Linux核心（The Linux Kernel)

（上）

"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

译者的话:

历尽半年多，《**Linux** 核心》终于和大家见面了.作为译者，心中非常高兴。基于**Linux** 核心**2.0.33,** 本书介绍了**Linux**核心是如何工作的。它不是一本关于核心的手册，而是描述了**Linux**核心中使用的原理，机制和**Linux**为什么使用这些原理和机制。希望本书能给读者带来些益处。

编译本书的过程中，我们没有局限于原作者的内容，加入了一些译者自己的理解。由于我们的专业和英语水平有限，疏漏之处在所难免。敬请读者谅解并望指出。

谢谢,

译者全体敬上，**12/1/1999**

【注1：】

当笔者整理修订这本经典的“The Linux Kernel”一书的中文版时，才意识到时光距当年动笔翻译已是九年之遥。希望这本书籍能对Linux核心的初学者有所帮助，也不辜负夜灯下的字里行间，各位译者曾经的科技报国之心。九年来，Linux核心的演变也是非常大的，特别是2.6核心。希望读者能在阅读本书的同时，阅读最新的Linux核心的相关文章或书籍，从而可以更好的把握Linux核心的演变，更加深刻的理解核心的机制为什么演变，要解决的是什么问题。这样有可以做到忘掉Linux核心大量细节的繁杂，把握操作系统核心的设计精华。

【注2：】

关于其他译者：

胡宁宁，毕业于美国卡内基梅隆大学计算机系，博士，现任职于美国Google Inc.
毕昕， 毕业于美国普渡大学计算机系，获硕士学位。现供职于美国GTE公司。

仲盛， 毕业于美国耶鲁大学计算机系，博士，现为美国纽约大学水牛城分校计算机系助理教授。

赵振平，毕业于南京大学计算机系，硕士。目前去向不祥。

周笑波，毕业于南京大学计算机系，博士，现为美国科罗拉多大学Colorado Springs 分校计算机系助理教授。

李群， 毕业于达特茅斯大学计算机系，博士，现为美国The College of William & Mary大学计算机系助理教授。

前言

Linux是Internet的产物，从属于一个学生(Linus Torvalds)的个人爱好演变成为一个当今最流行的免费操作系统。对许多人而言，Linux是乎是个迷。一个免费的东西怎么会有价值？在一个被一群软件巨头统治的(系统)软件王国里，一个由一些电脑hackers编写的操作系统如何能够参与竞争？一个由不同的国家许多不同的人编写的软件如何能够保持其稳定性和高效性？这里的答案是Linux具有非常好的可靠性，高效性和竞争能力。许多大学和研究机构都在用Linux来作计算。许多人们已在其PC上安装了Linux。绝大多数公司都或多或少地在使用Linux。Linux被广泛地用来浏览web站点，文件处理，发送email ，玩计算机游戏。Linux绝不是一个计算机界的玩具，而是一个由全世界的爱好者开发的非常完善的，专业化的操作系统。

Linux 的源头可以追溯到Unix 家族。1969年，贝尔实验室的研究人员Ken Thompson其开始在一台空闲的PDP-7机器上实验其多用户，多任务的操作系统(multi-user, multi-tasking operating system) 。不久Dennis Richie和其他两位同事加入了他的行列。他们与实验室中的其他同事一道开发出了最早期的Unix版本。Richie在早期的项目MULTICS 中发挥了很大的作用。Unix其实是MULTICS的双关语。早期的Unix是用汇编写的。第3版时采用了C语言。C语言是Richie设计并编写的，以用来作为编写操作系统设计的语言。用C改写过的Unix使得Unix可以被移植PDP-11/45 和DIGITAL 11/70 计算机上。Unix移植到DIGITAL 11/70是一个历史性的转折，使得Unix正式从实验室走向大型机计算环境。很快，绝大多数的计算机制造商都发布了其相应的Unix版本。

Linux诞生的原因极其简单。Linus Torvalds，Linux的作者和主要管理者，当时穷的只能够付的起Minix。Minix是一个非常简单，Unix风格的，被广泛用在教学上的操作系统。Linus对Minix的功能不是很满意(译者注：不知Andrew Tanenbaum看见这句话会有何感想。“后生可畏？”。有兴趣的读者可以访问Andrew的主页.)决定自己动手编写一个软件。因为在学校里每天用的都是Unix，所以他选择将Unix作为他的软件的模型.最开始的工作是在一台Intel 386 的PC机上。他的进展很迅速。Linus对他所做的事情充满了兴趣，并通过刚刚出现的，还局限在学术领域的计算机网络，将已有的代码共享给其他的学生。其他的人看见了Linus的软件并开始加入开发的行列。不同的人由于在使用Linux时碰到不同的问题，所以这个软件也就不断地被更新和完善。不久之后，Linux就成为一个完整的操作系统了。值得注意的是Linux中没有任

何Unix代码，而是根据POSIX标准重新编写的Linux中使用了许多在Cambridge, Massachusetts的Free Software Foundation的提供GNU软件。

多数人仅把Linux 当做一个简单的工具来使用。也有许多用户在Linux进行应用程序的开发。只有很少数的人敢于为Linux写设备驱动程序和核心的patches(译者注：这个词不太好翻译)。 Linus Torvalds接受来自任何人和地方的关于核心的补充和修改。这似乎有点象无政府主义。当Linus严把质量关并由他自己将新的代码加入核心。当然，在任何时候，从事Linux核心开发的人员毕竟是只有一少部份人。

大多数Linux用户似乎不关心这个操作系统是如何构造和运行的。这不是个明智的决定因为学习Linux是更多理解一个操作系统功能的有利途径。Linux不仅仅是设计的很好，更重要的是其源代码是公开的。这是因为虽然作者拥有这个软件的版权，但在Free Software Foundation's GNU Public License的基础上，源代码是可以免费获取的。对刚接触源码的人，起初会觉得迷惑当看见一些做kernel，mm，和net的子目录。它们含有些什么？这些代码是如何工作的？为了解决上述问题，我们需要的是对整个Linux的结构有个总体的了解。这也正是本书的目的：提供一个关于Linux核心如何工作的清楚的画面，从而使得当你在运行一些应用程序时，你可以明白操作系统核心处正在发生着什么(译者注：这一点是为什么要鼓励应用程序开发者了解操作系统的结构。例如，不理解操作系统的调度，很难想象一个应用程序开发者可以编出一个高效的多线程并发程序)。

这本书讲述的是Alpha AXP-based Linux版本(译者注：这是因为作者在写此书时Digital公司(后来被Compag吞并；著名的Alta Vista seearch engine出自Digital)任职。然而95%的Linux核心源代码是与具体的硬件平台无关的。换而言之，本书95%的内容是讲述与硬件无关的Linux核心。

关于读者

本书不要求读者必须具备任何知识和经验的前提。我们假设读者会在需要时补充相关的知识。具备一定的计算机知识和C语言功底将帮助读者更好地理解本书。

本书的组织

本书并不是关于Linux内部的手册。它是关于通常意义上的操作系统，特别是对于Linux的介绍。每一章节按照从”普通到特殊“的规则来布局。首先在具体介绍细节之前，我们给出一个对核心子系统的综述。

关于核心的具体算法被特意地略去。有兴趣的读者可以参阅具体的源代码。本书将重点放在核心数据结构和其之间的关系上。

本书每一章都相对独立，就象每一个Linux子系统完成相对独立的功能一样。当然章节之间也是相关联的。例如，你不能很好地描述一个进程如果不理解虚拟内存是如何工作的。

硬件基础章节给出一个对当代PC的简要介绍。一个操作系统必须工作在一个硬件基础上。操作系统中的某些功能是与硬件相关的。为了理解Linux操作系统，读者需要具备一定的底层硬件知识。

软件基础章节介绍了基本的软件知识原理并论及了汇编和C语言。它们都是编写一个操作系统不可缺少的工具。另外本章给出了对操作系统的功能和目标的一个综述。

内存管理章节描述了Linux如何管理系统中的物理和虚拟内存。

进程章节描述了什么是一个进程，Linux核心如何创见，管理和删除系统中的进程。

进程之间，进程与核心之间通过通讯来调整/安排它们的行为。Linux支持很多种进程间通讯--IPC(Inter-Process Communication)机制。信号与管道是其中的两种Linux还支持System V IPC机制。关于IPC机制的内容在IPC章节描述。

PCI标准是一个高效的PC总线。PCI章节讲述了Linux核心如何初始化合使用PCI总线和系统中的PCI 设备。

中断和中断处理章节探讨了Linux核心如何处理中断。尽管操作系统核心里提供一些通用的机制和接口来处理中断，一些关于中断处理的细节是与硬件体系结构有关的。

Linux的优点之一是它支持许多硬件设备。设备驱动程序章节描述了Linux核心如何控制系统中的物理设备。

文件系统章节讲述了Linux核心如何维护其文件系统中的文件，描述了虚拟文件系统(Virtual File System)和如何支持Linux核心中的真正文件系统。

网络与Linux是一对同义词。从某种真实的意义上讲，Linux是Internet或World Wide Web (WWW)的产物。Linux的开发和使用者通过web来交换信息和代码。Linux=通常被用来支持机构的网络需求。网络章节描述了Linux如何支持TCP/IP协议族。

核心机制章节探讨了Linux核心需要支持/完成的一些通用任务和机制，以用来使得核心其他部份有效地工作在一起。

模块章节讲述了Linux如何动态地装载功能模块，例如文件系统。

源代码章节描述了对应于特定的核心功能的Linux核心源代码地址。读者可以依据本章的介绍来开始阅读源代码。

本书的约定

serif font 用户必须输入的命令或文本。

Type font 数据结构和数据结构中的域。

本书中所引用的源程序都基于相对路径/usr/src/linux。如foo/bar.c，则其实际的路径是/usr/src/linux/foo/bar.c。

商标

X Window System is a trademark of the X Consortium and the Massachusetts Institute of Technology.

