第5章

Linux内核

Linux是一个一体化内核(Monolithic Kernel)系统。这里的“内核”指的是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被叫作Linux操作系统或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,它负责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 Linux内核概述

5.1.1 GNU/Linux的基本体系结构

GNU/Linux的基本体系结构如图5-1所示。从图5-1可以看到GNU/Linux被分成了两个空间。

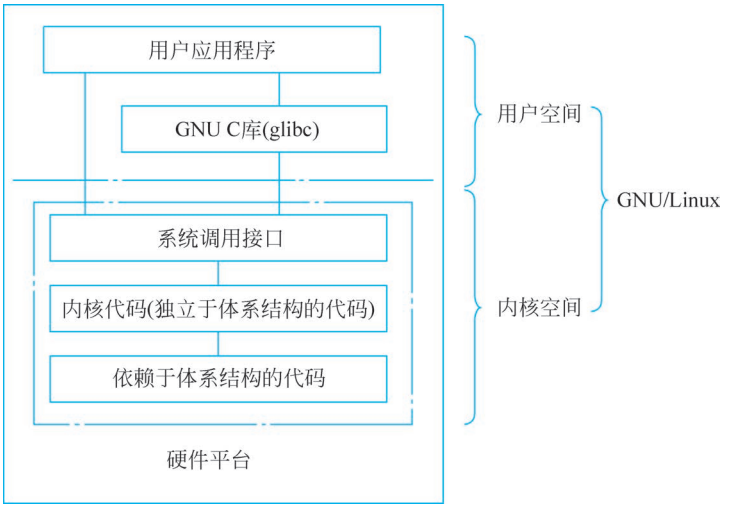

图5-1 GNU/Linux的基本体系结构

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)，负责执行用户应用程序。在该空间中，一般的应用程序是由 glibc 间接调用系统调用接口(System Call Interface，SCI)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 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，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，所有的资源访问都在内核的控制下执行，以免用户程序对系统资源进行越权访问，从而保障了系统的安全和稳定。从功能上来看，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代码。这些代码是 Linux 所支持的所有处理器体系结构所通用的。这些代码之下是依赖于体系结构的代码，构成了通常被称为板级支持包(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的平台，一般位于内核里的 arch 目录(./Linux/arch 目录)和 drivers 目录中。arch 目录含有诸如 x86、RISC-V、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 Linux 内核版本及特点

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，截至目前有两次变化：第一次是 1994 年的 1.0 版，第二次是 1996 年的 2.0 版。2011 年的 3.0 版发布，但这次在内核的概念上并没有发生大的改变。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 Bug、实现新的特性还是驱动时都会改变。

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 7 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的增加，发布周期是两三个月。考虑到对某个版本的 Bug 和安全漏洞的修复，有时会出现第四个数字。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，内核版本将升至 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——如在 3.0 发布的几个月之后发布 3.1，同时当需要修复 Bug 和安全漏洞时，增加一个数字(现在是第三个数)来表示，如 3.0.18。

如图 5-2 所示，在 Linux 内核官网上主要有 3 种类型的内核版本。

mainline:	6.4-rc6	2023-06-11	[tarball]		[patch]	[inc. patch] [view diff]
stable:	6.3.8	2023-06-14	[tarball]	[pgp]	[patch]	[inc. patch] [view diff]
stable:	6.2.16 [EOL]	2023-05-17	[tarball]	[pgp]	[patch]	[inc. patch] [view diff]
longterm:	6.1.34	2023-06-14	[tarball]	[pgp]	[patch]	[inc. patch] [view diff]
longterm:	5.15.117	2023-06-14	[tarball]	[pgp]	[patch]	[inc. patch] [view diff]
longterm:	5.10.184	2023-06-14	[tarball]	[pgp]	[patch]	[inc. patch] [view diff]
longterm:	5.4.247	2023-06-14	[tarball]	[pgp]	[patch]	[inc. patch] [view diff]
longterm:	4.19.286	2023-06-14	[tarball]	[pgp]	[patch]	[inc. patch] [view diff]
longterm:	4.14.318	2023-06-14	[tarball]	[pgp]	[patch]	[inc. patch] [view diff]
linux-next:	next-20230616	2023-06-16				

图 5-2　Linux 内核可支持版本一览

mainline 是主线版，目前主线版本为 6.4。

stable 是稳定版，由 mainline 在时机成熟时发布，稳定版会在相应版本号的主线上提供 Bug 修复和安全补丁。从 5.15 版本开始，内核开始全面支持基于 RISC-V 指令集架构的 VisionFive 2 单板计算机。

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，长期支持版的内核等到不再支持时，也会标记停止支持(End of Life，EoL)。

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。这样做的好处是系统各部分直接沟通、系统响应速度高、CPU 利用率好、实时性好。但是单内核的不足显而易见，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，但是大大减小了内核体积、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、支持内核线程以及动态装载内核模块等特点。

5.1.3　Linux 内核的主要架构及功能

Linux 内核的整体架构如图 5-3 所示。根据内核的核心功能，Linux 内核具有 5 个主要的子系统，分别负责如下的功能：进程管理、内存管理、虚拟文件系统(Virtual File System，VFS)、进程间通信和网络管理。

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 上实现多个进程的抽象。进程管理源码可参考./Linux/kernel 目录。

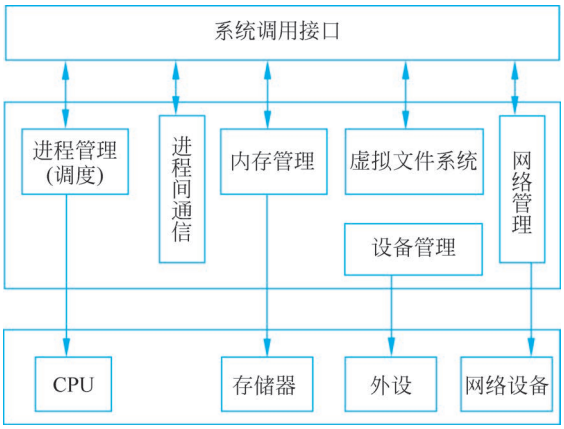

图 5-3 Linux 内核的整体架构

2. 内存管理

Linux 内核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 引入虚拟文件系统为用户提供统一、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，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这些设备。Linux 中的绝大部分对象可以被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 规范中标准的进程间通信(Inter Process Communication，IPC)机制和其他许多广泛使用的 IPC 机制实现进程间通信。IPC 不管理任何的硬件，它主要负责 Linux 系统中进程之间的通信，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)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被广泛应用。

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管理子系统的参与。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为其分配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

现网络文件系统的构建，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，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，以及系统的调试例程等。

