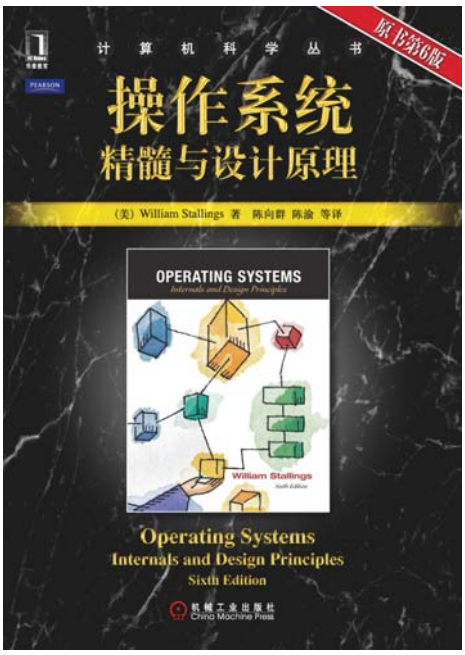
迷　你　书

——《操作系统：精髓与设计原理》



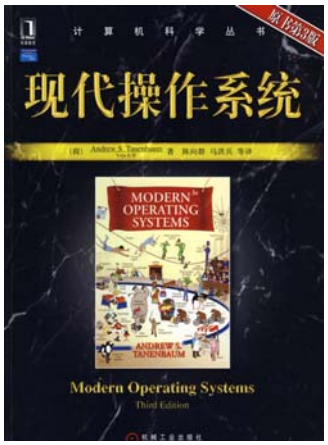
作者：William Stallings

译者：陈向群　陈渝等

ISBN：978-7-111-30426-5

定价：69.00

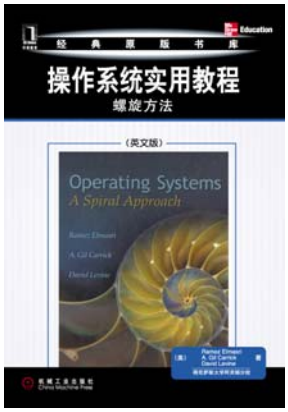
出版社：机械工业出版社



《现代操作系统：原书第 3 版》

- 操作系统领域的经典之作。
- Tanenbaum 教授作为三种操作系统的设计师或联合设计师。
- 在线操作系统练习：采用主流 Windows 操作系统以及开源工具。

ISBN：978-7-111-25544-4
定价：75.00



《操作系统实用教程：螺旋方法》

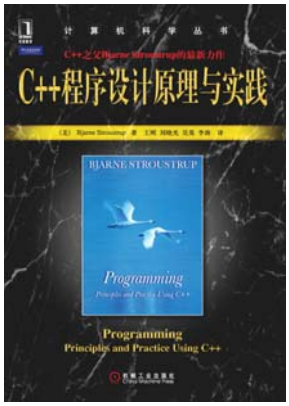
- 采用螺旋方法和深度导向方法讲解操作系统原理。
- 扩展知识。结合当时的行业历史，讲述所讨论的操作系统。
- 讨论算法级解决方案，而没有列出实际代码，便于使用不同编程语言实现

ISBN：978-7-111-31094-5
定价：45.00

《C++程序设计原理与实践》

- C++之父 Bjarne Stroustrup 最新力作
- 经典程序设计思想与 C++开发实践的完美结合
- 一本以 C++为载体讲述如何学习程序设计的书
- 自上市来评价颇高的一本书

ISBN：978-7-111-30322-0
定价：108.00



第0章　读者指南

USENET新闻组本书及相关Web站点包含了大量的资料，下面将给读者提供一个总体介绍。

0.1　本书概述

本书共分为八个部分：

第一部分　背景：提供关于计算机组织与系统结构的综述，重点讲述与操作系统设计相关的主题，并且概述了本书的其余部分操作系统（OS）的各个主题。

第二部分　进程：详细分析进程、多线程、对称多处理（SMP）和微内核，还讨论了单一系统中的并发机制，重点讲述了互斥和死锁。

第三部分　存储器：全面讲述存储器管理技术，包括虚拟存储器。

第四部分　调度：对多种进程调度方法进行分析比较，同时还讨论线程调度、SMP 调度和实时调度。

第五部分　输入/输出与文件：分析操作系统中有关输入/输出函数的控制，特别是磁盘输入/输出，它是决定系统性能的关键所在。本部分还给出了关于文件管理的综述。

第六部分　嵌入式系统：嵌入式系统的数量远远多于通用计算系统，因此存在许多独特的嵌入式操作系统。本章讨论了嵌入式操作系统的一般性原理，并且介绍了两个实例系统：TinyOS和eCos。

