Linux 核心（The Linux Kernel）

（中）

"First, they ignored us; then they laughed at us; then they fighted us; then we win."

--From 1st Linux Conference

原著：David A Rusling

编译：陈怀临等

第 4 章 进程

本章讲述什么是进程， 以及 Linux 核心是如何创建，管理和清除系统中的进程的.

在操作系统中， 进程是任务的执行者。 程序只是存贮在盘上的可执行映像里面的机器指令和数据的集合， 因此是被动的实体。 进程可以被看作正在运行的计算机程序。

进程是一个动态实体， 随着处理器执行着机器指令而不断变化。 除了程序中的指令和数据之外， 进程中还包括了程序计数器， CPU 的所有寄存器， 堆栈(包含着象过程参数，返回地址，保存的变量等临时数据)。 当前正在执行的程序， 也就是进程， 含有微处理器当前的所有活动。 Linux 是一个多重处理型的操作系统(multiprocessing， 或叫做多道)。

进程各司其职， 如果某个进程崩溃， 不会导致系统中别的进程崩溃。 每个进程在独立的虚拟地址空间中运行， 除非通过核心提供的安全的机制之外， 不能和别的进程相互作用。

进程在其生命周期内要使用许多系统资源， 它要用 CPU 运行指令， 用物理内存存贮指令和数据； 它会打开并使用文件系统中的文件， 直接或间接使用物理设备。 Linux 必须了解进程使用资源的情况以便合理地管理系统中的所有进程。 假如让某个进程独占大部份系统物理内存或者 CPU， 对别的进程就不公平。

系统中最重要的资源是 CPU， 通常只有一个。 作为一个多重处理操作系统， Linux 的目标是让系统中的每个 CPU 上面始终有一个进程在执行， 以充份利用 CPU。 如果进程数多于 CPU 数(通常总是这样)， 多余的进程必须等待有 CPU 空闲下来才能运行。

多重处理的想法很简单： 让进程一直执行直到它必须等待， 通常是等待使用一些系统资源； 当它可以使用这个资源时， 可以再让它运行。 在一个单一处理的操作系统(或叫做单道)中， 例如 DOS， CPU 在进然 b 等待资源的时候将无所事事，白白浪费时间。 在一个多重处理操作系统中， 内存中同时存在许多进程。 每当一个进程必须等待， 操作系统就把 CPU 分配给别的需要运行的进程。 系统中专门有一个调度器(scheduler)负责选出下一个要运行的进程。 Linux 使用很多调度策略来保证调度的公平。

Linux 支持很多不同的可执行的文件格式, 比如 ELF ， 还有 Java。 这些格式必须被透明地管理。

4.1 Linux 进程

Linux 系统为了管理进程，用 task_struct 数据结构表示每个进程 (任务和进程(task and process) 在 Linux 中是可以互换使用的术语)。 任务向量(task vector)是一个指针数组， 里面的指针指向系统中的每个 task_struct 数据结构。

这样就意味着系统中的最大进程数受到任务向量的大小的限制； 缺省它有 512 个入口。当创建新进程时， 新的 task_struct 从系统存储器中被分配出来并被加入任务向量。 为了便于查找， 一个 current 指针指向当前的进程。

除了普通进程， Linux 还支持实时进程。 所谓实时是指这些进程必须能够快速响应外部的事件。 调度器会区别对待实时进程和普通进程。 尽管 task_struct 数据结构相当大， 而且很复杂， 但是其中能够划分出很多功能区域：

State (状态)

进程执行时会根据不同的情形改变状态。 Linux 进程有下列状态：

1。Running 运行态

进程或者正在运行(它是系统的当前进程)， 或者是准备运行的(它正在等待被分到系统的 CPU 之一) 。

2。Waiting 等待态

进程正在等一个事件或一个资源。 Linux 中的等待态有性质不同的两种类型： interruptible (可中断的)和 uninterruptible (不可中断的)。 可中断的等待进程能被信号打断而不可中断的等待进程直接等待某种硬件条件， 在任何情形下都不能被中断。

3。Stoped 停止态

进程被停止了， 通常是通过接受一个信号的方法。 被调试的进程能处于一个停止态。

4。Zombie 僵死态

某个已经终止的进程， 由于一些原因， 仍然在任务向量中占有一个 task_struct 数据结构， 就处于僵死态。

Scheduling Information (调度信息)

调度程序需要这个信息以便相当决定系统中哪个进程最需要运行。

Identifiers (标识符)

每个进程有一个进程标识符。 进程标识符不是任务向量的一个索引， 它就是一个数字而已。 每个进程也有用户和组标识符， 它们是用来控制这个进程对系统中的文件和设备的访问的。

Inter-Process Communication (IPC, 进程间通讯)

Linux 支持 Unix 中经典的 IPC 机制， 如： signal (信号)， pipe (管道) 和 semaphore (信号灯)，并且支持 System V (Unix 的一种较流行的标准版本)中的 share memory (共享存储器)， semaphore (信号灯) 和 message queue (消息队列)。 Linux 所支持的 IPC 机制在第 IPC 章中有详细讲述。

Links (连接)

