第四章　进程描述

操作系统中最核心的概念是进程，本章将对进程进行全面的描述。首先从程序的角度引入进程，接着对 Linux 中的进程进行了概要描述，在此基础上，对 Linux 中核心数据结构 task_struct 进行了较全面的介绍。另外，详细描述了内核对进程的 4 种组织方式，最后介绍了系统中一种特殊的进程——内核线程。

4.1　进程和程序（Process and Program）

首先我们对进程作一明确定义：所谓进程是由正文段（Text）、用户数据段（User Segment）以及系统数据段（System Segment）共同组成的一个执行环境。

程序只是一个普通文件，是一个机器代码指令和数据的集合，这些指令和数据存储在磁盘上的一个可执行映像（Executable Image）中，所以，程序是一个静态的实体。这里，对可执行映像做进一步解释，可执行映像就是一个可执行文件的内容，例如，你编写了一个 C 源程序，最终这个源程序要经过编译、连接成为一个可执行文件后才能运行。源程序中你要定义许多变量，在可执行文件中，这些变量就组成了数据段的一部分；源程序中的许多语句，例如“ i++; for(i=0; i<10; i++); ”等，在可执行文件中，它们对应着许多不同的机器代码指令，这些机器代码指令经 CPU 执行，就完成了你所期望的工作。可以这么说：程序代表你期望完成某工作的计划和步骤，它还浮在纸面上，等待具体实现。而具体的实现过程就是由进程来完成的，进程可以认为是运行中的程序，它除了包含程序中的所有内容外，还包含一些额外的数据。程序和进程的组成如图 4.1 所示。

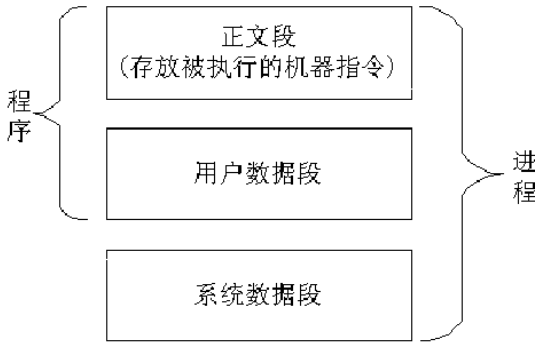

图 4.1 程序及进程的组成

程序装入内存后就可以运行了：在指令指针寄存器的控制下，不断地将指令取至 CPU 运

行。这些指令控制的对象不外乎各种存储器（内存、外存和各种 CPU 寄存器等），这些存储器中保存有待运行的指令和待处理的数据，当然，指令只有到 CPU 才能发挥其作用。可见，在计算机内部，程序的执行过程实际上就是一个执行环境的总和，这个执行环境包括程序中各种指令和数据，还有一些额外数据，比如说寄存器的值、用来保存临时数据（例如传递给某个函数的参数、函数的返回地址、保存变量等）的堆栈（包括程序堆栈和系统堆栈）、被打开文件的数量及输入输出设备的状态等。这个执行环境的动态变化表征程序的运行。我们就把这个环境称作“进程”，它代表程序的执行过程，是一个动态的实体，它随着程序中指令的执行而不断地变化。在某个特定时刻的进程的内容被称为进程映像（Process Image）。

Linux 是一个多任务操作系统，也就是说，可以有多个程序同时装入内存并运行，操作系统为每个程序建立一个运行环境即创建进程，每个进程拥有自己的虚拟地址空间，它们之间互不干扰，即使要相互作用（例如多个进程合作完成某个工作），也要通过内核提供的进程间通信机制（IPC）。Linux 内核支持多个进程虚拟地并发执行，这是通过不断地保存和切换程序的运行环境而实现的，选择哪个进程运行是由调度程序决定的。注意，在一些 UNIX 书籍中，又把“进程切换”（Process Switching）称为“环境切换”或“上下文切换”（Context Switching），这些术语表达的意思是相同的。

进程运行过程中，还需要其他的一些系统资源，例如，要用 CPU 来运行它的指令、要用系统的物理内存来容纳进程本身和它的有关数据、要在文件系统中打开和使用文件、并且可能直接或间接的使用系统的物理设备，例如打印机、扫描仪等。由于这些系统资源是由所有进程共享的，所以 Linux 必须监视进程和它所拥有的系统资源，使它们们可以公平地拥有系统资源以得到运行。

系统中最宝贵的资源是 CPU，通常来说，系统中只有一个 CPU，当然也可以有多个 CPU（Linux 支持 SMP___对称多处理机）。Linux 作为多任务操作系统，其目的就是让 CPU 上一直都有进程在运行，以最大限度地利用这一宝贵资源。通常情况下，进程数目是多于 CPU 数目的，这样其他进程必须等待 CPU 这一资源。操作系统通过调度程序来选择下一个最应该运行的进程，并使用一系列的调度策略来确保公平和高效。

进程是一个动态实体，如图 4.1 所示，进程实体由 3 个独立的部分组成。

（1）正文段（Text）：存放被执行的机器指令。这个段是只读的（所以，在这里不能写自己能修改的代码），它允许系统中正在运行的两个或多个进程之间能够共享这一代码。例如，有几个用户都在使用文本编辑器，在内存中仅需要该程序指令的一个副本，他们全都共享这一副本。

（2）用户数据段（User Segment）：存放进程在执行时直接进行操作的所有数据，包括进程使用的全部变量在内。显然，这里包含的信息可以被改变。虽然进程之间可以共享正文段，但是每个进程需要有它自己的专用用户数据段。例如同时编辑文本的用户，虽然运行着同样的程序编辑器，但是每个用户都有不同的数据：正在编辑的文本。

（3）系统数据段（System Segment）：该段有效地存放程序运行的环境。事实上，这正是程序和进程的区别所在。如前所述，程序是由一组指令和数据组成的静态事物，它们是进程最初使用的正文段和用户数据段。作为动态事物，进程是正文段、用户数据段和系统数据段的信息的交叉综合体，其中系统数据段是进程实体最重要的一部分，之所以说它有效地存放程序运行的环境，是因为这一部分存放有进程的控制信息。系统中有许多进程，操作系统

要管理它们、调度它们运行，就是通过这些控制信息。Linux 为每个进程建立了 task_struct 数据结构来容纳这些控制信息。

总之，进程是一个程序完整的执行环境。该环境是由正文段、用户数据段、系统数据段的信息交织在一起组成的。

4.2　Linux 中的进程概述

Linux 中的每个进程由一个 task_struct 数据结构来描述，在 Linux 中，任务（Task）和进程（Process）是两个相同的术语，task_struct 其实就是通常所说的“进程控制块”即 PCB。task_struct 容纳了一个进程的所有信息，是系统对进程进行控制的唯一手段，也是最有效的手段。

在 Linux 2.4 中，Linux 为每个新创建的进程动态地分配一个 task_struct 结构。系统所允许的最大进程数是由机器所拥有的物理内存的大小决定的,例如,在 IA32 的体系结构中，一个 512MB 内存的机器，其最大进程数可以达到 32KB，这是对旧内核（2.2 以前）版本的极大改进[①]。

Linux 支持多处理机（SMP），所以系统中允许有多个 CPU。Linux 作为多处理机操作系统时，系统中允许的最大 CPU 个数为 32。很显然，Linux 作为单机操作系统时，系统中只有一个 CPU，本书主要讨论单处理机的情况。

和其他操作系统类似，Linux 也支持两种进程：普通进程和实时进程。实时进程具有一定程度上的紧迫性，要求对外部事件做出非常快的响应；而普通进程则没有这种限制。所以，调度程序要区分对待这两种进程，通常，实时进程要比普通进程优先运行。这两种进程的区分也反映在 task_struct 数据结构中了。