5.1.4 Linux 内核源码目录结构

为了实现 Linux 内核的基本功能，Linux 内核源码的各个目录大致与此相对应，其组成如表 5-1 所示。

表 5-1 Linux 内核源码目录结构

目 录 名 称	目 录 说 明
arch	arch 目录包括所有与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，如 riscv 子目录就是 RISC-V CPU 及与之相兼容体系结构的子目录
block	block 目录包含一些 Linux 存储体系中关于块设备管理的代码。该目录用于实现块设备的基本框架和块设备的 I/O 调度算法
crypto	crypto 目录包含许多加密算法的源代码。例如，“sha1_generic.c”文件包含 SHA1 加密算法的代码
documentation	documentation 目录下是一些文档，是对目录作用的具体说明
drivers	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
fs	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
include	include 目录包括编译核心所需要的大部分头文件，如与平台无关的头文件在 include/Linux 子目录下
init	init 目录包含核心的初始化代码，要注意的是该代码不是系统的引导代码
ipc	ipc 目录包含核心进程间的通信代码
kernel	kernel 目录存放内核管理的核心代码。另外与处理器结构相关代码都放在 arch/*/kernel 目录下
lib	lib 目录包含核心的库代码，但是与处理器结构相关的库代码被放在 arch/*/lib/目录下
mm	mm 目录包含内存管理代码，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。需要值得注意的是，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
net	net 目录里是核心的网络部分代码，它包含网络协议代码，主要包括 IPv6、AppleTalk、以太网、Wi-Fi、蓝牙等代码，此外处理网桥和 DNS 解析的代码也在 net 目录中
samples	samples 目录存放一些内核编程范例

续表

目 录 名 称	目 录 说 明
scripts	scripts 目录包含用于配置核心的脚本文件
security	security 目录存放有关内核安全的代码
sound	sound 目录包含声卡驱动、存放声音系统架构的相关代码和具体声卡的设备驱动程序
tools	tools 目录包含了与内核交互的工具
usr	usr 目录包含早期用户空间代码
virt	virt 目录包含内核虚拟机代码

5.2 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包括打开的文件、挂起的信号、进程状态、地址空间等的总称。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，而且还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。由于在计算机中，CPU 资源是有限的，而众多的应用程序都要使用 CPU 资源，所以需要进程管理子系统对 CPU 进行调度管理。进程管理子系统包括 4 个子模块(如图 5-4 所示)，它们的功能如下所示。

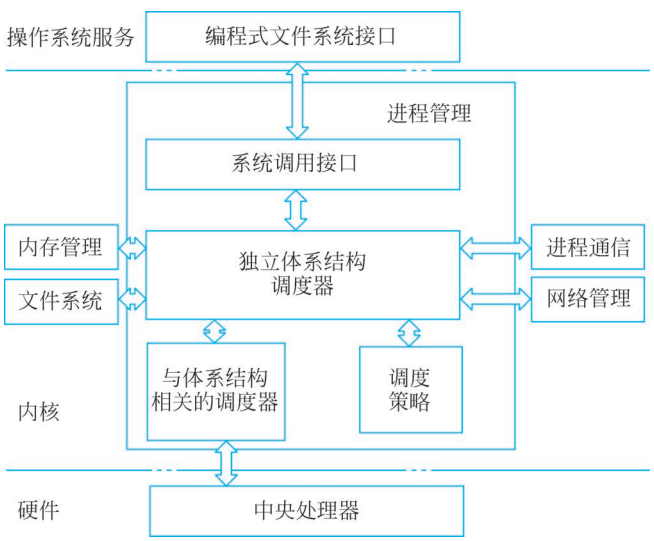

图 5-4 Linux 进程管理子系统的基本架构

(1) 调度策略模块。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模块。该模块涉及与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) 独立体系结构调度器模块。该模块涉及与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) 系统调用接口模块。进程管理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

对于 Linux 5.15 内核,系统最多可有 64 种进程同时存在。内核程序使用进程标识符(Process ID,PID)来标识每个进程。Linux 内核通过一个被称为进程描述符的 task_struct 结构体来管理进程,这个结构体记录了进程的最基本信息,它的所有域按其功能可以分为状态信息、链接信息、各种标识符、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许多描述进程属性的字段,而且还包含一系列指向其他数据结构的指针。内核把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当中,它定义在./include/Linux/sched.h 文件中。该结构体代码较长,这里只列出部分代码。

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
    struct thread_info          thread_info;
#endif
    unsigned int                __state;
#ifdef CONFIG_PREEMPT_RT
    unsigned int                saved_state;
#endif
    randomized_struct_fields_start
    void      *stack;
    refcount_t     usage;
    unsigned int                flags;
    unsigned int                ptrace;
#ifdef CONFIG_SMP
    int     on_cpu;
    struct __call_single_node     wake_entry;
#ifdef CONFIG_THREAD_INFO_IN_TASK
    unsigned int     cpu;      /* 当前 CPU */
#endif
    unsigned int                wakee_flips;
    unsigned long               wakee_flip_decay_ts;
    struct task_struct          *last_wakee;
    int     recent_used_cpu;
    int     wake_cpu;
#endif
    int     on_rq;
    int     prio;
```

```
    int     static_prio;
    int     normal_prio;
    unsigned int   rt_priority;
…
#define TASK_RUNNING              0x0000
#define TASK_INTERRUPTIBLE        0x0001
#define TASK_UNINTERRUPTIBLE      0x0002
#define EXIT_DEAD                 0x0010
#define EXIT_ZOMBIE               0x0020
#define EXIT_TRACE                (EXIT_ZOMBIE | EXIT_DEAD)
#define TASK_PARKED               0x0040
#define TASK_DEAD                 0x0080
#define TASK_WAKEKILL             0x0100
#define TASK_WAKING               0x0200
#define TASK_NOLOAD               0x0400
#define TASK_NEW                  0x0800
#define TASK_STATE_MAX            0x1000
#define TASK_KILLABLE             (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED              (TASK_WAKEKILL | __TASK_STOPPED)
#define TASK_TRACED               (TASK_WAKEKILL | __TASK_TRACED)
#define TASK_IDLE                 (TASK_UNINTERRUPTIBLE | TASK_NOLOAD)
…
```

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，要么正在准备执行。

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，除了不能通过接受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 表示当进程收到致命错误信号时唤醒进程。

TASK_WAKING 表示该任务正在唤醒，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 TASK_DEAD 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。

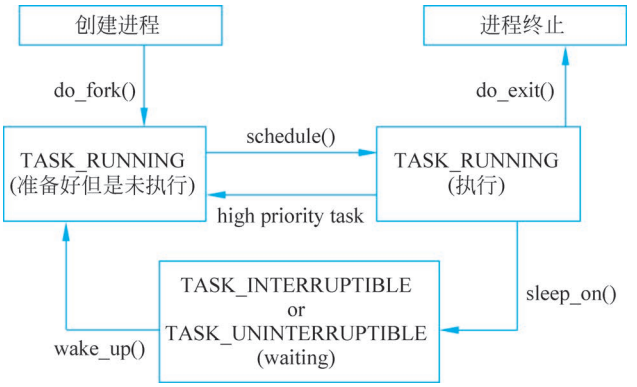

图 5-5 进程状态的切换过程

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。内核对进程、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。并且内核使用同一个函数 do_fork() 来分别创建这 3 种执行线程。执行线程通常是指任何正在执行的代码实例，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。

本小节首先介绍进程、线程和内核线程的概念，然后结合三者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，那么线程必须如同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而存在的，只不过它比较特殊，它和同类的进程共享某些资源，如进程地址空间、进程信号、打开的文件等。这类特殊的进程称之为轻量级进程。

按照这种线程机制的定义，每个用户态的线程都与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与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，称之为内核线程。由于在内核中，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，以 32 位 Linux 为例，内核线程只能使用大于 3 GB 的地址空间，而普通进程则可以使用整个 4 GB 的地址空间；还有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。

(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。

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，内核通过轻量级进程与其进行关联。轻量级进程之所以轻量，是因为它与其他进程共享上述所提及的进程资源，如进程 A 创建了线程 B，则线程 B 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，而且线程 B 的进程描述符中的某些代表资源指针会和进程 A 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 指针为 NULL。

5.2.4 kernel_clone 函数

进程、线程以及内核线程都有对应的创建函数，这三者所对应的创建函数最终在内核都是由 do_fork 函数进行创建的，在 5.10-rc1 版本里，kernel_clone 函数替换了原 do_fork 函数，kernel_clone 函数对于进程、线程以及内核线程的应用如图 5-6 所示。

从图 5-6 可以看出，内核中创建进程的核心函数为 kernel_clone，该函数的原型如下。

