第9章　内核机制与模块

9.1　内核机制

本节讲述Linux内核提供的几种通用任务和机制，正是在它们的支持下，内核的其他部分才得以协调一致地工作。

9.1.1　Bottom Half控制

核态(kernel)下有很多时候系统为了进行一项工作而无法再干其他的事情，中断处理就是很好的例子，当有中断时，处理机停止当前的工作，由操作系统将中断转交给相应的设备驱动程序，然后等待，因此设备驱动程序应快速完成中断的处理，提高系统效率。然而还有另外一些工作，系统不必为它们停止当前的处理，因为可以稍后再进行这些工作，　Linux的Bottom Half控制程序正是为了处理这些工作而设计的。图 1-9-1给出了Bottom Half控制程序所用的内核数据结构，由图中可看出，一共有 32个不同的Bottom Half控制程序，同时有一个包含32个指针的bh_base向量，每个指针指向一个 Bottom Half控制例程。bh_active和bh_mask通过其位值分别指出安装了哪些控制程序和哪些控制程序是活跃的：如果 bh_mask第N位被置1，则表明bh_base中的第N个指针指向了一个 Bottom Half例程；如果bh_active第N位被置1，则表明一旦调度进程许可，立即调用第 N个Bottom Half例程。这些索引值都是静态设定的，如定时器Bottom Half控制器具有最高优先级 (索引值0)，控制台Bottom Half控制器优先级稍低(索引值 1)等等。典型的 Bottom Half控制例程总与一个任务表相关联，比如　Immediate Bottom Half控制程序负责immediate任务队列(tq-immediate)的处理，该队列中包含了需立即执行的任务。

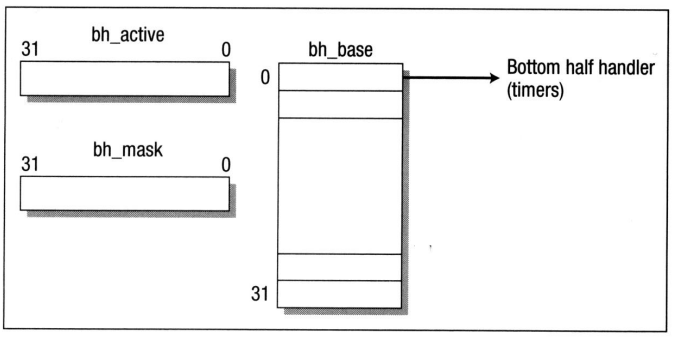

图1-9-1　Bottom Half控制数据结构

有一些内核的Bottom Half控制程序是由硬件设备所定义的，但另外一些则是通用的，如下所示：

TIMER	每次系统周期性定时器中断均被调用，以驱动内核定时器队列机制
CONSOLE	处理控制台消息
TQUEUE	处理tty消息
NET	处理通用网络运行
IMMEDIATE	为一些设备驱动程序设计的通用控制，用于将稍后进行的工作排队

一旦设备驱动程序或内核其他部分需要调度待执行工作，首先要加入工作到相应的系统队列中(例如定时器队列)，然后通知内核去执行某些 Bottom Half控制程序，这是通过设置bh_active中的相应位来实现的。如果驱动程序将某些工作加到了 immediate队列中，并希望执行Immediate Bottom Half以处理这些工作，则其会把 bh_active的第8位置1。在每次系统调用之后并尚未将控制器交给调用进程之前，都要检测 bh_active的各个位置，若发现某些位被置1，则调用相应的Bottom Half控制程序，检测顺序由第0位到第31位，调用完成之后bh_active中相应位清零。bh_active是暂时的，仅在两次调度进程调用之间有意义，对它的使用可避免无任何工作要做时盲目调Bottom Half控制程序。

9.1.2 任务队列

任务队列是内核用于延迟某些工作的一种方法。 Linux对于将工作排队稍后处理有一种通用机制。任务队列通常用于 Bottom Half控制程序的连接，当运行定时器队列 Bottom Half控制程序时则处理定时器队列。如图 1-9-2所示，任务队列是一种简单的数据结构，它包含了一个tq_struct数据结构的单向链表，每个 tq_struct数据结构中有一个例程地址和一个指向一些数据的指针，当处理任务队列中的各项时，将调用这个例程，并同时传递给它数据指针。