总之，包含进程所有信息的 task_struct 数据结构是比较庞大的，但是该数据结构本身并不复杂，我们将它的所有域按其功能可做如下划分：

- 进程状态（State）；
- 进程调度信息（Scheduling Information）；
- 各种标识符（Identifiers）；
- 进程通信有关信息（IPC，Inter_Process Communication）；
- 时间和定时器信息（Times and Timers）；
- 进程链接信息（Links）；
- 文件系统信息（File System）；
- 虚拟内存信息（Virtual Memory）；
- 页面管理信息（page）；
- 对称多处理器（SMP）信息；
- 和处理器相关的环境（上下文）信息（Processor Specific Context）；

[①] 在 Linux 2.2 及以前的版本中，用一个 task 数组来管理系统中所有进程的 task_struct 结构，因此，系统中进程的最大个数受数组大小的限制。

- 其他信息。

下面我们对 task_struct 结构进行具体描述。

4.3 task_struct 结构描述

1．进程状态（State）

进程执行时，它会根据具体情况改变状态。进程状态是调度和对换的依据。Linux 中的进程主要有如下状态，如表 4.1 所示。

表 4.1 Linux 进程的状态

内核表示	含义
TASK_RUNNING	可运行
TASK_INTERRUPTIBLE	可中断的等待状态
TASK_UNINTERRUPTIBLE	不可中断的等待状态
TASK_ZOMBIE	僵死
TASK_STOPPED	暂停
TASK_SWAPPING	换入/换出

（1）可运行状态

处于这种状态的进程，要么正在运行、要么正准备运行。正在运行的进程就是当前进程（由 current 所指向的进程），而准备运行的进程只要得到 CPU 就可以立即投入运行，CPU 是这些进程唯一等待的系统资源。系统中有一个运行队列（run_queue），用来容纳所有处于可运行状态的进程，调度程序执行时，从中选择一个进程投入运行。在后面我们讨论进程调度的时候，可以看到运行队列的作用。当前运行进程一直处于该队列中，也就是说，current 总是指向运行队列中的某个元素，只是具体指向谁由调度程序决定。

（2）等待状态

处于该状态的进程正在等待某个事件（Event）或某个资源，它肯定位于系统中的某个等待队列（wait_queue）中。Linux 中处于等待状态的进程分为两种：可中断的等待状态和不可中断的等待状态。处于可中断等待态的进程可以被信号唤醒，如果收到信号，该进程就从等待状态进入可运行状态，并且加入到运行队列中，等待被调度；而处于不可中断等待态的进程是因为硬件环境不能满足而等待，例如等待特定的系统资源，它任何情况下都不能被打断，只能用特定的方式来唤醒它，例如唤醒函数 wake_up（）等。

（3）暂停状态

此时的进程暂时停止运行来接受某种特殊处理。通常当进程接收到 SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU 信号后就处于这种状态。例如，正接受调试的进程就处于这种状态。

（4）僵死状态

进程虽然已经终止，但由于某种原因，父进程还没有执行 wait()系统调用，终止进程的信息也还没有回收。顾名思义，处于该状态的进程就是死进程，这种进程实际上是系统中的垃圾，必须进行相应处理以释放其占用的资源。

2．进程调度信息

调度程序利用这部分信息决定系统中哪个进程最应该运行，并结合进程的状态信息保证系统运转的公平和高效。这一部分信息通常包括进程的类别（普通进程还是实时进程）、进程的优先级等，如表 4.2 所示。

表 4.2　　进程调度信息

域名	含义
need_resched	调度标志
Nice	静态优先级
Counter	动态优先级
Policy	调度策略
rt_priority	实时优先级

在下一章的进程调度中我们会看到，当 need_resched 被设置时，在“下一次的调度机会”就调用调度程序 schedule()counter 代表进程剩余的时间片，是进程调度的主要依据，也可以说是进程的动态优先级，因为这个值在不断地减少；nice 是进程的静态优先级，同时也代表进程的时间片，用于对 counter 赋值，可以用 nice()系统调用改变这个值；policy 是适用于该进程的调度策略，实时进程和普通进程的调度策略是不同的；rt_priority 只对实时进程有意义，它是实时进程调度的依据。

进程的调度策略有 3 种，如表 4.3 所示。

表 4.3　　进程调度的策略

名称	解释	适用范围
SCHED_OTHER	其他调度	普通进程
SCHED_FIFO	先来先服务调度	实时进程
SCHED_RR	时间片轮转调度	

只有 root 用户能通过 sched_setscheduler（）系统调用来改变调度策略。

3．标识符（Identifiers）

每个进程有进程标识符、用户标识符、组标识符，如表 4.4 所示。

不管对内核还是普通用户来说，怎么用一种简单的方式识别不同的进程呢？这就引入了进程标识符（PID，process identifier），每个进程都有一个唯一的标识符，内核通过这个

标识符来识别不同的进程，同时，进程标识符 PID 也是内核提供给用户程序的接口，用户程序通过 PID 对进程发号施令。PID 是 32 位的无符号整数，它被顺序编号：新创建进程的 PID 通常是前一个进程的 PID 加 1。然而，为了与 16 位硬件平台的传统 Linux 系统保持兼容，在 Linux 上允许的最大 PID 号是 32767，当内核在系统中创建第 32768 个进程时，就必须重新开始使用已闲置的 PID 号。

表 4.4　　各种标识符

域名	含义
Pid	进程标识符
Uid、gid	用户标识符、组标识符
Euid、egid	有效用户标识符、有效组标识符
Suid、sgid	备份用户标识符、备份组标识符
Fsuid、fsgid	文件系统用户标识符、文件系统组标识符

另外，每个进程都属于某个用户组。task_struct 结构中定义有用户标识符和组标识符。它们同样是简单的数字，这两种标识符用于系统的安全控制。系统通过这两种标识符控制进程对系统中文件和设备的访问，其他几个标识符将在文件系统中讨论。

4．进程通信有关信息（IPC，Inter_Process Communication）

为了使进程能在同一项任务上协调工作，进程之间必须能进行通信即交流数据。

Linux 支持多种不同形式的通信机制。它支持典型的 UNIX 通信机制（IPC Mechanisms）：信号（Signals）、管道（Pipes），也支持 System V 通信机制：共享内存（Shared Memory）、信号量和消息队列（Message Queues）,如表 4.5 所示。

表 4.5　　进程通信有关信息

域名	含义
Spinlock_t sigmask_lock	信号掩码的自旋锁
Long blocked	信号掩码
Struct signal *sig	信号处理函数
Struct sem_undo *semundo	为避免死锁而在信号量上设置的取消操作
Struct sem_queue *semsleeping	与信号量操作相关的等待队列

这些域的具体含义将在进程通信一章进行讨论。

5．进程链接信息（Links）

程序创建的进程具有父/子关系。因为一个进程能创建几个子进程，而子进程之间有兄弟关系，在 task_struct 结构中有几个域来表示这种关系。

在 Linux 系统中，除了初始化进程 init，其他进程都有一个父进程（Parent Process）或称为双亲进程。可以通过 fork()或 clone()系统调用来创建子进程，除了进程标识符(PID)等必要的信息外，子进程的 task_struct 结构中的绝大部分的信息都是从父进程中拷贝，或说“克隆”过来的。系统有必要记录这种“亲属”关系，使进程之间的协作更加方便，例如父进程给子进程发送杀死（kill）信号、父子进程通信等，就可以用这种关系很方便地实现。