```
pid_t kernel_clone(struct kernel_clone_args *args)
{
    u64 clone_flags = args->flags;
    struct completion vfork;
```

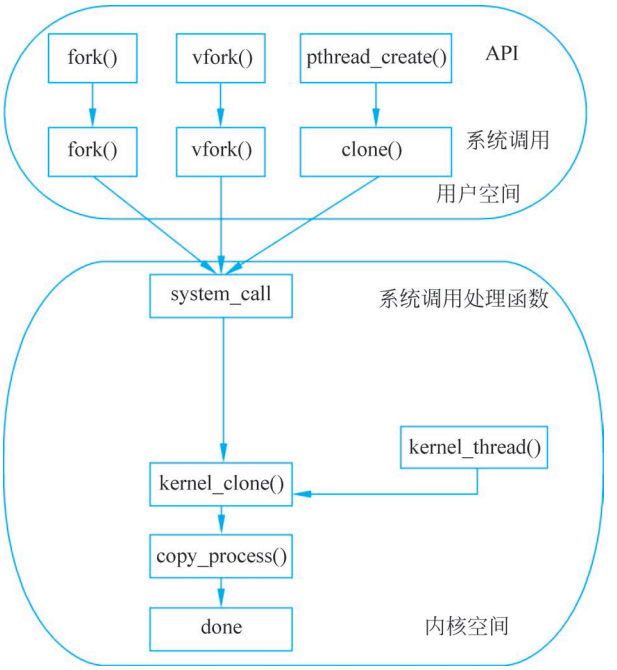

图 5-6　kernel_clone 函数对于进程、线程以及内核线程的应用

```
    struct pid *pid;
    struct task_struct *p;
    int trace = 0;
    pid_t nr;
    if ((args->flags & CLONE_PIDFD) &&
        (args->flags & CLONE_PARENT_SETTID) &&
        (args->pidfd == args->parent_tid))
        return -EINVAL;
    if (!(clone_flags & CLONE_UNTRACED)) {
        if (clone_flags & CLONE_VFORK)
            trace = PTRACE_EVENT_VFORK;
        else if (args->exit_signal != SIGCHLD)
            trace = PTRACE_EVENT_CLONE;
        else
            trace = PTRACE_EVENT_FORK;
        if (likely(!ptrace_event_enabled(current, trace)))
            trace = 0;
    }
//创建一个进程并返回 task_struct 指针
    p = copy_process(NULL, trace, NUMA_NO_NODE, args);
    add_latent_entropy();
    if (IS_ERR(p))
        return PTR_ERR(p);
        trace_sched_process_fork(current, p);
    //获取 pid
```

```
    pid = get_task_pid(p, PIDTYPE_PID);
    //获取虚拟的 pid
    pid_vnr(pid);
    if (clone_flags & CLONE_PARENT_SETTID)
        put_user(nr, args->parent_tid);

    if (clone_flags & CLONE_VFORK) {
        p->vfork_done = &vfork;
        init_completion(&vfork);
        get_task_struct(p);
    }
//将进程加入到就绪队列
    wake_up_new_task(p);
    if (unlikely(trace))
        ptrace_event_pid(trace, pid);
//等待子进程调用 exec()或 exit()
if (clone_flags & CLONE_VFORK) {
        if (!wait_for_vfork_done(p, &vfork))
            ptrace_event_pid(PTRACE_EVENT_VFORK_DONE, pid);
    }
    put_pid(pid);
    return nr;
}
```

该函数的参数 kernel_clone_args 的定义说明如下。

```
struct kernel_clone_args {
        /* 创建进程的标志位集合 */
        u64 flags;
        int __user *pidfd;
        /* 指向用户空间子进程 ID */
        int __user *child_tid;
        /* 指向用户空间父进程 ID */
        int __user *parent_tid;
        int exit_signal;
        /* 用户态栈起始地址 */
        unsigned long stack;
        /* 用户态栈大小,通常设置为 0 */
        unsigned long stack_size;
        /* 线程本地存储(Thread Local Storage) */
        unsigned long tls;
        pid_t *set_tid;
        size_t set_tid_size;
        int cgroup;
        struct cgroup *cgrp;
        struct css_set *cset;
};
```

5.2.5 进程的创建

在用户态程序中,可以通过 fork()、vfork()和 clone()这 3 个接口函数创建进程,这 3 个函数在库中分别对应同名的系统调用。系统调用函数进入内核后,会调用相应的系统调

用服务例程。

```
SYSCALL_DEFINE0(fork)
{
#ifdef CONFIG_MMU
        struct kernel_clone_args args = {
                .exit_signal = SIGCHLD,
        };
        return kernel_clone(&args);
#else
        return -EINVAL;
#endif
}
SYSCALL_DEFINE0(vfork)
{
        struct kernel_clone_args args = {
                .flags         = CLONE_VFORK | CLONE_VM,
                .exit_signal   = SIGCHLD,
        };
        return kernel_clone(&args);
}
SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
                 int __user *, parent_tidptr,
                 unsigned long, tls,
                 int __user *, child_tidptr)
{
        struct kernel_clone_args args = {
                .flags         = (lower_32_bits(clone_flags) & ~CSIGNAL),
                .pidfd         = parent_tidptr,
                .child_tid     = child_tidptr,
                .parent_tid    = parent_tidptr,
                .exit_signal   = (lower_32_bits(clone_flags) & CSIGNAL),
                .stack         = newsp,
                .tls           = tls,
        };
        return kernel_clone(&args);
}
```

通过上述系统调用服务例程的源码可以发现，3 个系统服务例程内部都调用了 kernel_clone 函数，主要差别在于参数所传的值不同这正好导致由这 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()。使用 fork()函数创建子进程时，子进程和父进程有各自独立的进程地址空间，fork 后会重新申请一份资源，包括进程描述符、进程上下文、进程堆栈、内存信息、打开的文件描述符、进程优先级、根目录、资源限制、控制终端等，复制给子进程。fork()函数会返回两次，一次在父进程，另一次在子进程。如果返回值为 0，说明是子进程；如果返回值为正数，说明是父进程。fork 系统调用只使用 SIGCHLD 标志位，子进程终止后发送 SIGCHLD 信号通知父进程。fork 是重量级调用，为子进程创建了一个基于父进程的完整

副本，然后子进程基于此运行，为了减少工作量采用写时复制技术。子进程只复制父进程的页表，不会复制页面内容，页表的权限为 RD-ONLY。当子进程需要写入新内容时会触发写时复制机制，为子进程创建一个副本，并将页表权限修改为 RW。由于需要修改页表，触发 page fault 等，因此 fork 需要 mmu 的支持。

(2) vfork()。使用 vfork()函数创建子进程时，子进程和父进程有相同的进程地址空间，vfork 会将父进程除 mm_struct 的资源复制给子进程，也就是创建子进程时，它的 task_struct-> mm 指向父进程，父进程与子进程共享一份同样的 mm_struct，vfork 会阻塞父进程，直到子进程退出或调用 exec 释放虚拟内存资源，父进程才会继续执行。vfork 的实现比 fork 多了两个标志位，分别是 CLONE_VFORK 和 CLONE_VM。CLONE_VFORK 表示父进程会被挂起，直至子进程释放虚拟内存资源。CLONE_VM 表示父进程与子进程运行在相同的内存空间中。由于没有写时复制，不需要页表管理，因此 vfork 不需要 MMU。

(3) clone()。使用 clone()创建用户线程时，clone 不会申请新的资源，所有线程指向相同的资源。创建进程和创建线程采用同样的 api 即 kernel_clone，带有标记 clone_filag 可以指明哪些是要复制。进程完全不共享父进程资源，线程完全共享父进程的资源，通过 clone_flags 标志复制父进程一部分资源，部分资源与父进程共享，部分资源与父进程不共享，而是位于进程和线程间的临界态。

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 kernel_clone 函数。

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，其本质是向 kernel_clone 函数提供特定的 flags 标志而创建的。

