# 第一章 绪论

## 1.1 Unix与其他类Unix系统的比较

## 1.2 硬件的依赖性

## 1.3 Linux版本

## 1.4 操作系统的基本概念

### 1.4.1 多用户系统

### 1.4.2 用户和组

在多用户系统中，每个用户在机器上都有私有空间；典型地，他用有一定数量的磁盘空间来存储文件、接收死人邮件信息等等。操作系统必须保证用户空间的私有部分仅仅对其私有者是可见的。特别是必须保证，没有用户能够开发一个用于侵犯其他用户是有空间的系统应用程序。

所有的用户由一个唯一的数字来标识，这个数字叫用户标识符(User ID, UID)。通常一个计算机系统只能由有限的人使用。当其中的某个用户开始一个工作会话时，操作系统要求输入一个登录名和用户口令，如果用户输入的信息无效，则系统拒绝访问。因为口令不是公开的，所以用户的保密性得到了保证。

为了和其他用户有选择地共享资料，每个用户是一个或者多个用户组的一名成员，组由唯一的用户组标识符(user group ID)标识。每个文件也恰好与一个组相对应。例如，可以设置这样的访问权限，拥有文件的用户具有对文件的读写权限，同组用户仅有只读权限，而系统中的其他用户没有对文件的任何访问权限。

任何类Unix操作系统都有一个特殊的用户，叫做root，即超级用户(superuser)。系统管理员必须以root的身份登录，以便处理用户账号，完成诸如系统备份、程序升级等维护任务。Root用户几乎无所不能，因为操作系统对它不适用通常的保护机制。尤其是，root用户能访问系统中的每一个文件，能干涉每一个正在执行的用户程序的活动。

### 1.4.3 进程

所有的操作系统都使用一种基本的抽象：进程（proces）。一个进程可以定义为：“程序执行时的一个实例”，或者一个运行程序的“执行上下文”。在传统的操作系统中，一个进程在地址空间（address space）中执行一个单独的指令序列。地址空间是运行进制引用的内存地址集合。现代操作系统允许具有多个执行流的进程，也就是说，在相同的地址空间可执行多个指令序列。

多用户系统必须实施一种执行环境，在这种环境里，几个进程能并发活动，并能竞争系统资源（主要是CPU）。允许进程并发活动的系统称为多道程序系统（mutiprogramming）或多处理系统（mutiprocessing）（注5）。区分程序和进程是非常重要的：几个进程能并发地执行同一个程序，而一个进程能顺序地执行几个程序。

在单处理器系统上，只有一个进程能占用CPU，因此，在某一时刻只能有一个执行流。一般来说，CPU的个数是有限的，因而只有少数几个进程能同时执行。操作系统中叫做调度程序（scheduler）的部分决定那个程序能执行。一些操作系统运行有非抢占式（nonpreemptable）进程，这就意味着，只当今资源放弃CPU时，调度程序才能被调用。但是，多用户系统中的进程必须是抢占式的（preemptable），操作系统记录下每个进程占有的CPU时间，并周期性的激活调度程序。

Unix是具有抢占式进程的多处理器操作系统。即使没有用户登录，没有程序运行，也还是有几个系统进程在监视外围设备。尤其是，有几个进程在监视系统终端等待用户登录。当用户输入一个登录名，监听程序就运行一个程序来验证用户的口令。如果用户身份得到证实，那么监听检查就创建另一个进程来执行shell，此时在shell下可以输入命令。当一个图形化界面被激活时，有一个进程就运行窗口管理器，界面上的每个窗口通常都由一个单独的进程来执行。如果用户创建了一个图形化shell，那么，一个进程运行图形化窗口，而第二个进程运行用户可以输入命令的shell。对每一个用户命令，shell进程都创建执行相应程序的另一个进程。

类Unix操作系统采用进程/内核模式。每个进程都自以为它是系统中唯一的进程，可以独占操作系统所提供的服务。只要进程发出系统调用（即对内核提出请求，参见第10章），硬件就会把特权模式由系统态变成内核态，然后进程以非常有限的目的开始一个内核过程的执行。这样，操作系统在进程的执行上下文中起作用，以满足进程的请求。一旦这个请求完全得到满足，内核过程将迫使硬件返回到用户态，然后进程从系统调用的下一条指令继续执行。

### 1.4.4 内核体系结构

如前所述，大部分Ubix系统的单块结构：每个内核层都被集成到整个内核程序中，并代表当前进程在内核态下运行。相反的，微内核（microkernel）操作系统只需要内核有一个很小的函数集，通常包括几个同步原语、一个简单的调度程序和进程间通信机制。运行在微内核之上的几个系统进程实现从前操作系统级实现的功能，如内存分配程序。设备驱动程序、系统调用处理程序等等。

尽管关于操作系统的学术研究都是面向微内核的，但这样的操作系统一般比单块内核的效率低，因为操作系统不同层次之间显示的消息传递要花费一定的代价。不过，微内核操作系统比单块内核有一定的理论优势。微内核操作系统迫使系统程序员采用模块化的方法，因为任何操作系统层都是一个相对独立的程序，这种程序必须通风定义明确而清晰的软件接口与其他层交互。此外，已有的微内核操作系统可以很容易地移植到其他的体系结构上，因为所有

与硬件相关的部分都被封装进微内核代码中。最后，微内核操作系统比单块内核更加充分地利用RAM，因为暂且不需要执行的系统进程可以被调出或撤销。

为了达到为微内核理论上的很多优点而又不影响性能，linux内核提供了模块（module）。模块是一个目标文件，其代码可以在运行时链接带内核或从内核解除链接。这种目标代码通常由一组函数组成，用来实现文件系统、驱动程序或其他内核上层功能。与微内核操作系统的外层不同，模块不是作为一个特殊的进程执行的。相反，与任何其他静态链接的内核函数一样，它代表当前进程在内核态下执行。

使用模块的主要优点包括：

**模块化方法**

因为任何模块都可以在运行时链接或解除链接，因此，系统程序员必须提出良定义的软件接口以访问由模块处理的数据结构。这使得开发新模块变得容易。

**平台无关性**

即使模块依赖于某些特殊的硬件特点，但它不依赖于某个固定的硬件平台。例如，符合SCSI标准的磁盘驱动程序模块，在IBM兼容PC与HP的Alpha机上都能很好地工作。