每个进程的 task_struct 结构有许多指针，通过这些指针，系统中所有进程的 task_struct 结构就构成了一棵进程树，这棵进程树的根就是初始化进程 init 的 task_struct 结构（init 进程是 Linux 内核建立起来后人为创建的一个进程，是所有进程的祖先进程）。表 4.6 是进程所有的链接信息。

表 4.6　　进程链接信息

名称	英文解释	中文解释 [指向哪个进程]
p_opptr	Original parent	祖先
p_pptr	Parent	父进程
p_cptr	Child	子进程
p_ysptr	Younger sibling	弟进程
p_osptr	Older sibling	兄进程
Pidhash_next、Pidhash_pprev		进程在哈希表中的链接
Next_task、 prev_task		进程在双向循环链表中的链接
Run_list		运行队列的链表

6．时间和定时器信息（Times and Timers）

一个进程从创建到终止叫做该进程的生存期（lifetime）。进程在其生存期内使用 CPU 的时间，内核都要进行记录，以便进行统计、计费等有关操作。进程耗费 CPU 的时间由两部分组成：一是在用户模式（或称为用户态）下耗费的时间、一是在系统模式（或称为系统态）下耗费的时间。每个时钟滴答，也就是每个时钟中断，内核都要更新当前进程耗费 CPU 的时间信息。

“时间”对操作系统是极其重要的 。读者可能了解计算机时间的有关知识，例如 8353/8254 这些物理器件，INT 08、INT 1C 等时钟中断等，可能有过编程序时截获时钟中断的成就感，不管怎样，下一章我们将用较大的篇幅尽可能向读者解释清楚操作系统怎样建立完整的时间机制、并在这种机制的激励下进行调度等活动。

建立了“时间”的概念，“定时”就是轻而易举的了，无非是判断系统时间是否到达某个时刻，然后执行相关的操作而已。Linux 提供了许多种定时方式，用户可以灵活使用这些方式来为自己的程序定时。

表 4.7 是和时间有关的域，上面所说的 counter 是指进程剩余的 CPU 时间片，也和时间

有关，所以这里我们再次提及它。表 4.8 是进程的所有定时器。

表 4.7　　与时间有关的域

域名	含义
Start_time	进程创建时间
Per_cpu_utime	进程在某个 CPU 上运行时在用户态下耗费的时间
Per_cpu_stime	进程在某个 CPU 上运行时在系统态下耗费的时间
Counter	进程剩余的时间片

表 4.8　　进程的所有定时器

定时器类型	解释	什么时候更新	用来表示此种定时器的域
ITIMER_REAL	实时定时器	实时更新，即不论该进程是否运行	it_real_value
			it_real_incr
			real_timer
ITIMER_VIRTUAL	虚拟定时器	只在进程运行于用户态时更新	it_virt_value
			it_virt_incr
ITIMER_PROF	概况定时器	进程运行于用户态和系统态时更新	it_prof_value
			it_prof_incr

进程有 3 种类型的定时器：实时定时器、虚拟定时器和概况定时器。这 3 种定时器的特征共有 3 个：到期时间、定时间隔和要触发的事件。到期时间就是定时器到什么时候完成定时操作，从而触发相应的事件；定时间隔就是两次定时操作的时间间隔，它决定了定时操作是否继续进行，如果定时间隔大于 0，则在定时器到期时，该定时器的到期时间被重新赋值，使定时操作继续进行下去，直到进程结束或停止使用定时器，只不过对不同的定时器，到期时间的重新赋值操作是不同的。在表 4.8 中，每个定时器都有两个域来表示到期时间和定时间隔：value 和 incr，二者的单位都是时钟滴答，和 jiffies 的单位是一致的，Linux 所有的时间应用都建立在 jiffies 之上。虚拟定时器和概况定时器到期时由内核发送相应的信号，而实时定时器比较特殊，它由内核机制提供支持，我们将在后面讨论这个问题。

每个时钟中断，当前进程所有和时间有关的信息都要更新：当前进程耗费的 CPU 时间要更新，以便于最后的计费；时间片计数器 counter 要更新，如果 counter<=0，则要执行调度程序；进程申请的延时要更新，如果延时时间到了，则唤醒该进程；所有的定时器都要更新，Linux 内核检测这些定时器是否到期，如果到期，则执行相应的操作。在这里，“ 更新 ”的具体操作是不同的：对 counter，内核要对它减值，而对于所有的定时器，就是检测它的值，内核把系统当前时间和其到期时间作一比较，如果到期时间小于系统时间，则表示该定时器到期。但为了方便，我们把这些操作一概称为“ 更新 ”，请读者注意。

请特别注意上面 3 个定时器的更新时间。实时定时器不管其所属的进程是否运行都要更新，所以，时钟中断来临时，系统中所有进程的实时定时器都被更新，如果有多个进程的实

时定时器到期，则内核要一一处理这些定时器所触发的事件。而虚拟定时器和概况定时器只在进程运行时更新，所以，时钟中断来临时，只有当前进程的概况定时器得到更新，如果当前进程运行于用户态，则其虚拟定时器也得到更新。

此外，Linux 内核对这 3 种定时器的处理是不同的，虚拟定时器和概况定时器到期时，内核向当前进程发送相应的信号：SIGVTALRM 、SIGPROF ；而实时定时器要执行的操作由 real_timer 决定，real_time 是 timer_list 类型的变量（定义：struct timer_list real_timer），其中容纳了实时定时器的到期时间、定时间隔等信息，我们将在下一章详细讨论这些内容。

7．文件系统信息（File System）

进程可以打开或关闭文件，文件属于系统资源，Linux 内核要对进程使用文件的情况进行记录。task_struct 结构中有两个数据结构用于描述进程与文件相关的信息。其中，fs_struct 中描述了两个 VFS 索引节点（VFS inode），这两个索引节点叫做 root 和 pwd，分别指向进程的可执行映像所对应的根目录（Home Directory）和当前目录或工作目录。file_struct 结构用来记录了进程打开的文件的描述符（Descriptor）。如表 4.9 所示。

表 4.9　　与文件系统相关的域

定义形式	解释
Sruct fs_struct *fs	进程的可执行映像所在的文件系统
Struct files_struct *files	进程打开的文件

在文件系统中，每个 VFS 索引节点唯一描述一个文件或目录，同时该节点也是向更低层的文件系统提供的统一的接口。

8．虚拟内存信息（Virtual Memory）

除了内核线程（Kernel Thread），每个进程都拥有自己的地址空间（也叫虚拟空间），用 mm_struct 来描述。另外 Linux 2.4 还引入了另外一个域 active_mm，这是为内核线程而引入的。因为内核线程没有自己的地址空间，为了让内核线程与普通进程具有统一的上下文切换方式，当内核线程进行上下文切换时，让切换进来的线程的 active_mm 指向刚被调度出去的进程的 active_mm（如果进程的 mm 域不为空，则其 active_mm 域与 mm 域相同）。内存信息如表 4.10 所示。

表 4.10　　虚拟内存描述信息

定义形式	解释
Struct mm_struct *mm	描述进程的地址空间
Struct mm_struct *active_mm	内核线程所借用的地址空间

9．页面管理信息

当物理内存不足时，Linux 内存管理子系统需要把内存中的部分页面交换到外存，其交换是以页为单位的。有关页面的描述信息如表 4.11。

表 4.11　　页面管理信息

定义形式	解释
Int swappable	进程占用的内存页面是否可换出
Unsigned long min_flat,maj_flt,nswap	进程累计的次（minor）缺页次数、主(major)次数及累计换出、换入页面数
Unsigned long cmin_flat,cmaj_flt,cnswap	本进程作为祖先进程，其所有层次子进程的累计的次（minor）缺页次数、主(major)次数及累计换出、换入页面数