```
pid_t kernel_thread(int ( * fn)(void * ), void * arg, unsigned long flags)
{
    struct kernel_clone_args args = {
        .flags          = ((lower_32_bits(flags) | CLONE_VM |
                        CLONE_UNTRACED) & ~CSIGNAL),
        .exit_signal    = (lower_32_bits(flags) & CSIGNAL),
        .stack          = (unsigned long)fn,
        .stack_size     = (unsigned long)arg,
    };
    return kernel_clone(&args);
```

kernel_thread 用于创建一个内核线程，它只运行在内核地址空间，且所有内核线程共享相同的内核地址空间，没有独立的进程地址空间，即 task_struct-> mm 为 NULL。通过 kernel_thread 创建的内核线程处于不可运行态，需要 wake_up_process()来唤醒并加载到就绪队列，kthread_run()是 kthread_create 和 wake_up_process 的封装，可创建并唤醒进程。

5.2.7 进程的执行——exec 函数族

fork()函数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，原调用进程的内容除了进程号外，其他全部被新的进程替换。

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 5-2 列举了 exec 函数族成员函数的语法。

表 5-2 exec 函数族成员函数的语法

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函数只有 execve()，其他 5 个都是库函数，它们最终都会调用 execve()这个函数。这里简要介绍 execve()执行的流程。

(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 结构的其他项作初始化。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

当进程终止时，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。进程的终止最终都要通过 do_exit()来完成(Linux/kernel/exit.c 中)。进程终止后，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,如果没有别的进程使用,即没有被共享,就彻底释放它们。

(2) 如果进程排队等候 IPC 信号,则离开队列。

(3) 分别递减文件描述符、文件系统数据、进程名字空间的引用计数。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,并把进程状态设成 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。此时进程存在的唯一目的就是向它的父进程提供信息。父进程检索到信息后,或者通知内核该信息是无关信息后,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。内核对这三者一视同仁,进行统一的调度。进程调度的主要原则如下。

(1) 公平:保证每个进程得到合理的 CPU 时间。

(2) 高效:使 CPU 保持忙碌状态,即总有进程在 CPU 上运行。

(3) 响应时间:使交互用户的响应时间尽可能短。

(4) 周转时间:使批处理用户等待输出的时间尽可能短。

(5) 吞吐量:使单位时间内处理的进程数量尽可能多。

(6) 负载均衡:在多核多处理器系统中提供更高的性能。

在 Linux 中,从调度的角度来看,进程主要分为两种:实时进程和普通进程。

(1) 实时进程:对系统的响应时间要求很高,它们需要短的响应时间,并且这个时间的变化非常小,典型的实时进程有音乐播放器、视频播放器等。

(2) 普通进程:包括交互进程和非交互进程,交互进程如文本编辑器等,非交互进程的后台维护进程对 I/O 的响应时间没有很高的要求,如编译器。

实时进程和普通进程在 Linux 内核运行时是共存的,实时进程的优先级为 0～99,实时进程的优先级不会在运行期间改变,而普通进程的优先级为 100～139,普通进程的优先级

会在内核运行期间进行相应的改变。

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，内核里可以通过 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放弃加上时间限制，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，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夕、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可能被调度出去。比如一个进程正在内核态运行，此时一个中断发生使另一个高权值进程就绪，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，切换到高权值进程。为支持内核代码可抢占，在 2.6 版内核后，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，以避免不合理的延迟发生。而在检查过程中，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，只要新的进程不需要持有该锁。

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。内核默认提供以下 5 个调度器，Linux 内核使用 struct sched_class 来对调度器进行抽象。

(1) Stop 调度器(stop_sched_class)：优先级最高的调度类，可以抢占其他所有进程，不能被其他进程抢占。

(2) Deadline 调度器(dl_sched_class)：使用红黑树，将进程按照绝对截止期限进行排序，选择最小进程进行调度运行。

(3) RT 调度器(rt_sched_class)：实时调度器，为每个优先级维护一个队列。

(4) 完全公平调度器(Completely Fair Scheduler，CFS)(cfs_sched_class)：完全公平调度器，采用完全公平调度算法，引入虚拟运行时间概念。

(5) IDLE-Task 调度器(idle_sched_class)：空闲调度器，每个 CPU 都会有一个 idle 线程，当没有其他进程可以调度时，调度运行 idle 线程。

Linux 内核提供了一些调度策略供用户程序来选择调度器，其中 Stop 调度器和 IDLE-Task 调度器，仅由内核使用，用户无法进行选择。调度策略主要有以下几种。

(1) SCHED_DEADLINE：限期进程调度策略，使 task 选择 Deadline 调度器来调度运行。

(2) SCHED_RR：实时进程调度策略，时间片轮转，进程用完时间片后加入优先级对应运行队列的尾部，把 CPU 让给同优先级的其他进程。

(3) SCHED_FIFO：实时进程调度策略，先进先出调度没有时间片，没有更高优先级的情况下，只能等待主动让出 CPU。

(4) SCHED_NORMAL：普通进程调度策略，使 task 选择 CFS 调度器来调度运行。

(5) SCHED_BATCH：普通进程调度策略，批量处理，使 task 选择 CFS 调度器来调度运行。

(6) SCHED_IDLE：普通进程调度策略，使 task 以最低优先级选择 CFS 调度器来调度运行。

进程调度的一般原理如图 5-7 所示、

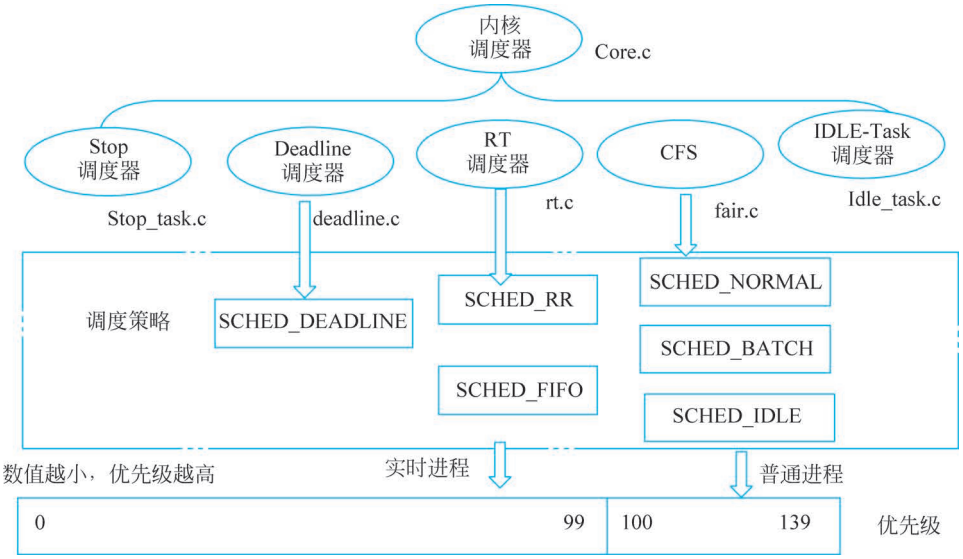

图 5-7 进程调度的一般原理

3. Linux CFS 调度

CFS 用于 Linux 系统中普通进程的调度。CFS 采用了红黑树算法来管理所有的调度实体 sched_entity，算法效率为 $O(\log(n))$。CFS 跟踪调度实体 sched_entity 的虚拟运行时间 vruntime，平等对待运行队列中的调度实体 sched_entity，将执行时间短的调度实体 sched_entity 排列到红黑树的左边。调度实体 sched_entity 通过 enqueue_entity()和 dequeue_entity()来进行红黑树的管理。

5.3 Linux 内存管理

5.3.1 Linux 内存管理概述

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，这种机制主要涵盖如下功能。

➢ 内存分配和回收。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，就要把逻辑地址转换成物理地址。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各程序执行中相互干扰，并保证对内存中信息访问的正确。

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，这种映射以进程为单位，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 5-8 所示。

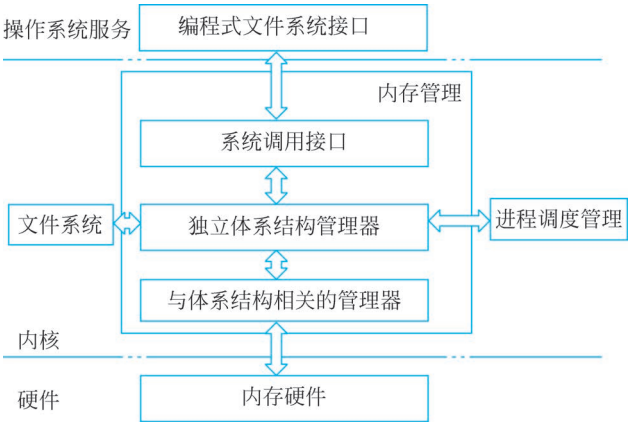

图 5-8　内存管理子系统结构

（1）与体系结构相关的管理器子模块，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，涉及体系结构无该部分，提供所有的内存管理机制，包括以进程为单位的 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，向用户空间应用程序提供内存的分配、释放，文件的映射等功能。

内存管理子系统使用节点（node）、区域（zone）、页（page）三级结构描述物理内存。在多核 CPU 中，节点是基于哪个 CPU 建立的，一般多少核的 CPU 就有多少个节点。内存节点结构体在 linux 内核 include/linux/mmzone.h 文件中定义，部分代码如下所示。

