第3章 进 程

本章描述了什么是进程以及Linux内核如何创建、管理和删除系统中的进程。

进程执行操作系统中的任务。一个程序是保存在一个磁盘的可执行映像中的机器代码指令和数据的集合，并且，实际上是一个被动实体；一个进程可以被认为是一个执行中的计算机程序。它是一个动态实体，总是随着机器代码指令随处理器的执行而处于变化之中。除了程序的指令和数据，进程还包括程序计数器和所有 CPU寄存器，以及含有例程参数、返回地址和保存的变量等临时变量的进程栈 (stack)。当前正执行的程序或说进程，包含所有处理器当前的行为。Linux是一个多进程操作系统。进程是独立的任务，有自己的权利和责任。若一个进程崩溃并不会引起系统中另一个进程崩溃。每个单独的进程运行在自己的虚拟地址空间，并且只能通过安全的内核管理机制和其他进程交互。

在进程的生存期(lifetime)内将使用许多系统资源。它将使用系统的 CPU来运行自己的指令并使用系统的物理内存来保存自己和自己的数据；它将打开和使用文件子系统中的文件并直接或间接地使用系统中的物理设备。 Linux必须跟踪进程本身和它拥有的系统资源，来保证它能公平地管理该进程和系统中其他进程。如果一个进程独自占有系统中的大部分物理内存或其CPU，对系统中其他进程来说将是不公平的。

系统中最宝贵的资源是CPU，通常只有一个。Linux是个多进程操作系统，它的目标是在每一时刻都有一个进程运行在系统的每个 CPU上，来极大化CPU利用率。若进程比CPU多(通常是这样)，其余的进程在它们可以运行前必须等待一个 CPU变空闲。多进程基于这样一个简单的思想：一个进程执行直到它必须等待，通常是等待一些系统资源；当它获得这个资源后，就可以再次运行。在单进程系统中，比如 DOS，CPU将简单地闲置等待，等待的时间将被浪费。在多进程系统中内存里同时保存多个进程。每当一个进程需要等待时，操作系统从该进程夺走CPU并给予另一个更有价值的进程。选择哪一个进程是下一次运行最合适的进程是调度器(scheduler)的事，Linux使用一些调度策略来保证公平。

Linux支持几种不同的可执行文件格式， ELF是一种，Java是另一种；并且它们要透明地管理，因为进程要使用系统的共享库。

3.1 Linux进程

为了使Linux可以管理系统中的进程，每个进程通过一个 task_struct数据结构表示 (任务(task)和进程(process)是Linux中两个互相通用的术语)。Task向量是一个指向系统中所有task_struct数据结构的指针数组。这意味着系统中最大进程受限于 Task向量的大小，缺省情况下它有512项。当进程被创建时，从系统内存中分配一个新的 task_struct并把它加入到task向量中。为了便于查找，当前运行的进程由 current指针来指向。

除了普通类型的进程， Linux还支持实时进程。这些进程必须对外部事件反应迅速 (从而得到名字“ 实时 ”)，并且它们被调度器区别于普通用户进程对待。尽管 task_struct数据结构很大很复杂，但它的字段可以被分成几个功能区。

• 状态　随着进程执行，它根据其环境改变状态。Linux进程有以下状态：

a. 运行　进程或者正在运行(它是系统中的当前进程)或者已经准备好运行(等待被分配到系统的一个CPU上)。

b. 等待　进程正在等待一个事件或一个资源。Linux区分两种类型的进程；可中断的和不可中断的。可中断等待进程可以被信号中断，而不可中断等待进程直接等待硬件条件，并且在任何环境下都不会被中断。

c. 停止　进程停止了，通常是通过接收一个信号。一个正在被调试的进程可以处于停止状态。

d. 死亡　这是一个被终止的进程，由于某种原因，仍在 task向量中有一个task_struct数据结构，正像其名字那样，它是一个死去的进程。

• 调度信息　调度器需要这些信息以公平地决定系统中哪个进程最值得运行。

• 标识　系统中每个进程有一个进程标识。进程标识不是 task向量的索引，它只是一个简单的数。每个进程还有用户和组标识，这些被用来控制本进程对系统中文件和设备的访问。

• 进程间通信　Linux支持经典的UNIXTM IPC机制如信号、管道和信号灯，以及 System V IPC机制如共享内存、信号灯和消息队列。Linux支持的IPC机制在第4章描述。

• 链接　在Linux系统中没有哪个进程与其他进程完全独立。除了初始进程，系统中每个进程都有一个父进程，新的进程不是被创建，它们是从先前的进程被复制 (或说克隆)。每一个表示一个进程的task_struct都有指针指向其父进程及其兄弟进程 (和它具有相同父进程的进程)以及它自己的子进程。你可以用 ptree命令查看Linux系统中运行的进程之间的家族关系：

```
init(1)-+-crond(98)
        |-emacs(387)
        |-gpm(146)
        |-inetd(110)
        |-kerneld(18)
        |-kflushd(2)
        |-klogd(87)
        |-kswapd(3)
        |-login(160)—bash(192)—emacs(225)
        |-lpd(121)
        |-mingetty(161)
        |-mingetty(162)
        |-mingetty(163)
        |-mingetty(164)
        |-login(403)—bash(404)—pstree(594)
        |-sendmail(134)
        |-syslogd(78)
        `-update(166)
```