10．对称多处理机（SMP）信息

Linux 2.4 对 SMP 进行了全面的支持，表 4.12 是与多处理机相关的几个域。

表 4.12　　与多处理机相关的域

定义形式	解释
Int has_cpu	进程当前是否拥有 CPU
Int processor	进程当前正在使用的 CPU
Int lock_depth	上下文切换时内核锁的深度

11．和处理器相关的环境（上下文）信息（Processor Specific Context）

这里要特别注意标题：和“处理器”相关的环境信息。进程作为一个执行环境的综合，当系统调度某个进程执行，即为该进程建立完整的环境时，处理器（Processor）的寄存器、堆栈等是必不可少的。因为不同的处理器对内部寄存器和堆栈的定义不尽相同，所以叫做“和处理器相关的环境”，也叫做“处理机状态”。当进程暂时停止运行时，处理机状态必须保存在进程的 task_struct 结构中，当进程被调度重新运行时再从中恢复这些环境，也就是恢复这些寄存器和堆栈的值。处理机信息如表 4.13 所示。

表 4.13　　与处理机相关的信息

定义形式	解释
Struct thread_struct *tss	任务切换状态

12．其他

(1) struct wait_queue *wait_chldexit

在进程结束时,或发出系统调用 wait4 时，为了等待子进程的结束，而将自己（父进程）睡眠在该等待队列上，设置状态标志为 TASK_INTERRUPTIBLE，并且把控制权转给调度程序。

(2) Struct rlimit rlim[RLIM_NLIMITS]

每一个进程可以通过系统调用 setlimit 和 getlimit 来限制它资源的使用。

(3) Int exit_code exit_signal

程序的返回代码以及程序异常终止产生的信号，这些数据由父进程（子进程完成后）轮流查询。

(4) Char comm[16]

这个域存储进程执行的程序的名字，这个名字用在调试中。

(5) Unsigned long personality

Linux 可以运行 X86 平台上其他 UNIX 操作系统生成的符合 iBCS2 标准的程序，personality 进一步描述进程执行的程序属于何种 UNIX 平台的“个性”信息。通常有 PER_Linux,PER_Linux_32BIT,PER_Linux_EM86,PER_SVR4,PER_SVR3,PER_SCOSVR3,PER_WYSEV386,PER_ISCR4,PER_BSD,PER_XENIX 和 PER_MASK 等,参见 include／Linux/personality.h>。

(6) int did_exec:1

按 POSIX 要求设计的布尔量，区分进程正在执行老程序代码，还是用系统调用 execve() 装入一个新的程序。

(7) struct linux_binfmt *binfmt

指向进程所属的全局执行文件格式结构，共有 a.out、script、elf、java 等 4 种。

综上所述，我们对进程的 task_struct 结构进行了归类讨论，还有一些域没有涉及到，在第六章进程的创建与执行一节几乎涉及到所有的域，在那里可以对很多域有更深入一步的理解。task_struct 结构是进程实体的核心，Linux 内核通过该结构来控制进程：首先通过其中的调度信息决定该进程是否运行；当该进程运行时，根据其中保存的处理机状态信息来恢复进程运行现场，然后根据虚拟内存信息，找到程序的正文和数据；通过其中的通信信息和其他进程实现同步、通信等合作。几乎所有的操作都要依赖该结构，所以，task_struct 结构是一个进程存在的唯一标志。

4.4　task_struct 结构在内存中的存放

task_struct 结构在内存的存放与内核栈是分不开的，因此，首先讨论内核栈。

4.4.1　进程内核栈

每个进程都有自己的内核栈。当进程从用户态进入内核态时，CPU 就自动地设置该进程的内核栈，也就是说，CPU 从任务状态段 TSS 中装入内核栈指针 esp（参见下一章的进程切换一节）。

X86 内核栈的分布如图 4.2 所示。

在 Intel 系统中，栈起始于末端，并朝这个内存区开始的方向增长。从用户态刚切换到内核态以后，进程的内核栈总是空的，因此，esp 寄存器直接指向这个内存区的顶端在图 4.2 中，从用户态切换到内核态后，esp 寄存器包含的地址为 0x018fc00。进程描述符存放在从

0x015fa00 开始的地址。只要把数据写进栈中，esp 的值就递减。

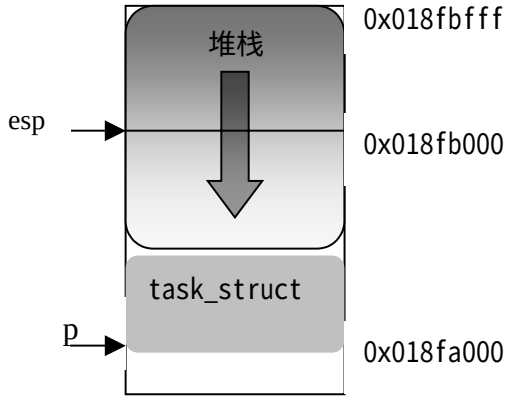

图 4.2　内核栈的分布图

在／include/linux/sched.h 中定义了如下一个联合结构：

```
union task_union {
   struct task_struct task;
   unsigned long stack[2408];
};
```

从这个结构可以看出，内核栈占 8KB 的内存区。实际上，进程的 task_struct 结构所占的内存是由内核动态分配的，更确切地说，内核根本不给 task_struct 分配内存，而仅仅给内核栈分配 8KB 的内存，并把其中的一部分给 task_struct 使用。

task_struct 结构大约占 1K 字节左右，其具体数字与内核版本有关，因为不同的版本其域稍有不同。因此，内核栈的大小不能超过 7KB，否则，内核栈会覆盖 task_struct 结构，从而导致内核崩溃。不过，7KB 大小对内核栈已足够。

把 task_struct 结构与内核栈放在一起具有以下好处：

- 内核可以方便而快速地找到这个结构，用伪代码描述如下：

task_struct = (struct task_struct *) STACK_POINTER & 0xffffe000

- 避免在创建进程时动态分配额外的内存。
- task_struct 结构的起始地址总是开始于页大小（PAGE_SIZE）的边界。

4.4.2　当前进程（current 宏）

当一个进程在某个 CPU 上正在执行时，内核如何获得指向它的 task_struct 的指针？上面所提到的存储方式为达到这一目的提供了方便。在 linux/include/ i386／current.h 中定义了 current 宏，这是一段与体系结构相关的代码：

```
tatic inline struct task_struct * get_current (void)
{
struct task_struct *current;
__asm__ ("andl %%esp,%0; ":"=r" (current) : "0" (~8191UL) ) ;
return current;
}
```

实际上，这段代码相当于如下一组汇编指令（设 p 是指向当前进程 task_struc 结构的指针）：

```
movl $0xffffe000, %ecx
andl %esp, %ecx
movl %ecx, p
```

换句话说，仅仅只需检查栈指针的值，而根本无需存取内存，内核就可以导出 task_struct 结构的地址。

在本书的描述中，会经常出现 current 宏，在内核代码中也随处可见，可以把它看作全局变量来用，例如，current->pid 返回在 CPU 上正在执行的进程的标识符。

另外，在 include/ i386/processor.h 中定义了两个函数 free_task_struct( )和 alloc_task_struct( )，前一个函数释放 8KB 的 task_union 内存区，而后一个函数分配 8KB 的 task_union 内存区。