感谢

I must thank the many people who have been kind enough to take the time to e-mail me with comments about this book. I have attempted to incorporated those comments in each new version that I have produced. Special thanks must go to John Rigby and Michael Bauer who gave me full, detailed review notes of the whole book. Not an easy task.

第1章硬件基础

一个操作系统必需紧密地和其支撑--硬件系统结合在一起。操作系统需要一些只能由硬件提供的一些服务。为了很好的理解Linux操作系统，读者需要明白一些低层的硬件知识。本章对当代PC系统的一些硬件作一个介绍。

1975年的1月，当“Popular Electronics”杂志在其封面上给出Altair 8080的照片后，一场革命就开始了。

Altair 8080当时的价格是非397美金。对于今天而言，其Intel 8080的处理器，256字节的内存，没有屏幕和键盘的配置是微不足道的。它的发明者Ed Roberts将其新的发明称之为“个人计算机”。

计算机爱好者看见了Altair得潜力并开始为它写软件和在其上为它配置硬件。对这些先驱者来说，这是一种摆脱呆版的基于批处理的大型机的自由。一些退学的在校生通过这些个人计算机一夜之间获得了巨大的财富。市场上出现了大量的硬件设备。软件hackers们非常高兴地为这些新机器编写软件。有趣的是，是IBM公司在1981年制造了当代PC--IBM PC并在1982年交给用户使用。当时IBM PC的配置是Intel 8088处理器， 64K内存(可扩展到 256K)，两个软磁盘和一个80字符，25行的CGA显示适配器。1983年，IBM推出了IBM PC-XT，含有一个10M的硬盘。不久之后，许多公司，如compaq，推出了IBM PC的兼容机。PC的体系结构变成了一个工业标准。这个工业标准使得大量的硬件公司在一个基础上进行竞争。从而使的PC价格变得越来越便宜。早期PC的许多结构特徵被现代PC所继承。例如，即使先进的Intel Pentium Pro系统在开始

时也运行在Intel 8086的地址模式下。当Linus Torvalds 开始写Linux时，他选择了当时最多的，价格也较合理的Intel 80386 PC。

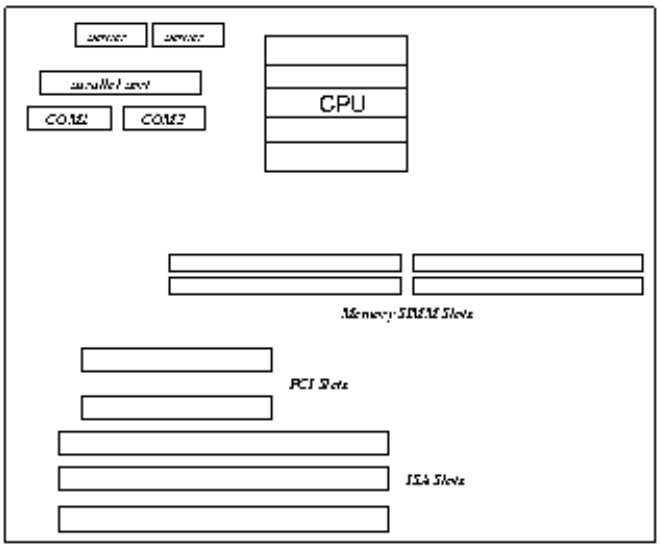

图1.1 一个典型的PC主板

从外面看一个PC，最明显的部件是一个系统主机，键盘，鼠标器和一个显示器。大多数系统还有CD ROM。如果你想要保护数据，还可以有一个磁带驱动器用来做数据备份。这些设备通称为外设。

虽然CPU是系统的主要控制部件，它不是系统中唯一具有智能的。所有的外设控制器，例如，IDE控制器，都有一定的智能成份。在一个PC的内部(如图1.1所示)，读者可以看见一个含有CPU的主板，内存和一些ISA或PCI外设控制器插槽。有些控制器，如IDE磁盘控制器，有可能被直接作在系统板上。

1.1 CPU

CPU，或微处理器，是任何一个计算机系统的心脏。CPU通过读取并执行内存中的指令来进行计算，执行逻辑运算和管理数量流。

(以下略去关于CPU的介绍，有兴趣的读者请参考有关书籍)

1.2 存储器

所有的系统都有一个存储器的层次结构。每一层的速度和大小不一样。最快的存储器是缓冲(cache)存储器，用来暂时存放主存储器的内容。这种存储器非常快但价格很贵，因此大多数处理器在芯片中只含有较

少的cache存储器。更多的则在系统板上。有些处理器在cache中混合存放数据和指令；有些则分开存放。一个cache为指令；其他一个 cache为数据。Alpha处理器中含有两个内部存储cache；D-cache为存放数据；I-cache为存放指令。外部的cache(B-cache)混合存放数据。最后，存储体系结构中的是主存储器。相对于外部cache,主存比较慢。如果与CPU内部的cache比较，主存的速度就象爬一样。

缓冲与主存之间必须保持一致性(coherent)。换句话说，如果主存中的一个字在cache中的一个或多个地方,系统必须保证cache中的内容与主存中的一致。cache一致性的工作一部份是有硬件完成的，一部份是有操作系统完成的。这一点对于大多数系统都是一样的。系统中的软硬件必须互相合作完成功能。

1.3 总线

系统板上的部件通过总线相连。系统总线份外三个逻辑功能部份：地址总线，数据总线和控制总线。地址总线指定数据传送的地址。数据总线负责运送要传送的数据. 数据总线是双向的。允许数据读进CPU和从CPU写出。控制总线包含一些信号线用来控制时序和系统中的其他控制信号。

1.4 控制器和外设

外设是真实的设备，例如显示卡或磁盘。外设都由系统主板上的控制器所控制。IDE磁盘由IDE控制器芯片控制；SCSI磁盘被SCSI磁盘控制器所控制。控制器之间，控制器与CPU通过总线相连。大多数系统通过PCI和ISA总线将系统的部件相连。控制器与CPU一样是一种处理器，可以看作是CPU的智能助手。CPU是系统的控制中心。

所有的控制器都不相同。但通常它们都有一些用来控制作用的寄存器。在CPU之上运行的软件必须能够读和写这些控制寄存器。一个寄存器可能包含一个用来描述错误的状态；另一个寄存器可能被用来作为控制，例如改变控制器的模式。总线上的每个控制器都可以分别地被CPU所访问。从而设备驱动程序软件可以写上述寄存器以控制这些控制器。

1.5 地址空间

系统总线将CPU与主存相连，这与将CPU与系统硬件外设相连的总线是分开的。总的来说，硬件外设所占据的存储空间叫做I/O空间。CPU可以既可以存取系统存储空间，也可以存取I/O空间。然而控制器只能在CPU的帮助下间接地访问系统主存。从设备的观点看，例如软盘控制器，它只能看见其控制寄存器所在的ISA空间的地址。一般而言，CPU使用不同的指令系统来存取系统存储器和I/O空间。例如，可能存在一个指令“从I/O地址0x3f0读一个字节到寄存器x”。CPU就是这样通过读和写来控制系统的硬件外设的。

在I/O空间中，一些常用的硬件外设(IDE控制器，串行口，软盘驱动器等)的寄存器地址已经被固定下来。例如0x3f0正好是串口一(com1)的一个控制寄存器。

有时候，控制器需要在系统主存之间传送大量的数据。例如将数据写入磁盘。这种情况下，直接储存存取(DMA)控制器被用来使得硬件外设直接存取系统内存。但是这个过程是在CPU的严格控制和监督之下。

1.6 定时器

所有的操作系统都需要知道时间。所以PC中含有一个特殊的设备叫做实时时钟(Real Time Clock)。RTC提供两种功能：每天的时间和准确的时间脉冲。RTC有其自己的电平。因此即使PC没开电源，RTC也在运行。这就是为什么你的PC的时间一直在更新的原因。RTC提供的时间脉冲使得操作系统可以准确的调度必须的工作。

第 **2** 章软件基础

程序就是一组执行特定任务的计算机指令。程序既可以用非常低级的计算机语言--汇编语言，也可以用高级的、独立于机器的语言如C来编写。操作系统是一种特殊的程序，它允许用户运行各种应用程序如制表程序和字处理程序。本章介绍基本的程序设计原理，并对操作系统的目标和功能做一综述。

