置位。当系统要删除未被使用的请求加载模块时，内核就扫描所有模块，一般只查看那些被标记为 autoclean 并处于运行状态的模块。如果某模块的 visited 标记被清除则该模块将被删除，并且此模块占有的内核内存将被回收。其他依赖于该模块的模块将修改各自的引用域，表示它们之间的依赖关系不复存在。

5.4.4 工具集 module-init-tools

在 Linux 2.6 中，工具 insmod 被重新设计并作为工具集 module-init-tools 中的一个程序，其通过系统调用 sys_init_module（可查看头文件 include/asm-generic/unistd.h）衔接了模块的版本检查、模块的加载等功能。module-init-tools 是为 Linux 2.6 内核设计的运行在 Linux 用户空间的模块加卸载工具集，其包含的程序 rmmod 用于卸载当前内核中的模块。表 5-2 列举了工具集 module-init-tools 中的部分程序。

表 5-2 工具集 module-init-tools 中的部分程序

名称	说　明	使用方法示例
insmod	装载模块到当前运行的内核中	#insmod[/full/path/module_name] [parameters]
rmmod	从当前运行的内核中卸载模块	#rmmod [-fw] module_name -f：强制将该模块删除掉，不论是否正在被使用 -w：若该模块正在被使用，则等待该模块被使用完毕后再删除
lsmod	显示当前内核已加装的模块信息，可以和 grep 指令结合使用	#lsmod 或者 #lsmod|grep XXX
modinfo	检查与内核模块相关联的目标文件，并打印出所有得到的信息	#modinfo [-adln] [module_name|filename] -a：仅列出作者名 -d：仅列出该 modules 的说明 -l：仅列出授权 -n：仅列出该模块的详细路径
modprobe	利用 depmod 创建的依赖关系文件自动加载相关的模块	#modprobe [-lcfr] module_name -c：列出目前系统上面所有的模块 -l：列出目前在/lib/modules/`uname-r`/kernel 当中的所有模块完整文件名 -f：强制加载该模块 -r：删除某个模块
depmod	创建一个内核可装载模块的依赖关系文件，modprobe 用它来自动加载模块	#depmod [-Ane] -A：不加任何参数时，depmod 会主动去分析目前内核的模块，并且重新写入/lib/modules/$(uname-r)/modules.dep 中。如果加-A 参数，则会查找比 modules.dep 内还要新的模块；找到后才会更新 -n：不写入 modules.dep，而是将结果输出到屏幕上 -e：显示出目前已加载的不可执行的模块名称

值得注意的是，在 module-init-tools 中可用于模块加载和卸载的程序 modprobe。程序 modprobe 的内部函数调用过程与 insmod 类似，只是其装载过程会查找一些模块装载的配

置文件，且 modprobe 在装载模块时可解决模块间的依赖性问题，也就是说，如果有必要，程序 modprobe 会在装载一个模块时自动加载该模块依赖的其他模块。

5.5 ARM-Linux 中断管理


视频讲解

5.5.1 ARM-Linux 中断的一些基本概念

1. 设备、中断控制器和 CPU

在一个完整的设备中，与中断相关的硬件可以划分为 3 类，它们分别是设备、中断控制器和 CPU 本身，图 5-13 展示了一个 SMP 系统中的中断硬件的组成结构。