4.5　进程组织方式

在 Linux 中，可以把进程分为用户任务和内核线程，不管是哪一种进程，它们都有自己的 task_struct。在 2.4 版中，系统拥有的进程数可能达到数千乃至上万个，尤其对于企业级应用（如数据库应用及网络服务器）更是如此。为了对系统中的很多进程及处于不同状态的进程进行管理，Linux 采用了如下几种组织方式。

4.5.1　哈希表

哈希表是进行快速查找的一种有效的组织方式。Linux 在进程中引入的哈希表叫做 pidhash，在 include/linux/sched.h 中定义如下：

```
#define PIDHASH_SZ (4096 >> 2)
extern struct task_struct *pidhash[PIDHASH_SZ];

#define pid_hashfn(x) ((((x) >> 8) ^ (x)) & (PIDHASH_SZ - 1))
```

其中，PIDHASH_SZ 为表中元素的个数，表中的元素是指向 task_struct 结构的指针。pid_hashfn 为哈希函数，把进程的 PID 转换为表的索引。通过这个函数，可以把进程的 PID 均匀地散列在它们的域（0 到 PID_MAX-1）中。

在数据结构课程中我们已经了解到，哈希函数并不总能确保 PID 与表的索引一一对应，两个不同的 PID 散列到相同的索引称为冲突。

Linux 利用链地址法来处理冲突的 PID：也就是说，每一表项是由冲突的 PID 组成的双向链表，这种链表是由 task_struct 结构中的 pidhash_next 和 pidhash_pprev 域实现的，同一链表中 pid 的大小由小到大排列。如图 4.3 所示。

哈希表 pidhash 中插入和删除一个进程时可以调用 hash_ pid( ) 和 unhash_ pid( ) 函数。对于一个给定的 pid，可以通过 find_task_by_pid()函数快速地找到对应的进程：

```
static inline struct task_struct *find_task_by_pid(int pid)
{
```

```
        struct task_struct *p, **htable = &pidhash[pid_hashfn (pid) ];
        for (p = *htable; p && p->pid != pid; p = p->pidhash_next)
                ;
        return p;
}
```

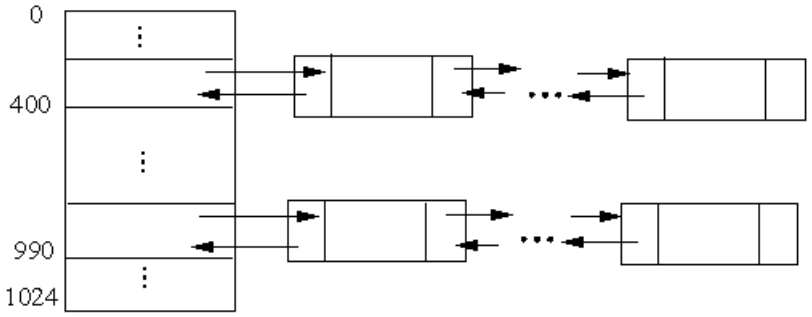

图 4.3　链地址法处理冲突时的哈希表

4.5.2　双向循环链表

哈希表的主要作用是根据进程的 pid 可以快速地找到对应的进程，但它没有反映进程创建的顺序，也无法反映进程之间的亲属关系，因此引入双向循环链表。每个进程 task_struct 结构中的 prev_task 和 next_task 域用来实现这种链表，如图 4.4 所示。

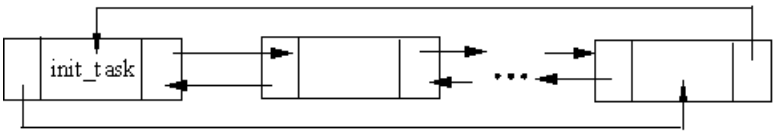

图 4.4　双向循环链表

宏 SET_LINK 用来在该链表中插入一个元素：

```
#define SET_LINKS (p)  do { \
       (p) ->next_task = &init_task; \
       (p) ->prev_task = init_task.prev_task; \
      init_task.prev_task->next_task =  (p) ; \
      init_task.prev_task =  (p) ; \
       (p) ->p_ysptr = NULL; \
       if  ( ( (p) ->p_osptr =  (p) ->p_pptr->p_cptr)  != NULL)  \
                (p) ->p_osptr->p_ysptr = p; \
         (p) ->p_pptr->p_cptr = p; \
       } while  (0)
```

从这段代码可以看出，链表的头和尾都为 init_task，它对应的是进程 0（pid 为 0），也就是所谓的空进程，它是所有进程的祖先。这个宏把进程之间的亲属关系也链接起来。另外，还有一个宏 for_each_task()：

```
#define for_each_task (p)  \
        for  (p = &init_task ;  (p = p->next_task)  != &init_task ; )
```

这个宏是循环控制语句。注意 init_task 的作用，因为空进程是一个永远不存在的进程，因此用它做链表的头和尾是安全的。

因为进程的双向循环链表是一个临界资源，因此在使用这个宏时一定要加锁，使用完后开锁。

4.5.3　运行队列

当内核要寻找一个新的进程在 CPU 上运行时，必须只考虑处于可运行状态的进程（即在 TASK_RUNNING 状态的进程），因为扫描整个进程链表是相当低效的，所以引入了可运行状态进程的双向循环链表，也叫运行队列（runqueue）。

运行队列容纳了系统中所有可以运行的进程，它是一个双向循环队列，其结构如图 4.5 所示。

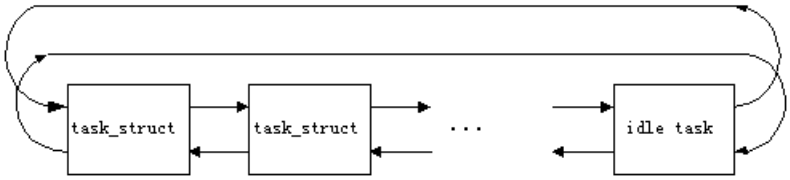

图 4.5　运行队列的结构

4.5.4　进程的运行队列链表

该队列通过 task_struct 结构中的两个指针 run_list 链表来维持。队列的标志有两个：一个是“空进程”idle_task，一个是队列的长度。

有两个特殊的进程永远在运行队列中待着：当前进程和空进程。前面我们讨论过，当前进程就是由 cureent 指针所指向的进程，也就是当前运行着的进程，但是请注意，current 指针在调度过程中（调度程序执行时）是没有意义的，为什么这么说呢？调度前，当前进程正在运行，当出现某种调度时机引发了进程调度，先前运行着的进程处于什么状态是不可知的，多数情况下处于等待状态，所以这时候 current 是没有意义的，直到调度程序选定某个进程投入运行后，current 才真正指向了当前运行进程；空进程是个比较特殊的进程，只有系统中没有进程可运行时它才会被执行，Linux 将它看作运行队列的头，当调度程序遍历运行队列，是从 idle_task 开始、至 idle_task 结束的，在调度程序运行过程中，允许队列中加入新出现的可运行进程，新出现的可运行进程插入到队尾，这样的好处是不会影响到调度程序所要遍历的队列成员，可见，idle_task 是运行队列很重要的标志。

另一个重要标志是队列长度，也就是系统中处于可运行状态（TASK_RUNNING）的进程数目，用全局整型变量 nr_running 表示，在/kernel/fork.c 中定义如下：

```
int nr_running=1;
```

若 nr_running 为 0，就表示队列中只有空进程。在这里要说明一下：若 nr_running 为 0，则系统中的当前进程和空进程就是同一个进程。但是 Linux 会充分利用 CPU 而尽量避免出