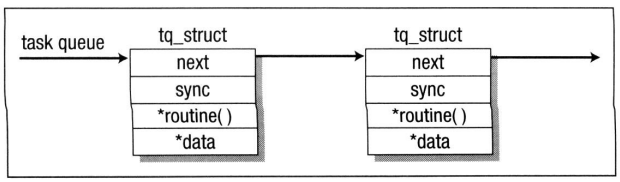

图1-9-2 任务队列

内核中的任何进程，例如设备驱动程序，都可以创建并使用任务队列，但其中有三种任务队列只能由内核进行创建和管理：

- timer 希望在下一个时钟节拍之后尽可能快地进行的工作使用本队列。每到一个时钟节拍，就检测本队列，看看其是否为空，若不空，则置定时器队列 Bottom Half控制程序为活跃的，当下一次运行调度进程时，则与其他 Bottom Half控制进程一起运行定时器队列 Bottom Half控制进程。不要把系统定时器与本队列混淆了，那是一个更为复杂的机制。
- immediate 调度进程运行活跃的Bottom Half控制程序时将同时处理本队列，从优先级上来看，Immediate Bottom Half控制程序低于定时器队列Bottom Half控制程序，因此它将在定时器任务处理之后执行。
- schedule 由调度进程直接运行，用于支持系统中的其他任务队列，从这点来讲，本队列的各个任务就是一个处理任务队列的例程。

每当处理完任务队列中的一项，就从队列中删除相应指针 (将其值置为NULL)。事实上，这种删除操作是一种原子(atomic)操作，不会被中断，这样队列中的每一项都依次调用其控制

例程。队列中的项通常是静态分配的数据，但对这些数据的删除并无由内存继承而来的回收机制。任务队列处理例程仅仅是把下一项加入表中，而正确释放掉所分配核心内存的工作是由这些任务自己来完成的。

9.1.3 定时器

操作系统有时需要安排将来的运行活动，因此需要提供一种机制，在该机制下，这些运行活动能够在相对准确的时间点上开始运行。任一个能支持操作系统的微处理器都必须支持可编程的内部定时器，该定时器将周期地中断处理器，这个定时器就是众所周知的系统时钟，它像一种节拍一样安排系统的各种活动。 Linux对时间有一种简单的观点：它从系统启动时开始以时钟节拍计时，所有的系统时间都基于这种计时方式，它的值可以通过全局变量 jiffies取到。

Linux有两种系统定时器，在某一系统时间同时被调用，但它们在实现上略有不同，图 1-9-3给出这两种机制。第一种，即老的定时器机制，有一个包含 32个指针的静态数据组和一个活跃定时器屏蔽码 (timer_active)，这些指针指向timer_struct数据结构，定时器程序与定时器表的连接是静态定义的，大多数定时器程序入口是在系统初始化时加入到定时器表中的；第二种，即新的定时器机制，使用了一个链表，表中的 timer_list数据结构以递增的超时数排序。