**节省内存使用**

当需要模块功能时，把它链接带正在运行的内核中，否则，将该模块解除链接。这种机制对于小型嵌入式系统是非常有用的。

**无性能损失**

模块的目标代码一旦被链接到内核，其作用域静态链接的内核的目标代码完全等价。因此，当模块的函数被调用时，无需显示地进行消息传递。（注6）

## 1.5 Unix文件系统概述

Unix操作系统的设计集中反映在其文件系统上，文件系统有几个有趣的特点。因为在后面的章节中会反复提到这些特点，所有我们先回顾最重要的几个特点。

### 1.5.1 文件

Unix文件是以字节序列组成的信息载体（container），内核不解释文件的内容。很多编程的库函数实现了更高级的抽象，例如，由字段构成的记录以及基于关键字编址的记录。然而，这些库中的程序必须依靠内核提供的系统调用。从用户的观点来看，文件被组织在一个树结构的命名空间中，如图1-1所示。

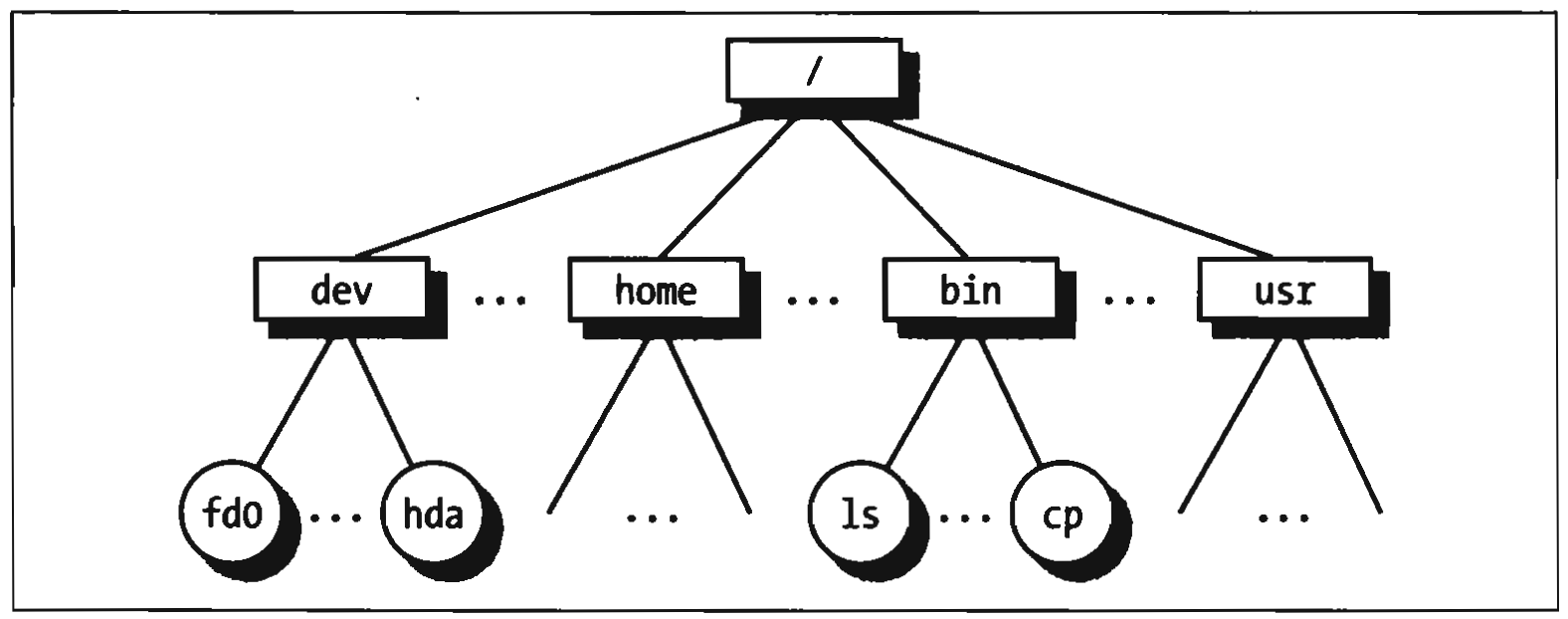


图1-1: 目录树示例

除了叶节点之外，树的所有节点都表示目录名。目录节点包含它下面文件及目录的所有信息。文件或目录名由除“/”和空字符“\0”之外的任意ASCII字符序列组成（注7）。大多数文件系统对文件名的长度都有限制，通常不能超过255个字符。与树的根相对应的目录被称为根目录（root directory）。按照惯例，它的名字是“/”。在同一目录中的文件名不能相同，而在不同目录的文件名可以相同。

Unix的每个进程都有一个当前工作目录（参见本章后面的“进程/内核模式”一节），它属于进程执行上下文（execution context），标识出进程所用的当前目录。为了标识一个特定的文件，进程使用路径名（pathname），路径名由斜杠及一列指向文件的目录名交替组成。如果路径名的第一个字符是斜杠，那么这个路径就是所谓的绝对路径，因为它的起点是根目录。否则，如果第一项是目录名或是文件名，那么这个路径名就是所谓的相对路径，因为它的起点是进程的当前目录。

当标识文件名时，也用符号“.”和“..”。他们分别表示当前工作模块和父目录。如果当前工作是根目录，“.”和“..”就是一致的。

### 1.5.2 硬件链接和软连接

包含在目录中的文件名就是一个文件的硬链接（hard link），或简称（link）。在同一目录或不同的目录中，同一文件可以有几个链接，因此对应几个文件名。

Unix命令：

$ ln p1 p2

用来创建一个新的硬链接，即由路径p1标识的文件创建一个路径名为p2的硬链接。

硬链接有两方面的限制：

* 不允许用户给目录创建硬链接。因为这可能把目录树变为环形图，从而就不可能通过名字定位一个文件。
* 只有在同一文件系统中的文件之间才能创建链接。这带来比较大的限制，因为现代Unix系统坑你包含了多种文件系统，这些问价系统位于不同的磁盘或分区，用户也许无法知道他们之间的物理划分。

为了克服这些限制，引入了软链接（soft link）[也称符号链接（symbolic link）]。符号链接是短文件，这些文件包含有另一个文件的任意一个路径名。路径名可以指定位于任意一个文件系统的任意文件或目录，甚至可以指向一个不存在的文件。

Unix命令：

$ ln -s p1 p2

创建一个路径名为p2的新软链接，p2指向路径名位p1。当这个命令执行时，文件系统抽出p2的目录部分，并在那个目录下创建一个名为p2的符号链接类型的新项。这个新文件包含路径名位p1。这样，任何对p2的引用都可以被自动转换层指向p1的一个引用。