2.1 计算机语言

2.1.1 汇编语言

CPU从内存中取出并运行的指令对人来说根本无法理解。它们是精确指示机器如何操作的机器代码。例如，十六进制数0x89E5是Intel80486的一条指令，把ESP寄存器的内容拷到EBP寄存器中。汇编器是最早发明的软件工具之一，它输入人类可以理解的源代码，汇编为机器代码。汇编语言显式地处理寄存器和数据操作，与特定的微处理器相关（应为与特定的处理器相关--译者注）。IntelX86微处理器的汇编语言就与Alpha AXP微处理器的汇编语言大相径庭。以下Alpha AXP汇编代码表示了程序可以进行的一种操作：

```
ldr r16, (r15) ; Line 1
ldr r17, 4(r15) ; Line 2
```

```
beq r16,r17,100 ; Line 3
str r17, (r15) ; Line 4
100: ; Line 5
```

第一条指令（见第一行）把寄存器15存放的地址中的内容装入寄存器16。下一条指令把内存中下一个位置的内容装入寄存器17。第三行把寄存器16和寄存器17的内容比较，如果相等，分支转向标号100。如果两个寄存器包含数值不等，程序继续运行第四行，把寄存器17的内容存到内存。如果两个寄存器包含数值相等，那么没有数据需要保存。编写汇编语言程序枯燥乏味、技巧性强而且易于出错。Linux核心只有很少的一点用汇编语言编写，目的是为了效率，这些部份是与特定机器相关的。

2.1.2 C语言和编译器

用汇编语言编写大型程序十分困难而且消耗大量时间。这样做易于出错，得到的程序也无法移植，限制在特定的处理器族上。用独立于机器的语言如C，会好得多。C允许你用逻辑算法和其操作的数据结构来描述程序。叫作编译器的特定程序读入C程序，并把它翻译成汇编语言，生成相应的机器代码。好的编译器所产生的汇编指令的效率接近于好的汇编语言程序员编写的汇编语言程序。大部份Linux核心是用C语言编写的。以下的C片段：

```
if (x != y)
x = y ;
```

与前一个例子中汇编代码的操作完全相同。如果变量x和y的内容并不完全相同，就把y的内容拷给x。C代码组织为例程，每一个例程执行一个任务。例程可以返回C支持的任何数值或者数据类型。象Linux核心这样的大型程序包含很多独立的C源模块，每个模块都有自己的例程和数据结构。这些C源代码模块把象文件系统处理这样的逻辑功能组合在一起。

C支持很多类型的变量。所谓变量，就是内存中的一个位置，可以用符号名字来引用。在以上C片段中，x和y指引了内存中的位置。程序员不关心变量究竟存放在内存中的何处，这是连接器（见下面所述）的任务。一些变量含有不同类型的数据、整数和浮点数，另一些则是指针。指针就是包含地址--其它数据在内存中的位置，的变量。考虑叫做x的变量，它可能处于内存地址0x80010000。你可以有一个指针，叫做px，指向x。px可能处于地址0x80010030，而px的值是0x80010000，变量x的地址。C允许你把相关的变量绑在一起，形成数据结构。例如，

```
struct {
int i ;
char b ;
} my_struct ;
```

是一个叫做my_struct的数据结构，它包含两个元素：叫做i的整数（32位数据）和叫做b的字符（8位数据）。

2.1.3 连接器

连接器是一种程序，它可以把几个目标模块和库连接在一起，产生一个独立的、自洽的程序。目标模块是汇编器或编译器生成的机器代码输出，含有可执行的机器代码和数据，以及允许连接器把模块连接起来的信息。例如一个模块可能含有程序中所有的数据库函数，而另外一个则含有命令行参数处理函数。连接器负责解决目标模块之间的引用，例如一个模块中引用的例程或数据结构事实上在另外一个模块之中。Linux核心就是一个与很多成员目标模块连接在一起的独立大程序。

2.2 什么是操作系统？

没有软件的计算机就是一堆发热的电子器件。如果说硬件是计算机的核心，那么软件就是计算机的灵魂。所谓操作系统，就是允许用户在其上运行应用软件的一组系统程序。操作系统对系统的真正硬件进行抽象，向系统的用户和应用程序给出一个虚拟机。在很现实的意义上说，软件提供了系统的特点。绝大部份PC能运行一个或多个操作系统，每一个擦系统都有一个完全不同的外观和风格。Linux是由一批功能上分离的部件组成，其中明显的一个是核心。但是即使是核心，如果脱离库和外壳程序也是没有用的。为了开始理解什么是操作系统，请考虑当你敲入以下的简单命令时会发生的情况：

```
$ ls
Mail c images perl
docs tcl
$
```

这里$是由登录外壳程序（在此例为bash）给出的提示符。这意味着它在等待你--用户，敲入命令。敲入ls后，键盘驱动程序识别出已经有字符输入。键盘驱动程序把这些字符传给外壳程序，外壳程序则通过寻找可执行程序的映象来处理这个命令。它在/bin/ls发现了映象，于是调用核心服务来把ls可执行程序的映象拖入虚拟内存，开始执行。ls的映象调用核心的文件子系统，以找出有哪些文件可以获得。文件系统有可能要使用放在cache中的文件系统的信息或者用磁盘驱动程序来从磁盘读出这些信息，甚至可能用网络驱动程序与远程机器交换信息，以找出本系统能够存取的远程文件的细节（文件系统可以通过"网络文件系统"NFS来远程mount）。无论是用哪种方式定位信息，ls都会把信息写出来，由视频驱动程序把它显示在屏幕上。以上看起来很复杂，但是说明了一个道理：即使是最简单的命令，也需要相当的处理，操作系统事实上是一组互相合作的函数，它们在整体上给用户以一个系统的完整印象。（以上一句是根据译者理解翻译的，未必忠实于原文--译者注）

2.2.1 内存管理

如果有无限的资源，例如内存，很多操作系统需要做的事情都是冗余的。操作系统的一个基本技巧是使一小块内存看起来象很多内存。这种表面上的大内存称为虚拟内存。其思想是使系统中运行的软件以为它在

很多内存上运行。系统把内存分成很多容易控制的页面，把一些页面交换到硬盘上。由于另外的一个技巧--多道处理，软件注意不到这一点。

2.2.2 进程

进程可以想象为一个活动中的程序。每一个进程是一个独立的实体，在运行一个特定程序。如果你看看你的Linux系统中的进程，你就会发现一大堆。例如，在我的系统中敲入ps可以显示如下进程：

$ ps

PID TTY STAT TIME COMMAND

158 pRe 1 0:00 -bash

174 pRe 1 0:00 sh /usr/X11R6/bin/startx

175 pRe 1 0:00 xinit /usr/X11R6/lib/X11/xinit/xinitrc --

178 pRe 1 N 0:00 bowman

182 pRe 1 N 0:01 rxvt -geometry 120x35 -fg white -bg black

184 pRe 1 <0:00 xclock bg grey geometry 1500-1500 padding 0 185 pRe 1 < 0:00 xload bg grey geometry 0-0 label xload 187 pp6 1 9:26 /bin/bash 202 pRe 1 N 0:00 rxvt geometry 120x35 fg white bg black 203 ppc 2 0:00 /bin/bash 1796 pRe 1 N 0:00 rxvt geometry 120x35 fg white bg black 1797 v06 1 0:00 /bin/bash 3056 pp6 3 < 0:02 emacs intro/introduction.tex 3270 pp6 3 0:00 ps $ 如果我的系统有很多CPU，每个进程（至少在理论上）可以运行在一个不同的CPU上。不幸的是，我只有一个CPU，所以系统只能求助于让每个进程轮流运行一小段时间的办法。这一小段时间称为时间片。这种技巧称为多道处理或者调度，它使得每个进程以为自己是唯一的进程。在进程之间进行保护，以便当一个进程崩溃或者错误时，不会影响其它程。操作系统给每个进程一个单独的、只有它自己能存取的地址空间，以达到这样的目的。

2.2.3 设备驱动程序

设备驱动程序构成了Linux核心的主要部份。就象操作系统的其它部份一样，设备驱动程序在高优先级的环境下运行，一旦发生错误就可能造成危险。设备驱动程序控制操作系统和其控制的硬件设备之间的互相作用。例如，在把块写到IDE硬盘时，文件系统使用一个通用块设备接口。驱动程序负责细节，控制与设备相关的事情。设备驱动程序是针对其控制的特定芯片的，所以如果你有一个NCR810 SCSI控制器，那么你就需要一个NCR810 SCSI驱动程序。 2.2.4 文件系统在Linux中，就象在Unix中一样，系统可以使用的不同的文件系统并非通过设备标识来存取（例如驱动器号或驱动器名），而是被组织在一个单一的分层树结构里，每个文件系统用一个实体来表示。文件系统通过把新的文件系统mount在某个目录下--例如 /mnt/cdrom，从而把新的文件系统加入树中。Linux的最重要特点之一是支持多个不同的文件系统，这使得其伸缩性好，易于与其它操作系统共存。最广为人知的Linux文件系统是 EXT2，受到所发行的绝大部份的Linux的支持。文件系统屏蔽了下层物理设备或文件系统类型的细节，给予用户察看硬盘上文件和目录的一个清楚视角。Linux透明地支持多种不同文件系统（例如MS-DOS和EXT2），把所有 mount上的文件和文