Linux 系统没有进程与别的进程完全无关。 除了初始化进程(init process)之外， 每个进程都有一个父进程(parent process)。 新进程不是被凭空创造出来的， 它们是从已有的进程拷贝得来， 或者是克隆得来的。 代表进程的每个 task_struct 中都有指针指向它的父进程， 兄弟进程(同一个父进程产生的进程之间是兄弟关系)， 以及自己的子进程。 你能使用 pstree 命令看到正在运行的进程的家庭关系， 下面是某次运行 pstree 命令得到的结果：

```
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， 系统中有一个以初始化进程的 task_struct 数据结构为根的双向链表， 把所有进程都链接在里面。 有了这样的表， Linux 核心就可以方便地查看系统中的每个进程。 这是为了支持 ps 和 kill 这样的命令(分别是列出系统中的进程的命令和向进程发送信号的命令(通常用于终止进程)).

Times and Timers (时钟和定时器)

在进程的生命周期内， 核心记录进程的创建时间并随时记录进程消耗的 CPU 时间。 每过一次时钟滴答(tick)的时间， 核心就更新当前进程在系统态和用户态所花的 CPU 时间(以 jiffy 为单位)。 Linux 也支持进程特定的间隔定时器， 进程可以使用系统调用设置定时器，当定时器所设置的时间间隔已到， 核心就会给进程发送一个信号。 这些定时器可以是一次性的或周期性地触发。

File system (文件系统)

进程可以打开和关闭文件。 进程的 task_struct 中包含了指向打开的文件的描述符(descriptor)的指针， 还有两个指向 VFS i 节点(inode)的指针。 VFS i 节点能够唯一描述文件系统中的一个文件或目录， 它也是文件系统所提供的统一的访问文件的接口。 关于 Linux 系统中怎样支持文件系统， 请参看第章文件系统。 第一个指针指向进程的根目录(进程的可执行映像文件所在的目录)， 第二个指向进程的当前目录或者叫 pwd 目录(得名于 Unix 中的 pwd 命令， 是 print working directory 之意。)。 VFS i 节点中有一个域用来记录有多少个进程指向它们。 现在你明白为什么当一个进程的 pwd 目录是你想删除的目录或者是这个目录的一个子目录的时候， 你就不能删除它的原因了吧？

Virtual memory (虚存)

大多数进程有一些虚存(核心线程和精灵(daemon)除外)， Linux 核心必须追踪虚存到系统物理内存上的映射关系。

Processor Specific Context (处理器特定的上下文)

进程可以被看作是系统的当前的各种状态的集合。 进程运行时要使用处理器的寄存器，堆栈等等。 这就是所谓的进程上下文。 当进程被挂起时(暂时不再运行)， 这个进程的 CPU 特定的上下文必须被保存到这个进程的 task_struct 中。 当进程被调度器重新启动时，它就从这里恢复它的上下文。

4.2 Identifiers 标识符

Linux 象所有的 Unix 一样， 使用用户(user)和组(group)标识符在来检查进程对系统中文件或者映像的访问权限。 Linux 系统中的文件都有所有权和许可权， 这些许可权描述了系统中的用户对那个文件有什么访问权限。 基本的许可权有读(read)， 写(write)和执行(execute)， 它们被分派到 3 类用户： 文件的主人(owner)， 属于某个特定组的所有进程，还有系统中的所有进程。 每一类用户可以有不同的许可权， 例如： 一个文件可以允许它的主人读写， 允许文件所在的组读并且不允许系统中的其它进程访问。

Linux 系统中， 使用组就能够把文件的权限分配到一组用户而不是简单地到一个用户或到所有的进程。 例如， 你可以为一个软件项目中的所有用户创建一个组， 并且只允许这个组中的用户能够读写该项目的源程序。 进程能属于若干组(缺省最多能够属于 32 个组)。每个进程的 task_struct 中有一个组向量(group_vector)来记录这些组。 只要进程所属的组中有一个具有访问权限， 这个进程就有权访问那个文件。

每个进程的 task_struct 中有 4 对用户和组的标识符：

1。 uid， gid

进程所代表的用户(也就是启动这个进程的用户)的用户标识符和组标识符。

2。 effective uid and gid (有效的 uid 和 gid)

有一些程序在执行的时候会把 uid 和 gid 改变为它们的自己的特定的某个 uid 和 gid(这些程序的可执行映像文件的 VFS i 节点中有一个属性规定了这样的行为)。 这些程序被

称为 "setuid" 程序。 它是限制系统服务(service)的权限一个方法， 尤其在实现为别的用户服务的网络精灵程序等类似的服务时很有用。 有效的 uid 和 gid 来自程序的映像文件本身，和启动它的用户无关。 核心在检查权限的时候会使用有效的 uid 和 gid.

3。 file system uid and gid (文件系统 uid 和 gid)

这两个标识符通常与有效的 uid 和 gid 一样， 当检查文件系统存取权限时会用上。这两个标识符是为了建立 NFS(Network File System， 网络文件系统)而使用的， 因为用户模式的 NFS 服务器需要像一个特别的进答 7b 一样来访问文件。 在这种情况下， 只有文件系统 uid 和 gid 被改变(有效的 uid 和 gid 不变)。 这样可以防止恶意的用户向 NFS 服务器发送 kill 信号。 Kill 信号会被以一个特别的有效 uid 和 gid 发送到进程。

4。 saved uid and gid (节省的 uid 和 gid)

这是 POSIX 标准中要求的两个标识符。 当然 b 序通过系统调用来改变 uid 和 gid 的时候必须要用它们来保存真实的 uid 和 gid 。

4.3 Scheduling 调度

进程执行时总是一会儿在用户态下， 一会儿在系统态下。 不同的硬件如何实现对这两种模式的支持不一定相同， 但是都有一种安全机制保证从用户态进入系统态然后再回到用户态。 用户态时进答 7b 的权限比较系统态要小。 每当进程进行系统调用的时候就会从用户态切换到系统态， 然后继续运行。 进入系统态之后， 核心代码开始执行， 为这个进程服务。 在 Linux 系统中， 进程不能从当前正在运行的进程那里强占执行的权利。 当执行的进程需要等待某个系统事件的时候， 它就让出 CPU 。 例如， 进答 7b 可能等待从一个文件中读出一个字符。 这个等待在系统调用内部， 处于系统态； 这时， 等待事件的的进然 b 将被核心暂停， 其它更着急的进程会被选中来运行。

进程总是要经常做系统调用所以就经常会这样等待。 尽管如此， 如果进程愿意， 它还是可以长时间地不做系统调用从而不合理地占用 CPU 的处理时间。 因此， Linux 系统要使用抢先式的调度。 在这种情况下， 每个进程被允许运行一小段时间， 比如 200ms， 如果时间到了， 核心就会暂停当前的进程， (不管它是不是愿意)， 选择别的进程来运行。这一小段时间就是所谓的 time slice (时间片)。

负责在系统中所有可以运行的进程中选择最该运行的进程的核心部份是调度器。可以运行的进程(runnable process)是指这个进程就在等待 CPU 来执行。

Linux 使用基于优先级的相当简单的调度算法在系统在当前的进程之间选择。 当选择了新进程来运行时， 它保存当前的进程的状态， 特定的处理器寄存器以及其它的上下文，到这个进程的 task_struct 数据结构中。 然后它恢复新进程的状态(这仍就是处理器相关的)， 把系统的控制交给这个进程， 开始运行它。 调度器为了能够公平地分配 CPU 时间， 它在每个进程的 task_struct 中保存了下列信息：

policy (策略)

在这个进程上使用的调度策略。 Linux 进程有两种类型， 普通和实时。 实时进程比其它所有的进程的优先级都要高。 如果有实时进程可以运行， 它将总是首先运行。 实时进程有 2 种调度策略， Round Robin (轮转式)和 First in First out (先入先出式)。在轮转式调度下， 每个 runnable 的实时进程轮流运行； 在先入先出式调度下， 每个 runnable 的进程依次运行， 次序就是它们进入运行队列式的顺序， 而且不会变化。

priority (优先级)

进程的优先级。它也是这个进程被允许运行的时间的总量(以 jiffy 为单位)。 通过系统调用和 nice 命令能够改变进程的优先级。

rt_priority (实时优先级)

实时进程的优先级高于其他类型的进程。 这个域允许调度器给每个实时进程以相对的优先级。 实时的进程的优先级可以通过系统调用来改变。

counter (计数器)

这是该进程被允许运行的时间的总量(以 jiffy 为单位)。 进程第一次贻 d 始运行时， 这个值就被设定为优先级的大小， 每次时钟中断一次， 这个数值就被减小。

核心内若干地方会运行调度器。 把当前的进程放入等待队列后会运行调度器； 在系统调用结束， 即将返回到用户态的时候， 也可能会运行。 如果系统定时器把当前的进程的 计数器减小到了零， 它也需要运行。 调度器运行时， 需要做的事情是：

kernel work (核心工作)

调度器运行 bottom half handler (一种推迟处理任务的机制)并处理调度器的任务队列。关于 bottom half handler 以及这些轻量的核心线程在 第 11 章 核心机制 中有详细讲解。

process current process (处理当前进程)

在选择其它进程运行之前，必须处理当前进程。 如果当前进程的调度策略是 Round Robin (轮转式)， 它就被放到运行队列的末尾，

如果任务是可以被中断的(INTERRUPTIBLE)， 并且自从最后一次调度它之后， 它收到了一个信号， 那么就设置它的状态为 RUNNING(运行)。

如果当前的进程执行超时了，那么它的状态变为 RUNING。

如果当前的进程就是 RUNNING，它将保持这个状态。

不处于 RUNNING 态并且也不是 INTERRUPTIBLE 的进程就被移出运行队列。 这意味着当调度器要寻找最需要运行的进程时， 不再会考虑它们。

Process selection (进程选择)

调度器查找整个运行队列来选择最需要运行的进程。 如果有实时进程(调度策略是实时的那些)， 它们就会获得比普通进程更高的权重量。 正常的进程的权重是它的 counter (或者优先级)， 而实时进程是 counter 加 1000。 这说明如果系统中有处于 runnable 状态的实时进程， 它们就会比普通的 runnable 的进程先执行。 当前的进程， 因为已经执行了一段时间， 经过了若干时间片， 它的 counter 就被减去了一些， 所以如果有同样优先级的进答 7b 的话， 它就要让位了。 这正是需要的。 如果若干进程有同样的优先级， 在队列前面的被先选中。 当前进程会被放到队列的最后。 在有许多优先级相同的进程的平衡的系统中，它们会被轮流运行。 这就是称为 Round Robin 的调度方案。然而， 进程会等待资源， 它们的运行顺序就会发生变化。

Swap processes (交换进程)

如果最需要运行的进程不是当前进程， 当前进程就必须被暂停， 新的进程将取而代之。 进程在运行时， 它在使用 CPU 的寄存器和系统的物理内存。 调用过程时， 它用寄存器传递参数， 并且可能需要把返回地址放在堆栈中。 因此， 当调度器运行时它是在当前进程的上下文(Context)中。 这时， CPU 处于特权态下， 也即核心态， 但是正在运行的仍然是当前进程。 如果要暂停它， 就必须把它的上下文保存进它的 task_struct 数据结构中。 然后， 新进程的机器状态的必须被装载。 这是和具体的系统相关的操作， 各种 CPU 的做法很不相同， 但是通常有一些硬件辅助来做这件事。

进程上下文的切换在调度器运行结束时进行。 所切换的上下文是与被调度进程有关的硬件环境在此时的一个快照。

如果刚才的进程或新的当前进程使用虚存， 系统的页表的项目可以需要更新。 同样地，这是和特定的机器体系结构相关的。 象 Alpha AXP 这样的处理器， 使用 Look-aside Tables (转换对照表)或者 cached Page Table Entries (缓冲页表项)， 必须刷新那些属于先前进程的表项。

4.3.1 多处理机系统中的调度

多个 CPU 的系统在 Linux 世界中是相当稀罕的， 但是 Linux 系统中已经做了很多工作使其成为一个 SMP(Symetric Multi Processing 对称多处理) 操作系统。 那就是说，有能力在系统的 CPU 之间平衡工作。 在调度器中做这种工作是最合适的。

多处理器系统中， 理想的情况是所有处理器均忙于运行进程。 每当一个 CPU 的当前的进程用尽它的时间片或必须等一个系统资源， 就会单独运行调度程序。 关于一个 SMP 统要注意的第一事情是系统中不止存在一个空闲的进程。 在单处理器系统中空闲的进程是在任务向量的第一任务， 在一个 SMP 系统中每个 CPU 都有一空闲的进程， 并且你可能有不止一个空闲的 CPU 。另外每个 CPU 有一个当前进程， 因此 SMP 系统必须追踪每个处理器上的当前进程和空闲进程。

SMP 系统中每个进程的 task_struct 包含它当前正运行在上面的处理器的标识以及上次运行在上面的处理器的标识。 虽然进程可以每次在不同的 CPU 上面运行， Linux 可以使用 processor_mask 来限制进程可以使用的 CPU 。 如果 processor_mask 的第 N 位被设置， 这个进程就能在处理器 N 上运行。 当调度器选择新进程， 它不会选择 processor_mask 中和当前的处理器对应的位被清除的进程。 调度器会略微照顾上次在这个处理器上面运行的进程， 因为把进程在不同的处理器之间移动通常会带来一定的性能损失。

4.4 Files (文件)

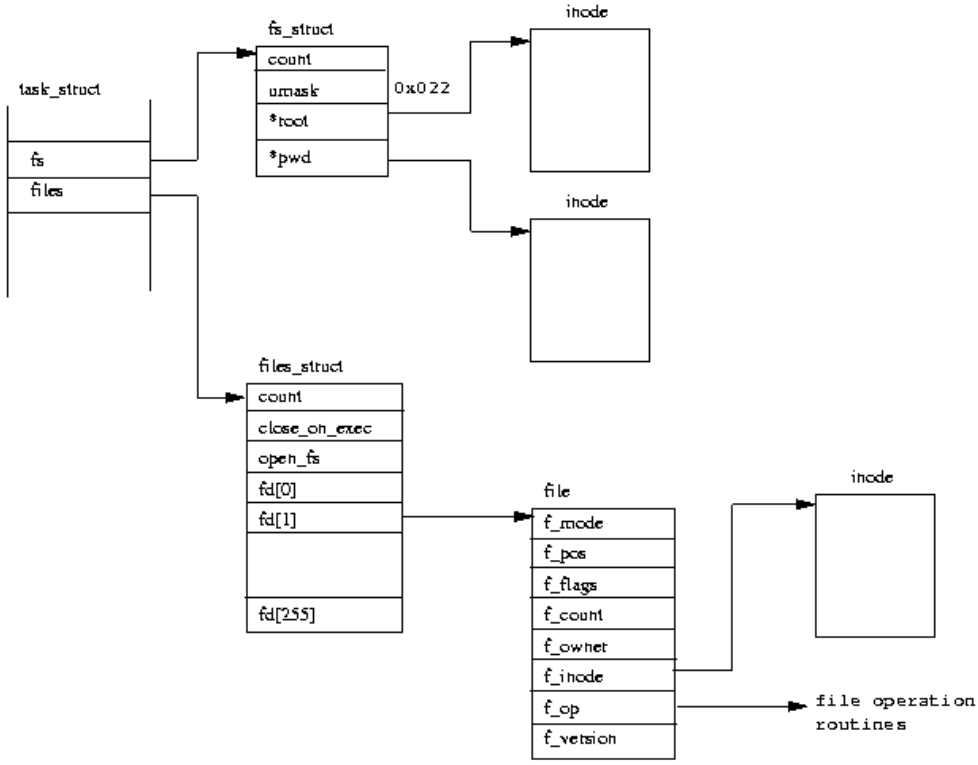

图 4.1 ：进程的文件

图 4.1 表明系统中的每个进程有 2 个数据结构描述文件系统相关的信息。

第一, fs_struct， 包含指针指向进程的 VFS i 节点 和它的 umask 。 umask 是创建新文件时使用的缺省模式， 可以用系统调用改变。

第二, files_struct， 包含进程当前正在使用的所有文件的信息。 然 b 序从 standard input (标准输入) 读并且写到 standard output (标准输出)。 任何错误消息应该输出到 standard error (标准错误)。 这些可以是文件， 终端输入/输出或一台真实的设备， 但是程序都把它们当作文件。 每个文件有它的自己的 descriptor (描述符)， files_struct 中包含可以指向 256 个文件数据结构的指针， 每个可以描述进程打开的一个文件。f_mode 描述文件是以什么模式被创建的：只读， 读写 或者 只写。 f_pos 记录下一个读或写操作的位置。 f_inode 指向描述该文件的 VFS i 节点， 而 f_ops 是一个指向例程地址的向量的指针， 每一个例程实现你希望在文件上做的一个操作， 例如， 一个写数据的例程。 这种对界面的抽象非常有用， 允许 Linux 系统支持各种各样的文件类型。 我们以后就会看到， Linux 中的 pipe (管道) 就是用这个机制实现的。

每打开一个文件， 在 files_struct 的一个空闲的文件指针被用来指向新文件结构。 Linux 进程启动的时候， 会有 3 个文件描述符已经打开， 它们是标准输入， 标准输出和标准错误， 通常都是从父进程中继承来的。 所有的文件访问都要使用系统调用， 它们使用或者

返回 file descriptor (文件描述符)。 文件描述符是到进程的 fd 向量的索引， 所以标准输入， 标准输出和标准错误的文件描述符是 0 ，1 和 2 。 文件的每次访问都要使用文件数据结构的文件操作例程和 VFS i 节点。

4.5 虚存

进程的虚存包含从许多来源来的可执行的代码和数据。

首先， 程序映像被装载。 例如 象 ls 一样的命令。 这个命令, 象所有的可执行的映像一样， 都由可执行代码和数据组成。 映像文件包含装载可执行的代码以及有关的程序数据到进程的虚存所需的全部信息。

第二， 进程运行时能分配(虚拟)存储器， 比如说保留它正在读的文件的内容。 这最新分配的， 虚拟的存储器要被连接进进程的已有的虚存才能使用。