现这种情况。

4.5.5 等待队列

在 2.4 版本中，引入了一种特殊的链表——通用双向链表，它是内核中实现其他链表的基础，也是面向对象的思想在 C 语言中的应用。在等待队列的实现中多次涉及与此链表相关的内容。

1．通用双向链表

在 include/linux/list.h 中定义了这种链表：

```
struct list_head {
        struct list_head *next, *prev;
};
```

这是双向链表的一个基本框架，在其他使用链表的地方就可以使用它来定义任意一个双向链表，例如：

```
struct foo_list {
     int data;
     struct list_head list;
};
```

对于 list_head 类型的链表，Linux 定义了 5 个宏：

```
#define LIST_HEAD_INIT (name)  { & (name) , & (name)  }

#define LIST_HEAD (name)  \
        struct list_head name = LIST_HEAD_INIT (name)

#define INIT_LIST_HEAD (ptr)  do { \
         (ptr) ->next =  (ptr) ;  (ptr) ->prev =  (ptr) ; \
} while  (0)

#define list_entry (ptr, type, member)  \
         ( (type *)  ( (char *)  (ptr) - (unsigned long)  (& ( (type *) 0) ->member) ) )

#define list_for_each (pos, head)  \
        for  (pos =  (head) ->next; pos !=  (head) ; pos = pos->next)
```

前 3 个宏都是初始化一个空的链表，但用法不同，LIST_HEAD_INIT()在声明时使用，用来初始化结构元素，第 2 个宏用在静态变量初始化的声明中，而第 3 个宏用在函数内部。

其中，最难理解的宏为 list_entry ()，在内核代码的很多处都用到这个宏，例如，在调度程序中，从运行队列中选择一个最值得运行的进程，部分代码如下：

```
static LIST_HEAD (runqueue_head) ;
struct list_head *tmp;
struct task_struct *p;

list_for_each (tmp, &runqueue_head)  {
    p = list_entry (tmp, struct task_struct, run_list) ;
```

```
        if (can_schedule (p) ) {
            int weight = goodness (p, this_cpu, prev->active_mm) ;
            if (weight > c)
                c = weight, next = p;
        }
    }
```

从这段代码可以分析出 list_entry(ptr, type, member)宏及参数的含义： ptr 是指向 list_ head 类型链表的指针，type 为一个结构，而 member 为结构 type 中的一个域，类型为 list_head，这个宏返回指向 type 结构的指针。在内核代码中大量引用了这个宏，因此，搞清楚这个宏的含义和用法非常重要。

另外，对 list_head 类型的链表进行删除和插入（头或尾）的宏为 list_del()/list_add()/list_ add_tail()，在内核的其他函数中可以调用这些宏。例如，从运行队列中删除、增加及移动一个任务的代码如下：

```
static inline void del_from_runqueue (struct task_struct * p)
{
        nr_running--;
        list_del (&p->run_list) ;
        p->run_list.next = NULL;
}
static inline void add_to_runqueue (struct task_struct * p)
{
        list_add (&p->run_list, &runqueue_head) ;
        nr_running++;
}

static inline void move_last_runqueue (struct task_struct * p)
{
        list_del (&p->run_list) ;
        list_add_tail (&p->run_list, &runqueue_head) ;
}

static inline void move_first_runqueue (struct task_struct * p)
{
        list_del (&p->run_list) ;
        list_add (&p->run_list, &runqueue_head) ;
}
```

2．等待队列

运行队列链表把处于 TASK_RUNNING 状态的所有进程组织在一起。当要把其他状态的进程分组时，不同的状态要求不同的处理，Linux 选择了下列方式之一。

- TASK_STOPPED 或 TASK_ZOMBIE 状态的进程不链接在专门的链表中,也没必要把它们分组，因为父进程可以通过进程的 PID 或进程间的亲属关系检索到子进程。
- 把 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE 状态的进程再分成很多类，其每一类对应一个特定的事件。在这种情况下，进程状态提供的信息满足不了快速检索进程，因此，有必要引入另外的进程链表。这些链表叫等待队列。

等待队列在内核中有很多用途，尤其对中断处理、进程同步及定时用处更大。因为这些

内容在以后的章节中讨论，我们只在这里说明，进程必须经常等待某些事件的发生，例如，等待一个磁盘操作的终止，等待释放系统资源或等待时间走过固定的间隔。等待队列实现在事件上的条件等待，也就是说，希望等待特定事件的进程把自己放进合适的等待队列，并放弃控制权。因此，等待队列表示一组睡眠的进程，当某一条件变为真时，由内核唤醒它们。等待队列由循环链表实现。在 2.4 版中，关于等待队列的定义如下（为了描述方便，有所简化）：

```
struct __wait_queue {
        unsigned int flags;
        struct task_struct * task;
        struct list_head task_list;
} ;
typedef struct __wait_queue wait_queue_t ;
```

另外，关于等待队列另一个重要的数据结构——等待队列首部的描述如下：

```
struct __wait_queue_head {
         wq_lock_t lock;
        struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;
```

在这两个数据结构的定义中，都涉及到类型为 list_head 的链表，这与 2.2 版定义是不同的，在 2.2 版中的定义为：

```
struct wait_queue {
        struct task_struct * task;
         struct wait_queue * next ;
} ;
typedef struct wait_queue wait_queue_t ;
typedef struct wait_queue *wait_queue_head_t ;
```

这里要特别强调的是，2.4 版中对等待队列的操作函数和宏比 2.2 版丰富了，而在你编写设备驱动程序时会用到这些函数和宏，因此，要注意 2.2 到 2.4 函数的移植问题。下面给出 2.4 版中的一些主要函数及其功能：

- init_waitqueue_head（）——对等待队列首部进行初始化
- init_waitqueue_entry（）——对要加入等待队列的元素进行初始化
- waitqueue_active（）——判断等待队列中已经没有等待的进程
- add_wait_queue（）——给等待队列中增加一个元素
- remove_wait_queue（）——从等待队列中删除一个元素

注意，在以上函数的实现中，都调用了对 list_head 类型链表的操作函数(list_del()/list_add()/list_add_tail())，因此可以说，list_head 类型相当于 C++中的基类型，这也是对 2.2 版的极大改进。

希望等待一个特定事件的进程能调用下列函数中的任一个：

sleep_on( )函数对当前的进程起作用，我们把当前进程叫做 P：

```
sleep_on (wait_queue_head_t *q)
{
         SLEEP_ON_VAR   ／＊宏定义，用来初始化要插入到等待队列中的元素＊／
         current->state = TASK_UNINTERRUPTIBLE;
         SLEEP_ON_HEAD  ／＊宏定义，把 P 插入到等待队列 ＊／
```

```
        schedule () ;
        SLEEP_ON_TAIL   ／＊宏定义把 P 从等待队列中删除 ＊／
}
```

这个函数把 P 的状态设置为 TASK_UNINTERRUPTIBLE，并把 P 插入等待队列。然后，它调用调度程序恢复另一个程序的执行。当 P 被唤醒时，调度程序恢复 sleep_on(　)函数的执行，把 P 从等待队列中删除。

- interruptible_sleep_on(　)与 sleep_on(　)函数是一样的，但稍有不同，前者把进程 P 的状态设置为 TASK_INTERRUPTIBLE 而不是 TASK_UNINTERRUPTIBLE，因此，通过接受一个信号可以唤醒 P。
- sleep_on_timeout(　) 和 interruptible_sleep_on_timeout(　)与前面情况类似，但它们允许调用者定义一个时间间隔，过了这个间隔以后，内核唤醒进程。为了做到这点，它们调用 schedule_timeout(　)函数而不是 schedule(　)函数。