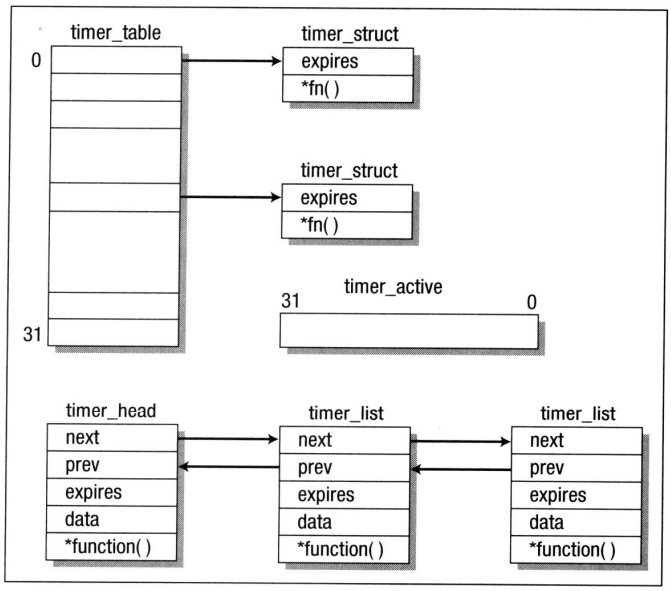

图1-9-3 系统定时器

两种机制都是用 jiffies值判断是否超时，因此如果一个定时器希望定时 5秒，那么它先要把5秒转化为jiffies的单位，然后把转换后的值加上当前系统时间以得到定时器超时的 jiffies值。由于每一系统时钟节拍都要把定时器 Bottom Half控制程序置为活跃，因此每当下次运行调度进程，都要处理定时器队列。定时器 Bottom Half控制程序可处理这两种系统定时器，对于老的系统定时器，它先检测 timer_active屏蔽码得到当前活跃的定时器，然后检测当前活跃的定时器是否超时，若是，则调用该定时器例程并把相应 timer_active中的位清零；对于新的系统

定时器，先检测timer_list链表数据中的入口，然后调用超时的定时器例程，并从链表中删除该项。新的定时器机制有一项优势：它允许向定时器例程传送一个参数。

9.1.4 等待队列

很多情况下处理器因等待某种系统资源而无法继续运行，例如：处理器需要一个描述目录的VFS索引节点，但该索引节点当前不在内存缓冲区中，这样处理器就必须先等到索引节点从磁盘中读到内存之后，才能继续运行。

对于这种等待的处理，Linux内核使用了一种简单的数据结构——等待队列(见图1-9-4)，其中包括一个指向task_struct的指针和一个指向队列中下一元素的指针。

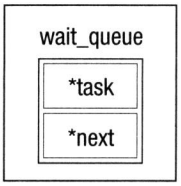

图1-9-4 等待队列

加入到等待队列中的进程可以是可中断，也可以是不可中断的。可中断进程在有定时器超时或等待信号的进程接收到了信号等事件发生时，可以被中断。通过进程状态参数值 (INTERRUPTIBLE或UNINTERRUPTIBLE)可判明该进程类型，由于调度进程中断了可中断进程的执行，而运行了另一进程，这样可中断进程将被挂起。

处理等待队列时，队列中每一个进程的状态都被置为 RUNNING，这时从运行队列中被移出的进程将重新进入运行队列，下次调度进程运行时，由于等待队列中的进程不需再等待了，因此它们将可以被调度运行。当处于等待队列中的一个进程准备运行时，首先要把自己从等待队列中移出。由于可以用等待队列实现对系统资源的并发访问，因此 Linux中用它来实现了信号量机制。

9.1.5 自旋锁

这是用于保护一个数据结构或一段代码的比较原始的方法，它限制了在一段时间内只允许有一个进程访问一块临界区。Linux中以一个整数域作为锁，来限制对数据结构的访问。每个要访问这些资源的进程必须首先将锁值由 0改为1，若锁的当前值已经是 1，则进程利用一个循环反复尝试对该锁的访问修改，直到成功。对内存区中锁的访问必须是原子操作：读锁值、检测锁值以及修改锁值的操作是不能被中断的。大多数 CPU体系结构是通过一条特殊的指令实现自旋锁(buzz lock)的，但也可以利用非缓存(uncached)主存来实现。

当占有临界区资源的进程使用完该资源之后，必须将锁值重置为 0，此时其他所有循环等待该锁的进程均可检测到其值为 0，但只有第一个检测到的进程有机会将锁值改为 1并访问临界区资源。

9.1.6 信号量

使用信号量(semaphore)是用来保护代码或数据结构的临界区资源。请不要忘记，只有核态下运行的进程才可以访问诸如描述文件目录的 VFS索引节点这样的临界区资源。如果允许一个进程去修改另一个进程正在使用的临界区资源，将是十分危险的，自旋锁可作为解决这种问题的一种方法，但自旋锁过于简单，使用这种方法会影响系统性能。在 Linux中，是用信号量来实现对临界区资源的并发访问的，未得到该资源的进程等待该资源被释放，并且这些等待进程将被挂起，其他进程照常运行。