第三， Linux 进程通常使用的公用代码库, 例如处理文件的例程。 每个进程有库的自己的拷贝， 这很不明智。 Linux 使用能同时被若干运行的进答 7b 使用的共享库。 共享库的代码和数据必须被连接到共享这个库的多个进程的虚拟地址空间。

在任何给定的时间段内， 进程不会使用在它的虚存中包含的所有代码和全部数据。 它可以包含仅仅在某些状况下被使用的代码， 例如在初始化期间或一个特别的事件发生时。 它可能仅仅使用了从共享库连接的一些例答 7b。 装载这些无用的东西进物理存储器， 实在是一种浪费。 考虑到系统中同时存在多个进程， 这将使系统很低效地运行。 为此, Linux 使用 demand paging (请求换页) 技术， 仅仅当进程试图访问某页时， 才把它装入物理内存。 因此， Linux 核心只要改变进程的页表， 把虚拟的空间标明为存在但是不在内存中就行了， 而不需要直接装载代码和数据进物理存储器。 当进程尝试访问这里的代码或数据时， 系统硬件将产生 page fault (页错) 并且把控制传递给 Linux 核心来处理。 因此， Linux 核心需要知道进程的虚拟地址空间的各个区域是从何处来的以及如何把它装入内存， 这样才能处理 page fault。

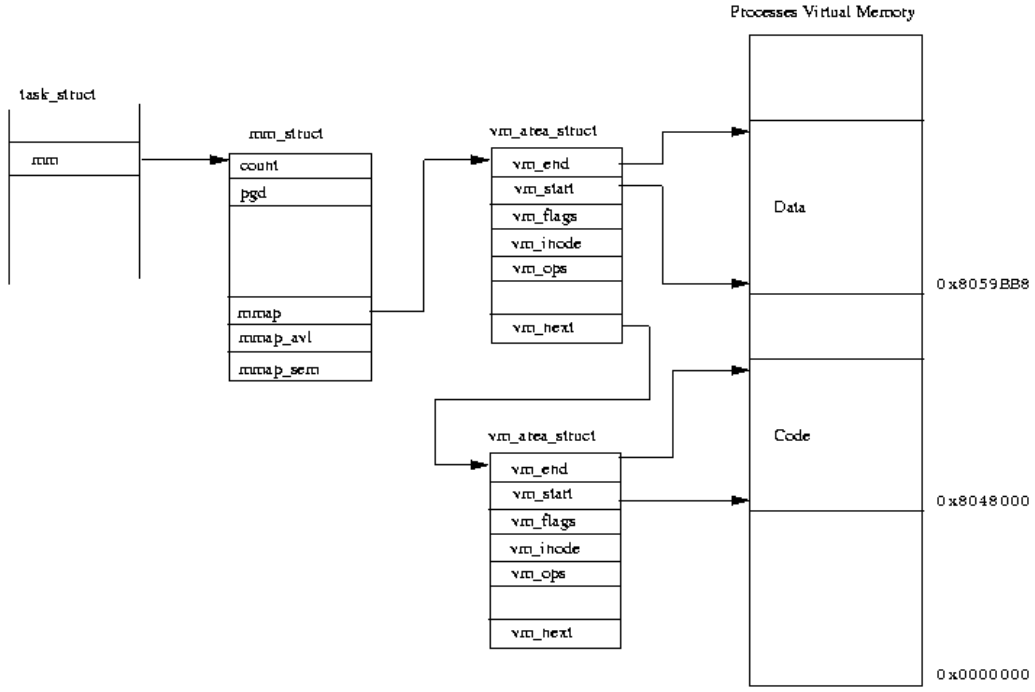

图 4.2 ： 进程的虚存

Linux 核心需要管理虚存的所有这些区域。 进程的虚存的内容在 mm_struct 数据结构中描述， 进程的 task_struct 有指针指向这个结构。 进程的 mm_struct 数据结构也包含已装载的可执行的映像的信息， 还有到进程的页表的指针。 进程的页表包含一些指针， 指到 vm_area_struct 数据结构的一个表， 每个表示进程虚存的一个区域。

这张链接的表是按虚存地址升序链接的， 图 4.2 显示了一个简单进程的虚存的布局以及管理它的核心数据结构。 因为虚存中的那些区域从若干来源， Linux 让 vm_area_struct 指向一套处理虚存的抽象接口的例程(经由 vm_ops)。 这样不管管理那存储器的内在的服务怎么不同， 进程的所有虚存都能用一致的方法处理。 例如有一个例程在进程试图存取存储器并且它不存在时， 将被调用， 这就是用来处理 page fault 的。

进程的 vm_area_struct 数据结构的会被 Linux 核心很频繁地调用。 这就使得寻找到 vm_area_struct 结构的时间对系统性能影响很大。为了加快存取， Linux 另外把 vm_area_struct 数据结构排列成一个 AVL (Adelson-Velskii 和 Landis)树 (也称平衡树)。 这棵树上， 每个 vm_area_struct (或节点) 有一左一右两个指针指到它的邻近的 vm_area_struct 结构。 左指针指向的节点虚拟地址小于右指针指向的节点。寻找正确的节点时， Linux 从树根开始， 根据每个节点的左右指针指向的地址的大小关系决定向何处去找， 直到找到为止。 当然, 没有免费的午餐， 把一个新的 vm_area_struct 插入到这棵树要花一些额外的处理时间。

当进程分配虚存时， Linux 实际上不为进程保留物理存储器。 相反, 它创建新的 vm_area_struct 数据结构描述虚存， 再连接进进程的虚存的表。 当进程试图在那个新虚存区域以内写时，系统将发生 page fault (页错)。 处理器将试图进行虚拟地冶 d 译码， 但

是因为这块存储器的没有页表入口， 它将失败并引发 page fault 异常， 让 Linux 核心来处理。 Linux 检查引用的虚拟的地址是否在当前的进程的虚拟的地址空间。 如果是， Linux 创造适当的 PTEs 并且为这个进程的分配物理存储器的一页。 代码或数据可能需要从文件系统或从交换磁盘拷贝到那个物理页。 进程然后在引起了 page fault 的指令处被重启并且, 这次因为存储器物理上存在， 它可以继续运行。

4.6 创建进程

当系统启动时， 它在核心态运行并且有，从某种意义上说， 仅仅一个进程， initial process (初始进程)。 象所有的进程一样， 初始进程的机器状态由堆栈， 寄存器等等表示。 当系统的另外的进程被创建并运行时， 这些将在初始进程的 task_struct 数据结构被保存。 系统初始化结束时， 初始进程启动一个核心线程(叫 init) 然后进入一个无事可做的空闲循环。 当没有别的事情做时， 调度器将运行这个空闲进程。 空闲进程的 task_struct 是唯一一个不被动态地被分配的， 当构造核心的时候， 它就静态地在核心里面定义并且被叫做 init_task， 相当含糊。

Init 核心线程或进程的进程标识符为 1， 是系统的第一个真正的进程。 它做一些系统初始化设置工作(例如打开系统控制台， 安装根文件系统)然后运行系统初始化程序。 这个程序是 /etc/init, /bin/init 或者 /sbin/init, 与你的系统有关。 init 程序使用 /etc/inittab 作为脚本文件来创建系统中的新进程。 这些新进程可能还要再创建新进程。 例如， 当用户试图登录时， getty 进程可能会创建 login 进程。 所有这些进程都是 init 核心线程的后代。

新进程通过克隆旧进程，或克隆当前进程来创建。 一个新任务通过系统调用(fork 或 clone）来创建。 克隆在核心态由核心来完成。 在系统调用结束时如果调度器选择了新进程，新进程就可以运行了。 新的 task_struct 数据结构在系统物理内存中分配， 而且有一页或多页物理内存页被用来作为克隆进程的堆栈(用户堆栈和核心堆栈)。 新的进程标识符被创建， 它在系统内唯一。 但是有理由让克隆出来的进程记住它的父进程。 新的 task_struct 被加入 task vector (任务向量)， 老进程的 task_struct 的内容被复制到克隆的进程的 task_struct.

当克隆进程时， Linux 允许两个进程共享资源而不是各自复制一份。 这包括进程的文件， 信号处理程序， 以及虚存。 当资源被共享时， 各自的计数域将被增加， 这样当两个进程全部释放资源的时候 Linux 才会回收它。

克隆进程的虚存比较困难。 新的 vm_area_struct 数据结构集合要被创建， 还有它们所拥有的 mm_struct 数据结构， 以及被克隆的进程的页表。 这时还没有进程的虚存的内容被复制。 这可能是个很困难的工作因为有的虚存在物理内存， 有的在可执行映像里， 有的在交换文件里。 为此， Linux Linux 使用称为 "copy on write" (写时复制) 的技术， 具体做法是当其中一个进程试图写共享虚存时才进行复制。 实现的方法是把可写的内存区域在页表中标为 "read only" (只读)， 在 vm_area_struct 数据结构中标为 "copy on write "。 当某个进程试图写时， 就会发生 page fault， 此时 Linux 就进行内存的复制， 并修改页表和虚存的数据结构。

4.7 时间和定时器

在进程的生命周期内， 核心记录进程的创建时间并随时记录进程消耗的 CPU 时间。 每过一次时钟滴答(tick)的时间， 核心就更新当前进程在系统态和用户态所花的 CPU 时间(以

jiffy 为单位)。 除了这些用于记账的定时器之外， Linux 也支持进然 b 特定的间隔定时器，当定时器所设置的时间间隔一到， 核心就会给进程发送信号。 有 3 种间隔定时器：

Real (实时)

定时器实时地走动。 当定时器到时， 进程会收到一个 SIGALRM 信号。

Virtual (虚拟)

当进程正在运行时定时器才走。 如果到时， 这个定时器会发送一个 SIGVTALRM 信号给进然 b。

Profile (活动总计)

当进程正在运行时或者当系统代表进程在执行时， 这个定时器就走动。 它会发送 SIGPROF 信号。

Linux 系统把间隔定时器的信息存放在进程的 task_struct 数据结构中。通过系统调用能够添加定时器， 启动， 停止以及读取定时器的当前的时间。

每当系统的时钟的一次滴答到来， 当前进程的所有间隔定时器的计数值就被减少， 如果时间间隔已到， 就会发送相应的信号给进程。

实时间隔定时器有点特别。 Linux 在核心中使用了定时器机制来处理它。 每个进程有自己的 timer_list 数据结构， 当实时间隔定时器运行时， 系统的 timer list (定时器列表)中把它排入了队列。 当定时器的时间间隔一到， 负责处理定时器事件的 bottom half handler 会把它从队列中删除， 然后调用调用间隔定时器的处理器(并不是 CPU, 而是一段代码)。 这个处理器这就产生了 SIGALRM 信号并且重启间隔定时器， 又把它加入系统定时器队列。 请参看 第 11 章 核心机制 中的具体讲解。

4.8 Executing programs 执行程序

象 Unix 系统一样， Linux 系统中的程序和命令通常是由一个命令解释器来执行的。 一个命令解释器是一个用户进程， 一般被称为 shell ， 因为它就象是系统的外壳， 被用户直接感受到。

Linux 系统中有许多命令解释器， 最流行的一些是 sh， bash 和 tcsh 。 除了一些内部命令之外， 例如 cd 和 pwd ， 一个命令就是一个可执行的二进制的文件。 对每个输入的命令， 命令解释器在进程的搜索路径中指定的目录中查找能够匹配的可执行的映像文件。 搜索路径由 PATH 环境变量定义。 如果找到了匹配的文件， 它就被装载执行。

命令解释器使用上面说的 fork 机制克隆自己。 新的子进程用所找到的可执行的二进制映像文件的内容替换自己原先的内容， 也就是命令解释器自身。 通常命令解释器等待命令完成， 也就是等待子进程退出。 你能让命令处理器不要等待， 只要把子进程放到后台运行就可以做到。 使用 control-Z 组合键， 它会导致一个 SIGSTOP 信号被送给子进程， 让它暂停。 然后你可以用 shell 命令 bg 把它放到后台。 命令解释器向它发送一个 SIGCONT 信号让它恢复运行， 它将一直在哪儿， 直到运行结束或者它需要做终端输入或输出。

一个可执行的文件能有许多格式或甚至是一个脚本文件。 脚本文件必须被识别出来并且用适当的解释器来处理。 例如 /bin/sh 解释 shell 脚本。 可执行的目标文件中包含可执行的代码和数据， 以及足够的信息以便操作系统能够装载并运行。 Linux 系统中使用的最多的目标文件格式是 ELF (参见下面的小节)。 但是理论上， Linux 灵活到几乎能处理任何格式的目标文件。

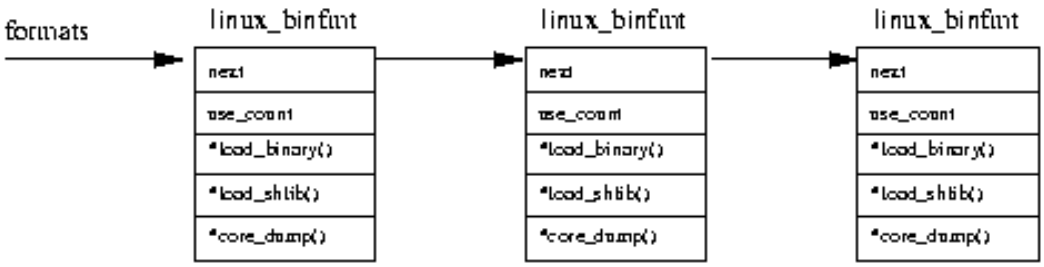

图 4.3 ：注册的二进制格式

就象文件系统， 格式由 Linux 支持了的二进制代码是在核被造了进核的任何一个造时间或可得到作为模块被装载。核坚持支持二进制的格式的一张表 (参见图 4.3) 并且当被尝试执行一个文件时，每二进制的格式接着被试用直到一个人工作。

通常被支持了的 Linux 二进制代码格式是 a.out 和 ELF 。可执行的文件不必须完全被读进存储器, 作为装载的需求被知道的技术被使用。当可执行的图象的每部份被进程使用，它被使存储器。图象的闲置的部份可以从存储器被丢弃。

4.8.1 ELF

ELF (Executable and Linkable Format) 目标文件格式， 由 Unix 系统实验室所设计，是 Linux 系统中最常用的格式。 虽然同其它的目标文件格式， 例如 ECOFF 和 a.out， 比较， ELF 在性能上略有损失， 但 ELF 更灵活。 ELF 可执行文件中包含可执行的代码(有时称为正文(text))， 还有数据。 除此之外， 还有表说明程序应该怎样被放进进程的虚存。 静态连接的映像可以用连接器(ld)构造， 或用连接编辑器， 结果成为一个包含运行时所需的全部代码和数据的单个的映像。 映像中还说明了映像在内存中的布局， 以及第一条指令在映像中的地址。

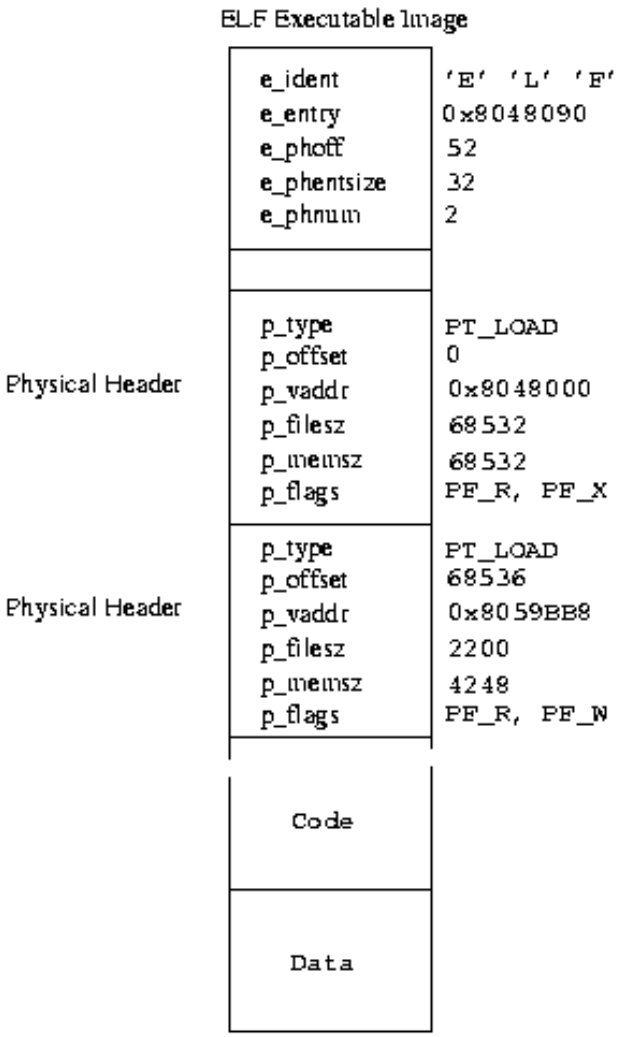

图 4.4 ： ELF 可执行文件文件格式

图 4.4 显示了一个静态连接的 ELF 可执行的映像的内部布局。

这是一个简单的 C 程序， 打印 "Hello, world!" 然后结束。 文件头说明它是一个 ELF 映像， 在文件头起始的 52 个字节是 2 个物理的头。 第一个物理头中指示在映像中的可执行的代码。 代码起始于虚地址 0x8048000 ， 有 65532 个字节。 因为它是为 printf 包含图书馆代码的所有的一幅静态地被连接了的图象，这是 () 输出"你好世界"的呼叫。 映像的入口点， 也就是程序的第一条指令， 不在映像的开始， 而是在虚地址 0x8048090 (e_entry) 处。 代码紧跟在第二物理的头之后。 这个物理头说明程序的数据， 要在装在虚地址 0x8059BB8 处。 数据是可读可写的。 你会注意到在文件中的数据块的大小是 2200 个字节(p_filesz)， 而在内存中所占的大小是 4248 个字节。 这是因为第一个 2200 个字节包含预初始化的数据而随后的 2048 个字节包含将由执行的代码来初始化的数据。

当 Linux 装载 ELF 可执行文件映像到进程的虚拟地址空间时， 它实际上没有真的装载映像。

它设置虚存数据结构， 进程的 vm_area_struct 树和它的页表。 当程序执行时， 页差错(page fault)将导致程序的代码和数据被装进物理内存。 程序中没用到的部份的部份决不会被装载进存储器。 当 ELF 二进制格式装载器检验认为这个映像确实是一个 ELF 可执行映像后， 它就从进程的虚存中刷新当前的可执行映像。 因为这个进程是一个克隆的映像(所有的进程都这样) 这旧映像就是父进程正在执行的程序。 刷新导致旧的虚存数据结构被废弃， 进程的页表被重新设置。 它也清除所有的信号 handler， 关闭已经打开的文件。 刷新过后， 进程就可以用新的可执行映像了。 不管可执行的映像是什么格式的， 进程的 mm_struct 中需要设置同样的信息。 有指向映像的代码和数据的开始和结束的指针。 这些值在读入 ELF 可执行映像的物理头时被得到， 它们所说明的程序段被映射到进程的虚拟地址空间。 此时，vm_area_struct 数据结构被设置， 进程的页表也被修改。mm_struct 数据结构中还包含指针指向传递给程序的参数以及进程的环境变量。

ELF 共享库

反之， 一个动态连接的映像， 并没有包含运行所必需的全部代码和数据。 部份代码和数据在共享库里， 当映像执行的时候会被连接进来。 这时， ELF 共享库的表也被连接进了映像。 Linux 使用若干动态的连接器， ld.so.1 ， libc.so.1 和 ld-linux.so.1 ， 都存放在 /lib 序目录下。库中包含公用的代码， 比如语言的子程序。 如果没有动态连接， 所有的程序需要把库中的这些代码各自复制一份， 这样会需要多得多的磁盘空间和虚拟内存。 有了动态连接， 每个被引用到的子程序都在 ELF 映像的表中保存了信息， 动态连接器根据这个信息知道怎样找到库中的代码并把它连接到程序的内存空间。

4.8.2 脚本文件

脚本文件是需要一个解释器来运行的可执行文件。 有各式各样的解释器可以在 Linux 中使用， 例如 wish， perl 和命令处理程序比如 tcsh 。 Linux 使用标准的 Unix 习惯，就是在脚本文件的第一行中包含解释器的名字。 因此， 一个典型的脚本文件将这样开头：

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

为了找到脚本指定的解释器， 脚本二进制代码装载器试图打开在脚本文件的第一行中指名的可执行的文件。 如果能打开它， 就让这个文件， 也就是一个解释器， 来执行这个脚本。 脚本文件的名字成为参数零(第一参数)并且所有其它的参数向后移动一个位置(原来第一参数成为新的第二参数， 依此类推)。 装入解释器的方法和 Linux 中装入一个可执行文件的方法是一样的。 Linux 试用每一种二进制格式直到某个格式能够成功为止。 这样， 从理论上，你能够安排若干个解释器以及二进制格式， 使 Linux 的二进制格式处理器变得非常灵活。

第 5 章 进程间通信的机制

进程之间、进程与核心之间互相通信，以协调它们的活动。Linux 支持一系列进程间通信机制，信号和管道是其中的两种，此外还有 SVR 的进程间通信机制。

5.1 信号

信号是 unix 系统最早使用的进程间通信方法之一。它们用来对一个或多个进程发送异步事件。信号可以由键盘中断产生，也可以由进程试图读取虚拟存储器中不存在的位置而引发。另外，信号也可以用于外壳程序向它们的子进程发送作业控制命令。有一组预先定义的信号，核心可以产生，具有相应优先权的进程也可以产生。使用 kill -l 命令可以列出系统的信号集合。例如，Intel 平台上列出：