```
struct bootmem_data;
typedef struct pglist_data {
    struct zone node_zones[MAX_NR_ZONES];              //内存区域数组
    struct zonelist node_zonelists[MAX_ZONELISTS];     //备用区域数组
    int nr_zones;                                      //该节点包含的内存区域数量
#ifdef CONFIG_FLAT_NODE_MEM_MAP
    struct page *node_mem_map;                         //指向物理页描述符数组
#ifdef CONFIG_PAGE_EXTENSION
    struct page_ext *node_page_ext;                    //页的扩展属性
#endif
#endif
#ifndef CONFIG_NO_BOOTMEM
    struct bootmem_data *bdata;                        //早期内存管理器
...
} pg_data_t;
```

每一个节点分成多个区域，采用数组 node_zones 表示。这个数组的大小为 MAX_NR_ZONES。内存区域结构体在 include/linux/mmzone.h 文件中定义，部分代码如下。

```
struct zone {
    unsigned long watermark[NR_WMARK];
    unsigned long nr_reserved_highatomic;
    long lowmem_reserve[MAX_NR_ZONES];
#ifdef CONFIG_NUMA
    int node;
#endif
    struct pglist_data     *zone_pgdat;
    struct per_cpu_pageset __percpu *pageset;
#ifndef CONFIG_SPARSEMEM
    unsigned long          *pageblock_flags;
#endif
    unsigned long          zone_start_pfn;
    unsigned long          managed_pages;
    unsigned long          spanned_pages;
    unsigned long          present_pages;
    const char             *name;
...
}
```

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。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 Linux 虚拟存储空间及分布

32 位的处理器具有 4 GB 大小的虚拟地址容量，即每个进程的最大虚拟地址空间为 4 GB，如图 5-9 所示。Linux 内核处于高端的 1 GB 虚拟内存空间处，而低端的 3 GB 属于用户虚拟内存空间，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，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。需要值得注意的是，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。要说明的是现在的 64 位处理器没有使用 64 位虚拟地址。因为目前应用程序没有那么大的内存需求，所以 ARM64 和 x86_64 处理器不支持完全的 64 位虚拟地址，而是使用了 48 位，也就是对应了 256 TB 的地址空间。RISC-V Linux 支持 sv32、sv39、sv48 等虚拟地址格式，分别代表 32 位虚拟地址、39 位虚拟地址和 48 位虚拟地址。RISC-V Linux 默认使用 sv39 格式。

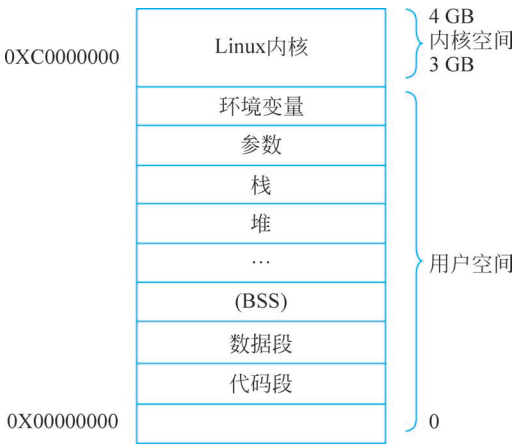

图 5-9 Linux 进程的虚拟内存空间及其组成（32 位平台）

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：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，所以只允许读取操作，而不允许写入（修改）操作。
- 数据段：用来存放可执行文件中已初始化的全局变量，即存放程序静态分配的变量和全局变量。
- BSS 段：包含程序中未初始化的全局变量，在内存中 BSS 段全部置零。
- 堆（heap）：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：用户存放程序临时创建的局部变量，也就是函数括弧“{}”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，所以栈特别方便用于保存/恢复调用现场。从这个意义上讲，堆栈被看成一个寄存、交换临时数据的内存区。

64 位 Linux 进程的虚拟内存空间及其组成和 32 位虚拟内存空间及其组成大致类似，

主要不同的地方有如下 3 点。

(1) 在用户态虚拟内存空间与内核态虚拟内存空间之间形成了一段地址是 0x0000 7FFF FFFF F000～0xFFFF 8000 0000 0000 的空洞,在这段范围内的虚拟内存地址是不合法的。

(2) 在代码段和数据段的中间有一段不可以读写的保护段,它的作用是防止程序在读写数据段时越界访问到代码段。

(3) 用户态虚拟内存空间与内核态虚拟内存空间均占用 128 TB,其中低 128 TB 分配给用户态虚拟内存空间,高 128 TB 分配给内核态虚拟内存空间。

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构来描述: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与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,代码较长,这里只列出部分。