件系统都组织在一个虚拟文件系统之中。所以，一般而言，用户和进程并不需要知道哪个文件在哪个文件系统之中，只需要直接去用它就可以了。块设备驱动程序隐藏了物理块设备类型之间的差别（例如IDE和SCSI的差别），从文件系统的角度来看，物理设备只是数据块的线形聚集。不同设备的块大小不同，例如软盘通常是512字节，而IDE设备通常是1024字节，但是系统的用户是看不到这一点的。无论放在什么设备上，EX2文件系统看起来都一样。 2.3 核心数据结构操作系统必须保存关于系统当前状态的很多信息。在系统中发生了事情之后，这些数据结构必须修改，以反映当前的真实状况。例如，在一个用户登录到系统之后，可能会创建一个新的进程。核心必须创建表示新进程的数据结构，并把它和表示系统中所有其它进程的数据结构连接在一起。通常这些数据结构存在于物理内存之中，只能由核心及其子系统存取。数据结构包括数据和指针--其它数据结构的地址或例程的地址。总而言之，Linux使用的数据结构看起来很复杂难懂。虽然其中某些可能为几个核心子系统所使用，但是每个数据结构都有其目的，所以这些数据结构事实上比刚看起来要简单。（以上一句是根据译者的理解翻译，未必正确和符合原文--译者注）理解Linux核心依赖于理解其数据结构以及核心中各种函数对数据结构的使用。本书对 Linux核心的描述就是基于其数据结构的。本书讨论了每个核心子系统的算法、完成工作的方法和对核心数据结构的使用。 2.3.1 链表 Linux使用一系列软件工程技术来把数据结构连接起来。在很多情况下要使用链表。如果每个数据结构描述某事物的一个实例或一次发生，例如一个进程或一个网络设备，那么核心就必须能够找到所有的实例。在链表中，根指针含有表中第一个数据结构，或者称为“元素”，的地址，而每个数据结构都含有一个指向表中下一个元素的next指针。最后一个元素的next指针为0或NULL，以示它已经是表尾。在双链表中，每个元素含有指向表中下一个元素的next指针和指向表中前一个元素的previous指针。使用双链表方便了增加或删除表中间的元素，当然，内存的存取也增加了。这是一个典型的操作系统中的折中：消耗内存存取与消耗CPU周期的折中。 2.3.2 Hash表链表可以方便地把数据结构绑在一起，但是浏览链表效率很低。如果你想查找特定的一个元素，很可能回不得不看完全表才找到需要的那个。Linux使用Hash技术来避开这个限制。所谓Hash表，是一个指针的数组或者向量。这里说的数组或者向量，是指内存中的一组顺序存放的东西。因此，书架可以说成是书的数组。数组使用索引进行存取，而索引就是数组中位置的偏移量。把书架的比喻继续下去，你可以通过在书架上的位置来描述每本书。例如，你可以要求拿第5本书。所谓Hash表，是指向数据结构的指针数组，采用数据结构中的信息作为Hash表的索引。如果你有描述村庄人口的数据结构，那么你可以采用人的年龄作为索引。为了寻找某个特定人的数据，你可以采用年龄作为人口Hash表的索引，然后按照指针找到含有此人细节的数据结构。（这里的指针相当于普通教科书上所说的Hash函数：Hash(ad)="*ad，--译者" 注）不幸的是，很可能村中的许多人年龄相同，所以Hash表的指针成了指向一个数据结构链的指针，每个数据结构描述相同年龄的一个人。然而，搜索这些短链还是比搜索全部数据结构要快。由于Hash表加快了普通数据的存取，Linux经常使用Hash表来实现cache。cache通常是总体信息的一部份，被抽出来需要加速存取。操作系统常用的数据结构要放在cache中保存。cache的不利之处在于使用和维护都比简单链表或Hash表更加复杂。假如能在cache 中找到数据结构（称为“命中”），那么好得很。假如找不到，那么所有相关数据结构都要被搜索，如果最终能找到，那么该数据结构将被加入cache中。把新的数据结构加入 cache 可能会需要挤出一个旧的cache项入口。Linux必须决定到底挤出哪一个才好，以尽可

能避免恰恰挤出下面要用的数据结构。 2.3.3 抽象接口 Linux核心经常对接口进行抽象。所谓接口，是例程和数据结构的集合，它通过某种特定方式进行操作。例如所有的网络设备驱动程序必须提供操作某些特定数据结构的某些特定例程。这样，就能有使用低层特殊代码的通用代码层。例如网络层是通用的，受到遵循标准接口的与设备相关的代码的支持。通常这些低层在启动时注册到高层。注册时一般要把一个数据结构加到一个链表中。例如核心中内置的每个文件系统在启动时或者（如果使用模块的话）首次使用时注册到核心。通过查看文件/proc/filesystems能够看到哪些文件系统已经注册。注册数据结构通常包含函数指针。这些都是进行特定任务的软件函数的地址。再次用文件系统注册作为一个例子，每个文件系统在注册时传给Linux核心的的数据结构包含与文件系统相关的函数地址，这些函数在该文件系统mount时必须调用。

第 **3** 章内存管理

内存管理子系统是操作系统中最重要的组成部份之一。从早期计算机开始，系统的实际内存总是不能满足需求，为解决这一矛盾，人们想了许多办法，其中虚存是最成功的一个。虚存让各进程共享系统内存空间，这样系统就似乎有了更多的内存。

虚存不仅使计算机的内存看起来更多，内存管理子系统还提供以下功能：

扩大地址空间
操作系统扩大了系统的内存空间。虚存能比系统的实际内存大许多倍。

内存保护
系统中每个进程都有它自己的虚拟地址空间。这些虚拟地址空间之间彼此分开，以保证应用程序运行时互不影响。另外，虚存机制可以对内存部份区域提供写保护，以防止代码和数据被其它恶意的应用程序所篡改。

内存映射
内存映射被用于将映像和数据文件映射到一个进程的虚拟地址空间中，也就是将文件内容连接到虚地址中。

公平分配内存

内存管理子系统公平地分配内存给正在运行的各进程。

虚存共享

尽管虚存允许各进程有各自的 (虚拟) 地址空间，但有时进程间需要共享内存。

例如，若干进程同时运行 Bash 命令。并非在每个进程的虚地址空间中，都有一个Bash的拷贝。在内存中仅有一个运行的Bash拷贝供各进程共享。又如，若干进程可以共享动态函数库。

共享内存也能作为一种进程间的通信机制(IPC)。两个或两个以上进程可以通过共享内存来交换数据。Linux 支持 Unix 系统 V 的共享内存 IPC标准。

3.1 一个抽象的虚存模型

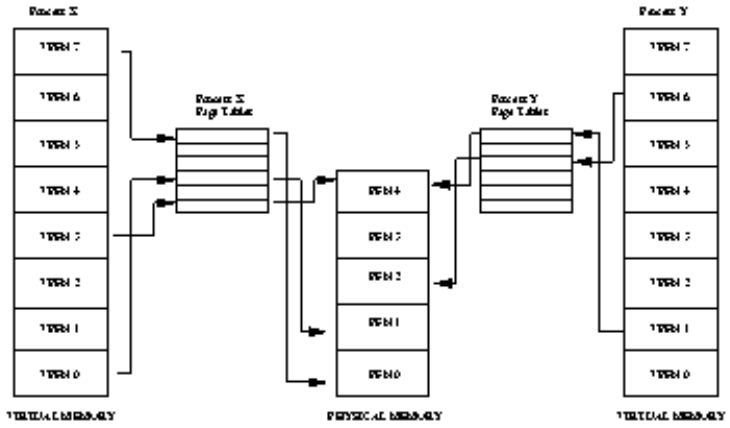

图 3.1 ： 虚地址与物理地址之间的映射

在分析 Linux 实现虚存的方法前，让我们先来看一个没有过多细节的抽象模型。

当处理器执行一段程序时，它先从内存中读出一条指令并对它进行解码。解码时可能需要在内存中的某一地址存取数据。然后处理器执行这条指令并移向下一条。可见处理器总是不断地在内存中存取数据或指令。

在虚存系统中，所有地址都是虚地址而非物理地址。处理器根据操作系统中的一组表格而把这些虚地址翻译成相应的物理地址。

为使这翻译的过程更容易，虚存和物理内存被划分成许多适当大小的块，叫做“页”(page)。为便于系统管理，这些页都是一样大小的。在 Alpha AXP 上的 Linux 系统中，每页有 8Kbyte，但在 Intel x86 系统中，每页有 4 Kbyte。每一页又被分配了一个各不相同的数字，叫页号 (PFN) 。

在本模型中，一个虚地址由两部份组成；偏移量和虚页号。如果页的大小是 4 Kbytes，那么虚地址的0至11位是偏移量，第12位以上是虚页号。每次处理器遇到虚地址时，它先取出偏移量和虚页号。然后，处理器把虚页号翻译成物理页号，再由偏移量得到正确的物理地址，最后存取数据。处理器需要使用页表来完成这整个过程。.