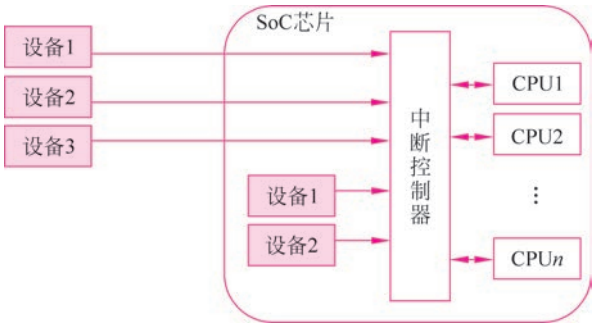

图 5-13 中断系统的硬件组成

- 设备：设备是发起中断的源，当设备需要请求某种服务的时候，它会发起一个硬件中断信号，通常，该信号会连接至中断控制器，由中断控制器做进一步的处理。在现代的移动设备中，发起中断的设备可以位于 SoC 芯片的外部，也可以位于 SoC 的内部。
- 中断控制器：中断控制器负责收集所有中断源发起的中断，现有的中断控制器几乎都是可编程的，通过对中断控制器的编程，用户可以控制每个中断源的优先级、中断的电气类型，还可以打开和关闭某一个中断源，在 SMP 系统中，甚至可以控制某个中断源发往哪一个 CPU 进行处理。对 ARM 架构的 SoC，使用较多的中断控制器是 VIC(Vector Interrupt Controller)，进入多核时代以后，GIC(General Interrupt Controller)的应用也开始逐渐变多。
- CPU：CPU 是最终响应中断的部件，它通过对可编程中断控制器的编程操作，控制和管理者系统中的每个中断。当中断控制器最终判定一个中断可以被处理时，它会根据事先的设定，通知其中一个或者是某几个 CPU 对该中断进行处理，虽然中断控制器可以同时通知数个 CPU 对某一个中断进行处理，实际上，最后只会有一个 CPU 响应这个中断请求，但具体是哪个 CPU 进行响应可能是随机的，中断控制器在硬件上对这一特性进行了保证，不过这也依赖于操作系统对中断系统的软件实现。在 SMP 系统中，CPU 之间也通过 IPI(Inter Processor Interrupt)进行通信。

2. IRQ 编号

系统中每一个注册的中断源，都会分配一个唯一的编号用于识别该中断，称之为 IRQ 编号。IRQ 编号贯穿在整个 Linux 的通用中断子系统中。在移动设备中，每个中断源的

IRQ 编号都会在 arch 相关的一些头文件中，例如，arch/xxx/mach-xxx/include/irqs.h。驱动程序在请求中断服务时，会使用 IRQ 编号注册该中断，中断发生时，CPU 通常会从中断控制器中获取相关信息，然后计算出相应的 IRQ 编号，然后把该 IRQ 编号传递到相应的驱动程序中。

5.5.2 内核异常向量表的初始化

ARM-Linux 内核启动时，首先运行的是 arch/arm/kernel/head.s，进行一些初始化工作，然后调用 main.c-> start_kernel()函数，进而调用 trap_init()（或者调用 early_trap_init()函数）以及 init_IRQ()函数进行中断初始化，建立异常向量表。

```
asmlinkage void __init start_kernel(void)
{
…
  trap_init();
…
  early_irq_init();
  init_IRQ();
…
}
```

接着系统会建立异常向量表。首先会将 ARM 处理器异常中断处理程序的入口安装到各自对应的中断向量地址中。在 ARM V4 及 V4T 以后的大部分处理器中，中断向量表的位置可以有两个位置：一个是 0x00000000，另一个是 0xffff0000。需要说明的是，Cortex-A8 处理器支持通过设置协处理 CP15 的 C12 寄存器将异常向量表的首地址设置在任意地址。可以通过 CP15 协处理器 c1 寄存器中 V 位（bit[13]）控制。V 位和中断向量表的对应关系如下：

V=0　～　0x00000000～0x0000001C

V=1　～　0xffff0000～0xffff001C

在 Linux 中，中断向量地址的复制由 trap_init()函数（或者调用 early_trap_init()函数）完成；对于 ARM 平台来说 trap_init()在 arch/arm/kernel/traps.c 中定义，为一个空函数。本节所使用内核版本使用了 early_trap_init()代替 trap_init()来初始化异常。代码如下所示。

```
void __init trap_init(void)
{
    return;
}
void __init early_trap_init(void)
{
    unsigned long vectors = CONFIG_VECTORS_BASE;

    extern char __stubs_start[], __stubs_end[];
    extern char __vectors_start[], __vectors_end[];
    extern char __kuser_helper_start[], __kuser_helper_end[];
    int kuser_sz = __kuser_helper_end - __kuser_helper_start;
```

```
/* __vectors_end至__vectors_start之间为异常向量表.__stubs_end至__stubs_start之间是异
常处理的位置.这些变量定义都在arch/arm/kernel/entry-armv.s中*/
    memcpy((void *)vectors,__vectors_start,__vectors_end - __vectors_start);
    memcpy((void *)vectors + 0x200,__stubs_start,__stubs_end - __stubs_start);
    memcpy((void *)vectors + 0x1000 - kuser_sz,__kuser_helper_start,kuser_sz);
    memcpy((void *)KERN_SIGRETURN_CODE,sigreturn_codes,
           sizeof(sigreturn_codes));
    memcpy((void *)KERN_RESTART_CODE,syscall_restart_code,
           sizeof(syscall_restart_code));
    flush_icache_range(vectors,vectors + PAGE_SIZE);
    modify_domain(DOMAIN_USER,DOMAIN_CLIENT);
}
```