### 1.5.3 文件类型

Unix文件可以是下列类型之一：

* 普通文件（regular file）
* 目录
* 符号链接
* 面向块的设备文件（block-oriented device file）
* 面向字符的设备文件（character-oriented device file）
* 管道（pipe）和命名管道（named pipe）（也叫FIFO）
* 套接字（socket）详细讨论

前三种文件类型是所有Unix文件系统的基本类型。其实现将在第十八章详细讨论。

设备文件与I/O设备以及集成到内核中的设备驱动程序相关。例如，当程序访问设备问价时，它直接访问与那个文件相关的I/O设备（参见第十三章）。

管道和套接字是用于进程间通信的特殊文件（参见本章后面的“同步和临界区”一节以及第十九章）。

### 1.5.4 文件描述符与索引节点

Unix对文件的内容和描述文件的信息给出了清楚的区分。除了设备文件和特殊文件系统文件之外，每个文件都由字符序列组成。文内容不包括任何控制信息，如文件长度或文件结束符（end-of-file，EOF）。

文件系统处理文件需要的所有信息包含在一个名为索引节点（inode）的数据结构中。每个文件都有自己的索引节点，文件系统用索引节点来标识文件。

虽然文件系统及内核函数对索引节点的处理可能随Unix系统的不同有很大的差异，但它们必须至少提供在POSIX标准中指定的如下属性：

* 文件类型（参加前一节）
* 与文件相关的硬链接个数
* 以字节为单位的文件长度
* 设备标识符（即包含文件的设备的标识符）
* 在文件系统中标识文件的索引节点号
* 文件拥有者的UID
* 文件的用户组ID
* 几个时间戳，标识索引节点状态改变的时间，最后访问时间及最后修改时间
* 访问权限和文件模式（参见下一节）

### 1.5.6 范围权限和文件模式

文件的潜在用户分为三种类型：

* 作为文件所有者的用户
* 同组用户，不包括所有者
* 所有剩下的用户（其他）

有三种类型的访问权限——读、写及执行，每组用户都有这三种权限。因此，文件访问权限的组合就用这九种不同的二进制来标记。还有三种附加的标记，即suid（Set User ID），sgid（Set Group ID），及sticky用来定义文件的模式。当这些标记应用到可执行文件时有如下含义：

suid

进程每执行一个文件时通常保持进程拥有者的UID。然而，如果设置了可执行文件suid的标志位，进程就获得了改文件的拥有者的UID。

sgid

进程执行一个文件是保持进程组的用户组ID。然而，如果设置了可执行文件sgid的标志位，进程就获得了该文件用户组的ID。

sticky

设置了sticky标志位的可执行文件相当于向内核发出了个请求，当程序执行结束后，依然将它保留在内存（注8）.

当文件由一个进程创建时，文件拥有者的ID就是该进程的UID。而其用户组ID可以是进程创建者的ID，这取决于父目录sgid标志位的值。

### 1.5.7 文件操作的系统调用

当用户访问一个普通文件或目录文件的内容时，他实际上是访问存储在硬件块设备上的一些数据。从这个意义上说，文件系统是硬盘分区物理组织的用户级视图。因为处于用户态的进程不能直接与硬件交互，所以每个实际的文件操作必须在内核态下进行。因此，Unix操作系统定义了几个与文件操作相关的系统调用。

所以Unix内核都对硬件块设备的处理效率给予极大的关注，其目的是为了获得非常好的系统整体性能。在后面的章节中，我们将描述Linux与文件操作相关的主题，尤其是讨论内核如何对文件相关的系统调用做出反应。为了理解这些内容，你需要知道如何使用文件操作的主要系统调用。下面对此给予描述。

#### 1.5.7.1打开文件

进程只能访问“打开的”文件。为了打开一个文件，进程调用系统调用：

fd = open(path, flag, mode)

其中的三个参数具有以下含义：

path

表示被打开文件的（相对或绝对）路径。

flag

指定文件打开的方式（例如，读、写、读/写、追加）。他也是指定是否应当创建一个不存在的文件。

mode

指定新创建文件的访问权限。

这个系统调用创建一个“打开文件”对象，并返回所谓文件描述符（file descriptor）的标识符。一个打开文件对象包括：

* 文件操作的一些数据结构，如指定文件打开方式的一组标志，表示文件当前位置的offset字段，从这个位置开始将进行下一个操作（即所谓的文件指针），等待。
* 进程可以调用的一些内核函数指针。这组允许调用的函数集合由参数flag的值决定。

我们将在第十二章中详细讨论打开的文件对象。在这里，我们仅描述一些POSIX语义所指的一般特性：

* 文件描述符表示进程与打开文件之间的交互，而打开文件对象包含了这种交互相关的数据。同一打开文件对象也许由同一个进程中的几个文件描述符标识。
* 几个进程也许同时打开同一个文件。在这种情况下，文件系统给每个文件分配一个单独的打开文件对象以及单独的文件描述符。当这种情况发生时，Unix文件系统对进程在同一文件上发出的I/O操作之间不提供任何形式的同步机制。然而，有几个系统调用，如flock()，可用来让进程在整个文件或部分文件上对I/O操作实施同步（参见第十二章）。

为了创建一个新的文件，进程也可以调用create()系统调用，它与open()非常类似，都是由内核来处理。

#### 1.5.7.2访问打开的文件

对普通Unix文件，可以顺序的访问，也可以随机的访问，而对设备文件和命名管道文件，通常只能顺序地访问。在这两种访问方式中，内核把文件指针存放在打开文件对象中，也就是说，当前位置就是下一次进行读或者写操作的位置。

顺序访问是文件的默认访问方式，即read()和write()系统调用总是从文件指针的当前位置开始读或写。为了修改文件指针的值，必须在程序中显式地调用lseek()系统调用。当打开文件时，内核让文件指针指向文件的第一个字节（偏移量位0）。

lseek()系统调用需要下列参数：

newoffset = lseek(fd, offset, whence);

其参数含义如下：

fd

表示打开文件的文件描述符

offset

指定一个有符号整数值，用来计算文件指针的新位置。

whence

指定文件指针新位置的计算方式：可以是offset加0，表示文件指针从文件头移动；也可以是offset加文件指针的当前位置，表示文件指针从当前位置移动；offset加文件最后一个字节的位置，表示文件指针从文件末尾开始移动。

read()系统调用需要以下参数：

nread = read(fd, buf, count);