利用 wake_up 或者 wake_up_interruptible 宏，让插入等待队列中的进程进入 TASK_RUNNING 状态，这两个宏最终都调用了 try_to_wake_up(　)函数：

```
static inline int try_to_wake_up (struct task_struct * p, int synchronous)
{
    unsigned long flags;
      int success = 0;
     spin_lock_irqsave (&runqueue_lock, flags) ; /＊加锁＊／
      p->state = TASK_RUNNING;
    if  (task_on_runqueue (p) )       ／＊判断 p 是否已经在运行队列＊／
            goto out;
      add_to_runqueue (p) ;        ／＊不在，则把 p 插入到运行队列＊／
     if  (!synchronous || ! (p->cpus_allowed &  (1 << smp_processor_id () ) ) )   ／
           reschedule_idle (p) ;
     success = 1;
  out:
    spin_unlock_irqrestore (&runqueue_lock, flags) ;  ／＊开锁＊／
    return success;
}
```

在这个函数中，p 为要唤醒的进程。如果 p 不在运行队列中，则把它放入运行队列。如果重新调度正在进行的过程中，则调用 reschedule_idle () 函数，这个函数决定进程 p 是否应该抢占某一 CPU 上的当前进程（参见下一章）。

实际上，在内核的其他部分，最常用的还是 wake_up 或者 wake_up_interruptible 宏，也就是说，如果你要在内核级进行编程，只需调用其中的一个宏。例如一个简单的实时时钟（RTC）中断程序如下：

```
static DECLARE_WAIT_QUEUE_HEAD (rtc_wait) ; ／＊初始化等待队列首部＊／

void rtc_interrupt (int irq, void *dev_id, struct pt_regs *regs)
{
        spin_lock (&rtc_lock) ;
        rtc_irq_data = CMOS_READ (RTC_INTR_FLAGS) ;
        spin_unlock (&rtc_lock) ;
        wake_up_interruptible (&rtc_wait) ;
}
```

这个中断处理程序通过从实时时钟的 I/O 端口（CMOS_READ 宏产生一对 outb/inb）读取

数据，然后唤醒在 rtc_wait 等待队列上睡眠的任务。

4.6 内核线程

内核线程（thread）或叫守护进程（daemon），在操作系统中占据相当大的比例，当 Linux 操作系统启动以后，尤其是 Xwindow 也启动以后，你可以用“ ps ”命令查看系统中的进程，这时会发现很多以“ d ”结尾的进程名，这些进程就是内核线程。

内核线程也可以叫内核任务，它们周期性地执行，例如，磁盘高速缓存的刷新，网络连接的维护，页面的换入换出等。在 Linux 中，内核线程与普通进程有一些本质的区别，从以下几个方面可以看出二者之间的差异。

- 内核线程执行的是内核中的函数,而普通进程只有通过系统调用才能执行内核中的函数。
- 内核线程只运行在内核态，而普通进程既可以运行在用户态，也可以运行在内核态。
- 因为内核线程指只运行在内核态，因此，它只能使用大于 PAGE_OFFSET（3G）的地址空间。另一方面，不管在用户态还是内核态，普通进程可以使用 4GB 的地址空间。

内核线程是由 kernel_thread( )函数在内核态下创建的，这个函数所包含的代码大部分是内联式汇编语言，但在某种程度上等价于下面的代码：

```
int kernel_thread (int  (*fn)  (void *) , void * arg,
                   unsigned long flags)
{
    pid_t p ;
    p = clone ( 0, flags | CLONE_VM ) ;
    if  ( p )         /* parent */
        return p;
    else {          /* child */
        fn (arg)  ;
        exit (  )  ;
   }
}
```

系统中大部分的内核线程是在系统的启动过程中建立的，其相关内容将在启动系统一章进行介绍。

4.7 进程的权能

Linux 用“ 权能（capability）”表示一进程所具有的权力。一种权能仅仅是一个标志，它表明是否允许进程执行一个特定的操作或一组特定的操作。这个模型不同于传统的“ 超级用户对普通用户 ”模型，在后一种模型中，一个进程要么能做任何事情，要么什么也不能做，这取决于它的有效 UID。也就是说，超级用户与普通用户的划分过于笼统。如表 4.13 给出了在 Linux 内核中已定义的权能。

表 4.13　　进程的权能

名字	描述
CAP_CHOWN	忽略对文件和组的拥有者进行改变的限制
CAP_DAC_OVERRIDE	忽略文件的访问许可权
CAP_DAC_READ_SEARCH	忽略文件/目录读和搜索的许可权

续表

名字	描述
CAP_FOWNER	忽略对文件拥有者的限制
CAP_FSETID	忽略对 setid 和 setgid 标志的限制
CAP_KILL	忽略对信号挂起的限制
CAP_SETGID	允许 setgid 标志的操作
CAP_SETUID	允许 setuid 标志的操作
CAP_SETPCAP	转移/删除对其他进程所许可的权能
CAP_LINUX_IMMUTABLE	允许对仅追加和不可变文件的修改
CAP_NET_BIND_SERVICE	允许捆绑到低于 1024TCP/UDP 的套节字
CAP_NET_BROADCAST	允许网络广播和监听多点传送
CAP_NET_ADMIN	允许一般的网络管理。
CAP_NET_RAW	允许使用 RAW 和 PACKET 套节字
CAP_IPC_LOCK	允许页和共享内存的加锁
CAP_IPC_OWNER	跳过 IPC 拥有者的检查
CAP_SYS_MODULE	允许内核模块的插入和删除
CAP_SYS_RAWIO	允许通过 ioperm(　) 和 iopl(　)访问 I/O 端口
CAP_SYS_CHROOT	允许使用 chroot(　)
CAP_SYS_PTRACE	允许在任何进程上使用 ptrace(　)
CAP_SYS_PACCT	允许配置进程的计账
CAP_SYS_ADMIN	允许一般的系统管理
CAP_SYS_BOOT	允许使用 reboot(　)
CAP_SYS_NICE	忽略对 nice(　)的限制
CAP_SYS_RESOURCE	忽略对几个资源使用的限制
CAP_SYS_TIME	允许系统时钟和实时时钟的操作
CAP_SYS_TTY_CONFIG	允许配置 tty 设备

任何时候，每个进程只需要有限种权能，这是其主要优势。因此，即使一位有恶意的用户使用有潜在错误程序，他也只能非法地执行有限个操作类型。

例如，假定一个有潜在错误的程序只有 CAP_SYS_TIME 权能。在这种情况下，利用其错误的恶意用户只能在非法地改变实时时钟和系统时钟方面获得成功。他并不能执行其他任何特权的操作。

4.8 内核同步

内核中的很多操作在执行的过程中都不允许受到打扰，最典型的例子就是对队列的操作。如果两个进程都要将一个数据结构链入到同一个队列的尾部，要是在第 1 个进程完成了一半的时候发生了调度，让第 2 个进程插了进来，就可能造成混乱。类似的干扰可能来自某个中断服务程序或 bh 函数。在多处理机 SMP 结构的系统中，这种干扰还有可能来自另一个处理器。这种干扰本质上表项为对临界资源（如队列）的互斥使用。下面介绍几种避免这种干扰的同步方式。

4.8.1 信号量

进程间对共享资源的互斥访问是通过“信号量”机制来实现的。信号量机制是操作系统教科书中比较重要的内容之一。Linux 内核中提供了两个函数 down()和 up（），分别对应于操作系统教科书中的 P、V 操作。