Linux信号量数据结构包含如下一些信息：

count	跟踪记录要使用该资源的进程个数。其值大于零时，表示资源可用；值小于或等于零时，要使用该资源的进程必须等待。它的初始值为 1，这意味着同一时刻只有一个进程可以使用资源。希望使用该资源的进程将其值减1，用完后将其值加1
waking	正在等待该资源的进程数
wait queue	等待该资源的进程均放入此队列
lock	访问waking域所使用的自旋锁

假定一个信号量的count初始值为1，第一个访问该资源的进程将发现 count值大于0，于是将其减1，此时count值为0，这个进程现在拥有了受信号量保护的临界区资源；当进程访问完了，再把count值加1，此时最优的情况是没有其他进程要取得该资源的拥有权， Linux对信号量的实现在这种最优但也是最常见的情况下可以高效工作。

如果有另一个进程要访问已被占用的临界区资源，它也要把该信号量的 count值减1，这样count值已小于0(－1)，它已无法访问，只能等待资源被释放。Linux把等待进程置为睡眠状态，直到占用资源的进程释放掉资源再将其唤醒。等待进程会把自己放入信号量的等待队列中，然后循环检测waking域的值并调用调度进程，直到waking域变为非零为止。

而拥有临界资源的进程释放资源时，先检测 count值以得知是否有进程在等待该资源，然后把count值加1。在最优的情况下，此时 count值又回复到了初始值 1，资源拥有者进程把waking域值加1并唤醒信号量等待队列中的进程；被唤醒的进程由 waking值为1得知现在该资源可用，这样它将把 waking值减1，然后顺次执行。 Linux通过信号量中的lock域利用自旋锁机制实现了对waking域的访问保护。

9.2 模块

本节讲述Linux内核在需要的时候，是如何动态载入像文件系统这样的函数的。

Linux是一个单内核操作系统，也就是说它是一个独立的大程序，其所有的内核功能构件均可访问任一个内部数据结构和例程。对于这样的操作系统，一种选择是采用微内核结构，内核被分为若干独立的单位，各单位之间通过严格的通信机制相互访问。这样的话，若要向内核中加入新的构件，就必须利用设置进程，例如想加入一个未建立到内核中的 NCR 810 SCSI的设备驱动程序，就必须用设置进程重建一个新的内核；而在 Linux中可针对用户需要，动态地载入和卸载操作系统构件。 Linux模块是一些代码的集成，可以在启动系统后动态链接到内核的任一部分，当不再需要这些模块时，又可随时断开链接并将其删除。 Linux内核模块通常是一些设备驱动程序、伪设备驱动程序 (如网络驱动程序)或文件系统。

对于Linux的内核模块，可以用insmod或rmmod命令显式地载入或卸载，或是由内核在需要时调用内核守护程序(kerneld)进行载入和卸载。进行动态载入工作的代码非常有效，它将最小化内核大小并增加内核灵活性。当调试一个新内核时，模块也非常有用，通过对它的动态载入即可省去每次的重建和重启内核工作。当然，有利必有弊，使用模块将降低一些系统性能并消耗一部分内存空间，因为载入模块额外多出一些代码和数据结构，并会间接地降低访问内核资源的效率。

一旦Linux模块载入后，就与内核其他部分没什么区别了，它会拥有同样的权利和义务，换句话说，它也能像核心代码或设备驱动程序一样使内核崩溃。

为了使用内核资源，模块必须要先找到资源。假如一个模块希望调用内核内存分配例程kmalloc( )，由于模块创建时并不知道kmalloc( )在内存中的何处，所以在模块载入时，内核必须先调整该模块对kmalloc( )的引用，否则该模块无法正常工作。内核中维护了一张所有内核资源的符号表，因此它可以在模块载入时解决载入模块对内核资源的引用的问题。 Linux允许模块的栈操作，由此一个模块即可以使用其他模块所提供的服务。例如， VFAT文件系统模块就需要使用FAT文件系统提供的服务，因为 VFAT文件系统从某种意义来讲是 FAT文件系统的扩展。一个模块对另一个模块的服务或资源的使用与其对内核服务或资源的使用非常相似，不同的只是这些服务和资源从属于另一个模块而已。每载入一个模块，内核就会修改符号表，将该模块所有的服务和资源加入进去，这样当下一个模块载入后，即可访问已载入模块的服务。