其参数含义如下：

fd

表示打开文件的文件描述符。

buf

指定在进程地址空间中缓冲区的地址，所读数据就放在这个缓冲区。

count

表示所读的字节数。

当处理这样的系统调用时，内核会尝试从拥有文件描述符fd的文件中读count个字节，其初始位置为打开文件的offset字段的当前值。在某些情况下可能遇到文件结束、空管道等待，因此内核无法成功地读取出全部count个字节。返回的nread值就是实际所读的字节数。给原来的值加上nread就会更新文件指针。write()参数与read()相似。

#### 1.5.7.3关闭文件

当进程无需再访问文件的内容时，就调用系统调用：

res = close(fd);

释放与文件描述符fd相对应的打开文件对象。当一个进程终止时，内核会关闭其所有仍然打开着的文件。

#### 1.5.7.4更名及删除文件

要重新命名或删除文件时，进程不需要打开它。实际上，这样的操作并没有对这个文件的内容起作用，而是对一个或多个目录的内容其作用。例如，系统调用：

res = remane(oldpath, newpath);

改变了文件链接的名字，而系统调用：

res = unlink(pathname);

减少了文件链接数，删除了相应的目录项。只有当链接数为0时，文件才被真正删除。

## 1.6 Unix内核概述

Unix内核提供了应用程序可以运行的执行环境。因此，内核必须实现一组服务及相应的接口。应用程序使用这些接口，而且通常不会与硬件资源直接交互。

### 1.6.1进程/内核模式

如前所述，CPU既可以运行在用户态下，也可以运行在内核态下。实际上，一些CPU可以有两种以上的执行状态。例如，Intel 80x86微处理器有四种不同的执行状态。但是，所有标准的Unix内核都仅仅利用了内核态和用户态。

当一个程序在用户态下执行时，它不能直接访问内核数据结构或内核的程序，然而，当应用程序在内核态下运行时，这些限制不在有限。每种CPU模型都为从用户态到内核态的转换提供了特殊的指令，反之亦然。一个程序执行时，大部分时间都处在用户态下，只有需要内核所提供的服务时才切换带内核态，当内核满足了用户程序的请求后，它让程序又回到用户态下。

进程是动态的实体，在系统内通常只有有限的生存期，创建、撤销及同步现有进程的任务都委托给内核中的一组例程来完成。

内核本身不是一个进程，而是进程的管理者。进程/内核模式假定：请求内核服务的进程使用所谓系统调用（system call）的特殊编程机制。每个系统调用都设置了一组识别进程请求的参数，然后执行与硬件相关的CPU指令完成从用户态到内核态的转换。

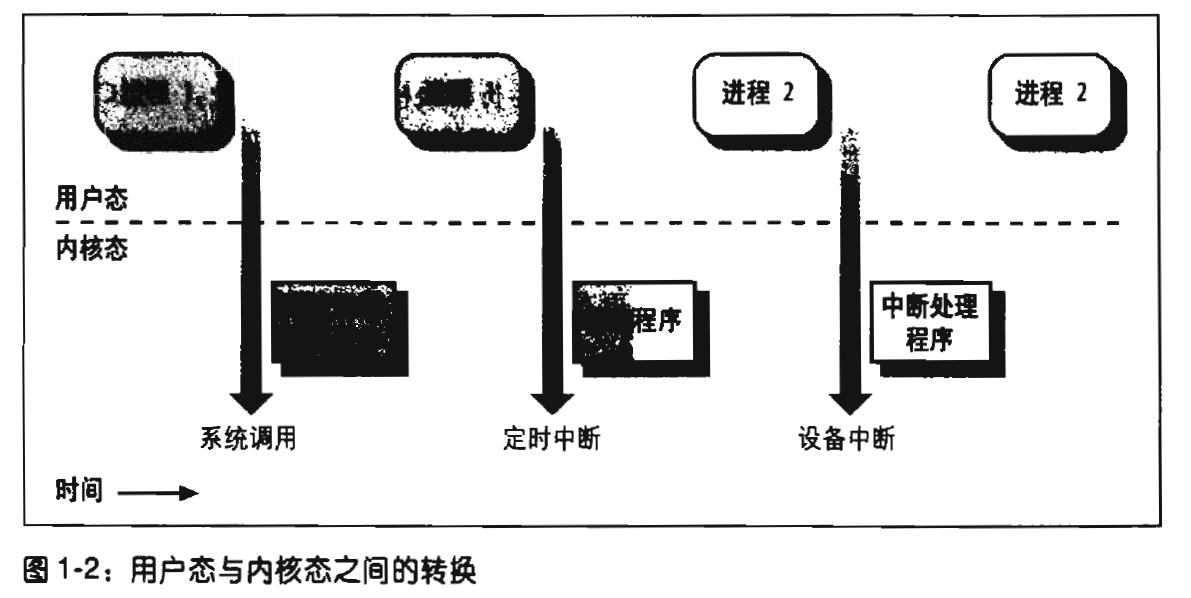
除用户进程外，Unix系统还包括几个所谓内核线程（kernel thread）的特权进程（被赋予特殊权限的进程），他们具有以下特点：

* 它们以内核态运行在内核地址空间。
* 它们不予用户直接交互，因此不需要终端设备。
* 它们通常在系统启动时创建，然后一直处于活跃状态直到系统关闭。

在单处理器系统中，任何时候只有一个进程在运行，它要么处于用户态，要么处在内核态。如果进程运行在内核态，处理器就执行一些内核例程。图1-2举例说明了用户态与内核态之间的相互转换。处于用户态的进程1发出系统调用之后，进程切换到内核态，系统调用被执行。然后，直到发生定时器中断且调度程序在内核态被激活，进程1才恢复在用户态下执行。进程切换发生，进程2在用户态开始执行，直到硬件设备发出中断请求。中断的结果是，进程2切换到内核态并处理中断。

Unix内核做的工作远不止处理系统调用。实际上，可以有几种方式激活内核例程：

* 进程调用系统调用
* 正在执行进程的CPU发出一个异常（exception）信号，异常是一些反常的情况，例如一个无效的指令。内核代表产生异常的进程处理异常。



* 外围设备向CPU发出一个中断（interrupt）信号以通知一个事件的发生，如一个要求注意的请求、一个状态的变化或一个I/O操作已经完成等。每个中断信号都是由内核中的中断处理程序（interrupt handler）来处理的。因为外围设备与CPU异步操作，因此，中断在不可预知的时间发生。
* 内核线程被执行。因为内核线程运行自傲内核态，因此必须任务其相应程序是内核的一部分。