```
1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGIOT       7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR
```

对于 Alpha 平台而言，数字又有所不同。进程可以选择忽略产生的大部份信号，除了两个特殊的之外：使进程停止运行的 SIGSTOP 和使进程退出的 SIGKILL。对此外的信号，进程可以任意选择处理的方法。进程可以阻塞信号，或者由自己的代码处理信号，或者交由核心来处理信号。如果是核心处理信号，它将进行这个信号要求的缺省处理。例如，进程收到 SIGFPE（浮点溢出）信号时，缺省处理是 core dump 并退出。信号之间没有天然的相对优先关系。如果两个信号同时为同一个进程产生，它们可以以任意顺序交给进程处理。进程也没有任何方法区别自己收到的是一个还是四十二个 SIGCONT 信号。Linux 用存储在进程的 task_struct 中的信息实现信号。所支持的信号数受到字长的限制，32 位处理器可有 32 个信号，64 位处理器就可有 64 位信号。当前未处理的信号保存在 signal 域中，blocked 中为阻塞信号的掩码。除了 SIGSTOP 和 SIGKILL 之外，一切信号都能够被阻塞。如果一个被阻塞的信号产生，除非解除其阻塞，否则它不会被处理。Linux 持有每一个进程如何处理每一个可能的信号的信息（放在每个进程的 task_struct 指向的一组 sigaction 数据结构中）。在 sigaction 之中，要么含有信号处理例程的地址，要么放置标志告诉系统：进程希望忽略这个信号，或者进程希望核心处理这个信号。进程通过系统调用来修改信号处理方法，这些系统调用把相应信号的 sigaction 或者 blocked 进行修改。并不是任何一个进程都能够发信号给所有进程，只有核心和超级用户进程才有此权力。普通进程只能发送信号给具有相同 uid 和 gid 的进程，或者同一个进程组中的进程。要产生一个信号，只要把 task_struct 中 signal 域的相应比特设置一下。如果进程没有阻塞信号，并且处于可中断的等待状态，那么它会被唤醒，转变为运行态，并确认它在运行队列中。这样，调度程序在下一次调度时将把它作为运行的候选进程之一。如果缺省处理是需要的，Linux 能够优化信号的处理。例如，如果产生了信号 SIGWINCH（X-Window 改变焦点），而缺省处理程序正在使用，那么什么也不会做。信号并非在产生后立刻送给进程，而是要等到进程重新被运行。每当一个进程从系统调用中退出，它的 signal 和 blocked 域都被检查，如果发现未被阻塞的信号，则将送给进程。这看起来似乎不太可靠，但是事实上每个进程总是在不断进行系统调用，例如把字符写到终端。如果愿意，进程可以选择等待信号，这是它处于可被信号的来临中断的挂起状态。

Linux 的信号处理代码通过查看 sigaction 结构以便决定处理方法。如果信号的处理被设置为缺省，那么核心将负责处理它。SIGSTOP 信号的缺省处理停止当前进程的运行并且使用调度程序选择下一个运行的进程。SIGFPE 信号的缺省处理使进程 core dump 并且让它退出。进程也可以选择指定自己的处理代码。该代码为一个每当信号产生时可调用的例程，sigaction 结构保存有这个例程的地址。核心必须调用进程的信号处理例程，如何实现这一点与具体的处理器相关，但是无论如何 CPU 必须注意到当前进程正处于核心态运行，并且即将返回用户态。通过对栈和寄存器的操作能解决这个问题。进程的程序计数器被设置到信号处理例程的地址，调用参数被通过调用帧或是寄存器传递。当进程得以继续时，看来似乎信号处理例程是被正常调用的。Linux 是 POSIX

兼容的，所以进程能够在信号处理例程调用时指定哪些信号被阻塞。这就意味着在信号处理例程中改变 blocked 掩码。例程结束时，blocked 掩码必须被恢复原有值。所以 Linux 增加了一个清理进程，该进程负责把原始 blocked 掩码恢复到接收信号进程的调用栈的顶端。某些情况下，几个信号处理例程需要被用堆栈方式调用，以便保证每个例程退出时，立刻调用下一个例程，直至清理例程被调用。对此，Linux 需要进行优化。

5.2 管道

普通的 Linux 外壳都允许重定向。例如