early_trap_init()函数的主要功能是将中断处理程序的入口复制到中断向量地址。其中，

```
extern char __stubs_start[],__stubs_end[];
extern char __vectors_start[],__vectors_end[];
extern char __kuser_helper_start[],__kuser_helper_end[];
```

这3个变量是在汇编源文件中定义的，在源代码包里定义在entry-armv.s中。

```
__vectors_start:
    swi  SYS_ERROR0
    b  vector_und + stubs_offset
    ldr  pc,.LCvswi + stubs_offset
    b  vector_pabt + stubs_offset
    b  vector_dabt + stubs_offset
    b  vector_addrexcptn + stubs_offset
    b  vector_irq + stubs_offset
    b  vector_fiq + stubs_offset
    .globl __vectors_end
__vectors_end:
```

本节关注中断处理(vector_irq)。这里要说明的是，在采用了MMU内存管理单元后，异常向量表放在哪个具体物理地址已经不那么重要了，只需要将它映射到0xffff0000的虚拟地址即可。在中断前期处理函数中会根据IRQ产生时所处的模式来跳转到不同的中断处理流程中。

Init_IRQ(void)函数是一个特定于体系结构的函数，对于ARM体系结构来说该函数的定义如下：

```
void __init init_IRQ(void)
{
    int irq;
    for (irq = 0; irq < NR_IRQS; irq++)
        irq_desc[irq].status |= IRQ_NOREQUEST | IRQ_NOPROBE;

    init_arch_irq();
}
```

这个函数将irq_desc[NR_IRQS]结构数组各个元素的状态字段设置为IRQ_NOREQUEST|IRQ_NOPROBE，也就是未请求和未探测状态。然后调用特定机器平台的

中断初始化函数 init_arch_irq()。init_arch_irq()实际上是一个函数指针，其定义如下：

```
void ( * init_arch_irq)(void) __ initdata = NULL;
```

5.5.3 Linux 中断处理

从系统的角度来看，中断是一个流程。一般来说，它要经过如下几个环节：中断申请并响应、保存现场、中断处理及中断返回。

1. 中断申请并响应

ARM 处理器的中断是由处理器内部或者外部的中断源产生，通过 IRQ 或者 FIQ 中断请求线传递给处理器。在 ARM 模式下，中断可以配置成 IRQ 模式或者 FIQ 模式。但是在 Linux 系统中，所有的中断源都被配置成了 IRQ 中断模式。要想使设备的驱动程序能够产生中断，首先需要调用 request_irq()来分配中断线。在通过 request_irq()函数注册中断服务程序的时候，将会把设备中断处理程序添加进系统，使在中断发生的时候调用相应的中断处理程序。下面是 request_irq()函数的定义。

```
include/Linux/interrupt.h
static inline int __ must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
    const char * name, void * dev)
{
    return request_threaded_irq(irq, handler, NULL, flags, name, dev);
}
```

从上述代码中可以发现，request_irq()函数是 request_threaded_irq()函数的封装，内核用这个函数来完成分配中断线的工作，其主要参数说明如下：

- irq——要注册的硬件中断号。
- handler——向系统注册的中断处理函数，它是一个回调函数，在相应的中断线发生中断时，系统会调用这个函数。
- irqflags——中断类型标志，IRQF_ * 是中断处理的属性。
- devname——一个声明的设备的 ASCII 名字，是与中断号相关联的名称，在/proc/interrupts 文件中可以看到此名称。
- dev_id——I/O 设备的私有数据字段，典型情况下，它标志 I/O 设备本身(例如，它可能等于其主设备号和此设备号)，或者它指向设备驱动程序的数据，这个参数会被传回给 handler()函数。在中断共享时会用到，一般设置为这个设备的驱动程序中任何有效的地址值或者 NULL。
- thread_fn——由 irq handler 线程调用的函数，如果为 NULL，则不会创建线程。这个函数调用分配中断资源，并使能中断线和 IRQ 处理。当调用完成之后，则注册的中断处理函数随时可能被调用。由于中断处理函数必须清除开发板产生的一切中断，因此必须注意初始化的硬件和设置中断处理函数的正确顺序。