第七部分　安全：对涉及计算机和网络安全的威胁和防护机制进行了概述。

第八部分　分布式系统：分析计算机系统网络化技术的主要趋势，包括TCP/IP、客户/服务器计算和集群，同时还介绍分布式系统开发中的一些主要设计领域。

本书试图使读者熟悉当代操作系统的设计原理和实现问题，因此单纯的概念和理论是远远不够的。为举例说明这些概念，并把它们与现实世界中必须做的设计选择联系起来，本书选用了两个操作系统作为运行实例：

- Windows Vista：可以在各种各样的个人计算机、工作站和服务器上运行的多任务操作系统。这是为数不多的几个从零开始设计的商用操作系统之一，因此它具有操作系统技术最新发展的鲜明风格。
- UNIX：一个多用户操作系统，最初是为小型计算机设计的，但后来广泛用于从微机到超级计算机的各种机器中。

关于这些示例系统的讨论贯穿于本书的全部内容，而不是集中在某一章或附录部分。这样，在讨论并发性的过程中，将讲述每个示例系统的并发机制，并讨论单个设计选择的动机。使用这种方法，可以通过真实的例子加深对某一特定章节中设计概念的理解。

0.2　读者和教师的学习路线图

读者可能会很自然地对本书中的主题顺序提出疑问。例如，有关调度的主题（第9章和第10章）与有关并发的主题（第5章和第6章）以及进程（第3章）密切相关，似乎应该安排在一起。

困难在于各个主题都是紧密关联的。例如，在讨论虚拟存储器时，如果能够参考调度问题中的缺页处理，则是非常有用的；当然，在讨论调度决策时，如果能够参考一些存储器管理的问题也是非常有用的。这类例子举不胜举：讨论调度需要输入/输出管理的部分内容，反之亦然。

图 0.1 给出了各个主题间一些重要的相互关系。粗线表示从设计和实现的角度考虑非常紧密的关系。基于这幅图，很显然应该从关于进程的最基本的讨论开始，正如我们在第 3 章中所做的。接下来的顺序就有一定的任意性，很多操作系统的书籍在一开始把所有与进程相关的内容放在一起，然后再处理其他主题，这无疑是正确的。但我相信存储器管理与进程管理具有同样重要的地位，因此，在本书中将这部分内容放在调度之前。

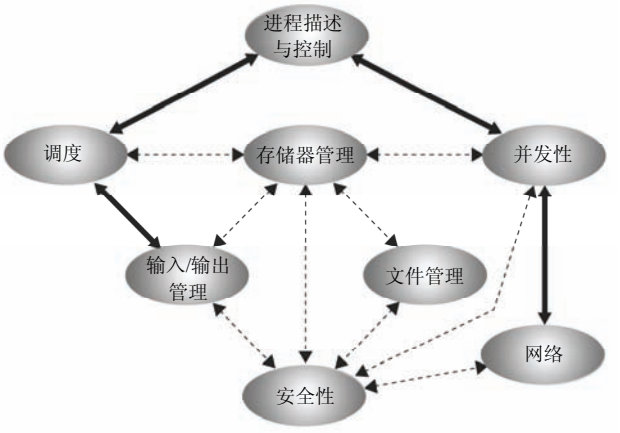


图 0.1　操作系统的各个主题

对学生而言，最理想的安排是，在按顺序学习完第 1 章到第 3 章后，并行地阅读和理解以下章节：第 4 章然后（任选）第 5 章；第 6 章然后第 7 章；第 8 章然后（任选）第 9 章；第 10 章。其余部分可以以任意顺序学习。当然，尽管人脑可能可以并行处理，但让学生同时打开同样的 4 本书，并翻到不同的 4 章进行学习几乎是不可能的（同时代价也过于昂贵）。既然必须按照线性顺序学习，我认为本书的顺序安排是最有效的。