```
struct mm_struct {
    struct {
        struct maple_tree mm_mt;
#ifdef CONFIG_MMU
        unsigned long (*get_unmapped_area) (struct file *filp,
                unsigned long addr, unsigned long len,
                unsigned long pgoff, unsigned long flags);
#endif
        unsigned long mmap_base;            /*内存映射区域的基址*/
        unsigned long mmap_legacy_base;     /*自底向上分配模式下的内存映射区域的基址 */
#ifdef CONFIG_HAVE_ARCH_COMPAT_MMAP_BASES /* 兼容内存映射的基址地址*/
        unsigned long mmap_compat_base;
        unsigned long mmap_compat_legacy_base;
#endif
        unsigned long task_size;            /* 任务的 vm 空间大小 */
        pgd_t * pgd;
#ifdef CONFIG_MEMBARRIER
        atomic_t membarrier_state;
#endif
        atomic_t mm_users;
        atomic_t mm_count;

#ifdef CONFIG_MMU
```

```
        atomic_long_t pgtables_bytes;           /* PTE 页表中的页 */
#endif
        int map_count;                          /* VMAs 的计数 */
        spinlock_t page_table_lock;             /* 保护页表和计数器 */
        struct rw_semaphore mmap_lock;
        struct list_head mmlist;
        unsigned long hiwater_rss;
        unsigned long hiwater_vm;
        unsigned long total_vm;                 /* 映射总页数 */
        unsigned long locked_vm;
        atomic64_t pinned_vm;
        unsigned long data_vm;
        unsigned long exec_vm;
        unsigned long stack_vm;                 /* VM 堆栈 */
        unsigned long def_flags;
        seqcount_t write_protect_seq;
        spinlock_t arg_lock;
        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 */
        ..... }
```

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,相应的操作都一致。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,其中包含本段的起始虚拟地址和结束虚拟地址,也包含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,vm_area_struct 结构是以链表形式链接的,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时;而红黑树用于在地址空间中定位特定内存区域时。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。

Linux 进程地址空间的管理模型如图 5-10 所示。

图 5-10 中的内存映射(memory map,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,以便可以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

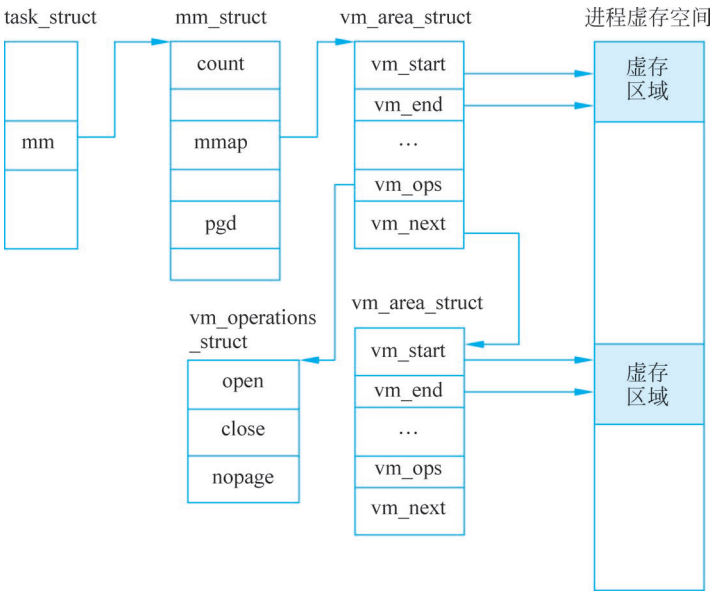

图 5-10 Linux 进程地址空间的管理模型

访问，也就是说，通过文件操作接口可以访问内存。vm_area_struct 结构体如下所示。

```
struct vm_area_struct {
    unsigned long vm_start;
    unsigned long vm_end;
    struct vm_area_struct * vm_next, * vm_prev;
    struct rb_node vm_rb;
    unsigned long rb_subtree_gap;
    struct mm_struct * vm_mm;
    pgprot_t vm_page_prot;
    unsigned long vm_flags;          /* mm.h 中的标志 */
    struct {
        struct rb_node rb;
        unsigned long rb_subtree_last;
    } shared;
```

2. Linux 的分页模型

由于分段机制与 Intel 处理器相关联，在其他的硬件系统上，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 中，Linux 采用了通用的四级页表结构，4 种页表分别为：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，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。

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，并

把这些块作为页(虚拟空间分页后的每个单位被称为页)或页帧(物理内存分页后的每个单位被称为页帧)进行管理。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 4 KB 单位划分页的,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 4 KB 的页单位下,32 位计算机的整个虚拟空间就被划分成 2^{20} 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 4 GB 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 1 GB 虚拟内存,后者为 3 GB 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。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,该函数实现当访问的进程虚拟内存并未真正分配页面时,该操作便被调用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上。当然,如果页被换出到外存,也会产生“缺页”异常,也就不用再建立页表。这种请求页机制利用内存访问的“局部性原理”,请求页带来的好处是节约了空闲内存、提高了系统的吞吐率。

5.3.4 物理内存管理(页管理)

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。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。但是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,因此能降低页地址快表(TLB)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,每个页块分别包含 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所示。

```
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对应的页框链表中，并且检查新插入的页框能否与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对应的页块链表中，并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的碎片。

Linux 内核中分配空闲页面的基本函数是 get_free_page/get_free_pages，它们是分配单页或分配指定的页面(2、4、8、…、512 页)。值得注意的是：get_free_page 在内核中分配内存，不同于 malloc 函数在用户空间中的分配方法。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可以以字节为单位分配，但内核在内部仍然是以页为单位分配的。

另外需要注意的是，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。内核空间物理页分配技术如图 5-11 所示。