如果希望针对目标设备设置线程化的 IRQ 处理程序，则需要同时提供 handler 和 thread_fn。handler 仍然在硬中断上下文被调用，所以它需要检查中断是否是由它服务的设备产生的。如果是，则返回 IRQ_WAKE_THREAD，这将会唤醒中断处理程序线程并执

行 thread_fn。这种分开的中断处理程序设计是支持共享中断所必需的。dev_id 必须全局唯一。通常是设备数据结构的地址。如果要使用的共享中断,则必须传递一个非 NULL 的 dev_id,这是在释放中断的时候需要的。

request_threaded_irq()函数返回 0 表示成功,返回－EINVAL 表示中断号无效或处理函数指针为 NULL,返回－EBUSY 表示中断号已经被占用且不能共享。

2. 保存现场

处理中断时要保存现场,然后才能处理中断,处理完之后还要把现场状态恢复后才能返回到被中断的地方继续执行。需要说明的是,在指令跳转到中断向量的地方开始执行之前,由 CPU 自动完成了必要工作之后,每当中断控制器发出产生一个中断请求,CPU 总是到异常向量表的中断向量处取指令来执行。将中断向量中的宏解开,代码如下所示。

```
    .macro vector_stub,name,mode,correction = 0
    .align 5
vector_irq:
  sub   lr,lr,#4
    @ Save r0,lr_<exception> (parent PC) and spsr_<exception>
    @ (parent CPSR)
  stmia sp,{r0,lr}      @ save r0,lr
  mrs   lr,spsr
  str   lr,[sp,#8]      @ save spsr
    @ Prepare for SVC32 mode.   IRQs remain disabled.
  mrs   r0,cpsr
  eor   r0,r0,          #(IRQ_MODE ^ SVC_MODE | PSR_ISETSTATE)
  msr   spsr_cxsf,r0
  and   lr,lr,          #0x0f
  mov   r0,sp
  ldr   lr,[pc,lr,lsl #2]
movs  pc,lr             @ branch to handler in SVC mode
  .long __irq_usr       @  0  (USR_26 / USR_32)
  .long __irq_invalid   @  1  (FIQ_26 / FIQ_32)
  .long __irq_invalid   @  2  (IRQ_26 / IRQ_32)
  .long __irq_svc       @  3  (SVC_26 / SVC_32)
  .long __irq_invalid   @  4
  .long __irq_invalid   @  5
  .long __irq_invalid   @  6
  .long __irq_invalid   @  7
  .long __irq_invalid   @  8
  .long __irq_invalid   @  9
  .long __irq_invalid   @  a
  .long __irq_invalid   @  b
  .long __irq_invalid   @  c
  .long __irq_invalid   @  d
  .long __irq_invalid   @  e
  .long __irq_invalid   @  f
```

可以看到,该汇编代码主要是把被中断的代码在执行过程中的状态(CPSR)、返回地址(lr)等保存在中断模式下的栈中,然后进入到管理模式下去执行中断,同时令 r0＝sp,这样可以在管理模式下找到该地址,进而获取 SPSR 等信息。该汇编代码最终根据被中断的代码所处的模式跳转到相应的处理程序中。需要注意的是,管理模式下的栈和中断模式下的

栈不是同一个。

还可以看出这是一段很巧妙的位置无关的代码，它将中断产生时，CPSR 的模式位的值作为相对于 PC 值的索引来调用相应的中断处理程序。如果在进入中断时是用户模式，则调用__irq_usr 例程，如果为系统模式，则调用__irq_svc；如果是其他模式，说明出错了，则调用__irq_invalid。接下来分别简要说明这些中断处理程序。