图 3.1 显示了两个进程的虚存地址空间。进程 X 和进程 Y 分别有各自的页表。页表记录各进程虚页和物理页之间的映射。如图：X 虚存的第0页对应物理地址的第4页。理论上，页表中每条记录包含以下信息：

-有效性标志。用以标识页表记录有效与否。
-页号。用以记录对应的物理内存页号。
- 存取控制信息。描述这页该怎样被使用。可否可写？可否包含可执行代码？

页表中使用虚页号作为偏移量。虚页 5 将是表中的第 6 条记录(0 是第一条记录)。

把一个虚地址翻译成物理地址时, 处理器必须先得出虚页号和偏移量。页的大小总是 2 的幂，这便于进行mask和移位操作。图 3.1中，假定页的大小是 0x2000 字节 (它是十进制的 8192)，在进程 Y的地址空间中有一虚地址 0x2194。那么处理器将把这个地址翻译成偏移量为 0x194，虚页号为1。

处理器使用虚页号作为检索进程页表记录的索引。如果对应那偏移量的页表记录是有效的，处理器就从中拿出物理页号。如果记录是无效的，表明进程想存取一个不在物理内存中的地址。在这种情况下，处理器不能翻译这虚地址，必须把控制权传给操作系统，让它处理。

当进程试图存取一个无法翻译的虚地址时，处理器将通知操作系统，这被称为一个页错。各种处理器处理页错的方法是不同的，但都会通知操作系统产生页错的虚地址和原因。

假设找到的是一有效的页表记录，处理器就取出物理页号并且乘以页的大小，得到内存中页的基地址。最后，处理器加上偏移量得到它需要数据的地址。

例如, 进程 Y 的虚存第1页被映射到内存第 4页，它从 0x8000（4 x 0x2000)开始。加上偏移量 0x194 字节就得到最后的物理地址是 0x8194。

由虚地址映射到物理地址时，虚存各页映射到系统内存中的顺序是任意的。例如, 在图 3.1 中，进程 X 的虚存第 0 页被映射到内存第1页，而虚存第 7页被映射到内存第 0页。这说明了虚存的一个有趣现象，虚存各页在物理内存中不必有任何顺序。

3.1.1 按需装载页(Demanding Paging)

虚存比实际内存大很多，所以操作系统一定要小心有效地使用内存。节省内存的一个方法是只装载被当前执行程序使用的虚页。例如，有一个用来查询数据库的程序。此时，并非所有数据库中的数据都需被装载进内存，只需要那些正在被访问的数据。如果正运行一条数据库搜索命令，那么就不必载入添加新记录的代码。当代码或数据被访问时才装载进内存，这叫作按需装载页(demand paging)。

当进程试图存取一个不在内存中的虚地址时，处理器不可能在页表中找到这一虚页的记录。例如，在图 3.1中，进程 X 的虚存第 2页没有对应的页表记录，如果尝试对这页进行读操作，那么处理器不能把虚地址翻译成物理地址。处理器就会通知操作系统页错发生了。

如果页错(faulting) 对应的虚地址是无效的，这意味着进程试图存取它不应该访问的虚地址。这也许是因为应用程序出了某些错误，例如试图在内存中任意进行写操作。在这种情况下，操作系统将终止这个错误进程，以保护其它进程。

如果页错(faulting) 对应的虚地址是有效的，只是它所在页目前不在内存中，操作系统必须将对应的页从磁盘载入内存。相对来说，磁盘存取会花很多时间，所以进程必须等待相当一会儿直到页被读入。这时候，如果有其它进程能运行，操作系统将选择其中之一。被取的页将被读入内存一空页中，并在进程页表中加入一条记录。然后，进程从产生页错的机器指令重新启动。这次处理器能将虚地址翻译成物理地址了，因此进程能继续运行下去。

Linux 使用按需装载页来读入可执行进程的映像。一个命令被执行时，包含它的文件被打开，它的内容被印射入进程的虚存。这操作需修改描述这进程内存映像的数据结构 (memory mapping)。然而，只有映像的第一部份被实际载入物理内存，余下部份被留在磁盘上。当映像执行时，它将不断产生页错， Linux 使用进程的内存映像表来决定哪块映像该被载入内存。

3.1.2 页交换 (Swapping)

当进程要装载一虚页进物理内存时，如果得不到空页, 操作系统必须从内存中丢弃别的页，为这页提供空间。

如果从内存中被丢弃的那页是从映像或数据文件中来的，并且映像和数据文件没被修改过，那这页不需再被保存，可以直接丢掉。如果进程再需要那页，它可以重新被从映像或数据文件中读入内存。

但如果该页已被修改了，操作系统必须保存这页的内容以便它以后能再被访问。这类页叫作脏 (dirty) 页，当它们被从内存中移出时，它们被作为特殊的交换文件 (swap file) 保存。相对于处理器和内存的速度，交换文件的存取时间是很长的，所以操作系统必须权衡是否需要把页写到磁盘上，还是保留在内存中以备后用。

如果交换算法的效率不高，那么thrashing现象就会发生。在这种情况下，页常常一
会儿被写到磁盘上，一会儿又被读回来，操作系统忙于文件存取而不能执行真正的工作。例如，图3.1 中，如果内存第 1页不断被访问，那它就不应该被交换到硬盘上。进程当前正在使用的页的集合被叫作工作集 (working set)。有效的交换算法将保证所有进程的工作集都在内存中。

Linux 使用最近最少使用算法(Least Recently Used) 来公平选择从内存中被丢弃的页。这个算法中，当页被存取时，它的年龄 (aging) 就变化了。页越多被存取，便越年轻；越少被存取就越旧。旧页通常是被丢弃的好候选。

3.1.3 共享虚存

虚存使得若干进程更容易共享内存。进程所有的内存访问都要通过页表，并且各进程有各自独立的页表。当多进程共享内存中一页时，物理页号就会同时出现在每个进程的页表中。

图3.1 中显示两进程共享物理第4页。对进程 X 而言，那是虚存的第 4页，对进程 Y而言，
那是虚存第 6页。这说明一个有趣的现象：被共享的物理页对应的虚存页号可以各不相同。

3.1.4 物理和虚拟地址模式

把操作系统运行在虚存中是不明智之举，如果操作系统还要为自己保存页表，那将是一场恶梦。因此，很多种处理器同时支持虚拟地址模式和物理地址模式。物理地址模式不需要页表，处理器不必做任何地址翻译。 Linux 内核被直接连在物理地址空间中运行。

Alpha AXP 处理器没有物理地址模式。相反，它把内存划分成若干区域并且指定其中两块为物理地址区。这段核地址空间叫作KSEG，包括所有0xfffffc0000000000以上的地址。在 KSEG执行的 (按定义，核代码) 或在那里存取数据的代码肯定是在核模式下执行。在 Alpha 上的 Linux 核被连接从0xfffffc0000310000开始执行。

3.1.5 存取控制

页表记录中也包含了存取控制信息。处理器使用页表记录来把虚地址翻译成物理地址的同时，它也很容易地使用其中的存取控制信息来检查进程是否在正确地访问内存。

在很多种情况下，你想要为内存的一段区域设置存取限制。一段内存，例如包含可执行的代码, 应为只读内存；操作系统应该不允许进程在它的可执行的代码上写数据。相反的，包含数据的页能被写，但是当指令试图执行那段内存时，应该失败。大多数处理器的执行代码有两种模式：核态和用户态。你将不想由一个用户执行核代码，或者让核数据结构被不是核态执行的代码所访问。

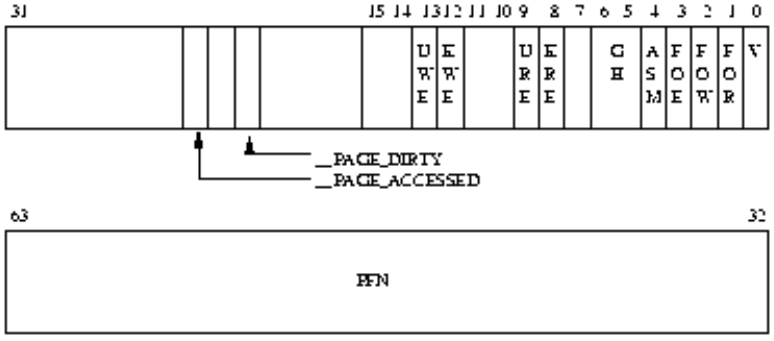

图 3.2 ： Alpha AXP 的页表记录 (Page Table Entry)