另外系统中所有的进程都保存在一个双向链表中，它的根是 init进程的task_struct数据结构。这个链表使得 Linux可以查看系统中每一个进程，它需要这样来为一些命令如 ps或kill提供支持。

• 时间和时钟　内核记住进程的创建时间以及在它生存期内消耗的 CPU时间。每次时钟

“滴答”时，内核更新保留在jiffies中的时间量，表示当前进程花费在系统和用户态下的时间。Linux还支持进程相关的间隔定时器。进程可以用系统调用来设置定时器，以便当定时器超时时给进程自己发送信号。定时器可以是一次触发的或周期性多发的。

- 文件系统　进程在需要的时候可以打开和关闭文件；进程的task_struct包含每个打开的文件的描述符指针以及两个VFS索引节点的指针。每个VFS索引节点唯一地描述文件系统中的一个文件或目录，并提供一个底层文件系统的统一接口。 Linux对文件系统的支持将在第7章描述。两个VFS索引节点指针第一个指向进程的根目录，第二个指向其当前的或称pwd目录。pwd从UNIX命令pwd——打印工作目录(print working directory)而来。这两个VFS索引节点使自己的count字段(field)递增来表示一个或多个进程在引用它们。这就是为什么不能删除被一个进程设成 pwd的目录的原因。同样的原因，也不能删除它的子目录。
- 虚拟内存　大多数进程有一些虚拟内存(内核线程和守护进程没有)，并且Linux必须跟踪虚拟内存如何映射到系统物理内存。
- 处理器相关上下文　一个进程可以被认为是系统当前状态的总和。每当一个进程运行时，它要使用处理器的寄存器、栈等等，这是进程的上下文 (context)。并且，每当一个进程被暂停时，所有的CPU相关上下文必须被保存在该进程的 task_struct中。当进程被调度器重新启动时其上下文将从这里恢复。

3.2 标识符

Linux像所有UNIXTM系统一样使用用户和组标识来检查对系统中文件和映像的访问权限。Linux系统中所有文件都有所有权和授权，这些授权描述系统中用户对该文件或目录有什么访问权限。基本的授权是读、写和执行并被赋予三种用户：文件的所有者、属于一个特定组的进程和系统中所有的进程。每种用户可以有不同的授权，例如一个文件可以有授权使得其所有者可以读写，文件的组成员可以读而系统中其他进程没有任何访问权。

组是Linux给一组用户(而不是一个用户或系统中所有进程)赋予访问文件或目录特权的方式。例如，你可以为一个软件项目的所有用户创建一个组，并可以设定只有他们能够读写该项目的源代码。一个进程可以属于几个组 (缺省最大值是32个)，并且它们被保存在每个进程task_struct中的groups向量中。只要进程所在的一个组有对某文件的访问权限，该进程就拥有对该文件的适当的组权利。

一个进程task_struct中有4对进程和组标识符：

- uid、gid　进程运行所代表用户的用户标识符和组标识符。
- 有效uid、gid　有些程序把从执行进程来的uid和gid改变成自己的(作为属性保存在描述可执行映像的VFS索引节点中)。这些程序被称为setuid程序，并且它们很有用，因为它提供一种限制访问某些服务的方式，特别是那些代表其他用户运行的进程，比如一个网络守护进程。有效uid和gid是来自setuid程序的而其uid和gid保持不变。当内核检查特权时就检查有效uid和gid。
- 文件系统uid和gid　这些通常和有效uid、gid相同，并在检查文件系统访问权利的时候被使用。它们在NFS安装的文件系统中需要，在那里用户模式下的 NFS服务器需要像一个特定进程一样访问文件。这种情况下只有文件系统uid和gid被改变(而不是有效uid和gid),

这样可以防止恶意用户向NFS服务器发送kill信号。Kill信号将被送到一个有特定有效uid和gid的进程。

- 保存的uid和gid　这是POSIX标准所要求的并被那些通过系统调用改变进程 uid和gid的进程使用。它们用来在初始uid和gid被改变期间保存真正的uid和gid。

3.3 调度

所有的进程都是部分运行在用户模式下，部分运行在系统模式下。这些模式怎样被底层硬件支持是随硬件不同而不同的，但通常都有一个安全的机制来从用户模式进入系统模式以及再返回。用户模式拥有的特权比系统模式少得多。每次进程进行一次系统调用时，它从用户模式切换到系统模式然后继续执行，这时内核代表进程执行。在 Liunx 中，进程不能抢先当前的、正在运行的进程，它们不能停止其运行以便使它们自己能运行。每个进程当必须等待某个系统事件时，会释放它正在其上运行的 CPU。例如，一个进程可能需要等待从一个文件中读入一个字符，这种等待发生在系统调用中。在系统模式下，进程使用一个库函数来打开和读取文件，接着进行系统调用来从打开的文件中读取字节。在这种情况下，等待的进程将被暂停而另一个更有价值的进程将被选中运行。