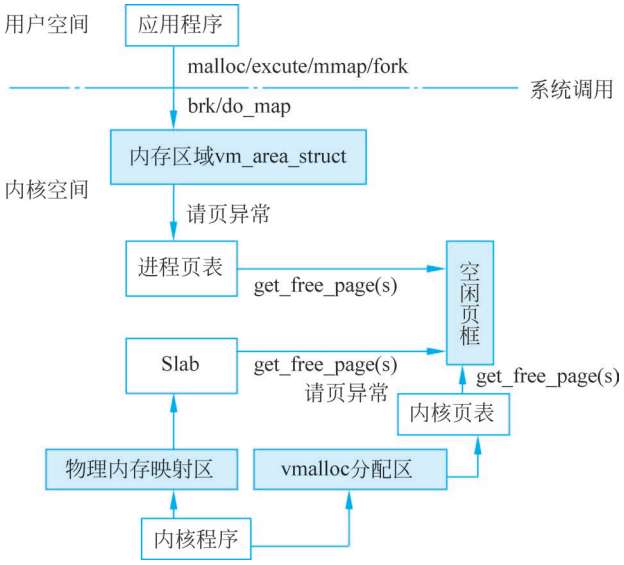

图 5-11 内核空间物理页分配技术

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，这样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较高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——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，差距是非常大的。一个整页中可以聚集多个这些小块内存，而且这些小块内存一样频繁地生成或者销毁。

为了满足内核对这种小块内存的需要，Linux 系统采用一种被称为 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。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，分配对象时从缓存池中取，释放对象时再放入缓存池。Slab 分配器是基于对象类型进行内存管理的，每一种对象被划分为一类，如索引节点对象是一类，进程描述符又是一类等。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”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)。

与传统的内存管理模式相比，Slab 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 分配器通过对类似大小的对象进行缓存，可以大大减少内部碎片。同时 Slab 分配器还支持通用对象的初始化，从而避免了为同一目的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 分配器来进行可能的优化。

Slab 分配器不仅只用来存放内核专用的结构体，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 分配器提供的接口 kmalloc 来完成(虽然它可分配 32 到 131 072 Byte 的内存)。从内核内存分配的角度来讲，kmalloc 可被看成 get_free_page(s)的一个有效补充，内存分配粒度更灵活。

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，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)函数类似于__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(vmalloc)

从内存管理理论角度而言，伙伴关系与 Slab 分配器的目的基本是一致的，它们都是为

了防止“分片”，分片又分为外部分片和内部分片。内部分片是系统为了满足一小段内存区连续的需要，不得不分配一大区域连续内存给它，从而造成空间浪费。外部分片是指系统虽有足够的内存，但却是分散的碎片，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。由前文可知，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，节约了空闲内存。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”——这里的情况很类似于用户空间分配虚拟内存，即内存逻辑上连续，其实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，它可以分配比 kmalloc 大得多的内存空间（可远大于 128 KB，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 3G（其到 PAGE_OFFSET），从 3G 到 vmalloc_start 这段地址是物理内存映射区域（该区域中包含内核镜像、物理页面表 mem_map 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个由 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，第三个参数为 flags 为传递的 vm_area-> flags 值，第四个参数 prot 为映射时的页保护。

（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以下两种。

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。前文所述的伙伴算法和 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 分配器中分配的对象（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（Page Frame Reclaiming Algorithm，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 Linux 模块

可加载内核模块（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 LKM 时并不需要重新编译内核。LKM 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。

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却不需要重新编译内核和启动系统。这里需要值得注意的是，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

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 版本中，两个函数可以起任意的名字，通过宏 module_init()和 module_exit()实现。唯一需要注意的是，函数必须在宏的使用前定义，如下所示。

```
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没有清理结束工作，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 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 的源码与 Makefile 文件。

hello.c 模块代码如下所示。

```
#include <linux/module.h>
#include <linux/kernel.h>
static int __init init_hello_module(void)          //__init 进行注明
{
    printk("**************** Start ***************\n");
    printk("Hello World! Start of hello world module!\n");
    return 0;
}
static void __exit exit_hello_module(void)         //__exit 进行注明
{
    printk("**************** End ***************\n");
    printk("Hello World! End of hello world module!\n");
}
MODULE_LICENSE("GPL");                             //模块许可证声明
module_init(init_hello_module);                    //module_init()宏,用于初始化
module_exit(exit_hello_module);                    //module_exit()宏,用于析构
```

Makefile 文件如下所示。

```
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 的内核表示

每一个内核模块在内核中都对应一个数据结构 module，所有的模块通过一个链表维护。下列代码来自 module.h 文件。部分成员列举如下。

```
struct module
{    enum module_state state;                      //状态
     struct list_head list;                        //所有的模块构成双链表
```

```
    char name[MODULE_NAME_LEN];                    //模块名字
    struct module_kobject mkobj;
    struct module_attribute * modinfo_attrs;
    const char * version;
    const char * srcversion;
    struct kobject * holders_dir;
const struct kernel_symbol * syms;                 //导出符号信息
const unsigned long * crcs;
    unsigned int num_syms;
    struct kernel_param * kp;                      //内核参数
    unsigned int num_kp;
unsigned int num_gpl_syms;/
const struct kernel_symbol * gpl_syms;
    const unsigned long * gpl_crcs;
#ifdef CONFIG_UNUSED_SYMBOLS
    const struct kernel_symbol * unused_syms;
    const unsigned long * unused_crcs;
    unsigned int num_unused_syms;
    unsigned int num_unused_gpl_syms;
    const struct kernel_symbol * unused_gpl_syms;
    const unsigned long * unused_gpl_crcs;
#endif
#ifdef CONFIG_MODULE_SIG
       bool sig_ok;
#endif
    unsigned int num_exentries;
    struct exception_table_entry * extable;        //异常表
    int ( * init)(void);                           //模块初始化函数指针
    void * module_init;
    void * module_core;                            //核心数据和代码部分,在卸载时会调用
    …
    struct task_struct * waiter;                   //等待队列,记录被卸载的进程
    void ( * exit)(void);                          //卸载退出函数,模块中定义的 exit 函数
     …
};
```

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：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，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，在当前模块路径的其他模块中查找。如果找到，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 5-12 来简要说明。

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

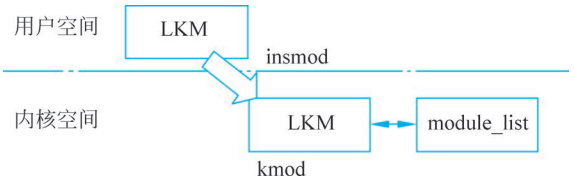

图 5-12 LKM 的加载

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。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。这些工作采取由 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，并将其插入内核模块链表的尾部，最后将新模块标志为 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，将再次申请，但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。如果新模块希望将其符号输出到系统中，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 命令卸载模块，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 5-13 来描述。

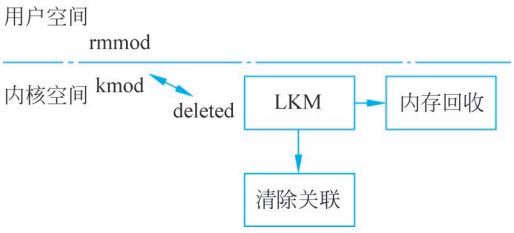

图 5-13 LKM 的卸载

内核中其他部分还在使用的模块不能被卸载。例如，系统中安装了多个虚拟文件分配

表(Virtual File Allocation Table,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志为 autoclean 并处于 running 状态的模块。如果某模块的 visited 标记被清除则该模块就将被删除,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,表示它们间的依赖关系不复存在。