存取控制信息被保存在 PTE中，并且不同的处理器，PTE的格式是不同的；图3.2 显示的是 Alpha AXP 的PTE。各位包含以下信息：

V 有效位。如果设置，表示这 PTE 是有效的。

FOE (Fault on Execute) 无论何时试图在这页执行指令时，处理器将报告页错，
并且把控制权传给操作系统。

FOW (Fault on Write) 当在这页上进行写操作时报页错。

FOR (Fault on Read) 当在这页上进行读操作时报页错。

ASM(Address Space Match) 地址空间匹配。当操作系统仅仅希望清除翻译缓冲区中
若干记录时，这一位被使用。

KRE 在核模式下运行的代码能读这页。

URE 在用户模式下运行的代码能读这页。

GH 粒度性，指在映射一整块虚存时，是用一个翻译缓冲记录还是多个。

KWE 在核模式下运行的代码能写这页。

UWE 在用户模式下运行的代码能写这页。

页号 在有效的PFE中，这域包含对应的物理页号 (page frame number)。对无效的PTEs ，如果这域不是零，
它包含了页在交换文件中的信息。

以下两位是 Linux 定义并使用的：

_PAGE_DIRTY 如果设置，页需要被写到交换文件中。

_PAGE_ACCESSED 由 Linux 标记这页是否曾被访问。

3.2 缓存

如果你按照上面理论模型，可以实现一个工作的系统，但不会特别高效。操作系统和处理器的设计者都在
努力提高系统性能。除提高处理器和内存的速度外，最好的途径是把有用的信息和数据保存在缓存中。
Linux 就使用了很多与内存管理有关的缓存：

缓冲区

缓冲区包含块设备驱动程序 (block device driver) 使用的数据缓冲区。

这些缓冲区有固定的大小 (例如 512 个字节) ，记录从一台块设备读或写的信息。一台块设备只能存取整块数据。所有的硬盘都是块设备。

缓冲区通过设备标识符和需要的块号的索引来迅速发现所需数据。块设备只能通过缓冲区进行存取操作。如果数据在缓冲区中，那么它就不需要再从块设备中被读(例如硬盘)，这样存取得更快。

页缓存

它被用来加快磁盘上映像和数据的存取。

它被用来一次缓存文件的一页，存取操作通过文件名和偏移量来实现。当页从磁盘被读进内存时，他们被缓存在页缓存中。

交换缓存

只有修改了的页，即脏(dirty) 页，被保存在交换文件中。

只要一页在被写进交换文件以后，没有再被修改，下次这页被换出内存时，可以直接被扔掉。对一个进行许多页面交换的系统，这将节省许多不必要的并且昂贵的磁盘操作。

硬件缓存

处理器中有一经常用到的硬件缓存：页表记录的缓存。通常情况下，处理器并不总是直接读页表，而是用页表缓存保留用到的记录。这些被叫做 Translation Look-aside Buffer，保存了系统中多个进程页表的拷贝。

当翻译地址时，处理器先试图找到一匹配的TLB 记录。如果它发现了一个，它能直接把虚地址翻译成物理地址，并且对数据进行存取操作。如果处理器不能发现一匹配的 TLB 记录，那就必须借助操作系统。它发信号给操作系统，报告有一个 TLB 疏漏。特定的机制将把异常信号送给操作系统的代码。操作系统为印射

的地址产生一个新的 TLB 记录。当异常被解决后，处理器将尝试再翻译那个虚地址。因为现在那个地址在 TLB 中有一个有效的记录，这次的地址翻译一定成功。

使用缓冲区，硬件缓存等的缺点是Linux 必须花费更多的时间和空间来维护这些缓存，如果缓存发生错误，系统将崩溃。

3.3 Linux 页表

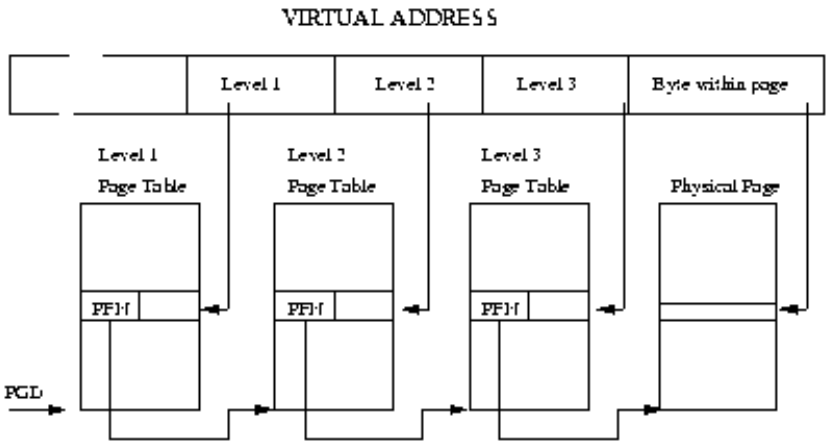

图 3.3： 3级页表

Linux页表有3层。每一层负责保存下一层页表所在的页号。图3.3 显示一个虚地址被分成了很多域；每个域记录在某一层页表中的偏移量。把一个虚地址翻译成物理地址时，处理器拿出每个域的内容把它变成页表中的偏移量，进而读出下层页表的所在页号。这样重复 3 次直到找到包含虚地址的物理页号。虚地址的最后一个域，叫做字节偏移量, 被用来在物理页内找到所需数据。

每个运行 Linux 的平台必须提供翻译宏(Translation macros) 以便内核可以检索页表，完成某种操作。这样，内核不需要知道各平台上页表记录的具体格式和它们是怎么被安排的。

这就是为什么 Linux 的 Alpha 处理器和Intel x 86 处理器使用一样的页表操作代码，而Alpha有3层页表，Intel x86处理器只有2层页表。

3.4 页的分配和回收

在系统中，对页有许多操作。例如, 当一段映像被装载进内存时，操作系统需要分配页。当映像执行完成并且被卸掉时，这些页将被释放。页的另外的用途是保存内核特定的数据结构，例如页表。页的分配和回收机制是维持虚存分系统效率的关键。

系统中所有物理内存页由 mem_map 数据结构描述，men_map由一列 mem_map_t 组成。在初始化时，每个 mem_map_t 描述系统中的一页。它重要的域如下(有关内存管理) ：

计数器 描述使用这页的用户数。如果计数器比一大，则这页被多进程共享。
年龄 描述页的年龄，被用来决定页是否是被丢弃或交换的好候选。
map_nr 描述这个 mem_map_t 对应的页的物理页号。

页分配代码使用矢量 free_area 来寻找并释放页。这机制支持整个缓冲区管理，对于代码来说，页的大小和处理器对页的操作机制是与其无关的。

free_area 每个单元都包含一种页块的信息。在数组的第一单元描述单个的页, 下一单元描述 2 页块，再下一单元描述 4 页的块，并以2的幂上升。表中每个单元作为一个队头，有指针指向mem_map 数组中的页。空的页块在这里排队。map是指向bitmap的一个指针，bitmap 记录了这种大小页块的分配情况。位图中，如果第 n 块页是空的，那么位 N 被置。

图 free_area_figure 显示的是 free_area 的结构，第0单元记录有一个空页，从第0页开始。第2单元记录有两个4页的空块，第一块从第4页开始，第二块从第56页开始。

3.4.1 页的分配

Linux 使用伙伴(Buddy) 算法来有效地分配和回收页块。页分配代码被用来分配一页或多页的块。页的大小总是 2 的幂，即能分配1页, 2 页, 4 页等等。只要系统中有足够满足请求的空页 (nr_free_pages >min_free_pages)，分配代码就能在 free_area 里找到所需大小的页块。free_area每个单元有一张分配图(bitmap)。例如，数组的单元 2 有描述长度为4的页块的分配图。

算法寻找所需大小的页块时，它先搜索 free_area 数据结构中那种页块的队列。如果所需大小的页块没有空，就在下一对列中寻找(页块的大小是所需的两倍)。继续这一过程直到free_area 中所有单元都被找过了或发现了一空页块。如果找到的空页块比所需的大，它必须被分割成正确的大小。

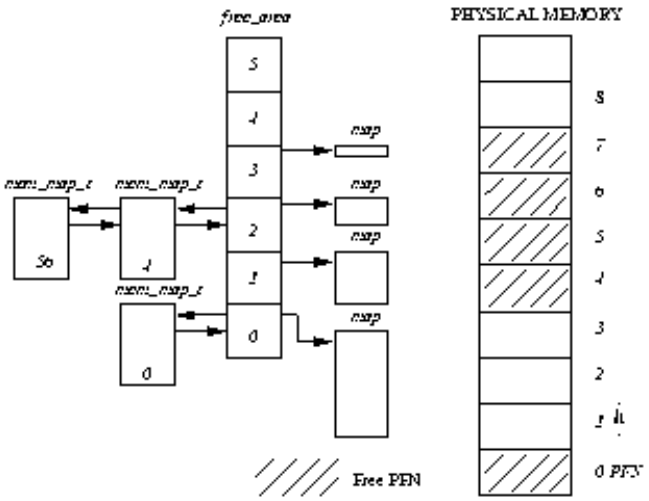

图 3.4： free_area 数据结构

例如, 在图3.4 中，如果需要一 2 页块，那么第一个空的 4 页块 (从第4页起) 将被分成两半。从第4页开始的 2 页块被返回给请求者；从第6页开始的 2 页块被排在free_area的空的两页块的队中。