进程总是在进行系统调用，所以经常需要等待。即使如此，若一个进程执行直到它等待才让出CPU的话，就仍可能使用了不合适的 CPU时间，所以Linux使用抢先调度。在这种方案中，每个进程被允许运行少量的时间 (200ms)当这段时间用完后另一个进程被选中来运行，而原先的进程要等待一会直到它可以再次运行。这段运行的少量时间被称为时间片 (time-slice)。

调度器必须从系统中可运行的进程中选取出最值得运行的进程。可运行的进程是指只等待CPU来运行的进程。Linux使用一个合理的基于简单优先权的调度算法来从系统中现有的进程中选择。当选中一个新进程来运行，它将保存当前进程的状态，处理器相关寄存器和其他上下文被保存在进程 task_struct数据结构中。然后它恢复新进程的状态 (同样这是处理器相关的)来运行并把系统控制交给该进程。调度器为了在系统中可运行进程间公平地分配 CPU时间，为每个进程在task_struct中保存有如下信息：

- policy　将被应用于本进程的调度策略。有两种 Linux进程：普通的和实时的。实时进程拥有比其他进程都要高的优先级。如果有一个实时进程准备好了运行，它总是先运行。实时进程可以有两种高度策略：“ 轮转(round robin) ”法和 “ 先进先出(first in first out) ”。在轮转法调度中每个可运行的实时进程依次运行；而在先进先出法中，实时进程按它们进入运行队列的顺序依次运行，并且该顺序永不会改变。
- priority　调度器将给予进程的优先级。它也是进程被允许运行时可以运行的时间量 (在jiffies中)。可以通过系统调用的方法和 renice命令来改变进程的优先级。
- rt_priority　Linux支持实时进程，并且它们在调度时拥有比系统中其他非实时进程更高的优先级。这个字段使调度器可以给每个实时进程一个相对优先级。实时进程的优先级可以用系统调用改变。
- counter　此进程允许运行的时间量(在jitties中)在第一次运行时被置成priority的值，然后在每个时钟 “ 滴答 ” 中递减。

调度器可以从内核中几个地方运行。它可以在把当前进程放到等待队列中后运行，也可以在一个系统调用结束并且刚好一个进程从系统模式返回进程模式时运行。另一个需要运行

的原因是系统定时器刚好把当前进程的counter置为0，每次调度器运行时它作以下工作：

- 内核工作　调度器运行Bottom Half控制程序，并处理调度器任务队列。这些轻量内核线程在第9章中详细描述。
- 当前进程　在另一个进程被选择运行前必须处理当前进程，如果当前进程的调度策略是轮转法，则它被放到运行队列尾；如果任务是可中断的，并且从最近一次被调度后收到了一个信号，则它的状态变为RUNNING。

如果当前进程时间用完了，那么其状态变为RUNNING。

如果当前进程是RUNNING，则它保持该状态。

既不是RUNNING也不是可中断的进程将从运行队列中移出。这意味着当调度器寻找最有价值的进程来运行时，它们不会被考虑。

- 进程选择　调度器查看运行队列中的所有进程来寻找最有价值的来运行。如果有实时进程(有实时调度策略的进程)那么它们将得到比普通进程更高的优先。也就是如果系统中有可运行的实时进程的话，在别的普通进程运行之前总是先运行这些实时进程。当前进程，已经消耗了一部分自己的时间片(其counter被减去了)，如果系统中有其他相同优先级进程的话将处于不利地位；事实上也应该是这样。如果有几个具有相同优先级的进程，则最靠近运行队列头的被选中。当前进程将被放到运行队列的尾部。在一个平衡的有许多相同优先级进程的系统中，每个进程将依次运行，这就是所谓的轮转法调度。然而，因为进程会等待资源，它们的运行顺序会改变。
- 切换进程　如果最有价值运行的进程不是当前进程，那么当前进程必须被暂停并使新进程运行。当进程运行时它正使用CPU和系统的寄存器和物理内存。每次它调用一个例程为它在寄存器中传递参数，并且可能把保存的值压到栈上，比如返回到调用例程的地址。所以，当调度器运行时它是在当前进程的上下文中运行。它将处于一个特权模式即核心模式，但运行的仍是当前进程。当该进程被暂停时，其所有的机器状态，包括程序计数器(PC)和处理器的所有寄存器，必须被保存在进程的task_struct数据结构中。然后，新进程的所有机器状态必须被装入。这是一个系统相关的操作，没有CPU用相同的方法完成此任务，但是通常有一些完成该动作的硬件帮助。

进程的切换发生在调度的最后。因此，被保存的原先进程的上下文是系统硬件上下文的一个快照(snapshot)，因为在调度最后还是在这个进程中运行。同样，当新进程上下文被装入时，它也是调度末尾情况的一个快照，包括此进程的程序计数器和寄存器内容。

如果先前进程或新进程使用虚拟内存，则系统的页表可能需要更新。这个动作又是体系结构相关的。像Alpha AXP这样使用转换旁视表或缓存的页表项的处理器，必须清空这些缓存的属于先前进程的表项。