最后补充一句，第 2 章特别是 2.3 节中给出了后续章节中涵盖的所有重要概念的介绍。因此，读完第 2 章后，可以灵活选择阅读其余章节的顺序。

0.3　Internet 和 Web 资源

在 Internet 中和本书的支持站点中有很多有用的资源，可以帮助读者跟上该领域的发展步伐。

本书的 Web 站点

WilliamStallings.com/OS/OS6e.html 是专门为本书所建立的 Web 页，有关该站点的详细描述请参阅本书开始处的内容，该站点特别为学生提供了两种文档：

- **伪代码**：本书为那些对 C 和 Java 均不熟悉的读者，用类 Pascal 的伪代码重新生成了所有的算法。这种伪代码语言非常直观，易于理解。
- **Vista、UNIX 与 Linux 说明**：正如前面所提到的，把 Windows 和各种不同版本的 UNIX 都作为运行实例研究，关于它们的讨论贯穿于本书的全部内容，而不是集中在某一章或附录部分。一些读者可能希望把所有这些资料放在一起以供参考，因此，本书中所有关于 Windows、UNIX 和 Linux 的资料都重新整理成为该站点中的三个文件。

只要发现了任何印刷错误和别的错误，在该 Web 站点中就可以找到勘误表。欢迎读者报告发现的任何错误。作者的其他书籍的勘误表以及优惠订购信息均在 WilliamStallings.com 中。

作者还在 WilliamStallings.com/studentSupport.html 中维护了一个 Computer Science Student Support Site 站点，该站点为计算机科学专业的学生及专业人员提供资料、信息和链接，链接和文档可分为以下 6 类：

- **数学**：包括基本数学知识的复习、排队论入门、计数系统入门和到很多数学站点的链接。
- **指引类**：为完成作业、编写技术报告和准备技术演示提供的建议和指导。
- **研究资源**：关于很多重要的论文、技术报告和参考文献的链接。
- **其他**：大量有用的资料和链接。
- **计算机科学职业**：对考虑计算机科学职业规划有用的链接和资料。
- **幽默和其他趣事**：工作之余有时也需要放松一下。

其他 Web 站点

还有很多站点提供了与本书相关的某些信息。在后续的章节中，在“推荐读物和 Web 站点”一节中可以找到一些特定 Web 站点的链接。由于 Web 站点的 URL 地址有可能更新，因此本书将不再包含这些 URL 地址。本书所列出的所有 Web 站点，都将列在本书的网站上。本书中未涉及的其他链接也会及时补充到 Web 站点上。

USENET 新闻组

很多 USENET 新闻组都致力于操作系统的某一方面或某个特定的操作系统。事实上所有的 USENET 组都有很高的噪信比，但仍然值得去试一试，看是否有满足自己需要的。其中，最相关的如下所示：

- comp.os.research：这是值得关注的最好的组，是关于研究主题的一个合适的新闻组。
- comp.os.misc：关于操作系统主题的一个全面的讨论。
- comp.unix.internals
- comp.os.linux.development.system

第一部分　背　　景

第一部分的目标是为本书的其余部分提供背景知识和上下文环境，给出关于计算机系统结构和操作系统核心的基本概念。

第一部分导读

第 1 章　计算机系统概述

操作系统处于中间位置，它的一边是应用程序、实用程序和用户，另一边是计算机系统的硬件。为了理解操作系统的功能和涉及的设计问题，必须首先对计算机组织与系统结构有一定的认识。第 1 章提供了对计算机系统中的处理器、内存和输入/输出原理的简要介绍。