3.4.2 页的回收

页分配时容易将大块连续的内存分成很多小块。页的回收代码须尽可能将小块的空内存重新组合成大块的。事实上，页块的大小对内存的重新组合很重要。

当一页块被释放时，系统会检查它旁边的和一样的大小的页块，看它们是否是空的。如果是，它们将被拼成一个大的整块。每次当两块内存被拼成了更大的空块时，页回收代码尝试将它们与其它空块继续组合，以得到更大的空间。这样得到的空页块可以满足任何对内存的需求。

例如，在图 3.1中，如果第 1 页被释放，那它将与第0页结合，并被放到 free_area 的两页空块的队中。

3.5 内存印射

当一映像被执行时，它的内容必须被读入进程的虚存。它调用的库函数也必须被读入虚存。这个可执行文件并非被实际读入内存, 相反它只是被连接入进程的虚存。然后，当程序的一部份被应用程序调用时，系统才将这部份映像读入内存。将映像连接到进程的虚地址空间叫做内存印射(memory mapping)。

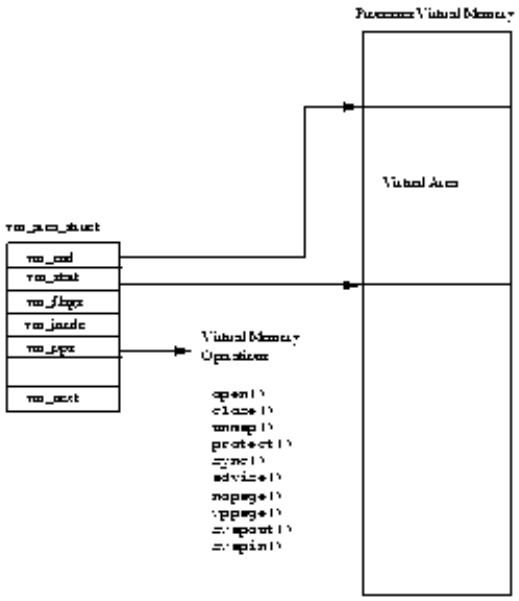

图 3.5 ： 虚存

每个进程的虚存空间由一个 mm_struct 数据结构表示。这包含当前正在执行的映像的信息 (例如 Bash)，还有很多指向 vm_area_struct 的指针。每个 vm_area_struct 数据结构
描述一段虚存区域的开始和结束，及进程对那段虚存的存取权限和允许的操作。这些操作是Linux 对这段虚存必须使用的一套例程。例如，当进程试图存取虚存中某页，但发现这页并不在内存中时，应执行的正确操作是 nopage 操作(通过页错)。 Linux使用nopage 操作可以按需将一页可执行映像载入内存。

当一段可执行映像被印射入进程的虚存时，会产生一组 vm_area_struct 数据结构。每个 vm_area_struct 数据结构代表可执行映像的一部份;可执行代码, 初始化数据 (变量)，未初始化数据等等。 Linux 支持很多标准的虚存操作，当 vm_area_struct 数据结构产生时，系统会把正确的虚存操作集与他们相联。

3.6 按需换页 (Demanding Paging)

当一部份可执行映像被印射入进程虚存后，它就可以开始执行了。可是这时只有映像的开始部份被实际读入内存，它将不断访问不在内存中的部份。当进程存取一个没有有效页表记录的虚地址时，那处理器将报页错给 Linux 系统。

页错描述页错发生的虚地址和引起的存取操作。

Linux 必须先找到代表页错发生区域的 vm_area_struct。由于搜索 vm_area_struct 数据结构对高效处理页错非常关键，所以所有 vm_area_struct 被连接成AVL树结构 (Adelson-Velskii and Landis)。如果没有 vm_area_struct 代表这页错发生的虚地址，表示这进程企图访问一个非法的虚地址。Linux 将发送 SIGSEGV 信号给进程，如果进程没有对应这个信号的处理程序，它将被终止。

Linux 再检查存取操作是否是被允许的。如果进程在用一个非法的方法存取内存，例如，写一个只读区域，它也将引起一个内存错误信号。

如果 Linux 确定页错是合法的, 它就会处理它。

Linux 必须首先区别映像是在交换文件中还是在磁盘上。它是通过页表记录来区别的。

如果那页的页表记录是无效的，但非空，说明产生页错的那页当前在交换文件中。例如， Alpha AXP 页表记录中，这样的记录有效位未置，但是PFN 域不为零。在这种情况下， PFN 域容纳的信息表示这页被保持在哪个交换文件中的哪里。本章后半部将讲述怎样处理在交换文件中的页。

并非所有的 vm_area_struct 数据结构都有一组虚存操作，即使有，也不一定有nopage 操作。缺损情况下，Linux 将分配一页新内存，并为这页增加一项页表记录。但如果这段虚存有 nopage 操作，Linux 将使用它。

通常 Linux 的 nopage 操作被用于把可执行映像通过页缓存读入内存。

当页被读入内存后，进程的页表将被更新。特别是如果处理器使用TLA 缓冲区的话，它可能需要通过硬件操作来完成更新。页错被处理后，进程在产生页错的指令处重新开始执行。

3.7 Linux 页缓存

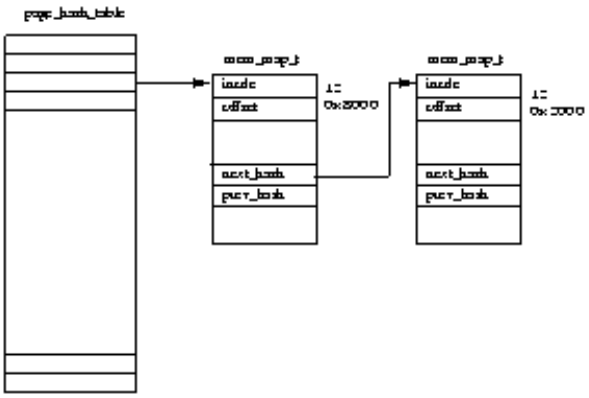

图 3.6： Linux 页缓存

Linux 页缓存的作用是加快从磁盘上存取文件的速度。每次系统读取文件的一页并将它放在页缓存中。图 3.6 显示页缓存包括 page_hash_table，它是一组指向 mem_map_t 的指针。

Linux 的每个文件由一 VFS inode 数据结构表示 (请参看文件系统章)，并且每个 VFS inode 是唯一的并且描述一个且仅一个文件。页表中的索引包括了文件的 VFS号及其在文件中的偏移量。

当从印像文件中读一页时，例如，按需装载一页回内存时，读操作将通过页缓存。如果页在缓存中，一个指向它的 mem_map_t 指针将被返回给处理页错的代码。否则，这页必须被从文件系统中读入内存。 Linux 需分配一页内存并从磁盘上读文件。

如果可能， Linux 将开始读文件的下一页。向前多读一页意味着如果进程是连续地访问文件，那么下一页将等在内存中。

页缓存中的内容将随着文件的存取而越来越多。当他们不再被任何进程使用时，这些页将被从缓存中移出。当 Linux 的空闲内存变得很少时，Linux 将减少页缓存的大小。

3.8 页的交换和释放

当空内存变得很少时， Linux 内存管理系统必须释放一些页。这任务由内核交换程序来完成(kswapd)。

内核交换程序是一种特殊的进程，是一个核线程。核线程是没有虚存的进程，他们在物理地址空间以核模式运行。内核交换程序不仅把页交换到系统的交换文件中，它的角色是保证系统有足够的内存而使内存管理系统可以高效工作。

内核交换程序被内核 init 进程在初始时启动，并等待内核交换定时器周期性地到期时开始运行。

每次定时器到期，内核交换程序检查系统中的空页数是否变得太低。交换程序使用两个变量，free_pages_high 和 free_pages_low 来决定是否它应该释放一些页。只要系统的空页数大于 free_pages_high，内核交换程序不做任何事情；它继续休息直到定时器再次到期。在做这项检查时，交换程序计算了正在往交换文件中写的页数。每次有一页等待写入交换文件时，计数器加1，当操作结束后，计数器减1。free_pages_low 和free_pages_high 在系统开始时被设置，并且与系统内存的页数有关。如果系统的空页数小于 free_pages_high 或甚至小于 free_pages_low , 核交换驻留程序将尝试 3 种方法以减少系统使用的页数：

减少缓冲区和页缓存的大小
换出系统 V 的共享页
换出并释放一些页

如果系统的空页数小于 free_pages_low , 核交换程序在它下次运行以前，将尝试释放 6 页，否则它将尝试释放 3 页。上面的方法将依次被使用直到有足够的页被释放。核交换程序将记住上一次它是用什么方法释放内存的，下一次将首先使用这个成功的方法。

在系统有足够的空页后，交换程序将休息直到它的定时器到期。如果上次空页数小于free_pages_low，它只休息一半时间。直到空页数多于 free_pages_low，核交换程序才恢复休息的时间。