3. 中断处理

ARM Linux 对中断的处理主要分为内核模式下的中断处理模式和用户模式下的中断处理模式。这里首先介绍内核模式下的中断处理。

内核模式下的中断处理，也就是调用__irq_svc 例程，__irq_svc 例程在文件 arch/arm/kernel/entry-armv.s 中定义，首先介绍这个例程的定义。

```
__irq_svc:
  svc_entry
#ifdef CONFIG_PREEMPT
  get_thread_info tsk
  ldr     r8,   [tsk, #TI_PREEMPT]       @ get preempt count
  add     r7,   r8,   #1                 @ increment it
  str     r7,   [tsk, #TI_PREEMPT]
#endif
   irq_handler
#ifdef CONFIG_PREEMPT
  str     r8,   [tsk, #TI_PREEMPT]       @ restore preempt count
  ldr     r0,   [tsk, #TI_FLAGS]         @ get flags
  teq     r8,   #0                       @ if preempt count != 0
  movne   r0,   #0                       @ force flags to 0
  tst     r0,   #_TIF_NEED_RESCHED
  blne  svc_preempt
#endif
  ldr     r4,   [sp, #S_PSR]             @ irqs are already disabled
#ifdef CONFIG_TRACE_IRQFLAGS
  tst     r4,   #PSR_I_BIT
  bleq  trace_hardirqs_on
#endif
  svc_exit r4                            @ return from exception
 UNWIND(.fnend)
ENDPROC(__irq_svc)
```

程序中用到了 irq_handler，它在文件 arch/arm/kernel/entry-armv.s 中定义。

```
  .macro irq_handler
  get_irqnr_preamble r5,lr
get_irqnr_and_base r0,r6,r5,lr
  movne r1,sp
  @
  @ routine called with r0 = irq number,r1 = struct pt_regs *
  @
#ifdef CONFIG_SMP
  test_for_ipi r0,r6,r5,lr
  movne r0,sp
  adrne lr,BSYM(1b)
```

```
    bne    do_IPI
#ifdef CONFIG_LOCAL_TIMERS
    test_for_ltirq r0,r6,r5,lr
    movne r0,sp
    adrne lr,BSYM(1b)
    bne    do_local_timer
#endif
#endif
    .endm
```

对于 ARM 平台来说，get_irqnr_preamble 是空的宏，irq_handler 首先通过宏 get_irqnr_and_base 获得中断号并存入 r0。然后将上面建立的 pt_regs 结构的指针，也就是 sp 值赋给 r1，将调用宏 get_irqnr_and_base 的位置作为返回地址。最后调用 asm_do_IRQ 进一步处理中断。get_irqnr_and_base 是平台相关的，这个宏查询 ISPR(IRQ 挂起中断服务寄存器，该寄存器与具体芯片类型有关，这里只是统称，当有需要处理的中断时，这个寄存器的相应位会置位，任意时刻，最多一个位会置位)，计算出的中断号放在 irqnr 指定的寄存器中。该宏结束后，r0＝中断号。这个宏在不同的 ARM 芯片上是不一样的，它需要读/写中断控制器中的寄存器。

在上述汇编语言代码中可以看到，系统在保存好中断现场，获得中断号之后，调用了函数 asm_do_IRQ()，从而进入中断处理的 C 程序部分。在 arch/arm/kernel/irq.c 中 asm_do_IRQ()函数定义如下：

```
asmlinkage void __exception asm_do_IRQ(unsigned int irq,struct pt_regs *regs)
{
     struct pt_regs *old_regs = set_irq_regs(regs);
     irq_enter();
     if (unlikely(irq >= NR_IRQS)) {
          if (printk_ratelimit())
               printk(KERN_WARNING "Bad IRQ %u\n",irq);
          ack_bad_irq(irq);
     } else {
          generic_handle_irq(irq);
     }
     irq_finish(irq);
     irq_exit();
     set_irq_regs(old_regs);
}
```

这个函数完成如下操作：

(1) 调用 set_irq_regs 函数更新处理器的当前帧指针，并在局部变量 old_regs 中保存老的帧指针。

(2) 调用 irq_enter()进入一个中断处理上下文。

(3) 检查中断号的有效性，有些硬件会随机地给一些错误的中断做一些检查以防止系统崩溃。如果不正确，则调用 ack_bad_irq(irq)，该函数会增加用来表征发生的错误中断数量的变量 irq_err_count。