### 1.6.2进程实现

为了让内核管理进程，每个进程由一个进程描述符（process descriptor）表示，这个描述符包含有关进程当前状态的信息。

当内核暂停一个进程的执行时，就把几个相关处理器寄存器的内容保存在进程描述符中。这些寄存器包括：

* 程序计数器（PC）和栈指针（SP）寄存器
* 通用寄存器
* 浮点寄存器
* 包含CPU状态信息的处理器控制寄存器（处理器状态字，Processor Status Word）
* 用来跟踪进程对RAM访问的内存管理寄存器

当内核觉得恢复执行一个进程时，它用进程描述符中合适的字段来装载CPU寄存器。因为程序计数器中所存的值是指向下一条将要执行的指令，所以进程从它停止的地方恢复执行。

当一个进程不在CPU上执行时，它在等待某一事件。Unix内核可以区分很多等待状态，这些状态通常由进程描述符队列实现。每个（可能为空）队列对应一组等待特定事件的进程。

### 1.6.3可重入内核

所有的Unix内核都是可重入的（reentrant）。这意味着若干个进程可以同时在内核态下执行。当然，在单处理器系统上只有一个进程在真正运行，但是许多进程可能在等待CPU或某一I/O操作完成时在内核态下被阻塞。例如，当内核代表某一进程发出一个读磁盘请求后，就让磁盘控制器处理这个请求并且恢复执行其他进程。当设备满足了读请求时，有一个中断就会通知内核，从而以前的进程可以恢复执行。

提供可重入的一种方式是编写函数，以便这些函数只能修改局部变量，而不能改变全局数据结构，这样的函数叫可重入函数。但是可重入函数不仅仅局限于这样的可重入函数（尽管一些实时内核正式如此实现的）。相反，可重入函数可以包含非重入函数，并且利用锁机制保证一次只能有一个进程执行一个非重入函数。

如果一个硬件中断发生，可重入内核挂起当前正在执行的进程，即使这个进程处于内核态。这种能力是非常重要的，因为这能提高发出中断设备控制器的吞吐量。一旦设备已发出一个中断，它就一直等待直到CPU应答它为止。如果内核能够快速应答，设备控制器在CPU处理中断时就能执行其他任务。

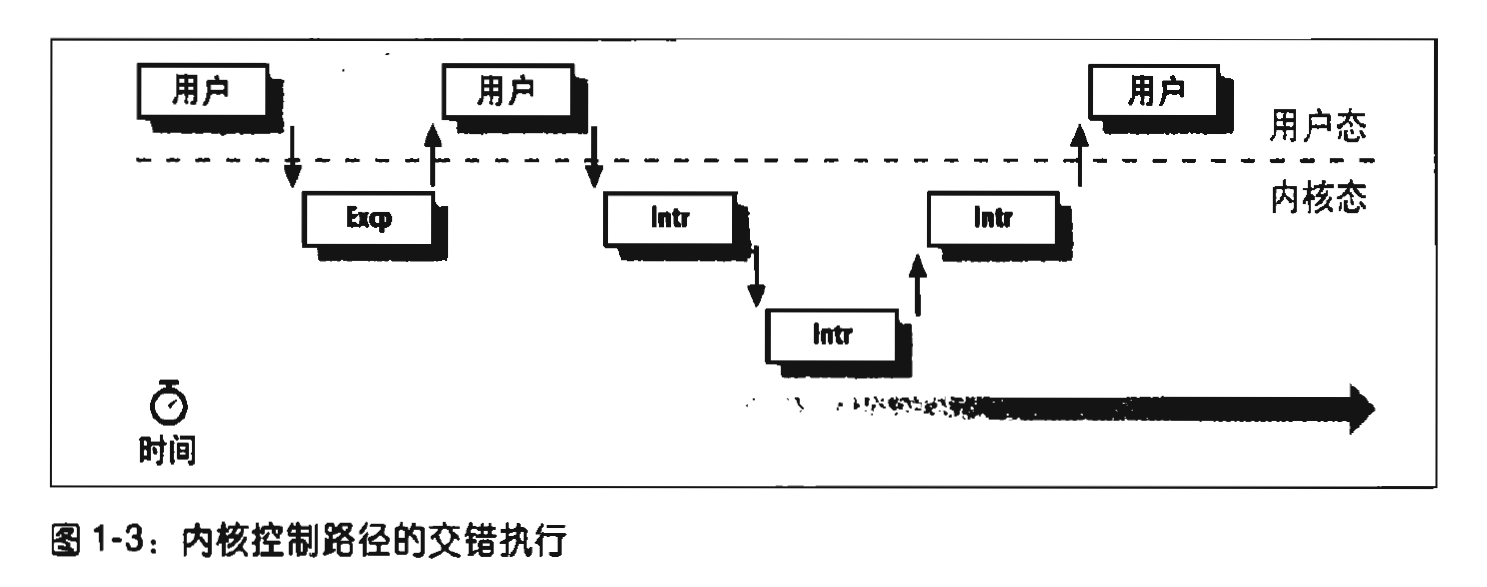
现在，让我们看一下内核可重入性及它对内核组织的影响。**内核控制路径**（kernel control path）表示内核处理**系统调用**、**异常**或**中断**所执行的指令序列。

在最简单的情况下，CPU从第一条指令到最后一条指令顺序地执行内核控制路径。然而，当前下述事件发生时，CPU交错执行内核控制路径：

* 运行在用户态下的进程调用一个系统调用，而相应的内核控制路径证实这个请求无法立即得到满足；然后，内核控制路径调用调度程序选择一个新的进程投入运行。结果，进程发生切换。第一个内核控制路径还没完成，而CPU又重新开始执行其他的内核控制路径。在这种情况下，两条控制路径代表两个不同的进程在执行。
* 当运行一个内核控制路径时，CPU检测到一个异常（例如，，访问一个不在RAM中的页）第一个控制路径被挂起，而CPU开始执行合适的过程。在我们的例子中，这种过程能给进程分配一个新页，并从磁盘读取它的内容。当这个过程结束时，第一个控制路径可以恢复执行。在这种情况下，两个控制路径代表同一个进程在执行。
* 当CPU正在运行一个启用了中断的内核控制路径时，一个硬件中断发生。第一个内核控制路径还没执行完成，CPU开始执行另一个内核控制路径来处理这个中断。当这个中断处理程序终止时，第一个内核控制路径恢复。在这种情况下，两个内核控制路径运行在同一进程的可执行上下文中所花费的系统CPU时间都是算给这个进程。然而，中断处理程序无需代表这个进程运行。
* 在支持抢占式调度的内核中，CPU正在运行，而一个更高优先级的进程加入就绪队列则中断发生。在这种情况，第一个内核控制路径还没执行完，CPU代表高优先级进程又开始执行另一个内核控制路径。只有把内核编译成支持可抢占式调度之后，才可能出现这种情况。