多处理器系统中的调度

在Linux世界中有多个CPU的系统很少，但却已经做了许多工作来使　Linux成为一个SMP(Symmetric Multi-Processing,对称多处理器)操作系统。也就是一个可以在系统中CPU之间平等地均衡负载的操作系统。这项均衡工作并不比调度器中的明显。

在一个多处理器系统中，希望所有处理器都忙于运行进程。每个处理器在其当前进程用尽它的时间片后或需要等待系统资源时将独立地运行调度器。　SMP系统中要注意的第一件事

是系统中不只一个空闲进程。在单处理器系统中空闲进程是 task向量中第一个任务；在SMP系统中每个CPU有一个空闲进程而你可以有多于一个的空闲 CPU。另外，每个CPU有一个当前进程，所以SMP系统必须为每个处理器跟踪当前和空闲进程。

在SMP系统中每个进程的task_struct包含进程正在运行的处理器号 (processor)和它上次运行的处理器号 (tast_processor)。一个进程每次被选中后可在不同的 CPU上运行，但可以用processor_mask来把一个进程限制在一个或多个处理器上。如果位 N被置位，则此进程可以在处理器N上运行。当调度器选择一个新的进程运行时 ,它不会考虑那些没有在其processor_mask中对应于当前处理器的位置位的进程。调度器同样给予上次运行在本处理器上的进程轻微的优先，因为当移动一个进程到不同的处理器上时经常会有性能开销。

3.4 文件

图1-3-1显示了系统中每个进程有两个数据结构描述文件系统相关信息。第一个 —— fs_struct，包含指向此进程VFS索引节点的指针和umask。umask是新文件被创建的缺省模式，它可以通过系统调用来改变。

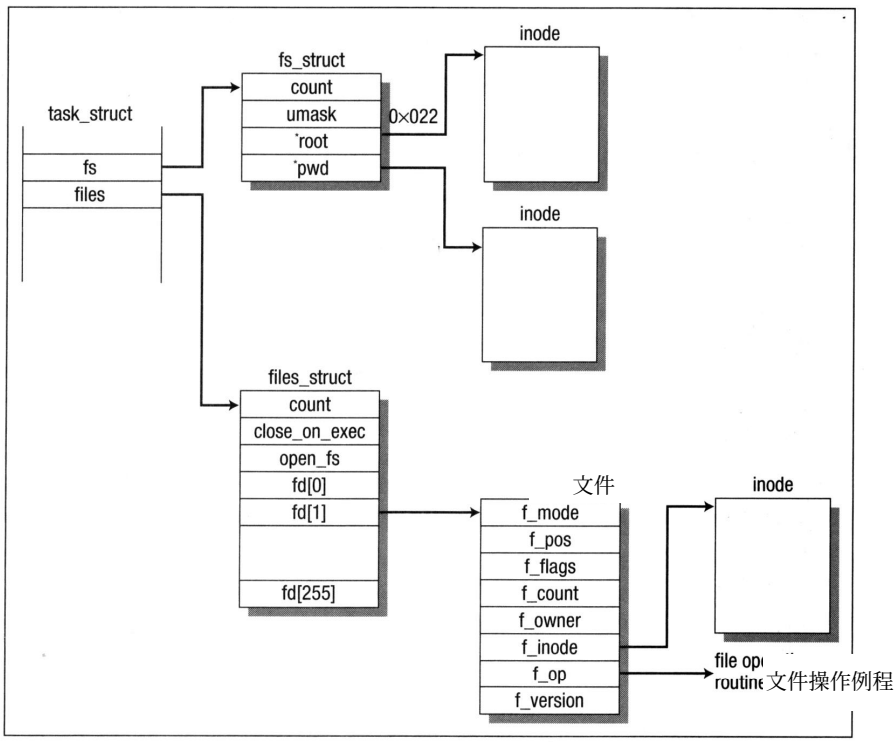

图1-3-1 进程的文件

第二个数据结构 —— Files_struct，包含此进程正在使用的所有文件的信息。程序从标准输入设备(standard input)读入而写出到标准输出设备(standard output)。任何错误消息应该输出到标准错误设备(standard error)。这些设备可以是文件、终端输入 /输出或一个真实的设备，但就程序而言它们都被当作文件对待。每个文件有自己的描述符并且 files_struct中包含至多256

个file数据结构的指针，每个数据结构描述一个被此进程使用的文件。 f_mode字段描述该文件是以什么模式创建的：只读、读写还是只写。 f_pos保存文件中下一个读或写将发生的位置。f_inode描述文件的VFS索引节点，而f_ops是一个例程地址向量的指针，每个代表一个想施加于文件的操作的函数。例如，有一个写数据函数。这个接口的抽象非常强有力并允许 Linux支持广泛的文件类型。后面我们会看到Linux中管道就是用这种机制实现的。