第 2 章　操作系统概述

关于操作系统设计的主题涉及很多领域，学习时很容易在大量的细节中迷失方向，且在讨论某个特定问题时容易丢掉问题的前后关系。第 2 章为读者提供了一个总览，方便读者在本书的任何一处找到其前后关系。本书从操作系统的目标和功能开始陈述，然后描述一些在历史上非常重要的系统和操作系统的功能。通过这样的论述，可在一个简单的环境中给出基本的操作系统设计原理，从而使不同的操作系统功能之间的关系变得十分清楚。本章还强调了现代操作系统的重要特征。通过贯穿本书的各种各样的主题论述，不仅讨论了操作系统设计中最基本的和完善的原理，而且还讨论了操作系统设计中最新的创新。本章的论述将告诉读者操作系统中已建立的和新的设计方法所必须解决的问题。最后，对 Windows、UNIX 和 Linux 进行了概述，并建立了这些操作系统的总体结构，为接下来更详细的讨论提供了前后联系。

第 1 章　计算机系统概述

操作系统利用一个或多个处理器的硬件资源，为系统用户提供一组服务，它还代表用户来管理辅助存储器和输入/输出（Input/Output，I/O）设备。因此，在开始分析操作系统之前，掌握一些底层的计算机系统硬件知识是很重要的。

本章给出了计算机系统硬件的概述并假设读者对这些领域已经比较熟悉，所以对大多数领域只进行简要概述。但某些内容对本书后面的主题比较重要，因此对这些内容的讲述将会比较详细。

1.1　基本构成

从最顶层看，一台计算机由处理器、存储器和输入/输出部件组成，每类部件有一个或多个模块。这些部件以某种方式互联，以实现计算机执行程序的主要功能。因此，计算机有 4 个主要的结构化部件：

- **处理器**（processor）：控制计算机的操作，执行数据处理功能。当只有一个处理器时，它通常指中央处理单元（CPU）。
- **内存**（main memory）：存储数据和程序。此类存储器通常是易失性的，即当计算机关机时，存储器的内容会丢失。相反，当计算机关机时，磁盘存储器的内容不会丢失。内存通常也称为**实存储器**（real memory）或**主存储器**（primary memory）。
- **输入/输出模块**（I/O module）：在计算机和外部环境之间移动数据。外部环境由各种外部设备组成，包括辅助存储器设备（如硬盘）、通信设备和终端。
- **系统总线**（system bus）：为处理器、内存和输入/输出模块间提供通信的设施。

图 1.1 描述了这些从最顶层看到的部件。处理器的一种功能是和存储器交换数据。为此，它通常使用两个内部（对处理器而言）寄存器：存储器地址寄存器（Memory Address Register，MAR），存储器地址寄存器确定下一次读写的存储器地址；存储器缓冲寄存器（Memory Buffer Register，MBR），存储器缓冲寄存器存放要写入存储器的数据或者从存储器中读取的数据。同理，输入/输出地址寄存器（I/O Address Register，简称 I/O AR 或 I/O 地址寄存器）确定一个特定的输入/输出设备，输入/输出缓冲寄存器（I/O Buffer Register，简称 I/O BR 或 I/O 缓冲寄存器）用于在输入/输出模块和处理器间交换数据。

内存模块由一组单元组成，这些单元由顺序编号的地址定义。每个单元包含一个二进制数，可以解释为一个指令或数据。输入/输出模块在外部设备与处理器和存储器之间传送数据。输入/输出模块包含内存缓冲区，用于临时保存数据，直到它们被发送出去。

1.2　处理器寄存器

处理器包含一组寄存器，它们提供一定的存储能力，比内存访问速度快，但比内存的容量小。处理器中的寄存器有两个功能：

- **用户可见寄存器**：优先使用这些寄存器，可以减少使用机器语言或汇编语言的程序员对内存的访问次数。对高级语言而言，由优化编译器负责决定哪些变量应该分配给寄存器，