图1-3显示了非交错的和交错的内核控制路径的几个例子。考虑以下三种不同的CPU状态

* 在用户态下运行一个进程（User）
* 运行一个异常处理程序或系统调用程序（Excp）
* 运行一个中断处理程序（Intr）



### 1.6.4进程地址空间

每个进程运行在它的私有进地址空间。在用户态下运行的进程涉及到私有栈、数据区和代码区。在当在内核态下运行时，进程访问内核的数据区和代码区，但使用另外的私有栈。

因为内核是可重入的，因此几个内核控制路径（每个都与不同的进程相关）可以轮流执行。在这种情况下，每个内核控制路径都引用它自己的私有内核栈。

尽管看起来每个进程访问一个私有地址空间，但有时进程之间也共享部分地址空间。在一些情况下，这种共享由进程显示地提出；在另外一些情况下，由内核自动完成共享以节约内存。

如果同一个程序（比如说编译程序）由几个用户同时使用，则这个程序只被装入内存一次，其指令由所有需要它的用户共享。当然，其数据不被共享，因为每个用户将有独立的数据。这种共享的地址空间由内核自动完成一节省内存。

进程件也能共享部分地址空间，以实现一种进程间通信，这就是由System V引入并且已经被Linux支持的“共享内存”技术。

最后，Linux支持mmap()系统调用，该系统调用允许存放在块设备上的文件或信息的一部分映射到进程的部分地址空间。内存映射为正常的读写传送数据方式提供了另一种选择。如果同一文件由几个进程共享，那么共享它的每个进程地址空间都包含它的内存映射。

### 1.6.5同步和临界区

实现可重入内核需要利用同步机制：如果内核控制路径对某个内核诗句结构进行操作时被挂起，那么，其他的内核控制路径就不应当再对该数据进行操作，除非它已经被重新设置成一致性（consistent）状态。否则，两个控制路径的交互作用将被破坏所存储的信息。

例如，假设全局变量V包含某个系统状态资源的可用项数。第一个内核控制路径A读取这个变量，并且确定仅有一个可用资源项。这时，另一个内核控制路径B被激活，并读取同一个变量V，V的值仍为1.。因此，B减1，并开始用这个资源项。然后，A恢复执行，因为A已经读到V的值，于是它假定自己可以对V减1并且获取B已经在使用的这个资源项。结果，V的值变为-1，两个内核控制路径使用相同的资源项可能已经导致灾难性的后果。

当某个计算结果取决于如何调度两个或多个进程时，相关代码就不是正确的。我们说存在一种竞争条件（race condition）。

一般来说，对全局变量的安全访问通过原子操作（atomic operation）来保证。在前面的例子中，如果两个控制路径读V并减1是一个单独的、不可中断的操作，那么，就不可能出现数据讹误。然而，内核包含的很多数据结构是无法用单一操作访问的。例如，用单一的操作从链表中删除一个元素是不可能的，因为内核一次至少访问两个指针。临界区（critical region）是这样的一段代码，进入这段代码的进程必须完成，之后另一个进程才能进入（注9）。

这些问题不仅出现在内核控制路径之间，也出现在共享公共数据的进程之间，集中同步技术已经被采用。一下集中讨论怎样同步内核控制路径。

#### 1.6.5.1非抢占式内核

在寻找彻底、简单地解决同步问题的方案中，大多数传统的Unix内核都是非抢占式的：当进程在内核态执行时，它不能被任意挂起，也不能被另一个进程代替。因此，在单处理器系统上，中断或异常程序不能修改的所有内核数据结构，内核对它们的访问都是安全的。

当然，内核态的进程能自愿放弃CPU，但是在这种情况下，它必须却白所有的数据结构都处在一致性状态。此外，当这种进程恢复执行时，它必须重新检查以前访问过的数据结构的值，因为这些数据结构有可能被改变。

如果内核支持抢占，哪呢在应用同步机制时，确保今天临界区前禁止抢占，退出临界区时启用抢占。

非抢占能力在多处理器系统上的低效的，因为运行在不同的CPU上的两个内核控制路径本可以并发地访问相同的书籍结构。

#### 1.6.5.2禁止中断

单处理器系统上的另一种同步机制是：在进入一个临界区之前禁止所有硬件中断，离开时在重新启用中断。这种机制尽管简单，但远不是最佳的。如果临界区比较大，那么在一个相对较长的时间内持续禁止中断就可能使所有的硬件活动处于冻结状态。

此外，由于在对处理器系统中禁止本地CPU上的中断是不够的，所以必须使用其他的同步技术。

#### 1.6.5.3信号量

广泛使用的一种机制是信号量（semaphore），它在处理器系统和多处理器系统上多有效。信号量仅仅是与一个数据结构相关的计数器。所有内核线程在视图访问这个数据结构之前，都要检查这个信号量。可以把这个信号量看成一个对象，其组成如下：

* 一个整数变量
* 一个等待进程的链表
* 两个原子方法：down()和up()

down()方法对信号量的值减1。如果这个新值小于0，该方法就把正在运行的进程加入到这个喜好链表，然后阻塞该进程（即调用调度程序）。up()方法对信号量当前值加1，如果这个值大于或等于0 ，则激活这个信号量链表中的一个或多个进程。

每个要保护的数据结构都有它自己的信号量，其初始值位1。当内核控制路径希望访问这个数据结构时，它在响应的信号量上执行down()方法。如果信号量的当前值不是负数，则允许访问这个数据结构。否则，把内核控制路径的进程加入到这个信号量的链表并阻塞该进程。当另一个进程在哪个信号量上执行up()方法时，允许信号量链表上的一个进程继续执行。

#### 1.6.5.4自旋锁

在多处理器系统中，信号量并不总是解决同步问题的最佳方案。系统不允许在不同CPU上运行的内核控制路径同时访问某些内核数据结构，在这种情况下，如果修改内核数据结构所需的时间比较短，那么信号量可能是低效的。为了检查信号量，内核必须把进程插入到信号链表中然后挂起它。因为这两种操作比较费时，完成这些操作时，其他的内核控制路径可能已经释放了信号量。