```
$ ls | pr | lpr
```

把 ls 命令的输出文件名通过管道作为 pr 命令的标准输入，后者对之进行分页（$$paginate？$$），最后 pr 的标准输出又通过管道送入 lpr 的标准输入，lpr 把结果打在缺省打印机上。所以，管道就是连接一个进程的标准输出到另外一个进程的标准输入的单向字节流。进程无法知道这个重定向，仍然正常工作。负责在进程之间建立临时管道的是外壳。

在 Linux 中，管道是通过两个指向同一个虚拟文件系统 i 节点的 file 结构来实现的，i 节点本身则指向内存中的一个物理页。图 5.1（略）显示，每个 file 数据结构含有指向不同的文件操作例程向量的指针，一个用于写管道，另一个用于读管道。这就隐藏了与读写普通文件的一般的系统调用之间的区别。当写进程在写管道时，字节被拷到共享数据页上，而当读进程在当读管道时，字节被从共享数据页上拷出来。Linux 必须对共享数据页的存取进行同步，它使用锁、等待队列和信号来保证读写进程之间的轮流。

当写进程想写管道时，它使用标准写库函数。这些库函数都传递文件描述符，而文件描述符是进程的 file 数据结构集合的索引，每一个代表一个打开的文件或者一个打开的管道。Linux 系统调用使用描述这个管道的 file 数据结构指向的例程。那个写例程使用表示管道的 i 节点中保存的信息来管理写要求。如果有足够的空间供所有的字节写入管道，只要管道没有为读进程锁住，Linux 将会为写进程锁住管道，并且把所有的待写字节从进程的地址空间拷到共享数据页中。如果管道为读进程锁住，或者没有足够的数据空间，那么将使当前进程睡眠在管道 i 节点的等待队列，调用调度程序运行另外一个进程。进程的状态是可中断的，所以它能收到信号，能在写数据空间变得足够或是管道被解锁之后被写进程唤醒。写完数据之后，管道的 i 节点被解锁，睡眠在 i 节点等待队列的读进程将被唤醒。从管道读数据与写数据非常类似。允许进程做非阻塞读（依赖于打开文件或管道的模式），在此情况下，如果没有数据可读或者管道被锁住，将返回一个错误。这意味着进程可以继续运行。另一种方法是等在管道 i 节点的等待队列里直到写进程完成工作（即阻塞读--译者注）。当两个进程都完成了管道上的工作，管道 i 节点将被与共享数据页一起丢弃。

Linux 也支持“有名”管道，也被称为 FIFO，因为管道的工作方式是先入先出的。最早写入这种管道的数据也最早被读出。与一般管道不同的是，FIFO 不是临时对象，而是文件系统中的实体，可以用 mkfifo 命令创建出来。只要进程有足够的存取权限，就能自由地使用 FIFO。打开 FIFO 的方式和打开管道的方式也稍有不同。一个管道（包括它的两个 file 数据结构，它的虚拟文件系统 i 节点和共享数据页）是一次性产生的，而 FIFO 是已经存在的，由用户负责它的打开和关闭。如果在写进程打开 FIFO 之前，读进程先打开了它，或者读进程去读一个没有被写入数据的管道，Linux 必须加以处理。除此之外，FIFO 与管道完全相同，因为它们采用的数据结构和操作是一致的。

5.3 套接字

5.3.1 SVR 的进程间通信机制

Linux 支持三类 SVR 首创的进程间通信机制：消息队列、信号灯和共享内存。这些 SVR 进程间通信机制都共用相同的认证方法。进程只能通过系统调用向核心传送一个唯一的引用标识，才能存取这些资源。这些 SVR 进程间通信对象的存取通过存取权限来控制，与文件存取的权限控制非常类似。由对象的创造者通过系统调用来设置对象的存取权限。在每一种机制之中，对象的引用标

识被用作资源表的索引。当然，索引本身并不简单，还需要一些操作来产生。所有表示 SVR 进程间通信对象的 Linux 数据结构都包含一个 ipc_perm 结构，该结构包含了所有者和创造者进程的用户和组标识。对象的存取模式（所有者、组和其它）以及对象的 key（这句似乎少了些什么，但是原文如此--译者注）。这个 key 只是用于定位对象的引用标识的一种方法。支持两类 key:公开 key 和秘密 key。如果 key 是公开的，那么系统中的任何进程，只要有足够的存取权限，都能找到对象的引用标识。SVR 进程间通信对象绝对不能用 key 来引用，而只能用引用标识来引用。

5.3.2 消息队列

消息队列允许一个或者多个进程读/写消息。Linux 维护一个消息队列表--msgque 向量，其中每一个元素指向一个 msqid_ds 数据结构，该数据结构将完整地描述消息队列。当创建一个消息队列时，从系统内存中分配出一个新的 msqid_ds 数据结构，插入向量之中。

每一个 msqid_ds 数据结构包含了一个 ipc_perm 数据结构和指向队列中消息的一批指针。另外，Linux 保存有队列修改时间，例如最后一次写队列的时间等等。msqid_ds 也包含两个等待队列，一个用于队列的写者进程，另一个用于队列的读者进程。每当进程试图写消息到写队列中时，它的有效用户标识和组标识将与队列的 ipc_perm 数据结构中的存取模式进行比较。如果进程能够写队列，那么消息将被从进程的地址空间中拷到一个 msg 数据结构中，并且把该数据结构放到消息队列的尾部。根据应用进程之间的约定，每个消息被用一个类型标记出来，这里的类型划分是与应用有关的。然而，由于 Linux 限制了能向队列中写的消息的长度和数量，队列中剩余的空间可能不足容纳这次要写的消息。这是，进程将被加入消息队列的写等待队列之中，调用调度程序来选择一个新的进程运行。当有消息从队列中读出后，写等待队列中的进程将被唤醒。读消息队列也类似。同样，需要检查进程对于写队列的存取权限。读进程可以选择读取队列中的第一个消息，而不计其类型，或者选择只读特定类型的消息。如果没有消息满足读进程的标准，那么它将被加入消息队列的读等待队列，然后运行调度程序。当一个新消息写入队列时，进程将被唤醒，重新运行。

5.3.3 信号灯

最简单类型的信号灯是内存中一个可以被一个或者多个进程测试并设置的位置。就进程而言，测试并设置操作是不可中断的，或者说是原子的。测试并设置操作的结果是信号灯当前值加上了所设置的数值，这个数值可以随便是正的或负的。根据测试并设置操作的结果，进程可能会被迫睡眠，直到另一个进程改变信号灯的值为止。信号灯可以用于实现关键区--一次只能有一个进程进入运行的关键代码区域。

例如，假设你有很多进程在同时读写一个数据文件的记录，你想对文件的存取进行严格的协调。你可以用一个初始值是 1 的信号灯，在文件操作代码的前后，放上两个信号灯操作。第一个信号灯操作是测试并且减少信号灯的值，第二个信号灯操作是测试并且增加信号灯的值。实际运行时，存取文件的第一个进程将试图减少信号灯的值，它当然会成功，这时信号灯的值变成了 0。于是进程能够继续下去，使用数据文件。这时，如果另外一个进程也想使用文件，当它试图减少信号灯的值的时候，它会失败，返回结果-1。该进程将会被挂起，直到第一个进程完成该数据文件的操作。第一个进程完成数据文件操作时，它增加信号灯的值，使之回到 1。这时等待进程可以被唤醒，它增加信号灯数值的尝试将会成功。

每一个 SVR 信号灯对象描述一个信号灯序列，Linux 使用 semid_ds 数据结构来代表之。系统中所有的 semid_ds 数据结构都被 semary 向量中的一组指针所指引。每一个信号灯序列中含有 sem_nsems 个信号灯，每一个信号灯用 sem_base 指向的一个 sem 数据结构描述。所有有权操作信号灯序列的进程可以通过系统调用来对它们进行操作。系统调用可以指定很多操作，每个操作用三个输入来描述：信号灯索引，操作值和一组标志。信号灯索引是信号灯序列中的索引，操作值是将被加到信号灯当前值上的数值。首先 Linux 测试是否所有的操作都能成功。操作能够成功当且仅当操作值加到当前值上之后结果大于 0，或者操作值与当前值都是 0。如果其中的任何信号灯操作失败，Linux 将挂起进程，除非操作标志要求系统调用是非阻塞的。如果需要挂起进程，

Linux 将保存信号灯操作的状态，并把当前进程送入等待队列。实现的方法是创立并填写一个 sem_queue 数据结构，放在信号灯对象的等待队列之中（使用 sem_pending 和 sem_pending_last 指针），并调用调度程序运行另外一个进程。（这句话是译者根据自己理解翻译的，未必确切--译者注）

如果所有的信号灯操作都成功并且当前进程不需要被挂起，那么 Linux 继续下去，对信号灯序列中适当的成员进行操作。现在 Linux 必须检查所有的等待、悬挂的进程能否进行它们的操作了。它依次看每一个信号灯操作等待队列 sem_pending，测试这些操作这一次是否会成功。如果能成功，则从队列中删除 sem_queue 数据结构，进行信号灯操作，并唤醒睡眠进程，使它在调度程序下一次运行时具备候选资格。Linux 从头检查等待队列，直到发现无法再进行任何信号灯操作，也不可能有更多进程被唤醒。信号灯还有一个死锁的问题。当一个进程进入关键区，改变信号灯的数值之后，由于瘫痪或者被杀而无法离开关键区，就会发生死锁。Linux 防止死锁的方法是维护信号灯序列的矫正表。这里的思想是用这些矫正值把信号灯恢复到操作之前的原有状态。矫正值放在 sem_undo 数据结构里，同时在信号灯序列的 semid_ds 数据结构和进程的 task_struct 数据结构之中排队。

每一个信号灯操作都要求有一个矫正值。Linux 为每个进程对每个信号灯序列的操作最多保存一个 sem_undo 数据结构。如果需要的进程没有此数据结构，则在需要时创建一个。新的 sem_undo 数据结构同时排在进程的 task_struct 数据结构和信号灯序列的 semid_ds 数据结构之中。当对信号灯序列进行操作时，操作值的相反数会被加在进程的 sem_undo 数据结构的矫正值序列中相应于这个信号灯的那个。所以，如果操作值是 2，矫正值加上的就是-2。进程被删除时，Linux 处理它们的 sem_undo 数据结构，对信号灯进行矫正。如果删除一组信号灯，那么 sem_undo 数据结构仍然排在进程的 task_struct 之中，但是信号灯序列的标识被置为无效。遇到这种情况，信号灯清除代码就只要丢弃 sem_undo 数据结构即可。

5.3.4 共享内存

共享内存允许一个或者多个进程通过同时出现在它们的虚地址空间内的内存进行通信。虚存的页面由各个进程的页表的入口指引，并不需要共享内存在每一个进程的虚拟内存的地址都相同。与所有的 SVR 进程间通信对象一样，共享内存的存取由 key 和存取权限检查来控制。一旦内存被共享，无法对进程如何使用它进行检查。必须依赖其它机制，如信号灯，来对内存的存取进行同步。

每一个新创建的共享内存区域由一个 shmid_ds 数据结构代表。这些数据接被保存在 shm_segs 向量之中。shmid_ds 数据结构描述了共享内存区域的大小，使用共享内存区域的进程的数量，和关于共享内存如何映射到进程的地址空间的信息。正是共享内存的创建者控制着其存取权限和其 key 是否公开。如果创建者具有足够的存取权限，它可以把共享内存锁定在物理内存之中。

每一个希望共享内存的进程必须通过系统调用与虚拟内存相联，从而产生一个 vm_area_struct 数据结构，为此进程描述该共享内存。进程可以选择把共享内存放在它的虚拟地址空间的何处，也可以任由 Linux 选择一块足够大的空间。新的 vm_area_struct 结构放在 shmid_ds 所指的 vm_area_struct 结构表之中。vm_next_shared 和 vm_prev_shared 指针把这些结构串联起来。虚拟内存在相联时并没有真正创建出来，而是在第一次有进程试图存取时创建出来。

当第一次有进程存取共享虚拟内存的一个页面时，发生一个缺页错误。在 Linux 处理缺页错误时，它会找到描述该虚拟内存的 vm_area_struct 数据结构，其中包含了指向处理该类共享虚拟内存的例程的指针。共享内存缺页错误处理代码查看该 shmid_ds 列出的页面对应的页表入口的表，确定是否有此页存在。如果不存在，就分配一个物理页面，在页表中为它创建一个入口。该入口不仅放入当前进程的页表之中，也放入该 shmid_ds 之中。这意味着当下一个试图存取该内存的进程得到一个缺页错误时（这里似应说明为同一虚存页面--译者注），共享内存错误处理代码将使用这个新创建的物理页面。所以，第一个存取共享内存某页使得它被创建，此后其它进程的存取使它被加入相应进程的虚拟地址空间。当进程不再想使用虚拟内存时，就与它断联。只要还有其

它进程还在使用该内存，断联就只会影响当前进程。它的 vm_area_struct 被从 shmid_ds 数据结构中删除、去配，并修改当前进程的页表，使原来使用的共享内存区域无效。当最后一个共享该内存的进程与它断联，当前在物理内存中的共享内存页面被释放，共享内存的 shmid_ds 数据结构也释放。

如果共享虚拟内存没有锁定在物理内存，就更复杂一些。这时，共享内存的页面在内存使用频繁时可以被换出到系统的交换磁盘上。共享内存如何换入和换出虚拟内存，见“内存管理”一章。

第 6 章 PCI

Peripheral Component Interconnect(PCI),外设部件连接，是一个标准，描述的是如何将一个系统中的外设以一种结构化的，可控制化的方式连接在一起。PCI 标准刻划了系统外设部件连接的电气方案，和在该标准下外设部件的行为归约。本章探讨 Linux 核心如何初始化系统的 PCI 总线和 PCI 设备。

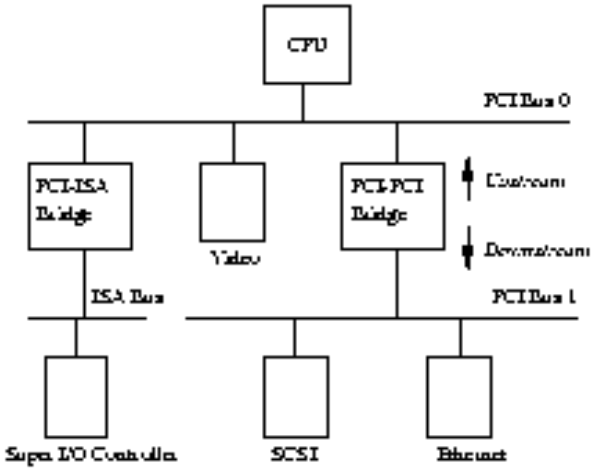

图 6.1 基于 PCI 总线的系统