(4) 若传递的中断号有效，则会调用 generic_handle_irq(irq)来处理中断。

(5) 调用 irq_exit()来推出中断处理上下文。

(6) 调用 set_irq_regs(old_regs)来恢复处理器的当前帧指针。

接下来介绍用户模式下的中断处理流程。中断发生时,CPU 处于用户模式下,则会调用__irq_usr 例程。

```
    .align 5
__irq_usr:
    usr_entry
   kuser_cmpxchg_check
    get_thread_info tsk
#ifdef CONFIG_PREEMPT
   ldr     r8,   [tsk,   #TI_PREEMPT]       @ get preempt count
   add     r7,   r8,     #1                 @ increment it
   str     r7,   [tsk,   #TI_PREEMPT]
#endif
    irq_handler
#ifdef CONFIG_PREEMPT
   ldr     r0,   [tsk,   #TI_PREEMPT]
   str     r8,   [tsk,   #TI_PREEMPT]
   teq     r0,   r7
 ARM( strne r0,[r0, -r0])
 THUMB(   movne r0, #0)
 THUMB(   strne r0,[r0] )
#endif
#ifdef CONFIG_TRACE_IRQFLAGS
   bl trace_hardirqs_on
#endif
    mov   why, #0
   b   ret_to_user
 UNWIND(.fnend)
ENDPROC(__irq_usr)
```

由该汇编代码可知,如果在用户模式下产生中断,则在返回的时候,会根据需要进行进程调度;如果中断发生在管理等内核模式下则不会进行进程调度。

4. 中断返回

前面已经分析过中断返回,此处不再赘述。这里只补充说明一点:如果是从用户态中断进入的则先检查是否需要调度,然后返回;如果是从系统态中断进入的则直接返回。

5.5.4 内核版本 2.6.38 后的中断处理系统的一些改变——通用中断子系统

在通用中断子系统(generic IRQ)出现之前,内核使用_do_IRQ 处理所有的中断,这意味着在_do_IRQ 中要处理各种类型的中断,这会导致软件的复杂性增加,层次不分明,而且代码的可重用性也不好。事实上,到了内核版本 2.6.38 以后,_do_IRQ 这种方式已经逐步在内核的代码中消失或者不再起决定性作用。通用中断子系统的原型最初出现于 ARM 体系中,一开始内核的开发者们把 3 种中断类型区分出来,它们分别是电平触发中断(level type)、边缘触发中断(edge type)和简易的中断(simple type)。

后来又针对某些需要回应 EOI(End Of Interrupt)的中断控制器加入了 fast eoi type,针对 SMP 系统加入了 per cpu type 等中断类型。把这些不同的中断类型抽象出来后,成为

了中断子系统的流控层。为了使所有的体系架构都可以重用这部分的代码，中断控制器也被进一步地封装起来，形成了中断子系统中的硬件封装层。图 5-14 表示通用中断子系统的层次结构。接下来简要介绍这些层次。

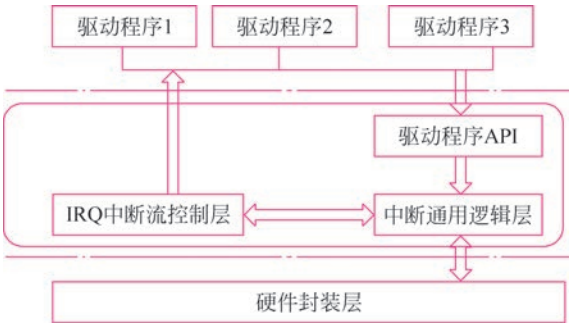

图 5-14 通用中断子系统的层次结构

1. 硬件封装层

它包含了体系架构相关的所有代码，包括中断控制器的抽象封装，体系结构相关的中断初始化，以及各个 IRQ 的相关数据结构的初始化工作，CPU 的中断入口也会在体系结构相关的代码中实现。中断通用逻辑层通过标准的封装接口（实际上就是 struct irq_chip 定义的接口）访问并控制中断控制器的行为，体系相关的中断入口函数在获取 IRQ 编号后，通过中断通用逻辑层提供的标准函数，将中断调用传递到中断流控制层中。