3.8.1 减少页缓存和缓冲区的大小

在页缓存和缓冲区中保存的页是被释放的最佳候选。页缓存保存着内存映像文件，很可能包括了许多没用的页。同样，缓冲区中，它保存读写物理设备的数据，也很可能包含许多不需要的数据。当系统的内存页快用完时，从这些缓存丢弃页是相对容易的 (不同于从内存交换页)，因为它们不需要写物理设备。丢弃这些页除了使访问设备和内存的速度减慢一些以外，没有其它的副作用。并且如果对各进程公平对待的话，对各进程的影响是相同的。

每次内核交换程序尝试缩小这些缓存时，它先检查在 mem_map 中的页块，看是否有页可以被从内存中释放。如果内核交换程序经常作交换操作，也就是系统空页数已经非常少了，它会先检查大一些的块。页块会被轮流检查；每次减少缓存时检查一组不同的页块。这被称作时钟算法，像钟的分针一样轮流检查 mem_map 中的页。

检查一页是看它是否在页缓存或缓冲区中。应该注意共享页在这时候不能被释放，并且一页不能同时在两个缓存中。如果页不在任何一个缓存中，那么就检查 mem_map 中的下一页。

页被缓存在缓冲区中 (或页内的缓冲区被缓存)是为更有效地分配和回收缓存。缩小内存代码将尝试释放被检查页中的缓冲区。

如果所有的缓冲区都被释放了，那么对应它们的内存也就被释放了。如果被检查的页在 Linux 页缓存中，它将被从页缓存中移出并释放。

当足够的页被释放后，内核交换程序将等到下一个周期再运行。因为释放的页都是进程的虚存部份 (他们是被缓存的页), 所以没有页表记录需要更新。如果没有释放足够的页，那么交换程序将试着释放一些共享页。

3.8.2 交换出系统 V 的共享页

系统 V 共享内存提供了进程间的通信机制。进程间如何共享内存，请参看IPC章。系统 V 的共享区域被描述成一个 shmid_ds 数据结构。这包含一根指向一组 vm_area_struct 数据结构的指针，每个 vm_area_struct 对应共享这区域的一个进程。vm_area_struct 数据结构描述了每个进程在各自虚存的哪里共享系统 V 的这个区域。每个 vm_area_struct 由 vm_next_shared 和vm_prev_shared 指针相连。每个 shmid_ds 数据结构还包括一组页表记录，描述这些共享页是对应内存中的哪些页。

内核交换程序也使用时钟算法来换出系统 V 的共享页。每次它运行时，它记得上次换出的是哪个共享页。它将其记录在两个索引中，第一个是 shmid_ds 数据结构的索引, 第二个是系统的这段共享内存的页表记录的索引。这保证它公平地对待系统 V 的所有共享页。

由于共享页的物理页号在每一个共享进程中都有记录，内核交换程序必须修改这些页表，显示页已不在内存中了，而被保存在交换文件中。对于每个换出的共享页，内核交换程序是顺着 vm_area_struct 的指针找到这共享页在各个进程中的页表记录。如果这共享的系统 V 的页对应的页表记录是有效的，交换程序将把它

改成无效，标为在交换文件中，再将对应这页的计数器减1。被换出的系统 V 的共享页仍包括两个索引，第一个是 shmid_ds 数据结构的索引, 第二个是系统中共享这段内存的进程的页表记录的索引。

如果各进程的页表修改过后，页的计数器变成0，那么这页就可以被写入交换文件了 shmid_ds 中各页表记录的值将变为交换文件中的地址，在交换文件中的页的记录包括其对应交换文件的索引和偏移量。当这页要被重新读回内存时，这些信息将被使用。

3.8.3 换出及释放(进程的)页

交换程序检查系统中每一个进程，看它们是不是好的候选。好的候选是那些能被换出的进程或那些能从内存中换出并释放若干页的进程。只有当这些页不能从其它地方得到时，它们才会被写进交换文件。

许多映像的内容是可以从映像文件中读出的。例如, 一段映像的可执行指令决不会被修改，所以不用被写进交换文件。这些页能被直接释放；当他们再被进程调用时，他们将被从可执行映像中重新读入内存。

一旦确定了换出的进程，交换程序将检查它所有的页表记录，找出不是共享或被锁的区域。

Linux 并不换出它所选择进程的所有可交换页；相反它仅移出其中的一小部份。

如果页在内存中被锁住了，它们就不能被换出或释放。

Linux 交换算法使用页的年龄 (aging)。每页有一个记数器 (保持在 mem_map_t 数据结构中)，告诉交换程序是否应将它移出。当它们闲置时，页会变老；当被访问时，页变年轻。交换程序仅仅移出旧页。缺省状态下，当一页被分配时，起始年龄是3，每次它被访问，它的年龄从 3 增加直到最大值 20。每次内核交换程序运行时，它把所有页的年龄数减1。这些缺省操作都能被改变，它们被存储在 swap_control 数据结构中。

如果页是旧的 (年龄 = 0)，交换程序就进一步处理它(将它移出内存)。脏页也可以被移出。Linux 用PTE中的特定位来标示 (见 3.2图)。然而, 并非所有的脏页必须被写进交换文件。进程的每个虚存区域都可以有它们自己的交换操作 (由 vm_area_struct 中的 vm_ops 指出)，这个特定的操作将被调用。否则，交换程序将分配一页交换文件，并将那页写到磁盘上。

页对应的页表记录将被改为无效，但包含了它在交换文件中的信息，它将指出是哪个交换文件，并且偏移量是多少。无论采取什么交换方法，原来的物理页将被放回 free_area。.乾净的 (not dirty) 页可以直接被释放并放回 free_area以备后用。

如果有足够的页被换出或释放, 交换程序就又开始休息。下一次它运行时，它将检查系统中的下一个进程。这样，交换程序对每个进程都移出几页，直到系统内存恢复正常，这比移出一整个进程来的公平。

3.9 交换缓存

当将页移入交换文件中时，并非所有情况，Linux 都需进行写操作。有时一页既在交换文件中，又在内存中。这种情况是由于这页本来被移到了交换文件中，后又因为被调用，重又被读入内存。只要在内存中的页没被修改过, 在交换文件中的拷贝仍然是有效。

Linux 使用交换缓存来记录这些页。交换缓存是一张页表记录的表，每条记录对应一页。每条页表记录描述被换出的页在哪个交换文件中及其在文件中的位置。如果一交换缓存记录非零，表示在交换文件中的那页没被修改过，如果页被修改了(被写)，它的记录将被从交换缓存中移出。

当 Linux 需要移出一页内存到交换文件中时，它先查询交换缓存, 如果这页有一个有效的记录，它就不需要把页写到交换文件中了。因为自从它上次被从交换文件中读出后，在内存中没被修改过。

交换缓存中的记录描述已被移到交换文件中的页。它们被标为无效，但是告许Linux 页在哪个交换文件以及在交换文件的哪一页。

3.10 移入页

保存在交换文件中的脏页可能会被再次调用。例如，一个应用程序要将某些内容写到一已移出的页中。当这页被换到交换文件中时，描述这页的页表记录已被标记为“无效”。这样，存取不在内存中的虚地址将引起页错。页错是由处理器发信号给操作系统，告诉操作系统它不能把某个虚地址翻译成物理地址，并告之引起页错的虚地址及原因 (不同的处理器是用不同的格式传递这些信的)，同时，处理器把控制权交给操作系统。

操作系统用特定(与处理器有关)的代码来找到引起页错的虚地址对应的 vm_area_struct 数据结构。在这个过程中，系统检索进程所有的 vm_area_struct。这段代码对时间的要求很高，所以vm_area_struct 应被合理组织起来，以缩短所需的时间。

系统执行了以上操作，证实了引起页错的虚地址是有效的后，处理页错的其它代码是与处理器无关的。

下一步，(系统)处理代码寻找虚页对应的页表记录。如果它发清b页表记录指示这页在交换文件中, Linux 就把这页读回内存。页表记录的格式因处理器的不同而各不相同，但“有效位”都应该是无效，并都保存着有关这页在交换文件中的信息。Linux 需要利用这些信息来把页重新载入内存。

此时，Linux 知道了引起页错的虚地址及其对应的页表记录，记录中保存着有关交换文件的信息。而将页从交换文件中读回内存的函数通常由vm_area_struct 中的指针指向。这种函数叫移入(swapin) 函数。如果能从vm_area_struct 中找到这函数，Linux 就会调用它。例如，因为系统 V 的页的格式与一般的页不同，所以系统 V 中移出的页需要特殊处理，这时就需要调用它们的移入函数。然而，某页可能没有对应的移入函数，在这种情况下， Linux 将认为它是一普通的页，而不需要做任何特别处理。

系统将分配内存中的一空页并从交换文件中把这页读回来，交换文件中的地址信息是从无效的页表记录中取回的。

如果引起页错的不是写操作，那么这页将被留在交换缓存中，它的页表记录不会被标为“可写”。如果后来这页被写了，那么会产生另一个页错，这时，页被标成“dirty”，并被从交换缓冲中删去。如果这页没被修改过，而它又需要被换出，Linux将不会再把这页写到交换文件中，因为它已经在那儿了。

如果引起页错的是写操作，页将被从交换缓存中删除，它的页表记录将被标成“dirty”和“可写(writable)”。