图 6.1 是一个基于 PCI 总线系统的例子的逻辑框图。PCI 总线和 PCI 桥负责将系统中的部件连接在一起。CPU 连在 PCI 总线 0；在主 PCI 总线(PCI 0)上挂着视频设备。PCI-PCI 桥，一个特殊的 PCI 设备将主 PCI 总线于次 PCI 总线(PCI 1)相连。用 PCI 归约的术语来讲，PCI 总线 1 被称作 PCI-PCI 桥的 downstream，PCI 总线 0 叫做桥的 upstream。在第二个 PCI 总线上，是系统的 SCSI 和 Ethernet 设备。这个 PCI-PCI 桥，第二个 PCI 总线和其上的两块设备在物理上都可以在一个 PCI 卡上。系统中的 PCI-ISA 桥支持 ISA 设备。上图所示 ISA 总线下挂着一个多功能 I/O 控制器，控制系统的键盘，鼠标器和软驱。

6.1 PCI 地址空间

CPU 与 PCI 设备需要存取在它们之间共享的内存。设备驱动程序使用这片内存来控制 PCI 设备并用来传送信息。一般而言，这些共享内存中包含设备的控制和状态寄存器。这些寄存器用来控制设备和读取设备的状态。例如，PCI SCSI 设备驱动程序读取 SCSI 设备的状态寄存器以探测该设备是否已就绪可以将一个数据块写入 SCSI 磁盘。又例如，设备驱动程序可以在控制寄存器中写入控制数据从而使设备开始运转在设备电源被开启之后。

上述的共享内存可以是 CPU 的系统内存。但如果这样的话，每次 PCI 设备存取内存时，CPU 将被阻塞，等待 PCI 设备存取的结束。一般而言，在某个特定时刻，只能一个系统部件存取一个特定的内存。所以，上述方法会使系统性能降低。另外，允许系统的外设不在一个良好的控制下存取主内存不是一个好的方法。这将会是非常危险的事情。一个“淘气”的设备可以使得系统非常不稳定。

因此，外设一般拥有它们自己的内存空间。CPU 可以存取这个空间。但是反之外设存取系统内存必须在 DMA(Direct Memory Access)的控制之下。ISA 设备可以存取两种地址空间，ISA I/O(Input/Output)和 ISA memory。PCI 可以有三种：PCI I/O，PCI memory 和 PCI Configuration 空间。所有的这些地址空间都可以被 CPU 所存取。其中设备驱动程序要使用 PCI I/O 和 PCI memory 空间。Linux 核心中的 PCI 初始化代码要用到 PCI Configuration 空间。

6.2 PCI 配置头(Configuration Header)

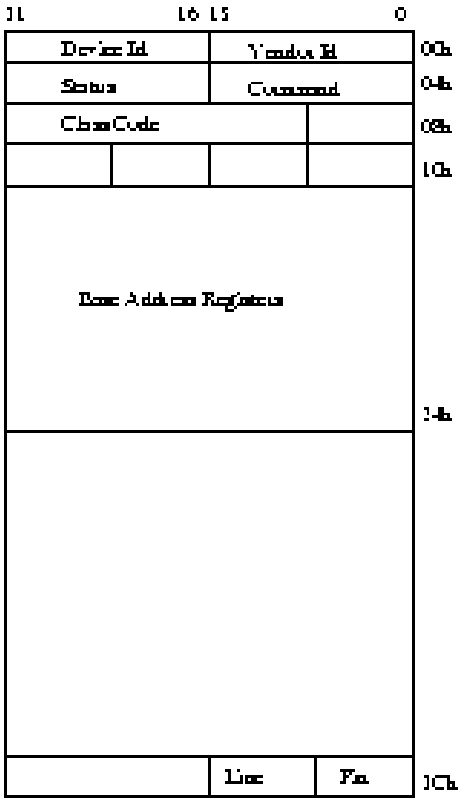

图 6.2 PCI Configuration Header (配置头)

系统中的每个 PCI 设备，包括 PCI-PCI 桥，都有一个配置数据结构在 PCI Configuration 地址空间中。这个 PCI Configuraton Header(配置头)被系统用来定位和控制一个设备。至于这个数据结构具体位于 PCI Configuration 空间的何处，依赖于设备位于 PCI 拓朴结构中的位置。例如，一个 PCI 视频(Video)卡插在 PCI 母板(motherboard)上的一个 PCI 槽中，它的数据配置头结构将会在一个地方；如果被插在另一个地方，其数据配置头将会位于 PCI Configuration 空间中的另外一个地方。当然，这没有关系。不论 PCI 设备和 PCI 桥在哪里，系统都将检测到，并使用它们的配置头中的状态和配置寄存器来对它们进行配置。

一般来讲，每个 PCI 插槽的 PCI 配置头都位于在 PCI Configuration 空间的一个偏移量(Offset)处，这个偏移量是与每个 PCI 插槽的位置顺序相关的。例如，PCI 板上的第一个 PCI 插槽的 PCI 配置在偏移量 0；第二个 PCI 插槽的配置在偏移 256 处(所有的配置头的大小是 256 个字节)。系统中提供一与硬件有关的机制，使得 PCI 配置代码可以试图检测在一个给定的 PCI 总线上所有可能的 PCI 配置头，从而知道哪个 PCI 插槽上目前有设备，哪个插槽上暂无设备。这是通过读取配置头上的某个域而完成的(一般是“Vendor Identification" 域)。如果一个插槽上为空，上述操作会返回一些错误返回值，如 0xFFFFFFFF。

图 6.2 所示的是一个完整的 256 字节的 PCI 配置头数据结构。它包含下列数据域：

Vendor Identification(厂商标识)
: 一个唯一的数字标识描述一个 PCI 设备的出处。Digital 的 PCI 厂商标识是 0x1011; Intel 的是 0x8086。

Device Identification(设备标识)
: 一个唯一的数字标识用来描述一个设备。例如 Digital 的 21141 快速 Ethernet 网卡有一个设备标识 0x0009。

Status(状态)
: 这个域给出一个设备的状态。这个域的每个位的含义是由一个标准来定义的。

Command(命令)
: 系统通过在这个域中写入数据来控制设备。例如，让设备存取 PCI I/O 空间。

Class Code(设备类代码)
: 这个域用来标定一个设备的类型。每一种设备都对应一个标准的类。如 video(视频),SCSI 等等。SCSI 设备的类码是 0x0100。

Base Address Registers(基地址寄存器)
: 这些寄存器用来决定和分配一个设备可用的存储空间类型(如，PCI I/O 和 PCI memory)大小和位置。

Interrupt Pin(中断管脚)
: 在每个 PCI 卡上，有 4 个物理管脚可以传送中断信号到 PCI 总线上。其标准的标号是 A，B，C 和 D。这个”Interrupt Pin"域描述的是这个 PCI 设备正在用那个物理中断管脚。通常来讲，对一个特定的设备，其使用的中断管脚是被固定好的。也就是说，每次系统重新启动时，该设备使用同样的中断管脚。这个信息使得中断处理子系统知道如何管理这个设备的中断。(译者注：请参阅中断处理章节。注意 PCI 设备于 ISA 设备在中断方面的区别)

Interrupt Line(中断线)
: 设备的 PCI 配置头数据结构的"Interrupt Line"域用来在 PCI 初始化代码，设备驱动程序和 Linux 中断处理子系统之间传递一个中断处理。这个域中的值对于设备驱动程序而言是无意义的，但其可以使得中断例程正确地将一个中断从一个 PCI 设备传递(route)到 Linux 中相应的设备驱动程序的中断处理代码中。请参阅第七章关于 Linux 如何处理中断。

6.3 PCI I/O 和 PCI Memory 地址

设备使用这两种地址空间来与其在 Linux 核心中运行的设备驱动程序进行通信。例如，DECchip 21141 快速 Ethernet 设备映射其内部寄存器到 PCI I/O 地址空间中。从而其设备驱动程序可以读和写这些寄存器来控制该设备。Video 驱动器经常使用大量的 PCImemory 空间来存放视频信息。

在 PCI 系统初始化完成，使用上述的"command"命令允许设备存取地址空间之前，系统不能存取这块地址。值得注意的是只有 PCI 配置代码才读和写 PCI Configuration 地址。Linux 设备驱动程序只能读和写 PCI I/O 和 PCI memory 地址空间。

6.4 PCI-ISA 桥

PCI-ISA 桥通过将对 PCI I/O 和 PCI memory 地址空间的存取转换到对 ISA I/O 和 ISA memory 地址空间的存取来支持对 ISA 设备的支持。许多系统中含有一些 ISA 总线槽和几个 PCI 总线槽。随着时间的推移，这种为了向后兼容的机器配置将会消失。系统将会只支持 PCI 系统。在 ISA 地址空间中(ISA I/O 和 ISA memory),系统中的 ISA 设备的寄存器地址被固定在某个地方(自从早期的 Intel 8080 PC 开始)。例如，一个$5000 的基于 Alpha AXP 的计算机的软盘控制器与早期的 IBM PC 的在 ISA 地址空间中占据的是同一片地址。PCI 归约(Specification)通过在 PCI I/O 和 PCI memory 地址空间的低端为 ISA 外设的使用保留一片区域并通过 PCI-ISA 桥来将对这片保留的 PCI 地址空间的存取映射到对系统中 ISA 地址的存取上。(译者注：细心的读者不难发现，这种通过加"Layer",或"映射"的方法在计算机软硬件系统中几乎无处不见。系统就是这样一层一层的抽象出更高的概念提供给更高的层使用，直到用户层，从而使得一切细节的复杂性变的越来越透明。)

6.5 PCI-PCI 桥

PCI-PCI 桥是一种特殊的 PCI 设备。它将系统中的 PCI 总线粘合作一起。简单的系统只有通常一个 PCI 总线。一个 PCI 总线上可支持的 PCI 设备的数目是有限的。使用 PCI-PCI 桥可以解决上述问题，允许将更多的 PCI 总线加入系统从而支持更多的 PCI 设备。这对于高性能的服务器来说是非常重要的。Linux 全面地支持 PCI-PCI 桥机制。

6.5.1 PCI-PCI 桥：PCI I/O 和 PCI memory 窗口

PCI-PCI 桥只负责传递一部份对 PCI I/O 和 PCI memory 读和写的请求到 downstream(请参见第一节)。例如，图 6.1 中，仅当读和写请求中的 PCI I/O 或 PCI memory 地址属于 PCI-PCI 桥 SCSI 或 Ethernet 设备时 PCI-PCI 桥才将这些总线上的请求从 PCI 总线 0 传递到 PCI 总线 1。其他的将被忽略。这种过滤机制可以避免地址在系统中没必要的繁衍。为了做到这点，每个 PCI-PCI 桥必须正确地被设置好它所负责的 PCI I/O 和 PCI memory 地址的起始和大小。当一个读或写请求落在其负责的范围之内，这个请求将被映射到次一级的 PCI 总线上。系统中的 PCI-PCI 桥一旦设置完毕，如果 Linux 中的设备驱动程序存取的 PCI I/O 和 PCI memory 地址只在这些窗口之内，这些 PCI-PCI 桥是不可见的(译者注：窗口在这里的含义是指每个 PCI-PCI 桥都仅负责一定范围的空间映射；上述原文是："Once the PCI-PCIBridges in a system have been configured then so long as the Linux device drivers only access PCI I/O and PCIMemeory space via these windows, the PCI-PCI Bridges are invisible.")。这是个很重要的特性使得 Linux PCI 设备驱动程序开发者的工作容易些。然而，这也使得 Linux 配置 PCI-PCI 桥变的有点迷惑。我们将在下面的章节中看到这一点。

6.5.2 PCI-PCI 桥：PCI 配置周期(Configuration Cycles)和 PCI 总线计数方法(Bus Numbering)

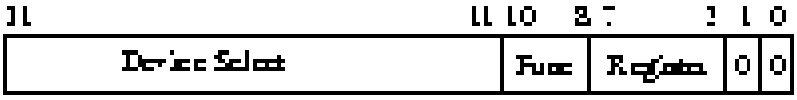

图 6.3 PCI 配置周期类型 0

图 6.4 PCI 配置周期类型 1

既然 CPU 的 PCI 初始化代码可以存取那些不在主 PCI 总线上的设备，那么必须存在一个机制使得这些桥能够判断是否将一个 PCI 配置周期从它们的(译者注：它们指的是 PCI-PCI 桥设备)主接口传递它们的次接口。所谓一个“周期”指的是一个出现在 PCI 总线上的一个地址。PCI 归约(Specification)定义了两种 PCI 配置寻址格式；类型 0 和类型 1。图 6.3 和图 6.4 所示分别是这两种寻址类型。PCI 配置周期类型 0 中不含有总线号码，被所有的设备当作针对当前这个 PCI 总线上的 PCI 配置周期。类型 0 地址中的 31-11 位被用来作为设备选择域。一种设计方案是每一位对应一个设备。在这种情况下，位 11 表示插槽 0 上的 PCI 设备。位 12 表示插槽 2 上的 PCI 设备并以此类推。另一种方法是直接地将设备的插槽号码写入位 31-11 中。具体采用哪种机制依赖于系统的 PCI 存储控制器(memory controller)。

类型 1 的 PCI 配置周期地址中含有一个 PCI 总线号码。当这种类型的配置周期(或命令)出现在一个 PCI 总线上时，除了 PCI-PCI 桥之外，所有其他的 PCI 设备会将其忽略。所有”看见“配置命令类型 1 的 PCI-PCI 桥可能选择将类型 1 的配置周期传递到其 downstream PCI 总线上或忽略之。选择的决定依赖于 PCI-PCI 桥的配置。没一个 PCI-PCI 桥有一个主 PCI 总线接口号和一个第二 PCI 总线接口号。主总线接口是那个离 CPU 更近的；第二总线接口是那个离 CPU 较远的。每个 PCI-PCI 桥还拥有一个次级总线号码。这个数目代表着在这个 PCI-PCI 桥第二总线接口之下的所有 PCI 总线中的最大数。换种方式讲，这个次级总线数目是这个 PCI-PCI 桥的 PCI 总线 downstream 的最大数目。当一个 PCI-PCI 桥看见一个类型 1 PCI 配置周期时，桥的行为如下：

* 忽略这个命令，如果指定的总线号码不在第二和次级总线号码之间。(包含边界值)

* 将其转换成类型 0 配置命令，如果指定的总线号码是桥的第二级总线接口。

* 将其(原封不动地)传递到第二级总线接口，如果指定的总线号码大于该桥的第二总线号但小于或等于次级总线号。