2. 中断流控制层

所谓中断流控制，是指合理并正确地处理连续发生的中断，比如一个中断在处理中，同一个中断再次到达时如何处理，何时应该屏蔽中断，何时打开中断，何时回应中断控制器等一系列的操作。该层实现了与体系和硬件无关的中断流控制处理操作，它针对不同的中断电气类型（如电平、边缘等），实现了对应的标准中断流控制处理函数。在这些处理函数中，最终会把中断控制权传递到驱动程序注册中断时传入的处理函数或者中断线程中。

3. 中断通用逻辑层

该层实现了对中断系统几个重要数据的管理，并提供了一系列的辅助管理函数。同时，该层还实现了中断线程的实现和管理，共享中断和嵌套中断的实现和管理，另外还提供了一些接口函数，它们将作为硬件封装层和中断流控制层以及驱动程序 API 层之间的桥梁，例如，以下 API：generic_handle_irq()、irq_to_desc()、irq_set_chip()、irq_set_chained_handler()。

4. 驱动程序 API

该部分向驱动程序提供了一系列的 API，用于向系统申请/释放中断，打开/关闭中断，设置中断类型和中断唤醒系统的特性等操作。驱动程序的开发者通常只会使用到这一层提供的这些 API 即可完成驱动程序的开发工作，其他的细节都由另外几个软件层较好地“隐藏”起来了，驱动程序开发者无须再关注底层的实现。

5.6 本章小结

本章主要介绍 ARM-Linux 内核的相关知识。内核是操作系统的灵魂，是我们了解和掌握 Linux 操作系统的最核心所在。ARM-Linux 内核是基于 ARM 处理器的 Linux 内核，

Linux 内核具有 5 个子系统，分别负责如下功能：进程管理、内存管理、虚拟文件系统、进程间通信和网络管理。本章主要从进程管理、内存管理、模块机制、中断管理这几个方面阐述 ARM-Linux 内核。限于篇幅，本章只是简要对内核的主要子模块进行了阐述，更多详细信息可参考 Linux 官网和阅读内核源代码。

习题

1. 什么是内核？内核的主要组成部分有哪些？

2. Linux 内核的五大主要组成模块之间存在什么关系？请简要描述。

3. 请在 Linux 官网上查阅当前内核主线版本和可支持版本情况，并比较最新版本与主线版本的差异。

4. Linux 内核 2.6 版本的主要特点有哪些？

5. 进程、线程和内核线程之间主要区别是什么？什么是轻量级进程？

6. Linux 内核的进程调度策略是什么？

7. 什么是 LKM？它的加载和卸载是如何进行的？

8. 可加载模块的最大优点是什么？

9. 在一个单 CPU 的计算机系统中，采用可剥夺式（也称抢占式）优先级的进程调度方案，且所有任务可以并行使用 I/O 设备。表 5-3 列出了 3 个任务 T1、T2、T3 的优先级和独立运行时占用 CPU 与 I/O 设备的时间。如果操作系统的开销忽略不计，这 3 个任务从同时启动到全部结束的总时间为多少毫秒，CPU 的空闲时间共有多少毫秒？

表 5-3 单 CPU 的任务优先级分配

任　务	优　先　级	每个任务独立运行时所需的时间
T1	最高	对每个任务： 占用 CPU 12ms，I/O 使用 8ms，再占用 CPU 5ms
T2	中等	
T3	最低	

10. 在某工程中，要求设置一绝对地址为 0x987a 的整型变量的值为 0x3434。编译器是一个纯粹的 ANSI 编译器。编写代码完成这一任务。

11. 下段代码是一段简单的 C 循环函数，在循环中含有数组指针调用。

```
CodeA
    void increment(int * restrict b,     int * restrict c)
    {       int i;
        for(i = 0; i < 100; i++)
        {
            c[i] = b[i] + 1;
        }
    }
```

请改写上述代码段，以实现如下功能：循环 100 次变成了循环 50 次（loop unrolling），减少了跳转次数；数组变成了指针，减少每次计算数组偏移量的指令；微调了不同代码操作的执行顺序，减少了流水线暂缓（stall）的情况；从＋＋循环变成了－－循环，这样可以使用 ARM 指令的条件位，为每次循环减少了一条判断指令。