哪些变量应该分配给内存。一些高级语言（如 C 语言）允许程序员建议编译器把哪些变量保存在寄存器中。

- **控制和状态寄存器**：用以控制处理器的操作，且主要被具有特权的操作系统例程使用，以控制程序的执行。

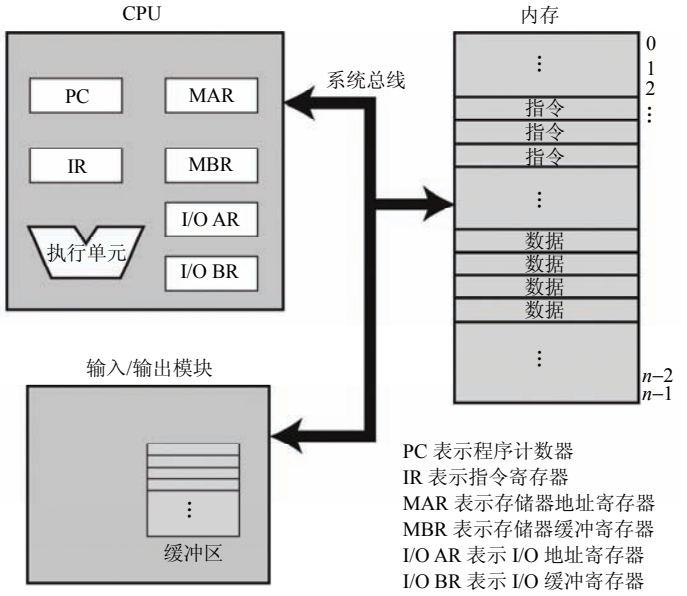


图 1.1　计算机部件：顶层视图

这两类寄存器并没有很明显的界限。例如，对某些处理器而言，程序计数器是用户可见的，但对其他处理器却不是这样。但为了方便起见，以下的讨论使用这种分类方法。

1.2.1　用户可见寄存器

用户可见寄存器可以通过由处理器执行的机器语言来引用，它一般对所有的程序都是可用的，包括应用程序和系统程序。通常可用的寄存器类型包括数据寄存器、地址寄存器和条件码寄存器。

数据寄存器（data register）可以被程序员分配给各种函数。在某些情况下，它们实际上是通用的，可被执行数据操作的任何机器指令使用。但通常也有一些限制，例如对浮点数运算使用专用的寄存器，而对整数运算使用其他寄存器。

地址寄存器（address register）存放数据和指令的内存地址，或者存放用于计算完整地址或有效地址的部分地址。这些寄存器可以是通用的，或者可以用来以某一特定方式或模式寻址存储器。如下面的例子所示：

- **变址寄存器**（index register）：变址寻址是一种最常用的寻址方式，它通过给一个基值加一个索引来获得有效地址。
- **段指针**（segment pointer）：对于分段寻址方式，存储器被划分成段，这些段由长度不等的字块组成[⊖]，段由若干长度的字组成。一个存储器引用由一个特定段号和段内的偏移量组成；在第 7 章关于内存管理的论述中，这种寻址方式是非常重要的。采用这种寻址

⊖ 术语字无通用的定义。一般来说，字是字节或位的一个有序集合，字节或位是在给定的计算机上存储、发送或操作信息的通用单位。一般地，若处理器有一个定长指令集，则指令长度等于字长。

方式，需要用一个寄存器保存段的基地址（起始地址）。可能存在多个这样的寄存器，例如一个用于操作系统（即当操作系统代码在处理器中执行时使用），一个用于当前正在执行的应用程序。

- **栈指针**（stack pointer）：如果对用户可见的栈[⊖]进行寻址，则应该有一个专门的寄存器指向栈顶。这样就可以使用不包含地址域的指令，如入栈（push）和出栈（pop）。

对于有些处理器，过程调用将导致所有用户可见的寄存器自动保存，在调用返回时恢复保存的寄存器。由处理器执行的保存操作和恢复操作是调用指令和返回指令执行过程的一部分。这就允许每个过程独立地使用这些寄存器。而在其他的处理器上，程序员必须在过程调用前保存相应的用户可见寄存器，通过在程序中包含完成此项任务的指令来实现。因此，保存和恢复功能可以由硬件完成，也可以由软件完成，这完全取决于处理器的实现。

1.2.2　控制和状态寄存器

有多种处理器的寄存器用于控制处理器的操作。在大多数处理器上，大部分此类寄存器对用户不可见，其中一部分可被在控制态（或称为内核态）下执行的某些机器指令所访问。

当然，不同的处理器有不同的寄存器结构，并使用不同的术语。在这里我们列出了比较合理和完全的寄存器类型，并给出了简要的说明。除了前面提到过的 MAR、MBR、I/O AR 和 I/O BR 寄存器（如图 1.1 所示）外，下面的寄存器是指令执行所必需的：