因此，如果我们想访问在图 6.9 中总线 3 上的设备 1，我们必须从 CPU 产生一个类型 1 的 PCI 配置命令。桥 1 将原封不动地将此命令传递到总线 1；桥 2 将忽略此命令，但桥 3 将接受这个命令并将其转换成一个类型 0 的配置命令，然后发送到设备 1 所在的 PCI 总线 3 上。

关于在 PCI 配置期间如何分配总线号码依赖于具体的操作系统。然而不论什么样的分配方案，对系统中所有的 PCI-PCI 桥来说，必须满足下列要求：

在 PCI-PCI 桥后面的所有 PCI 总线的编号必须在该桥的第二总线接口号码和次级总线号码之间

如果违反这个规则，PCI-PCI 桥将不会正确地传递和翻译类型 1 的 PCI 配置命令。系统将不能正确地发现和初始化系统中的 PCI 设备。为了完成这个赋值方案，Linux 依照一个特殊的顺序来配置这些 PCI 设备。从图 6.6 开始我们描述了 Linux PCI 桥和总线的号码赋值方案。并举出了一个例子。

6.6 Linux PCI 初始化

Linux PCI 初始化代码逻辑上分为三个部份：

PCI 设备驱动程序

这个伪设备驱动程序从总线 0 开始查询 PCI 系统并且定位系统中所有的 PCI 设备和桥。它建立一个　　可以用来描述这个 PCI 系统拓朴层次的数据结构链表。并且对所有的发现的桥编码。

PCI BIOS

这个软件层提供在 bib-pci-bios 归约中描述的服务。虽然 Alpha AXP 不提供 BIOS 服务，在其 Linux　版本中包含了相应的功能。

PCI Fixup

与特定系统相关的 PCI 初始化修补代码

6.6.1 Linux 核心的 PCI 数据结构

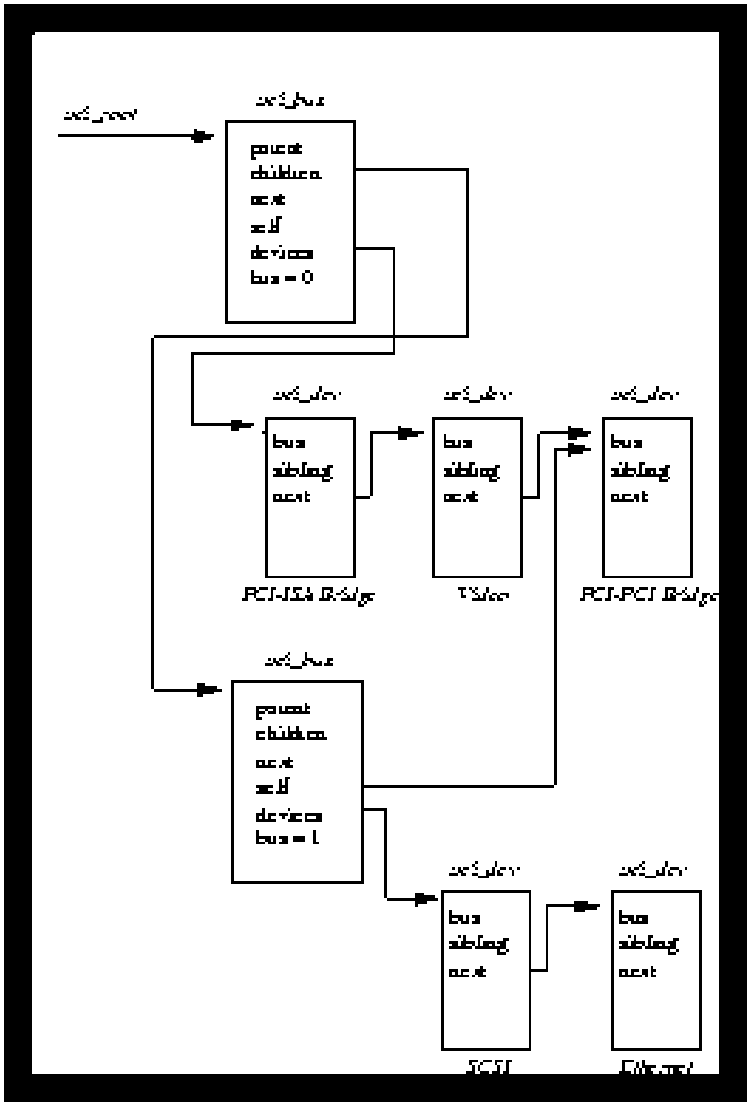

图 6.5 Linux 核心 PCI 数据结构

当 Linux 核心初始化 PCI 系统时，它建立一些可以描述系统 PCI 拓朴的数据结构。图 6.5 所示是反映图 6.1 系统的数据结构之间的关系。

每一个 PCI 设备(包括 PCI-PCI 桥)用一个 pci_dev 数据结构来描述。每个 PCI 总线用一个 pci_bus 结构来描述。这样的结果是产生了一个 PCI 总线树状关系结构。每个 PCI 总线结构 pci_bus 下挂着在该总线上的 PCI 设备。因为除了主 PCI 总线，总线 0，PCI 总线只能通过 PCI-PCI 桥来存取，每个 pci_bus 中含有一个指向其上的 PCI 设备(PCI 桥)的指针(这些设备 pci_bus 结构用链表连在一起，如图 6.5)。一个 PCI 设备是其"父"PCI 总线的"孩子"。(请注意图 6.5 中指针 children)

图 6.5 中没有显示出来的一个指针是 pci_devices。它用来指向系统中所有的 PCI 设备。系统中所有的 PCI 设备将其 pci_dev 数据结构加入到这个队列中。这个队列被 Linux 核心用来快速查找系统中的 PCI 设备。

6.6.2 PCI 设备驱动程序

PCI 设备驱动程序并不是真正的，严格意义上的驱动程序。它是在系统初始化时被调用的一个操作系统函数。PCI 初始化代码必须扫描系统中所有的 PCI 总线，寻找系统中所有的 PCI 设备(包括 PCI-PCI 桥设备)。

它使用 PCI BIOS 代码来发现它正在扫描的 PCI 总线上的每个插槽上是否已有设备安装。如果在一个插槽上发现了一个设备，一个用来描述该设备的 pci_dev 数据结构将被创建并且加入到被 pci_devices 所指向的队列中。

PCI 初始化代码从 PCI 总线 0 开始扫描。它通过读取”Vendor Identification"和"Device Identification"来试图发现每一个插槽上的设备(请参阅 6.2)。

如果发现了一个 PCI-PCI 桥，则创建一个 pci_bus 数据结构并且连入到由 pci_root 指向的 pci_bus 和 pci_dev 数据结构组成的树中。PCI 初始化代码通过设备类代码 0x060400 来判断一个 PCI 设备是否是 PCI-PCI 桥。然后，Linux 核心开始构造这个桥设备另一端的 PCI 总线和其上的设备。如果还发现了桥设备，就以同样的步骤来进行构建。这个处理过程称之为深度优先算法。系统的 PCI 拓朴在广度查询之前，先进行深度优先查找。

请参阅图 6.1，由上述算法可知，在构造 PCI 总线 0 上的 Video 设备之前，Linux 将首先构造 PCI 总线 1 和其上的 Ethernet 和 SCSI 设备(译者注：这里的前提是：图 6.1 中 PCI-PCI 桥所在的插槽号码小于 video 卡所在的 PCI 插槽的号码)

当 Linux 查询 downstream PCI 总线时，它必须构造 PCI-PCI 桥的第二级和次级总线编号的号码。下面我们将对此进行相信描述。

构造 PCI-PCI 桥---对 PCI 总线号码进行赋值

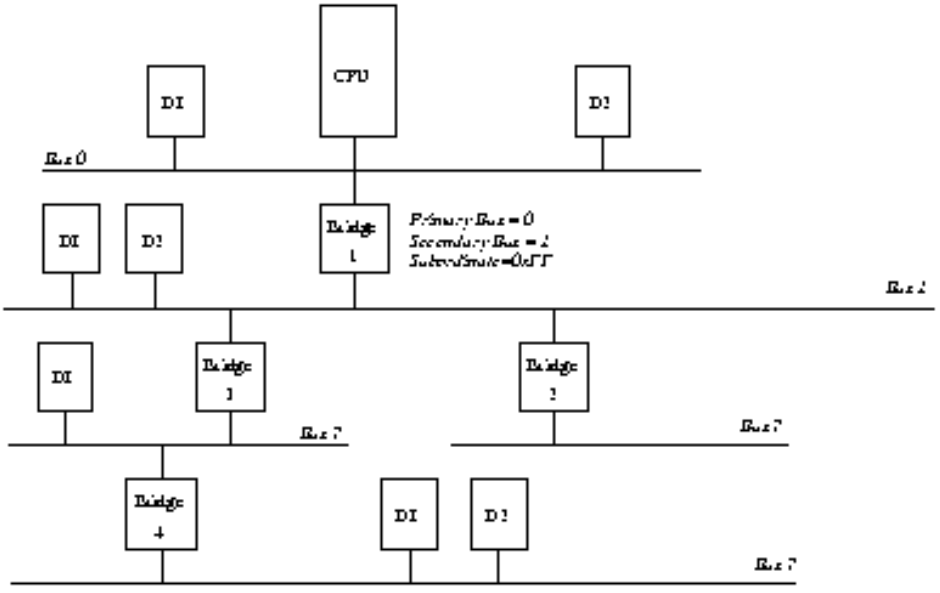

图 6.6 构造一个 PCI 系统：第一步

PCI-PCI 桥要想正确传递对 PCI I/O，PCI Memory 或 PCI Configuration 地址空间的读和写请求，必须知道下列信息：

Primary Bus Number(主总线号)
　　该 PCI-PCI 桥的紧接的 upstream 总线的编号。
Secondary Bus Number(第二级总线号)
　　该 PCI-PCI 桥的紧接的 downstream 总线的编号。
Subordinate Bus Number(次级总线号)
　　该桥的 downstream 总线中最大的总线编号。
PCI I/O 和 PCI Memory 窗口
　　对于该桥的所有 downstream 地址中的 PCI I/O 和 PCI Memory 地址空间的窗口的基址和大小。

存在的问题是当你想要配置一个 PCI-PCI 桥的时候，你不知道这个桥的次级总线接口号码。你不知道该桥下是否还有其他的 PCI-PCI 桥。即使你知道，也不清楚如何对它们进行赋值。解决方案是利用上述讲过的深度递归算法来扫描每个总线。每当发现 PCI-PCI 桥就对它们进行赋值。当发现一个 PCI-PCI 桥时，它的第二级 PCI 总线接口号可以被确定。然后我们暂时先将其次级总线接口号赋值为 0xFF。紧接着，开始扫描该 PCI-PCI 桥的 downstream 桥。这个过程看起来有点复杂。但下面的例子将给出清晰的解释。

PCI-PCI 桥的赋值--第一步

以图 6.6 的拓扑结构为例，扫描时首先发现的桥是 Bridge1(桥 1)。桥 1 的 downstream PCI 总线号码被赋值 1。自然该桥的第二级总线号码也是 1。其次级总线号码被暂时赋值为 0xFF。上述赋值的含义是所有类型 1 的含有 PCI 总线 1 或更高(<255)的号码的 pci 配置地址将被桥 1 传递到 pci 总线 1 上。如果 pci 总线号是 1，桥 1 还负责将配置地址的类型转换成类型 0。否则，就不做转换。上述动作就是开始扫描总线 1 时 Linux 初始化 代码所完成的对总线 0 的配置工作。

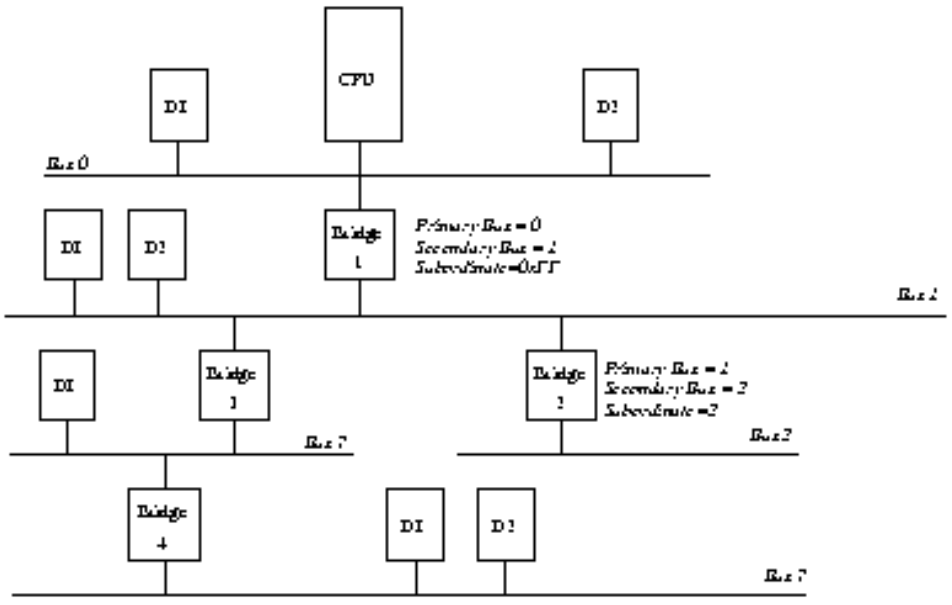

图 6.7 构造一个 PCI 系统：第二步

PCI-PCI 桥的赋值--第二步

Linux 使用深度优先算法进行扫描。所以初始化代码开始扫描总线 1。从而 PCI-PCI 桥 2 被发现。因为在桥 2 下面不再发现有 PCI-PCI 桥，所以桥 2 的次级总线号是 2，等于它的第二总线接口号。图 6.7 显示了在这个时刻总线和 PCI-PCI 桥的赋值情况。

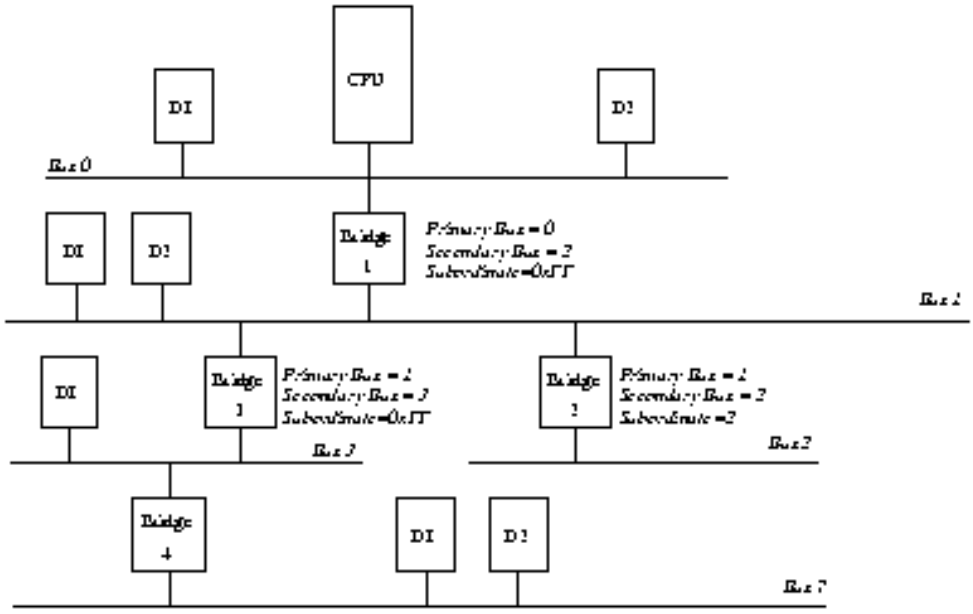

图 6.8 构造一个 PCI 系统：第三步

PCI-PCI 桥的赋值--第三步

PCI 初始化代码从总线 2 的扫描中回来接着进行扫描总线 1。这时，另外一个 PCI-PCI 桥，桥 3，被发现。它的主总线号被赋值为 1；第二级总线号为 3。因为总线 3 上还发现了桥，所以桥 3 的次级总线号被暂时赋值 0xFF。

图 6.8 显示了这个时刻系统配置的状态。到目前为止，含有总线号 1，2 和 3 的类型 1PCI 配置周期都可以被正确地传送到相应的总线上。

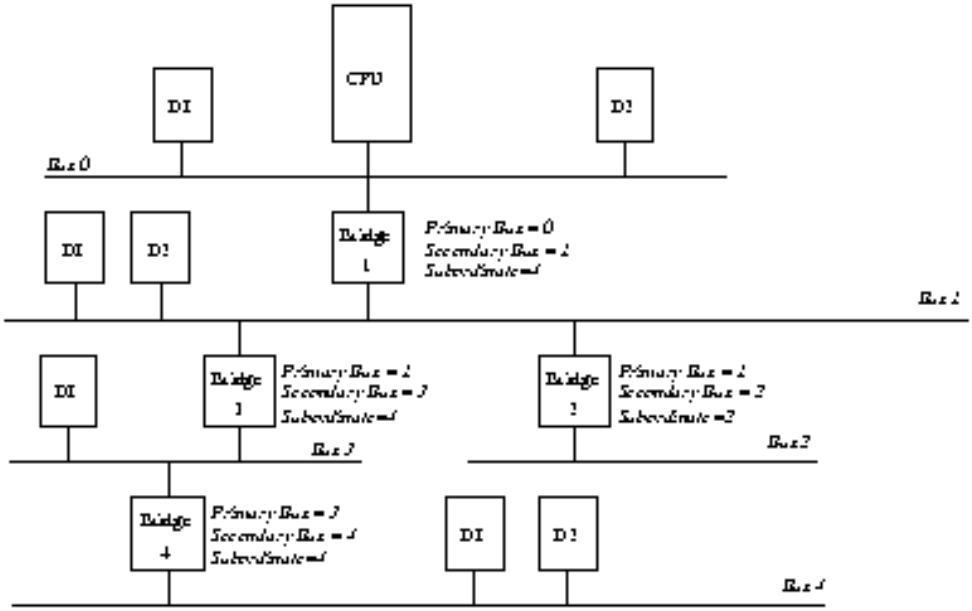

图 6.9 构造一个 PCI 系统：第四步

PCI-PCI 桥的赋值--第四步

现在 Linux 开始扫描 PCI 总线 3，桥 3 的 downstream.PCI 总线 3 上有另外一个 PCI-PCI 桥，桥 4。因此桥 4 的主总线号的值为 3。第二总线号为 4。由于桥 4 下面没有别的桥设备，所以桥 4 的次级总线号为 4。然后初始化代码回到 PCI-PCI 桥 3。这时就将桥 3 的次级总线号从 0xFF 改为 4，表示总线 4 是从桥 3 往下走的最远的 PCI-PCI 桥。最后，PCI 初始化代码将 4 以同样的道理赋值给桥 1 的次级总线号。图 6.9 反映了系统最后的状态。

6.6.3 PCI BIOS 函数

PCI BIOS 函数是一些在所有平台上都通用的一些标准例程。例如，对于 Intel 和 Alpha AXP 系统，它们都一样。BIOS 函数的存在使得 CPU 可以存取所有的 PCI 地址空间。

只有 Linux 核心代码和设备驱动程序可以使用这些函数。

6.6.4 PCI Fixup(补充或修补)

相对于 Intel 系统，Alpha AXP 系统的 PCI fixup 代码要作更多的事情。对于 Intel 系统基本上 PCI fixup 什么也不做。

对于 Intel 系统，系统的 BIOS 在启动时，已经基本上将 PCI 系统构造好了。这使得 Linux 只需将配置映射过来就好了。对于非 Intel 系统，Linux 还需做如下构建：

*为每个 PCI 设备分配 PCI I/O 和 PCI Memory 空间。
*为系统中的每个 PCI-PCI 桥，配置相应的 PCI I/O 和 PCI Memory 地址窗口。
*为每个设备的配置头产生“Interrupt Line"值；这些值控制设备的中断处理。

下面我们讲述上述行为的实现。

查询设备所需的 PCI I/O 和 PCI Memory 空间大小

系统对每个找到的 PCI 设备查询设备所需的 PCI I/O 和 PCI Memory 空间大小。为了做到这一点，每个基址寄存器先全写入 1 然后再读。设备将在没有用的位上返回 0 值。从而我们可以得知地址空间的大小。

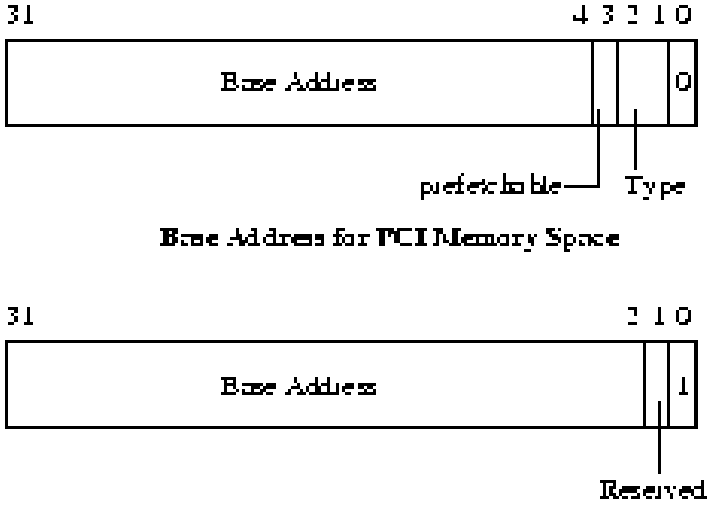

图 6.10 PCI 配置头：基地址寄存器

基地址寄存器分两种类型，以表示一个寄存器是位于 PCI I/O 空间或 PCI Memory 空间。这是通过寄存器的位 0 来设置的。图 6.10 所示是对应于 PCI I/O 和 PCI Memory 的两种形式的基址寄存器。

为了探测一个给定的基地址寄存器要申请的地址空间的大小，可以通过上述先向寄存器写入全 1 然后再读取的方法。返回值即是该基地址寄存器所申请的空间大小。这种设计还保证了所有的地址空间都是 2 的幂数从而且是自然对齐的。

例如当初始化 DECChip 21142 PCI 快速 Ethernet 设备时，我们会知道它需要 0x100 字节的 PCI I/O 或 PCI Memory 空间。Linux PCI 初始化代码将负责分配这片内存。然后，21142 的控制和状态寄存器就可以在这些地址上被访问。

为 PCI-PCI 桥和 PCI 设备分配 PCI I/O 和 PCI Memory 空间

象所有的存储空间一样，PCI I/O 和 PCI Memory 空间也是非常有限的，或稀少的。PCI Fixup 代码必须非常有效地为每个设备分配其申请的空间。PCI I/O 和 PCI Memory 必须以自然对齐的方式来被分配。例如，如果一个设备申请 0xB0 字节的 PCI I/O 空间，它必须对齐在一个是 0xB0 倍数的地址上。另外，对任何一个桥，其所需要的 PCI I/O 和 PCI Memory 必须分别对齐 4K 和 1M 的边界。由于一个桥的所有的 downstream 设备的地址空间都必须位于 PCI-PCI 桥的地址空间内，所以有必要提供一个有效的算法来进行控制。

Linux 使用的算法依赖于由 PCI 设备驱动程序建立的总线/设备树状数据结构中的每个设备分配的空间。空间是朝上增长的。系统使用一个递归算法来扫描 pci_bus 和 pci_dev 数据结构。扫描从 PCI 总线的根开始(其指针是 pci_root)。具体的行为如下：

*分别依照 4K 和 1M 字节的边界，调整当前的 PCI I/O 和 PCI Memory 的基址。

*对当前总线上的每个设备：
- 。分配 PCI I/O 和 Memory 空间
- 。相应调整全局的 PCI I/O 和 PCI Memory 基址
- 。使能(enable)设备使用被分配的空间

*递归地对该总线下面的所有总线进行空间分配。注意这会改变 PCI I/O 和 PCI Memory 基址。

*分别依照 4K 和 1M 字节的边界，调整当前的 PCI I/O 和 PCI Memory 的基址。并且计算出当前 PCI-PCI 桥的 PCI I/O 和 PCI Memory 空间窗口的基址和大小。

*将上一步骤计算出来的值对当前的 PCI 桥进行赋值。

*打开桥的对 PCI I/O 和 PCI Memory 地址过滤功能。这意味着如果一个在桥的主总线上的对 PCI I/O 和 PCI Memory 地址的寻址落在这个桥的 PCI I/O 和 PCI Memory 窗口内，该寻址指令将被传递到桥的第二级总线上。