在这些情况下，多处理器操作系统使用了自旋锁（spin lock）。自旋锁和信号量非常相似，但没有进程链表；当一个进程发现锁被另一个进程锁着时，它就不停地“旋转”，执行一个紧凑的循环指令直到锁打开。

当然，自旋锁在单处理器环境下是无效的。当内核控制路径试图访问一个上锁的数据结构时，它开始无休止循环。因此，内核控制路径可能因为正在修改受保护的数据结构而没有机会继续执行，也没有机柜释放这个自旋锁。最后的结构可能是系统挂起。

#### 1.6.5.5避免死锁

与其他控制路径同步否进程或内核控制路径很容易进入死锁（deadlock）状态。举一个最简单的死锁的例子，进程p1获得访问数据结构a的权限，进程p2获得访问b的权限，但是p1在等待b，而p2在等待a。进程之间其他更复杂的循环等待的情况也可能发生。显然，死锁情形会导致受影响的进程或内核控制路径完全出去冻结状态。

只要涉及到内核设计，当所用内核信号量的数量较多时，死锁就成为一个突出的问题。在这种情况下，很难保证内核控制路径在各种可能方式下的交错执行不出现死锁状态，有几种操作系统（包括linux）通过按规定的顺序请求信号量来避免死锁。

### 1.6.6信号和进程间通信

Unix信号（signal）提供了把系统时间报告给进程的一种机制。每种事件都有自己的信号编号，通常用一个符号常量来表示，例如SIGTERM。有两种系统事件：

* 异步通告

例如，当用户在中断按下中断键（通常为CTRL-C）时，即向前台进程发出中断信号SIGINT。

* 同步错误或异常

例如，当进程访问内存非法地址时，内核向这个进程发送一个SIGSEGV信号。

POSIX标准定义了大约20中不同的信号，其中，有两种是用户自定义的，可以当作用户态下进程通信和同步的原语机制。一般来说，进程可用以两种方式对接收到的信号做出反应：

* 忽略该信号
* 异步地执行一个指定的过程（信号处理程序）

如果进程不指定选择何种方式，内核就根据信号的编号执行一个默认操作。五种坑能的默认操作是：

* 终止进程
* 将执行上下文的进程地址空间 的内容希尔一个文件（核心转储，core dump）,并终止进程
* 忽略信号
* 挂起进程
* 如果进程曾被暂停，这恢复它的执行

因为POSIX语义允许进程暂停阻塞信号，因此内核信号的处理相当精细。此外，SIGKELL和SIGSTOP信号不能直接由进程处理，也不能由进程忽略。

AT&T的Unix system V引入了在用户态下其他种类的进程间通信机制，很多Unix内核也采用了这些机制：信号量、消息队列及共享内存。它们被统称为System V IPC。

内核把它们做为IPC资源来实现：进程要获得一个资源，可以调用shmget()，semget()或msgget()系统调用。与文件一样，IPC资源是持久不变的，进程创建者、进程拥有者或超级用户进程必须显示地释放这些资源。

这里的信号量与本章“同步和临界区”一节中所描述的信号量是相似的，只是它们用在用户态下的进程中。消息队列允许进程利用msgsnd()及msgget()系统调用交换消息，msgsnd()表示把消息插入到指定的队列中，msgget()表示从队列中提取消息。

POSIX标准（IEEE Std 1003.1-2001）定义了一种基于消息队列的IPC机制，这就是所谓的POSIX消息队列。它们和System V IPC小弟队列是相似的，但是，它们对应用程序提供一个风简单的基于文件的接口。

共享内存为进程之间交换和共享内存提供了最快的方式。通过调用shmget()系统调用来创建一个新的共享内存，其大小按需设置。在获得IPC资源标识符后，进程调用shmat()系统调用，其返回值是进程的地址空间中新区域的起始地址。当进程希望把共享内存从地址空间分离出去时，就调用shmdt()系统调用。共享内存的实现依赖于内核对进程地址空间的实现。

### 1.6.7进程管理

Unix在进程和它正在运行的程序之间做出一个清晰的划分。fork()和\_exit()系统调用分别用来创建一个新进程和终止一个进程，而调用exec()系统调用则是装入一个新程序。当这样一个系统调用执行以后，进程就在所装入程序的全新地址空间恢复运行。

调用fork()的进程是父进程，而新进程是它的子进程。父子进程能相互找到对方，因为描述每个进程的数据结构都包含有两个指针，一个直接指向它的父进程，另一个直接指向它的子进程。

实现fork()一种天真的方式就是将父进程的数据和代码都复制，并把这个拷贝赋予子进程。这会相当费时。当前依赖硬件分页单元的内核采用写时复制（copy-on-write）技术，即把页的复制延迟到最后一刻（也就是说，直到父或子进程需要时才写进页）。我们将在第九章“写时复制”一页中描述Linux是如何实现这一技术的。

\_exit()系统调用终止一个进程。内核对这个系统调用的处理是通过释放进程所拥有否资源并向父进程发出SIGCHILD信号（默认操作为忽略）来实现。

#### 1.6.7.1僵死进程（zombie process）

父进程如何查询其子进程是否终止了呢？wait4()系统调用允许进程等待，直到其中的一个子进程结束；它返回已终止子进程的进程标识符（Process ID，PID）。

内核在执行这个系统调用的时，检查子进程是否已经终止。引入僵尸进程的特殊状态是为了表示终止的进程；父进程执行完wait4()系统调用之前，进程就一直停留在那种状态。系统调用处理程序从进程描述符字段中获取有关资源使用的一些数据；一旦得到数据，就可以释放进程描述符。当进程执行wait4()系统调用时如果没有子进程结束，内核就通常把该进程设置成等待状态，一直到子进程结束。

很多内核也实现了waitpid()系统调用，它允许进程等待一个特殊的子进程。其他wait4()系统调用的变体也是相当通用的。

在父进程发出wait4()调用之前，让内核保存子进程有关信息是一个良好的习惯，但是，假设父进程终止而没有发出wait4()调用呢？这些信息占用了一些内存中非常有用的位置，而这些位置本来可以用来为活动着的进程提供服务。例如，很多shell允许用户在后台启动一个命令然后退出，正在运行这个shell命令的进程终止，但它的子进程继续运行。

解决的办法是使用一个名为init的特殊系统进程，它在系统初始化的时候被创建。当一个进程终止时，内核改变其现有子进程的进程描述符指针，使这些子进程成为init的孩子。Init监控所有子进程的执行，并且按常规发布wait4()系统调用，其副作用就是除掉所有僵死的进程。