5.4.4 模块主要命令

表 5-3 列举了模块相关的主要命令。

表 5-3 模块相关的主要命令

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	显示当前内核已加载的模块信息,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,如果真找到,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

这里值得注意的是,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，也就是说如果有必要，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 Linux 中断管理

5.5.1 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 5-14 展示了一个对称多处理（Symmetrical Multi-Processing，SMP）系统的硬件组成。

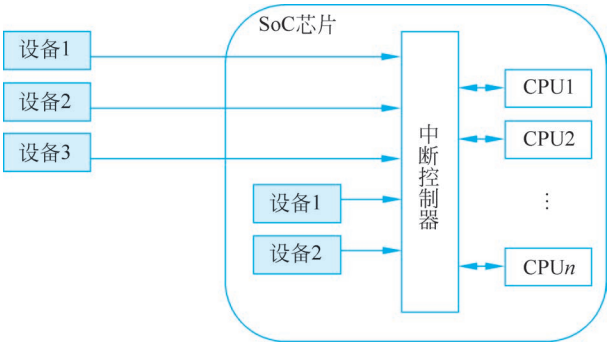

图 5-14 SMP 系统的硬件组成

- 设备：设备是发起中断的源，当设备需要请求某种服务时，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、中断的电器类型，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，称之为中断请求(Interrupt Request，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，如 arch/xxx/mach-xxx/include/irqs.h。驱动程序在请求中断服务时，它会使用 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 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_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，然后整合这些中断类型构建成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 5-15 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。

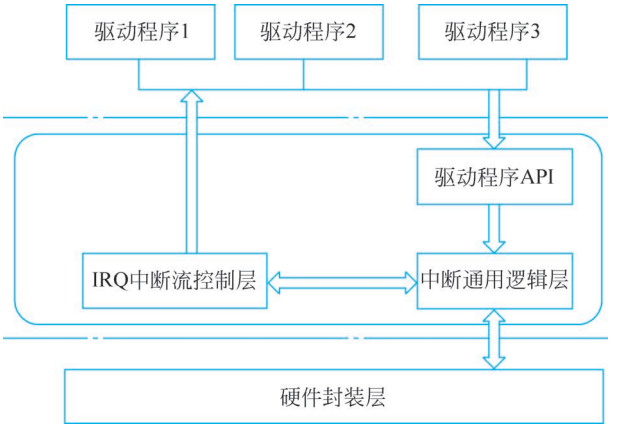

图 5-15 通用中断子系统的层次结构

硬件封装层：包含体系架构相关的所有代码，包括中断控制器的抽象封装、arch 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 的中断入口会在 arch 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，把中断调用传递到中断流控层中。

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，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（电平、边缘等），实现了对应的标准中断流控处理函数，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。

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层以及驱动程序 API 层之间的桥梁，如以下 API：generic_handle_irq()、irq_to_desc()、irq_set_chip()、irq_set_chained_handler()。

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，驱动程序开发者无须再关注底层的实现。

5.5.3 主要数据结构

Linux 中断主要涉及的数据结构包括 irq_desc、irq_chip 和 irqaction。首先分析 irq_desc 的部分代码。irq_desc 结构体用来描述中断源，其中的 action 成员是一个指向由 irqaction 结构体组成的一个单向链表的头的指针。若一个 IRQ 只被一个中断源使用，那么该链表的长度就是 1，当有多个设备共享一个中断源时，该链表就会由多个 irqaction 结构体组成。dpth 成员描述 irq_desc_t 的当前用户的个数，主要用来保证事件正在处理的过程中 IRQ 不会被禁止。

```
struct irq_desc {
    irq_flow_handler_t      handle_irq;          /* 指向中断函数 */
    struct irqaction        *action;             /* action 链表，用于中断处理函数 */
    unsigned int            status_use_accessors;
    unsigned int            core_internal_state__do_not_mess_with_it;
    unsigned int            depth;
    unsigned int            wake_depth;
    unsigned int            tot_count;
    unsigned int            irq_count;           /* IRQs 侦测 */
    unsigned long           last_unhandled;
    unsigned int            irqs_unhandled;
    atomic_t                threads_handled;
    int                     threads_handled_last;
    raw_spinlock_t          lock;
    struct cpumask          *percpu_enabled;
    const struct cpumask    *percpu_affinity;
……
};
```

irq_chip 结构体，用于访问底层硬件，下面是部分代码。

```
struct irq_chip {
    struct device     * parent_device;
    const char        * name;
    unsigned int    ( * irq_startup)(struct irq_data * data);      //启动中断
    void            ( * irq_shutdown)(struct irq_data * data);    //关闭中断
    void            ( * irq_enable)(struct irq_data * data);      //使能中断
    void            ( * irq_disable)(struct irq_data * data);     //禁止中断
    void            ( * irq_ack)(struct irq_data * data);         //响应中断
    void            ( * irq_mask)(struct irq_data * data);        //屏蔽中断源
    void            ( * irq_mask_ack)(struct irq_data * data);    //屏蔽中断源并响应中断
    void            ( * irq_unmask)(struct irq_data * data);      //开启中断源
    void            ( * irq_eoi)(struct irq_data * data);
……
    } ;
```

irqaction 结构定义如下所示。

```
struct irqaction {
        irq_handler_t handler;              //相当于用户注册的中断处理函数
        unsigned long flags;                //中断标志
        cpumask_t mask;                     //中断掩码
        const char * name;                  //中断名称，产生中断的硬件的名字
        void * dev_id;                      //设备 id
        struct irqaction * next;            //指向下一个成员
        int irq;                            //中断号，
        struct proc_dir_entry * dir;        //指向 IRQn 相关的/proc/irq/
};
```

Linux 中断的处理，总体来说可以分为两部分。

（1）围绕 irq_desc 中断描述符建立连接关系，这个过程包括：中断源信息的解析（通过设备树）、硬件中断号到 Linux 中断号的映射、irq_desc 结构的分配及初始化（内部各个结构的组织关系）、中断的注册（填充 irq_desc 结构，包括 handler 处理函数）等，总而言之，就是完成静态关系创建，为中断处理做好准备。

（2）当外设触发中断信号时，中断控制器接收到信号并发送到处理器，此时处理器进行异常模式切换，并逐步从处理器架构相关代码逐级回调。如果涉及中断线程化，则还需要进行中断内核线程的唤醒操作，最终完成中断处理函数的执行。

5.6 本章小结

本章主要介绍 Linux 内核的相关知识。内核是操作系统的灵魂，是了解和掌握 Linux 操作系统的最核心所在。Linux 内核具有 5 个子系统，分别负责如下的功能：进程管理、内存管理、虚拟文件系统、进程间通信和网络接口。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 Linux 内核。由于篇幅限制，本章只简要对内核的主要子模块进行了阐述，更多更详细的信息可参考 Linux 官网和阅读内核源码。