每次一个文件被打开时， files_struct中的空闲file指针之一就被用来指向新的 file结构。Linux进程期望在启动时有三个文件描述符被打开了，它们就是标准输入设备、标准输出设备和标准错误设备，并且通常它们是从创建此进程的父进程继承得来的。所有对文件的访问是通过传递或返回文件描述符的标准系统调用进行的。这些描述符是进程 fd向量的索引，所以标准输入设备、标准输出设备和标准错误设备分别对应文件描述符 0、1和2。每次对文件访问都是使用file数据结构的文件操作例程以及VFS索引节点来达到需求。

3.5 虚拟内存

一个进程的虚拟内存包含来自多处的可执行代码和数据。首先是被装入的程序映像，比如一个命令如fs。与其他所有可执行映像一样，这个命令由可执行代码和数据组成。映像文件包含把可执行代码和相关联的程序数据装入进程虚拟内存中所需要的所有信息。其次进程在其处理过程中可以分配(虚拟)内存来使用，比如保存它读取的文件的内容。这新分配的虚拟内存需要被链接到进程已有的虚拟内存以便使用。第三， Linux进程使用公共用途代码库，比如文件处理例程。每个进程有自己的库副本是没有意义的， Linux使用可以被几个运行的进程同时使用的共享库。这些共享库中的代码和数据必须被链接到此进程的虚拟地址空间，以及其他共享这些库的进程的虚拟地址空间。

在一个给定的时间段中一个进程不会使用包含在其虚拟内存中的全部代码和数据，它可能仅使用在特定情况下使用的代码(如初始化时或处理特定事件时)，也可能只使用共享库中一部分例程。将全部这些代码和数据装入物理内存将是浪费：它们不可能被同时使用。随着系统中进程数的增多，这种浪费被成倍地扩大，系统将非常低效地运行。事实上， Linux使用一种称为请求调页(demand-Paging)的技术：只有当进程要使用时其虚拟内存才被装入到物理内存。所以，不是直接把代码和数据装入物理内存， Linux内核只修改进程的页表，标识出虚拟内存页存在但不在内存中。当进程想要访问代码或数据时，系统硬件将产生页故障并把控制交给Linux内核来解决。因此，对于进程地址空间中的每一个内存区， Linux都需要知道该虚拟内存来自何处，以及如何把它装入内存以解决页故障。

Linux内核需要管理所有这些虚拟内存区，并且每个进程虚拟内存的内容通过其task_struct中指向的一个mm_struct数据结构来描述。进程的mm_struct数据结构还包含装入的可执行映像的信息和一个指向进程页表的指针。它包含指针指向一个 vm_area_struct数据结构列表，每个vm_area_struct代表此进程中的一个虚拟内存区。

这个链表是按虚拟内存中上升顺序排列，图 1-3-2显示了一个简单进程的虚拟内存布局，以及管理它的内核数据结构。因为这些虚拟内存区来自多处， Linux 使vm_area_struct指向一个虚拟内存处理例程的集合(通过vm_ops)来抽象接口。这样进程所有的虚拟内存可以用统一的方法处理，而不管内存管理的底层服务如何变化。例如当进程试图访问不存在的虚拟内存时将有一个例程被调用，这就是页故障的处理方法。