#### 1.6.7.2进程组和登录会话

现代Unix操作系统引入看进程组（process group）的概念，以表示一种“作业(job)”的抽象。例如，为了执行命令行：

$ ls | sort | more

Shell支持进程组，例如bash，为三个响应的进程ls、sort及more创建了一个新的组。Shell以这种方式作用于这三个进程，就好像它们是一个单独的实体（更准确地说的作业）。每个进程描述符包含进程组ID的字段。每一进程组可以有一个领头进程（即其PID与这个进程组的ID相同的进程）。新创建的进程最初被插入到其父进程的进程组中。

现代Unix内核也引入了登录会话（login session）。非正式地说，一个登陆会话包含值指定终端已经开始工作会话的那个进程的所有后代进程——通常情况下，登录会话就是shell进程为用户创建的第一条命令。进程组中所有进程必须在同一登录会话中。一个登录会话可以让几个进程组同时处于活动状态，其中，只有一个进程组一直处在前台，这意味着该进程组可以访问终端，而其他活动着的进程组在后台。当一个后台进程试图访问终端时，它将收到SIGTTIN或SIGTTOUT信号。在很多shell命令中，用内部命令bg和fg把一个进程组放在后台或前台。

### 1.6.8内存管理

内存管理是迄今为止Unix内核中最为复杂的活动。在本书中，我们将用超过三分之一的篇幅来描述Linux是如何实现它的。本节只说明一些与内存管理相关的主要问题。

#### 1.6.8.1虚拟内存

所有新近的Unix系统都提供了一种有用的抽象，叫虚拟内存（virtual memory）。虚拟内存作为一种逻辑层，处于应用程序的内存请求与硬件内存管理单元（Memory Management Unit，MMU）之间。虚拟内存有很大用途和优点：

* 若干个进程所可以并发地执行。
* 应用程序所需要内存大于可用物理内存时也可以运行。
* 程序只有部分代码装入内存时进程可以执行它。
* 运行可以共享库函数或程序的一个单独内存映像。
* 进程是可重定位的，也就是说，可以把程序放在物理内存的任何地方。
* 程序员可以编写与机器无关的代码，因为他们不必关心有关物理内存的组织结构。

虚拟内存子系统的主要成分是虚拟地址空间（virtual address space）的概念。进程所用的一组内存地址不同于物理内存地址。当进程使用一个虚拟地址时（注10），内核和MMU协同定位其在内存中的实际物理地址。

现在的CPU包含了能自动把虚拟地址空间转换层物理地址的硬件电路。为了达到这个目标，把可用的RAM划分成长度为4K或8K的页框（page frame），并且引入一组页表来指定虚拟地址与物理地址之间的对应关系。这些电路使用内存分配变得简单，因为一块连续的虚拟地址请求可用通过分配一组非连续的物理地址页框而得到满足。

#### 1.6.8.2随机访问存储器（RAM）的使用

所有的Unix操作系统都将RAM毫无疑义地划分为两部分，其中若干兆字节专门用于存放内核映像（也就是内核代码和内核静态数据结构）。RAM的其余部分通常由虚拟内存系统来处理，并且用在一下三种可能的方面：

* 满足内核对缓冲区、描述符及其他动态内核数据结构的请求。
* 满足进程对一般内存区的请求及对文件内存映射的请求。
* 借助于高速缓存从磁盘及其他缓冲设备获得较好的性能。

每种请求类型都是重要的。但从另一方面来说，因为可用RAM是有限的，所有必须在请求类型之间做出平衡，尤其是当可用内存没有剩下多少时。此外，当可用内存达到临界阈值时，可以调用页框回收（page-frame-reclaiming）算法释放其他内存，那么那些页框是最适合回收的页框呢？正如我们将在第十七章看到的一样，对这个问题既没有简单的答案，也没有多少理论的支持，唯一可用的解决办法是开发经过仔细调节的经验算法。

虚拟内存系统必须解决一个主要问题是内存碎片。理想情况下，只有当空闲页框数太少时，内存请求才会失败。然而，通常要求内核使用物理上连续的内存区域，因此，即使有足够的可用内存，但它不能作为一个连续的大块使用时，内存请求也会失败。

#### 1.6.8.3内核内存分配器

内核内存分配器（Kernel Memory Allocator, KMA）是一个子系统，它试图满足系统中所有部分对内存的请求。其中一些请求来自内核其他子模块，它们需要一些内核使用的内存，还有一些请求来自于用户程序的系统调用，用来增加用户进行的地址空间。一个好的KMA应该具有下列特点：

* 必须快。实际上，这是最重要的属性，因为它由所有的内核子系统（包括终端处理程序）调用。
* 必须把内存是浪费减到最少。
* 必须努力减轻内存的碎片（fragmtation）问题。
* 必须能与其他内存管理子系统合作，以便借用和释放页框。

基于各种不同的算法技术，已经提出几种KMA,包括：

* 资源图分配算法（allocator）
* 2的幂次方空闲链表
* McKusick-Karels分配算法
* 伙伴（Buddy）系统
* Mach的区域（Zone）分配算法
* Dynix分配算法
* Solaris的Slab分配算法

我们将在第八章中看到，Linux的KMA在伙伴系统之上采用了Slab分配算法。

#### 1.6.8.4进程虚拟地址空间处理

进程的虚拟地址空间包括了进程可以引用的所有虚拟内存地址。内核通常用一组内存区描述符描述进程虚拟地址空间。例如，当进程通过exec()系统调用开始某个程序的执行时，内核分配给进程的虚拟地址空间由一下内存区组成：

* 程序的可执行代码
* 程序的初始化数据
* 程序的未初始化数据
* 初始程序栈（即用户态栈）
* 所需共享库的可执行代码和数据
* 堆（又程序动态请求的内存）

所有现代Unix操作系统都采用了所谓的请求调页（demand paging）的内存分配策略。有了请求调页，进程可以在它的页还没有在内存时就开始执行。当进程访问一个不存在的页时，MMU产生一个异常；异常处理程序找到受影响的内存区，分配一个空闲的页，并用适当的数据把它初始化。同理，当进程通过系统调用malloc()或brk()（由malloc()在内部调用）系统动态地请求内存时，内核仅仅修改进程的堆内存区的大小。只有试图引用进程的虚拟内存地址而产生异常时，才给进程分配页框。

#### 1.6.8.5高速缓存

### 1.6.9设备驱动程序