以图 6.1 PCI 系统为例，我们给出 PCI Fixup 代码的工作如下：

校准 PCI 基址

PCI I/O 在 0x4000; PCI Memory 在 0x100000。这使得 PCI-ISA 桥将接受所有低于这些值的寻址，作为 ISA 寻址周期。

Video 设备

这个设备需要 0x200000 字节的 PCI Memory。因为为了和要求的空间大小对齐，我们从 0x20000 地址开始分配 0x200000 空间。PCI Memory 的基址移到 0x400000。PCI I/O 的基址还是 0x4000。

PCI-PCI 桥

现在碰到了 PCI-PCI 桥并对其分配 PCI 内存。 注意在这里我们不需要调整基地址。

Ethernet 设备

该设备为其 PCI I/O 和 PCI Memory 空间各要求 0xB0 字节。在 PCI Memory 的基址 0x400000 和 PCI I/O 的基址 0x4000 的基础上进行分配。结果是 PCI I/O 基址的值为 0x40B0。PCI Memory 基址为 0x4000B0。

SCSI 设备

该设备要求 0x1000 PCI Memory 空间。系统依照对齐的要求，在 0x401000 的基础上开始分配。从而 PCIMemory 的基址被调整至 0x402000。PCI I/O 的基址不变。

PCI-PCI 桥的 PCI I/O 和 PCI Memory 窗口

现在来设置桥的 PCI I/O 和 PCI Memory 的窗口大小。PCI I/O 的窗口在 0x4000 与 0x40B0 之间。PCI Memory 的窗口在 0x400000 与 0x402000 之间。这将使得桥总线上忽略对 Video 的寻址，而将传递对 SCSI 和 Ethernet 的寻址。

第七章中断与中断处理

第七章　中断与中断处理

本章讲述 Linux 内核如何处理中断。

虽然通常操作系统都提供一些通用的机制和接口来处理中断，大多数中断处理的细节是与具体的设备体系结构有关的。

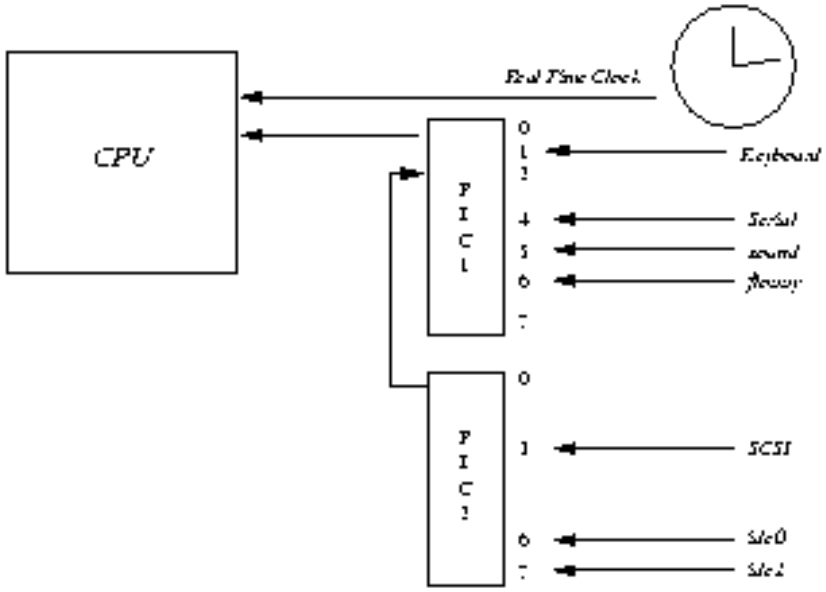

图 7.1

Linux 支持许多不同的硬件。视频设备驱动显示器，IDE 设备驱动磁盘等等。你可以同步地驱这些设备：发出一个操作请求然后等待操作的完成(比如将一块内存的内容写进磁盘)。这种方法虽然在逻辑和实践上都行的通，但是效率很差。在等待你请求的操作返回的时候，操作系统什么也不能作(busydoingnothing)，浪费了许多 CPU 时间(译者：外设的速度比 CPU 通常慢很多)。一个更好的，更有效的方法是：操作系统发出外设操作请求，然后去做别的事情。当外设请求完成并返回时，中断操作系统。通过这种方案，系统里可以同时支持许多设备请求，而不是严格的同步。

不管系统采用什么 CPU，我们必须用一些硬件来提支持设备中断 CPU。大多数处理器(比如 AlphaAXP)采用类似的方法：CPU 管蒌 7d(Pin)中的一些上的电压的变化(例如从+5 伏到-5 伏)可以导致 CPU 停止它正在处理的事情，而转到一段特殊的代码过程上去处理中断。在那些 CPU 管脚中，有一个管脚连接在一个内部定时器上，从而可以每秒接收 1000 次的中断。其他的负责中断的管脚会连在相应的其他设备上，比如 SCSI 驱动器。

在传递中断信号到一个 CPU 中断管脚前，系统常采用一个中断控制器并用它将众多的设备中断组合起来。这样就节省了 CPU 的中断管脚并且给系统设计带来了灵活性。中断控制器常用其的 Mask 寄存器和状态寄存器来控制来自设备的中断信号。通过设置 Mask 寄存器的位操作可以允许或屏蔽一些外设的中断。状态寄存器可以用来查询当前系统中已激活的待处理的中断。

系统中有一些中断管脚是固定连接的，比如，实时时钟的定时器可能被永久地连在中断控制器的管脚 3 上。当然，这些管脚具体连接的是什么设备取决于插在 ISA 或 PCI 插槽上的是设备控制器。比如，中断控制器的管脚 4 可能对应于 PCI 插槽 0，而在此插槽上有可能今天是一块 Ethernet 网卡，明天是一个 SCSI 控制器。对于这样每一种设备都提供不同的，为特定设备而写的中断处理过程，操作系统必须提供足够的灵活性来处理。

大多数通用微处理器采用同样的方式处理中断：当一个硬件中断发生时，CPU 停止它正在处理的指令，跳转到内存中的一个地址。在那个地址处，含有中断处理过程或一条可以指向中断处理过程的指令。这段代码一般运行在 CPU 的一种特殊模式---中断模式。一般而言，在这种模式下，其他的中断不会被接受。当然也存在例外的情况。一些 CPU 将中断按优先级划分，从而在处理低优先级的中断时，更高级的中断可以被处理。换句话说，最低级的那断中断处理过程必须非常细心的编写。例如，需要一个自己的栈空间以用来保存 CPU 的执行状态(所有的 CPU 寄存器和上下文)在 CPU 被剥夺并处理更高一级的中断之前。(译者注：1。上下文：context.这里的上下文讲的是用户进程切换到核心态时的上下文。2。中断处理是一个过程，通常依附在一个进程的上下文中。3。这里讲的栈通常是指一个进程在核心态下的核心栈。所有核心态下的“过程”调用包括中断处理过程都在这个栈上处理。)。有些 CPU 提供一套特殊的寄存器集。这套寄存器集只存在于中断模式下。从而中断过程处理代码可以利用它们来保存大多数的，需要保存的上下文。(译者注：现代操作系统调度中，一个很大的代价发生在进程上下文切换中。感兴趣的读者可以访问 UCBerkeley 的 NOW 项目中的 co-schedule 部份。提供一套寄存器有利于性能优化。)

当中断处理完毕后，CPU 的状态被恢复；CPU 将继续从断点处执行(译者住：这个断点有可能是会到用户态，也有可能仍然在核心态，比如，继续完成系统调用或处理低一级的中断请求。)。所以，中断处理程序要尽可能的高效以防止堵塞其他的中断。

7.1 可编程中断控制器

系统设计师可以随意选择他们希望的中断控制器硬件。IBM-PC 系列用的是 INTEL 的 82C59A-2CMOS 可编程中断控制器系列。这种控制器自从有了 PC 就存在了，提供了一套可编程的，在 ISA 地址空间里，地址是周知(Well-Known)的寄存器。任何一个现代的逻辑 chipsets 为这些寄存器保留着同样的 ISA 内存地址。非 INTEL 处理器系统，例如基于 AlphaAXP 的 PC 就不受上述限制。它们采用不同的中断控制器。

图 7.1 所示是两个 8 位的控制器连在一起，PIC1 和 PIC2。每一个控制器有一个 Mask 寄存器和一个中断状态寄存器，Mask 与中断状态寄存器的地址分别在 ox21,oxA1 和 ox20,oxA0。对一个 Mask 的某一位置 1 将允许一个中断；置 0 将屏蔽一个中断。例如，位 3 置 1 将使能(Enable)中断 3。反之将屏蔽中断 3。遗憾的是，中断 Mask 寄存器是只能写，不可读，你不能读回刚刚写

进去的位值。这意味著 Linux 必须在核心中保存一份当前 Mask 寄存器的备份。每次核心先改写这个“最近的”备份然后一次性的刷新 Mask 寄存器。

当一个中断到来时，中断处理过程读图中的两个中断状态寄存器(ISR)。系统把在 0x20 的 ISR 当做这个 16 位中断状态寄存器的低 8 位，在地址 0xA0 上的 ISR 为高 8 位。所以如果在 oxA0 上的 ISR 的第一位被置一的话，系统认为来了一个中断 9。PIC1 的第 2 位被用来连接 PIC2。所以任何 PIC2 的中断都会导致 PIC1 的第 2 位被置 1。

7.2 中断处理数据结构的初始化

核心的中断处理数据结构的设置由设备驱动程序(DeviceDriver)来负责，因为是它们需要控制中断。设备驱动程序利用 Linux 核心中的一些服务例程(译者注：核心中一些预先编好的功能函数。)来请求，使能或屏蔽一个中断。

这些各自不相同的设备驱动程序通过调用上述例程来等级它们的中断处理过程地址。

对于 PC 体系结构，一些中断的中断号是固定的，约定好的。初始化时，驱动器只要申请这个中断就可以。比如软盘驱动器将固定使用 IRQ6。有时候设备驱动程序不知道设备将使用那个中断。对于 PCI 设备驱动程序而言，这不是个问题因为设备占用的中断号可以被知道。但对于 ISA 设备驱动程序就不是那么容易知道。Linux 通过允许设备驱动程序探测中断号来解决上述问题。

首先，这个设备驱动程序通过一些操作使得这个设备发出中断。然后使能所有的，系统中还没有分配出去的中断号。这意味著这个设备发出的中断通过中断控制器会被系统接收。然后 Linux 读取 ISR 的内容并将当前值传递给上述的设备驱动程序。一个非 0 值将意味著一个或多个中断已经发生。这时，设备驱动程序重新屏蔽所有未分配的中断口。

ISA 设备驱动程序在知道它的设备占用的中断号后，就可以象正常一样去注册它的中断处理过程了。

基于 PCI 的系统比起基于 ISA 的系统有更多的灵活性。ISA 设备一般通过设置硬件板上的跳线(Jumpers)来设置中断。跳线设置后，在系统初始化后，核心程序中这个中断号是已经固定的分配给这个设备了(译者注：如果没有中断冲突的化)。然而，PCI 设备的中断是在系统启动时，通过 PCIBIOS 或 PCI 子系统在初始化时来分配的。每一个 PCI 设备卡有四个中断管脚，A,B,C 和 D。通常设备缺省使用管脚 A。每一个 PCI 插槽的 A,B,C 和 D 中断管脚都被引向中断控制器。所以 PCI 插槽 4 的管脚 A 可能映射在中断控制器的管脚 6 上，管脚 B 可能映射在中断控制器的管脚 7 上。

PCI 中断的如何映射跟不同的系统有关。任何一个系统都要提供一些代码用来解释 PCI 中断映射拓朴。基于 INTEL 的 PC 通过 BIOS 代码。对于没有 BIOS 的系统(基于 AlphaAXP 的系统)，Linux 核心将会负责处理上述任务。

上述 PCI 设置代码将每块 PCI 设备相对应的 IRQ 号写入一个 PCI 配置头(ConfigurationHeader)数据结构。IRQ 号的获得是通过 PCI 中断映射拓朴，PCI 插槽和哪一个 PCI 中断管脚正被使用而推导出来的。对每块 PCI 设备，它用的 IRQ 号将被固定下来并写入其相应的 PCI 配置头数据结构的值域中--"interruptline"。当这个设备运行时，它读取这个信息然后向 Linux 核心要求占有这个中断的处理权。

在一个系统中，有可能同时存在许多 PCI 中断源。例如当 PCI 桥的情况下。所以就有可能中断源的数目超过系统提供的可编程中断控制器的管脚数目。这种情况下，PCI 设备之间可能要共用一些中断口，中断控制器一个管脚将接收来自多个 PCI 设备的中断。Linux 允许第一个申请占有一

个特定中断口的中断源愿不愿意将这个中断口被其他设备共享。共享的中断口信息都存放在一些叫做 irqaction 的数据结构中。irqaction 结构的地址可在一个向量 irq_action 中找到。当一个共享的中断发生时，Linux 将调用挂在这个中断上的，所有的设备的，中断处理程序。因此任何一个可以支持共享中断的设备驱动程序(所有的 PCI 驱动程序)都必须能够支持其中断处理过程被调用虽然在那个时刻这个设备没有中断发生。

7.3 中断处理

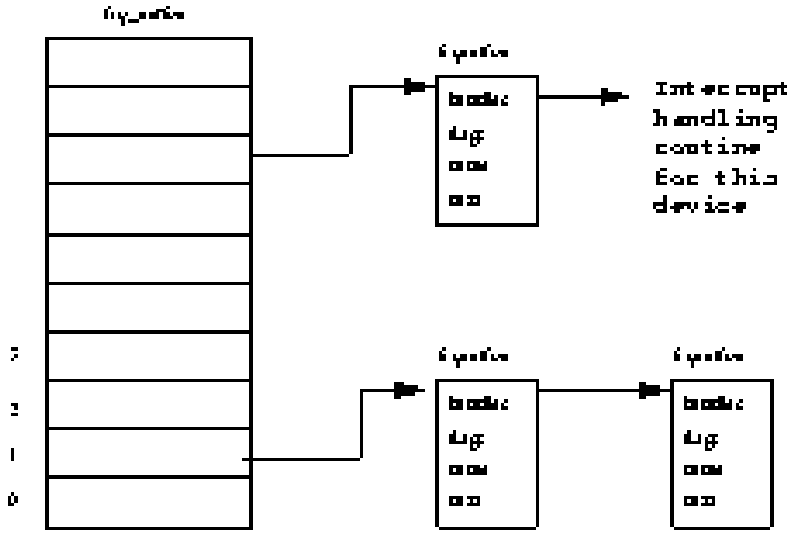

图 7.2

Linux 中断处理子系统的一个首要任务是当处理中断时，将指令控制指向正确的中断处理代码过程。完成上述任务的代码必须了解系统的中断分部情况。例如，如果软盘驱动控制器用的中断口是中断控制器的管脚 6，那么当接收到一个中断信号 6 时，系统必须将 CPU 执行地址转到软盘设备驱动程序代码处。Linux 使用一系列指针指向含有中断处理例程的数据结构。这些例程分别属于系统中不同的设备驱动程序。每一个设备驱动程序在初始化时负责申请它所需要的中断号。如图 7.2 所示，irq_action 是一个指针向量指向 irqaction 数据结构。每一个 irqaction 数据结构含有为这个中断口(译者注：irq_action 向量的下标加 1)服务的处理程序的信息(包含中断处理程序的入口地址)。至于系统支持的中断数目和中断如何被处理，对于不同的(硬件)体系结构和操作系统，方法不一样。Linux 的中断处理代码是与体系结构有关的。irq_action 向量的大小依赖于系统中中断源的数目。

当一个中断发生时，Linux 首先必须通过读取当前的 ISR(中断状态寄存器)来决定中断的来源。然后核心把这个中断源映射到 irq_action 向量一个偏移量上。例如，一个来自软驱的中断 6 将被映射到向量的第 7 个入口。如果对于一个发生了的中断没有一个中断处理句柄相对应，Linux 核心将记载一个错误。否则，核心将通过查询所有的"挂"这个中断口上 irqaction 结构并调用相应的中断处理例程。(译者注：如果是线性的链表查询的话，可以在这里做一些算法上的优化。如将最近常发生中断的那个中断源的数据结构移到链表的前面。很多现成的算法可以用上来。)

当一个设备驱动程序的中断处理例程被 Linux 核心调用以后，它必须迅速地解决为什么来了中断并做出反应。为了找出中断的原因，设备驱动程序会读取这个中断设备的状态寄存器。这个设备有可能正在汇报一个错误或一个请求的操作完成。比如，一个软盘控制器汇报对一个指定的磁盘扇区

的磁头定位已经结束。一旦中断的原因被查明，设备驱动程序有可能需要采取更多的工作去响应这个中断。如果是这样，Linux 核心提供机制允许设备驱动程序推延其操作。从而可以避免 CPU 花费太多的时间在中断模式下。有关这方面的细节请参阅设备驱动程序章节。