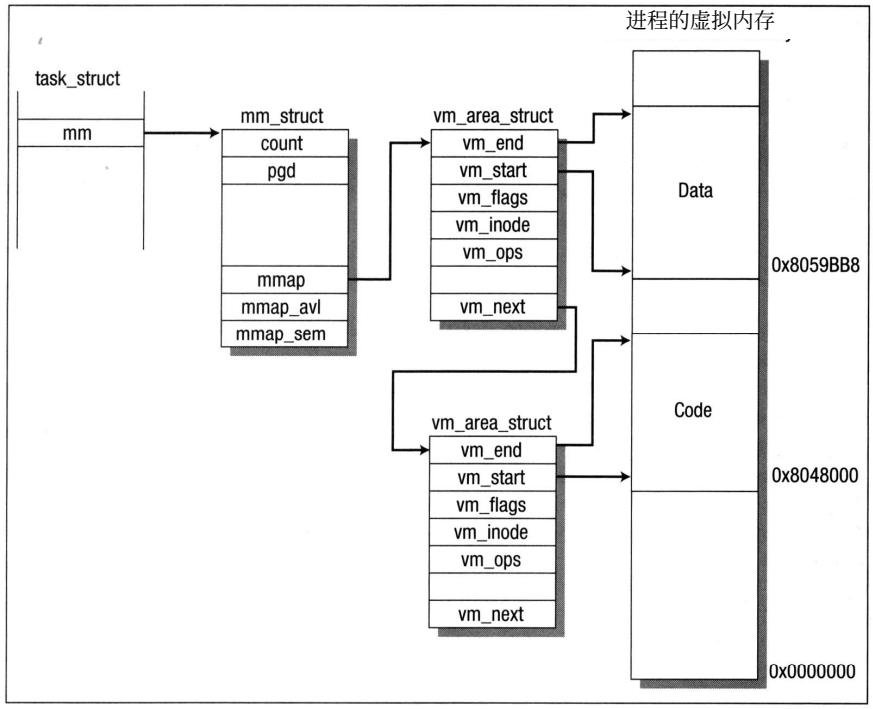

图1-3-2 进程的虚拟内存

随着Linux为进程创建新的虚拟内存区以及解决对不在系统物理内存中的虚拟内存引用的问题，进程的vm_area_struct数据结构集合将不断被 Linux内核访问。这使得查找到正确的vm_area_struct所花费的时间对系统性能很重要。为了加快这种访问， Linux同样把vm_area_struct数据结构组织成一个 AVL(Adelson-Velskii和Landis)树。这种树被组织成每个vm_area_struct(或节点)有一个左指针和一个右指针指向其邻接 vm_area_struct结构。左指针指向一个具有比自己低的起始虚拟地址的节点，而右指针指向一个具有比自己高的起始虚拟地址的节点。为了找到正确的节点， Linux从树的根节点开始沿每个节点的左、右指针进行直到它找到正确的 vm_area_struct。当然，世界上没有免费的午餐，在这种树中插入一个新的vm_area_struct将花费额外的处理时间。

当一个进程分配虚拟内存时， Linux并不真正为它保留物理内存。它只是创建一个新vm_area_struct数据结构来描述虚拟内存，这个结构被链入进程的虚拟内存列表。当进程试图写一个位于新分配虚拟内存区域的虚拟地址时，系统将产生页故障。处理器试图转换该虚拟地址，但是因为没有此内存的页表项，它将放弃并产生一个页故障异常，留给 Linux内核来解决。Linux查看被引用的虚拟地址是否位于当前进程的虚拟内址空间。如果是 Linux创建适当的PTE并为此进程分配一页物理内存。代码或数据可能需要从文件系统或交换硬盘上读入物理内存。然后进程可以从引起页故障的那条指令处重启，并且因为这次内存物理地址存在，所以它可以继续执行。

3.6 创建进程

当系统启动时它在核心模式下运行，并且在某种意义上，只有一个进程 —— 初始进程。像

所有进程一样，初始进程有一个机器状态，用栈、寄存器等表示。当系统中其他进程被创建执行时，这些将被保存在初始进程的task_struct数据结构中。在系统初始化的结尾，初始进程启动一个内核线程(称为init)，然后就空闲等待，什么也不做。每当没有其他事情时调度器就运行这个空闲的进程。空闲进程的task_struct是唯一非动态分配的，它是在内核建造时静态定义，并且很奇怪地被称为init_task。

Init内核线程或进程具有进程标识符1，因为它是系统中第一个真正的进程。它完成一些系统的初始配置(如打开系统的控制盘以及安装根文件系统)，然后执行系统初始化程序——/etc/init、/bin/init或/sbin/init之一，至于使用哪一个则由系统而定。 Init程序使用/etc/inittab作为一个脚本文件来创建系统的新进程，这些新进程自己可以继续创建新进程。例如 getty进程在用户试图登陆时，可以创建一个login进程。系统中所有进程都是init内核线程的后代。

新进程是通过克隆旧进程或说克隆当前进程而创建。新任务通过一个系统调用 (fork或clone)而创建，克隆过程发生在核心模式下的内核中，在系统调用的末尾会有一个新进程等待调度器选中它并运行。一个新的task_struct数据结构被从系统物理内存分配，以及一页或多页物理内存作为克隆的进程的栈(用户的和核心的)。一个进程标识符可能会被创建：一个在系统进程标识符集合中唯一的标识符。然而，克隆的进程完全有理由保存其父进程的标识符。新的task_struct新加入到task向量并且老的(当前)进程的task_struct的内容被复制到克隆出的task_struct中。

当克隆进程时，Linux允许两个进程共享资源而不是有两份独立的副本。这种共享可用于进程的文件、信号处理器和虚拟内存。当资源被共享时，它们的 count字段将被递增，以便在两个进程都结束访问它们之前Linux 不会回收这些资源。举例来说，如果克隆的进程将共享虚拟内存，其task_struct将包含指向原先进程的 mm_struct的指针并且该mm_struct将把其count字段递增以表明当前共享该页的进程数。

克隆一个进程的虚拟内存是很有技巧性的。一个新的 vm_area_struct集合以及它们拥有的mm_struct数据结构，还有被克隆的进程的页表必须被产生出来。在这时没有进程的任何虚拟内存被复制。复制将是一件很困难和冗长的工作，因为一些虚拟内存在物理内存，一些在进程正在执行的可执行映像中，还可能有一些在交换文件中；相反， Linux使用称为“写时复制(copy on write)”的技巧，这意味着仅当两个进程试图写它时虚拟内存才会被复制。任何虚拟内存只要没被写，即使它可以被写，也将在两个进程间共享而不会引起任何危害。只读的内存(比如可执行代码)总是被共享。为了使“写时复制”工作，可写区域将其页表项标识为只读并且vm_area_struct数据结构描述为它们被标识成”写时复制”。当一个进程试图写该虚拟内存时将产生一个页故障。正是在此时 Linux将产生该内存的一个副本，并修改两个进程的页表和虚拟内存数据结构。