当要卸载一个模块时，内核需要知道当前该模块是否被使用，并且还要能够通知该模块它将卸载，这样它就能释放掉所申请的所有系统资源。模块卸载后，内核从符号表中删除所有该模块所提供的资源和服务。

除了崩溃内核的可能性，模块还会带来另外一种危险：载入了一个与当前系统版本不同的模块会如何？若该模块以一个错误参数调用了系统例程，就会出问题，为防止这种情况发生，内核在载入模块前，会对该模块的版本号进行严格的检测。

9.2.1 模块载入

有两种载入模块的方法：一种是用 insmod命令手工载入，另一种方法更为灵活，是在需要时自动载入，这种方法也称为需求载入，当内核发现需要载入某个模块时，它会要求内核守护程序去载入相应的模块。

内核守护程序是一个拥有超级用户权限的一般用户进程，当它启动后 (系统启动时)会打开一个指向内核的内部进程间通信 (Inter-Process Communication, IPC)通道，内核用该通道通知内核守护程序进行各种操作。内核守护程序的主要工作是载入和卸载模块，它也做其他一些任务，如打开和关闭使用电话线的 PPP连接。内核守护程序并非亲自做这些工作，而是调用相应的程序(如insmod)来完成，它只是一个内核代理，自动地安排调度各项工作。

insmod工具在加载之前，先要打开欲加载的内核模块，需要时才被加载的模块保存在/lib/modules/kernel-version中，内核模块其实是一些链接的对象文件，与系统中其他的程序是一样的，只不过它们是作为重分配的映象被链接的，也就是说，它们并非从一特定地址开始运行。内核模块可以是 a.out或elf格式的对象文件， insmod要进行一次系统调用来查找内核导出符号，这些符号保存在符号名值对中。内核维护了一张模块表，模块表中第一个模块的数据结构内有内核导出符号表， module_list指针指向该符号表。只有特定的符号被加入到符号表中，并在编译和链接内核时创建，并非所有内核中的符号都导出到其模块上。“ request_irq ”就是一个符号，当一个驱动程序希望控制特定的系统中断时，就要调用该内核例程，通过查看/proc/ksyms文件或使用ksyms工具可以很方便地看到导出内核符号的名和值， ksyms工具能显示出所有的导出内核符号或仅仅被已载入模块所导出的符号。 insmod将模块读入虚存中，然后利用内核中的导出符号解决模块对内核进程的引用问题，解决方法是在内存中对模块映像进行修补：fnsmod把符号地址物理地写入模块的相应位置中。

解决了模块对导出内核符号的引用的问题之后， insmod通过系统调用为新的内核申请足

够的空间，内核即为该模块分配一个新的模块数据结构和足够的核心内存空间，并将其加到内核模块表尾。图1-9-5示出了载入两个模块(VFAT和FAT)后的内核模块表，图中未给出模块表中的第一个模块，该模块是一个伪模块，只用于保存内核导出符号表。可以用 lsmod命令列出所有已载入的模块及其相互的依赖关系，lsmod只是简单地重新组织一下/proc/modules文件，该文件是通过内核模块数据结构表创建的，它在内存中的地址被映射到了 insmod进程的地址空间中，便于该进程对它的访问。insmod把模块复制到为其分配的空间中，并重定位该模块，这样它就可以从内核地址上开始运行了。若一个模块在一个系统中需要载入两次时，为避免其两次载入拥有同一个地址，这种处理是必须的，由于这种重定位，则须用相应的地址对模块映像进行修补。