- **程序计数器**（Program Counter，PC）：包含将取指令的地址。
- **指令寄存器**（Instruction Register，IR）：包含最近取的指令内容。

所有的处理器设计还包括一个或一组寄存器，通常称为程序状态字（Program Status Word，PSW），它包含状态信息。PSW 通常包含条件码和其他状态信息，如中断允许/禁止位和内核/用户态位。

条件码（condition code，也称为**标记**）是处理器硬件为操作结果设置的位。例如，算术运算可能产生正数、负数、零或溢出的结果，除了结果自身存储在一个寄存器或存储器中之外，在算术指令执行之后，也随之设置一个条件码。这个条件码之后可作为条件分支运算的一部分被测试。条件码位被收集到一个或多个寄存器中，通常它们构成了控制寄存器的一部分。机器指令通常允许通过隐式访问来读取这些位，但不能通过显式访问进行修改，这是因为它们是为指令执行结果的反馈而设计的。

在使用多种类型中断的处理器中，通常有一组中断寄存器，每个指向一个中断处理例程。如果使用栈实现某些功能（例如过程调用），则需要一个系统栈指针（参见附录 1B）。第 7 章讲述的内存管理硬件也需要专门的寄存器。最后，寄存器还可以用于控制 I/O 操作。

在设计控制和状态寄存器结构时需要考虑很多因素，一个关键问题是对操作系统的支持。某些类型的控制信息对操作系统来说有特殊的用途，如果处理器设计者对所用操作系统的功能有所了解，那么可以设计寄存器结构，对操作系统的特殊功能提供硬件支持，如存储器保护和用户程序之间的切换等。

另一个重要的设计决策是在寄存器和存储器间分配控制信息。通常把存储器最初的（最低的）几百个或几千个字用于控制目的，设计者必须决定在昂贵、高速的寄存器中放置多少控制信息，在相对便宜、低速的内存中放置多少控制信息。

⊖ 栈位于内存中，它是一组连续的存储单元，对它的访问类似于处理一叠纸，只能从顶层放置或取走。有关栈处理的详细内容，请参阅附录 1B。

1.3　指令的执行

处理器执行的程序是由一组保存在存储器中的指令组成的。按最简单的形式，指令处理包括两个步骤：处理器从存储器中一次读（**取**）一条指令，然后执行每条指令。程序执行是由不断重复的取指令和执行指令的过程组成的。指令执行可能涉及很多操作，这取决于指令自身。

一个单一的指令需要的处理称为一个**指令周期**。如图 1.2 所示，可使用简单的两个步骤来描述指令周期。这两个步骤分别称做**取指阶段**和**执行阶段**。仅当机器关机、发生某些未发现的错误或者遇到与停机相关的程序指令时，程序执行才会停止。

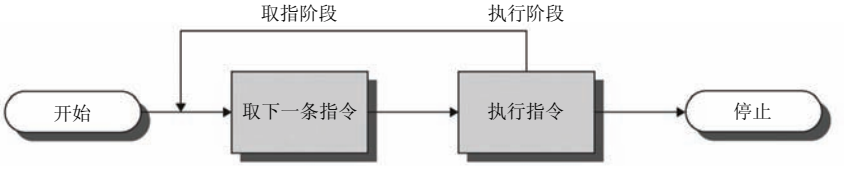


图 1.2　基本指令周期

1.3.1　取指令和执行指令

在每个指令周期开始时，处理器从存储器中取一条指令。在典型的处理器中，程序计数器（Program Counter，PC）保存下一次要取的指令地址。除非有其他情况，否则处理器在每次取指令后总是递增 PC，使得它能够按顺序取得下一条指令（即位于下一个存储器地址的指令）。例如，考虑一个简化的计算机，每条指令占据存储器中一个 16 位的字，假设程序计数器 PC 被设置为地址 300，处理器下一次将在地址为 300 的存储单元处取指令，在随后的指令周期中，它将从地址为 301、302、303 等的存储单元处取指令。下面将会解释这个顺序是可以改变的。