3.7 时间和定时器

内核保持跟踪一个进程的创建时间以及它在生存期中消费的 CPU时间。每个时钟“滴答”一下，内核就更新jiffies中当前进程花费在系统模式和用户模式下的时间量。

除了这些计数定时器外，进程支持进程相关的间隔定时器。进程可以使用这些定时期，在每次它们超时时给自己发送各种各样的信号。 Linux支持三种类型的间隔定时器：

- 真实的 这种定时器按真实时间跳动，当它超时时发送给进程一个 SIGALRM信号。

- 虚拟的 这种定时器仅在进程运行时才跳动，当它超时时发送给进程一个 SIGVTALRM 信号。
- 描述的 这种定时器在进程运行时和系统代表进程执行时都将跳动，当它超时时将发送 SIGPROF信号。

一种或全部间隔定时器都可以运行， Linux将所有必需信息保存在进程的 task_struct数据结构中。可以进行系统调用来设置这些间隔定时器，以及启动、停止它们或读取其当前的值。虚拟和描述定时器以相同方式处理。每个时钟“ 滴答 ”，当前进程的间隔定时器被递减并且如果它们超时，就发送相应的信号。

真实定时器有些不同， Linux将使用第9章中描述的定时机制。每个进程有自己的 time_list 数据结构，并且当真实间隔定时器运行时，它被排队到系统的定时器列表。当定时器超时时，定时器底层部分处理器把它移出队列并调用间隔定时器处理器。这将产生 SIGALRM信号并重启间隔定时，把它加入到系统定时器队列尾部。

3.8 执行程序

在Linux中，像UNIX中一样，程序和命令通常由命令解释器执行。命令解释器是一个像其他进程一样的用户进程，它被称为shell。Linux中有许多shell，一些最流行的如sh、bash和tcsh。除了一些内建的命令如cd和pwd之外，一个命令就是一个可执行二进制文件。对于每个输入的命令，shell在保存于PATH环境变量中的进程的搜索路径目录中寻找一个名字匹配的可执行映像。如果文件找到了，它就被装入并执行。shell使用上面描述的fork机制克隆自身，然后新的子进程用刚找到的可执行映像文件的内容替换它正执行的 shell二进制映像。通常 shell等待命令结束，或说等待子进程退出。你可以通过把子进程推到后面来使 shell再次运动。敲入comtrol_z，将使一个SIGSTOP信号发送到子进程，使它停止。然后使用 shell命令bg把它推到后面， shell发送一个SIGCONT信号重新启动它，它将留在后面直到结束或需要做终端输入或输出。

可执行文件可以有许多种格式甚至是一个脚本文件。脚本文件必须被识别出来并运行适当的解释器来处理，例如， /bin/sh解释shell脚本。可执行目标文件包含可执行代码和数据，以及足够的信息使操作系统可以把它们装入内存并执行。 Linux中最常用的目标文件格式是ELF，但是，理论上Linux足够灵活以处理几乎任何目标文件格式。

像对文件系统一样， Linux支持的二进制格式或是在内核建造时内置进内核的，或者可以作为模块被装入。内核保持一个所支持的二进制格式列表 (见图1-3-3)，当试图执行一个文件时，依次试用每一种二进制格式直到有一种成功。通常Linux支持的二进制格式是a_out和ELF。可执行文件不必完全被装入内存，而是使用了一种称为请求调页的技术。随着每一部分可执行映像被进程使用，它被读入内存。未被使用的部分映像可以从内存中淘汰。

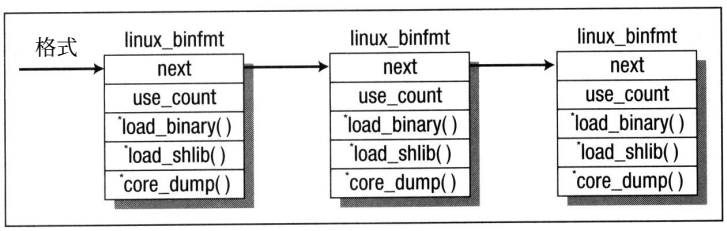

图1-3-3 注册的二进制格式

3.8.1 ELF

由Unix系统实验室设计的ELF(Executable and Linkable Format，可执行和可链接格式)目标文件格式已牢固树立了Linux中最常使用的格式的地位。尽管和其他目标文件格式如 ECOFF和a.out相比有轻微性能开销，但ELF被认为更灵活。ELF可执行文件包含可执行代码，有时被称为正文(text)，以及数据(data)。可执行映像中的表格描述程序应该怎样被放到进程的虚拟内存中。静态链接映像被链接器(ld)或链接编辑器建造成一个单一的映像，包含运行此映像所需的所有代码和数据。映像还指明此映像在内存中的布局，以及此映像中第一条执行的代码的地址。