信号量在内核中定义为 semaphore 数据结构，位于 include/i386/semaphore.h：

```
struct semaphore {
        atomic_t count;
            int sleepers;
        wait_queue_head_t wait;
 #if WAITQUEUE_DEBUG
        long __magic;
 #endif
 };
```

其中的 count 域就是“信号量”中的那个“量”，它代表着可用资源的数量。如果该值大于 0，那么资源就是空闲的，也就是说，该资源可以使用。相反，如果 count 小于 0，那么这个信号量就是繁忙的，也就是说，这个受保护的资源现在不能使用。在后一种情况下，count 的绝对值表示了正在等待这个资源的进程数。该值为 0 表示有一个进程正在使用这个资源，但没有其他进程在等待这个资源。

Wait 域存放等待链表的地址，该链表中包含正在等待这个资源的所有睡眠的进程。当然，如果 count 大于或等于 0，则等待队列为空。为了明确表示等待队列中正在等待的进程数，引入了计数器 sleepers。

down()和 up()函数主要应用在文件系统和驱动程序中，把要保护的临界区放在这两个函数中间，用法如下：

```
down () ;
临界区
up () ;
```

这两个函数是用嵌入式汇编实现的，非常麻烦，在此不予详细介绍。

4.8.2 原子操作

避免干扰的最简单方法就是保证操作的原子性，即操作必须在一条单独的指令内执行。

有两种类型的原子操作，即位图操作和数学的加减操作。

1．位图操作

在内核的很多地方用到位图，例如内存管理中对空闲页的管理，位图还有一个广泛的用途就是简单的加锁，例如提供对打开设备的互斥访问。关于位图的操作函数如下：

以下函数的参数中，addr 指向位图。

- void set_bit(int nr, volatile void *addr)：设置位图的第 nr 位。
- void clear_bit(int nr, volatile void *addr)：清位图的第 nr 位。
- void change_bit(int nr, volatile void *addr)：改变位图的第 nr 位。
- int test_and_set_bit(int nr, volatile void *addr)：设置第 nr 位，并返回该位原来的值，且两个操作是原子操作，不可分割。
- int test_and_clear_bit(int nr, volatile void *addr)：清第 nr 为，并返回该位原来的值，且两个操作是原子操作。
- int test_and_change_bit(int nr, volatile void *addr)：改变第 nr 位，并返回该位原来的值，且这两个操作是原子操作。

这些操作利用了 LOCK_PREFIX 宏，对于 SMP 内核，该宏是总线锁指令的前缀，对于单 CPU 这个宏不起任何作用。这就保证了在 SMP 环境下访问的原子性。

2．算术操作

有时候位操作是不方便的，取而代之的是需要执行算术操作，即加、减操作及加 1、减 1 操作。典型的例子是很多数据结构中的引用计数域 count（如 inode 结构）。这些操作的原子性是由 atomic_t 数据类型和表 4.14 中的函数保证的。atomic_t 的类型在 include/i386/atomic.h，定义如下：

```
typedef struct { volatile int counter; } atomic_t;
```

表 4.14　　原子操作

函数	说明
atomic_read(v)	返回*v
atomic_set(v,i)	把*v 设置成 I
Atomic_add(i,v)	给*v 增加 I
Atomic_sub(i,v)	从*v 中减去 I
Atomic_inc(v)	给*v 加 1
Atomic_dec(v)	从*v 中减去 1
Atomic_dec_and_test(v)	从*v 中减去 1，如果结果非空就返回 1；否则返回 0
Atomic_inc_and_test_greater_zero(v)	给*v 加 1，如果结果为正就返回 1；否则就返回 0
Atomic_clear_mask(mask,addr)	清除由 mask 所指定的 addr 中的所有位
Atomic_set_mask(mask,addr)	设置由 mask 所指定的 addr 中的所有位

4.8.3 自旋锁、读写自旋锁和大读者自旋锁

在 Linux 开发的早期，开发者就面临这样的问题，即不同类型的上下文（用户进程对中断）如何访问共享的数据，以及如何访问来自多个 CPU 同一上下文的不同实例。

在 Linux 内核中，临界区的代码或者是由进程上下文来执行，或者是由中断上下文来执行。在单 CPU 上，可以用 cli/sti 指令来保护临界区的使用，例如：

```
unsigned long flags;

save_flags(flags);
cli();
/* critical code */
restore_flags(flags);
```

但是，在 SMP 上，这种方法明显是没有用的，因为同一段代码序列可能由另一个进程同时执行,而 cli()仅能单独地为每个 CPU 上的中断上下文提供对竞争资源的保护，它无法对运行在不同 CPU 上的上下文提供对竞争资源的访问。因此，必须用到自旋锁。

所谓自旋锁，就是当一个进程发现锁被另一个进程锁着时，它就不停地“旋转”，不断执行一个指令的循环直到锁打开。自旋锁只对 SMP 有用，对单 CPU 没有意义。

有 3 种类型的自旋锁：基本的、读写以及大读者自旋锁。读写自旋锁适用于“多个读者少数写者”的场合，例如，有多个读者仅有一个写者，或者没有读者只有一个写者。大读者自旋锁是读写自旋锁的一种，但更照顾读者。大读者自旋锁现在主要用在 Sparc64 和网络系统中。

本章对进程进行了全面描述，但并不是对每个部分都进行了深入描述，因为在整个操作系统中，进程处于核心位置，因此，内核的其他部分（如文件、内存、进程间的通信等）都与进程有密切的联系，相关内容只能在后续的章节中涉及到。通过本章的介绍，读者应该对进程有一个全方位的认识：

（1）进程是由正文段（Text）、用户数据段（User Segment）以及系统数据段（System Segment）共同组成的一个执行环境。

（2）Linux 中用 task_struct 结构来描述进程，也就是说，有关进程的所有信息都存储在这个数据结构中，或者说，Linux 中的进程与 task_struct 结构是同意词，在英文描述中，有时把进程（Process）和线程（Thread）混在一起使用，但并不是说，进程与线程有同样的含义，只不过描述线程的数据结构也是 task_struct。task_struct 就是一般教科书上所讲的进程控制块.。

（3）本章对 task_struct 结构中存放的信息进行了分类讨论，但并不要求在此能掌握所有的内容，相对独立的内容为进程的状态，在此再次给出概述。

- TASK_RUNNING：也就是通常所说的就绪（Ready）状态。
- TASK_INTERRUPTIBLE：等待一个信号或一个资源（睡眠状态）。
- TASK_UNINTERRUPTIBLE：等待一个资源（睡眠状态），处于某个等待队列中。
- TASK_ZOMBIE：没有父进程的子进程。
- TASK_STOPPED：正在被调试的任务。

（4）task_struct 结构与内核栈存放在一起，占 8KB 的空间。

(5) 当前进程就是在某个 CPU 上正在运行的进程，Linux 中用宏 current 来描述，也可以把 curennt 当作一个全局变量来用。

(6) 为了把内核中的所有进程组织起来，Linux 提供了几种组织方式，其中哈希表和双向循环链表方式是针对系统中的所有进程（包括内核线程），而运行队列和等待队列是把处于同一状态的进程组织起来。

(7) Linux 2.4 中引入一种通用链表 list_head，这是面向对象思想在 C 中的具体实现，在内核中其他使用链表的地方都引用了这种基类型。

(8) 进程的权能和内核的同步我们仅仅做了简单介绍，因为进程管理会涉及到这些内容，但它们不是进程管理的核心内容，引入这些内容仅仅是为了让读者在阅读源代码时扫除一些障碍。

(9) 进程的创建及执行将在第六章的最后一节进行讨论。