取到的指令被放置在处理器的一个寄存器中，这个寄存器称做指令寄存器（Instruction Register，IR）。指令中包含确定处理器将要执行的操作的位，处理器解释指令并执行对应的操作。大体上，这些操作可分为 4 类：

- **处理器-存储器**：数据可以从处理器传送到存储器，或者从存储器传送到处理器。
- **处理器-I/O**：通过处理器和 I/O 模块间的数据传送，数据可以输出到外部设备，或者从外部设备输入数据。
- **数据处理**：处理器可以执行很多与数据相关的算术操作或逻辑操作。
- **控制**：某些指令可以改变执行顺序。例如，处理器从地址为 149 的存储单元中取出一条指令，该指令指定下一条指令应该从地址为 182 的存储单元中取，这样处理器要把程序计数器设置为 182。因此，在下一个取指阶段中，将从地址为 182 的存储单元而不是地址为 150 的存储单元中取指令。

指令的执行可能涉及这些行为的组合。

考虑一个简单的例子，假设有一台机器具备图 1.3 中列出的所有特征，处理器包含一个称为累加器（AC）的数据寄存器，所有指令和数据长度均为 16 位，使用 16 位的单元或字来组织存储器。指令格式中有 4 位是操作码，因而最多有 2^4=16 种不同的操作码（由 1 位十六进制[⊖]数字表示），操作码定义了处理器要执行的操作。通过指令格式的余下 12 位，可直接访问的存储器大小最大为 2^{12}＝4096（4K）个字（用 3 位十六进制数字表示）。

⊖ 有关记数系统（十进制、二进制、十六进制）的详细信息，可在 WilliamStallings.com/StudentSupport.html 中的 Computer Science Student Resource Site 站点找到。

a）指令格式

b）整数格式

程序计数器（PC）=指令地址
指令寄存器（IR）=正在执行的指令
累加器（AC）=临时存储体

c）内部 CPU 寄存器

0001=从存储器中加载 AC
0010=把 AC 的内容存储到存储器中
0101=从存储器中加到 AC 中

d）部分操作码列表

图 1.3　一台理想机器的特征

图 1.4 描述了程序的部分执行过程，显示了存储器和处理器的寄存器的相关部分。给出的程序片段把地址为 940 的存储单元中的内容与地址为 941 的存储单元中的内容相加，并将结果保存在后一个单元中。这需要三条指令，可用三个取指阶段和三个执行阶段描述：

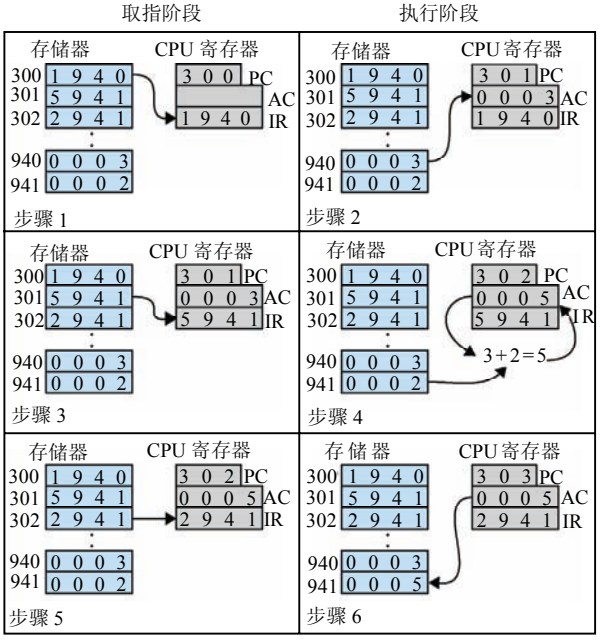


图 1.4　程序执行的例子（存储器和寄存器的内容以十六进制表示）

1）PC 中包含第一条指令的地址为 300，该指令内容（值为十六进制数 1940）被送入指令寄存器 IR 中，PC 增 1。注意，此处理过程使用了存储器地址寄存器（MAR）和存储器缓冲寄存器（MBR）。为简单起见，这些中间寄存器没有显示。