图1-3-4显示了一个静态链接的可执行映像的布局。它是一个简单的打印“ hello world ”，然后就退出的程序。

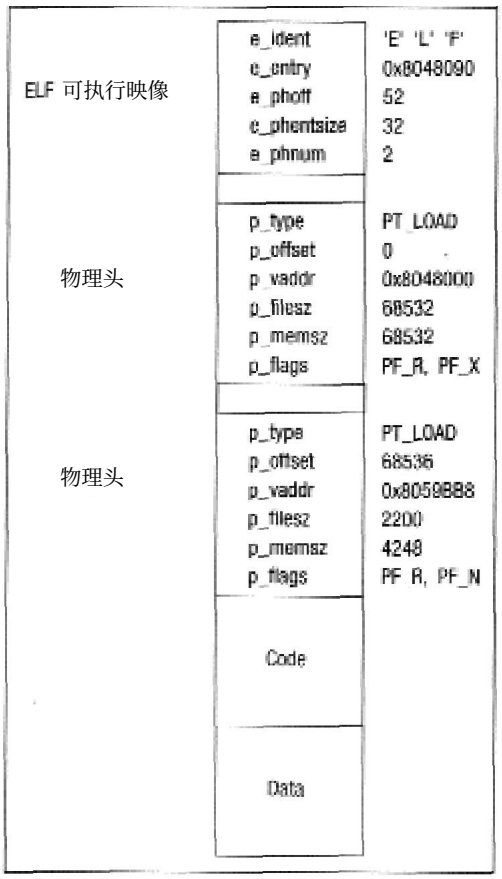

图1-3-4 ELF可执行文件格式

头信息说明它是一个 ELF映像，在文件开始 52字节处(e_phoff)有两个物理头(e_phnum为2)。第一个物理头描述映像中的可执行代码。它从虚拟地址0x8048000处开始，共有 65532字节。这是因为它是一个静态链接和映像，包含了输出“ hello world ”的printf( )调用的全部库代码。这个映像的入口点，即程序的第一条指令，不是映像的开始处而是在虚拟地址 0x8048090(e_entry)处，代码紧接第二个物理头开始。这个物理头描述了程序的数据并将装入到虚拟内存中地址0x8059BB8处。这段数据既是可读又是可写的。读者将注意到该数据在文件中的大小是2200字节(p_filesz)，而它在内存中的大小是4248字节。这是因为前面的 2200字节包含预初始化的数据，而后面 2048字节包含的数据将被执行的代码初始化。

当Linux把一个ELF可执行映像装入进程的虚拟地址空间时，它并不实际地装入映像。它设置好虚拟内存数据结构，进程的 vm_area_struct树及其页表。当程序被执行时，页故障将引起程序的代码和数据被取到物理内存中，程序中未被使用的部分将永远不会被装入内存。一旦ELF二进制格式装入器发现该映像是一个有效的ELF可执行映像，它就把进程的当前可执行映像从其虚拟内存中冲掉。因为这个进程是被克隆的映像(所有的进程都是)，这个老的映像是其父进程正在执行的程序，比如是命令解释器 bash，把老的可执行映像冲掉，将淘汰老的虚拟内存数据结构并重置进程的页表。它同时清除任何建立的信号处理器，并关闭任何打开的文件。最后进程已经为新的可执行映像作好准备。不管可执行映像是什么格式，相同的信息都将被设置到进程的mm_struct中。其中有映像的代码和数据的起始和结束指针。这些值在读取ELF可执行映像的物理头时可以得到，它们描述的程序段被映射

到进程的虚拟地址空间。也就在这时 vm_area_struct数据结构被设置，进程的页表被修改。Mm_struct数据结构也含有指向将被传到此程序的参数的指针以及指向此进程环境变量的指针。

ELF共享库

另一方面，一个动态链接映像并不包含运行时需要的所有代码和数据。一部分被保存在运行时被链接到映像的共享库中。在把共享库链接到映像时，动态链接器也要使用 ELF共享库的表格。Linux使用几种动态链接器：ld.so.1、libc.so.1和ld-linux.so.1。所有这些都可以在/lib下找到。库中包含通常使用的代码如语言子例程。如果没有动态链接，所有程序将需要有自己的这些库的副本，并需要更多的磁盘空间和虚拟内存。在动态链接中。每个引用的库例程信息都被包括在 ELF映像的表格中。这些信息指示动态链接器如何定位库例程以及如何把它链到程序的地址空间中。

3.8.2 脚本文件

脚本文件是需要解释器来运行的可执行文件。 Linux有许多可用的解释器，例如 wish、perl及命令shell如tcsh。Linux使用标准UNIX的习惯：用脚本文件的第一行包含解释器的名字。所以，一个典型的脚本文件将像下面这样开始：

```
#!/usr/bin/wish
```

脚本二进制装入器试着为脚本找到解释器。它通过试着打开在脚本第一行中提到的可执行文件来进行。如果能够打开，装入器将有一个其 VFS索引节点的指针，它可以继续让其解释脚本文件。脚本文件的名字作为参数 0(第一个参数)，而其他参数依次排列(原先的第一个参数成为新的第二个参数，等等)。装入解释器是用与 Linux装入其所有可执行文件一样的方式完成。Linux依次试用每种二进制格式直到一种成功。这意味着在理论上可以有几种解释器和二进制格式，使得Linux二进制格式处理器成为很灵活的软件。