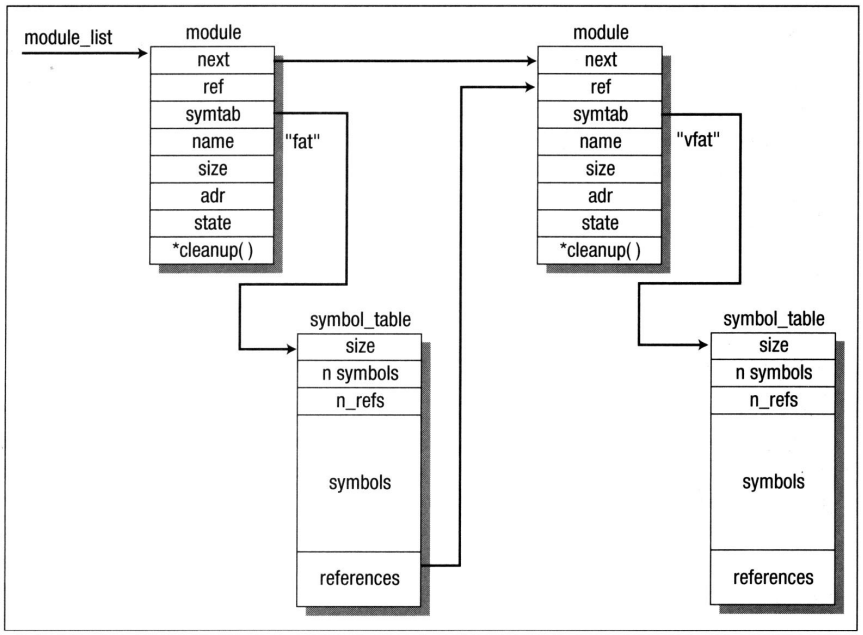

图1-9-5 内核模块表

新模块也要向内核导出符号，insmod将会为这些模块映像建立一张表，每一个内核模块必须包括模块初始化和清除例程，对于未导出符号的模块，insmod必须知道其地址以便告知内核。若一切正常，insmod即开始初始化模块，并通过系统调用将模块的初始化和清除例程地址交给内核。

新模块加入内核后，必须刷新内核字符并修改正在被其使用的模块。被其他模块引用的模块应维护一张引用表，该表在它们的符号表尾部，并可通过 module数据结构中的指针访问该表。从图1-9-5中可看出VFAT文件系统模块需要使用FAT文件系统模块，因此FAT模块中包含有一个对VFAT模块的引用，该引用是在载入 VFAT模块载入后加上的。内核会调用模块的初始化例程，若调用成功，将继续安装该模块，模块的清除例程地址保存在其 module数据结构中，核心卸载模块时将调用该例程，最后，置模块状态为 RUNNING。

9.2.2 模块卸载

可以用rmmod命令卸载模块，但对于需要时载入类型的模块，当其不再需要时，会由

kerneld自动将其从系统中删除。每次空闲定时器超时， kerneld都会利用系统调用来要求将所有当前未被使用的需要时载入类型的模块删除，定时器的值是在启动 kerneld时设定的。例如，如果对一个已mount的iso9660 CD ROM进行了unmount操作，则iso 9660模块不久即会被删除。

当一个模块正被其他内核构件所使用，则不能将其卸载。例如，当 mount了一个或多个VFAT文件系统之后，是无法卸载VFAT模块的，看一下lsmod的输入，会发现每一个模块有一与其相关的计数，例如：

```
Module:         #pages:  Used by:
msdos               5                 1
vfat                4                 1 (autoclean)
fat                 6    [vfat msdos] 2 (autoclean)
```

该计数是针对使用该模块的内核实体的。上例中， vfat和msdos模块都要使用fat模块，因此fat模块的计数是2；由于各有一个mount的文件系统使用vfat和msdos，由此它们的计数是1，若再载入一个VFAT文件系统， vfat模块的计数就变成2。模块在其映像的第1个longword中保存其计数值。

计数域是轻度重载的，因为该域也用以保存 AUTOCLEAN和VISITED标志，这两种标志都是指需要时载入类型的模块。有其他系统组件使用某个模块时，其标志就为 VISITED，每当kerneld通知系统删除不再使用的模块时，系统都要搜索所有模块以找到可能的删除模块，这样的查找结果是那状态为 RUNNING并且标志为 AUTOCLEAN的模块，这些模块中VISITED标志已清除的模块被删除，其他的模块则清除它们的 VISITED标志。

假定一个模块可被卸载，则会调用它的清除例程以释放其所占用的内核资源，它的module数据结构标记为DELETED并从内核模块链中将其删除，其他所有被该模块使用的模块都要修改它们的引用表，表明不再被它使用，还要释放掉为它分配的内存。