2）IR 中最初的 4 位（第一个十六进制数）表示需要加载 AC，剩下的 12 位（后三个十六进制数）表示地址为 940。

3）从地址为 301 的存储单元中取下一条指令（5941），PC 增 1。

4）AC 中以前的内容和地址为 941 的存储单元中的内容相加，结果保存在 AC 中。

5）从地址为 302 的存储单元中取下一条指令（2941），PC 增 1。

6）AC 中的内容被存储在地址为 941 的存储单元中。

在这个例子中，为把地址为 940 的存储单元中的内容与地址为 941 的存储单元中的内容相加，一共需要三个指令周期，每个指令周期都包含一个取指阶段和一个执行阶段。如果使用更复杂的指令集合，则只需要更少的指令周期。大多数现代的处理器都具有包含多个地址的指令，因此指令周期可能涉及多次存储器访问。此外，除了存储器访问外，指令还可用于 I/O 操作。

1.3.2　I/O 函数

I/O 模块（例如磁盘控制器）可以直接与处理器交换数据。正如处理器可以通过指定存储单元的地址来启动对存储器的读和写一样，处理器也可以从 I/O 模块中读数据或向 I/O 模块中写数据。对于后一种情况，处理器需要指定被某一 I/O 模块控制的具体设备。因此，指令序列的格式与图 1.4 中的格式类似，只是用 I/O 指令代替了存储器访问指令。

在某些情况下，允许 I/O 模块直接与内存发生数据交换，以减轻在完成 I/O 任务过程中的处理器负担。此时，处理器允许 I/O 模块具有从存储器中读或往存储器中写的特权，这样 I/O 模块与存储器之间的数据传送无需通过处理器完成。在这类传送过程中，I/O 模块对存储器发出读命令或写命令，从而免去了处理器负责数据交换的任务。这个操作称为直接内存存取（Direct Memory Access，DMA），详细内容请参阅本章后面部分。

1.4　中断

事实上所有计算机都提供了允许其他模块（I/O、存储器）中断处理器正常处理过程的机制。表 1.1 列出了最常见的中断类别。

中断最初是用于提高处理器效率的一种手段。例如，大多数 I/O 设备比处理器慢得多，假设处理器使用如图 1.2 所示的指令周期方案给一台打印机传送数据，在每一次写操作后，处理器必须暂停并保持空闲，直到打印机完成工作。暂停的时间长度可能相当于成百上千个不涉及存储器的指令周期。显然，这对于处理器的使用来说是非常浪费的。

表 1.1　中断的分类

类　别	说　明
程序中断	在某些条件下由指令执行的结果产生，例如算术溢出、除数为 0、试图执行一条非法的机器指令以及访问到用户不允许的存储器位置
时钟中断	由处理器内部的计时器产生，允许操作系统以一定规律执行函数
I/O 中断	由 I/O 控制器产生，用于发信号通知一个操作的正常完成或各种错误条件
硬件故障中断	由诸如掉电或存储器奇偶错误之类的故障产生

这里给出一个实例，假设有一个 1GHz CPU 的 PC 机，大约每秒执行 10^9 条指令[⊖]。一个典型的硬盘的速度是 7200 转/分，这样大约旋转半转的时间是 4 ms，处理器比这要快 4 百万倍。

图 1.5a 显示了这种事件状态。用户程序在处理过程中交织着执行一系列 WRITE 调用。竖实线表示程序中的代码段，代码段 1、2 和 3 表示不涉及 I/O 的指令序列。WRITE 调用要执行一个 I/O 程序，此 I/O 程序是一个系统工具程序，由它执行真正的 I/O 操作。此 I/O 程序由三部分组成：

⊖ 关于数字前缀的用法，如吉（Giga）和太（Tera），请参阅位于 WilliamStallings.com.StudentSupport.html 处的 Computer Science Student Resource Site 站点中的支持文档。

- 图中标记为 4 的指令序列用于为实际的 I/O 操作做准备。这包括复制将要输出到特定缓冲区的数据，为设备命令准备参数。
- 实际的 I/O 命令。如果不使用中断，当执行此命令时，程序必须等待 I/O 设备执行请求的函数（或周期性地检测 I/O 设备的状态或轮询 I/O 设备）。程序可能通过简单地重复执行一个测试操作的方式进行等待，以确定 I/O 操作是否完成。

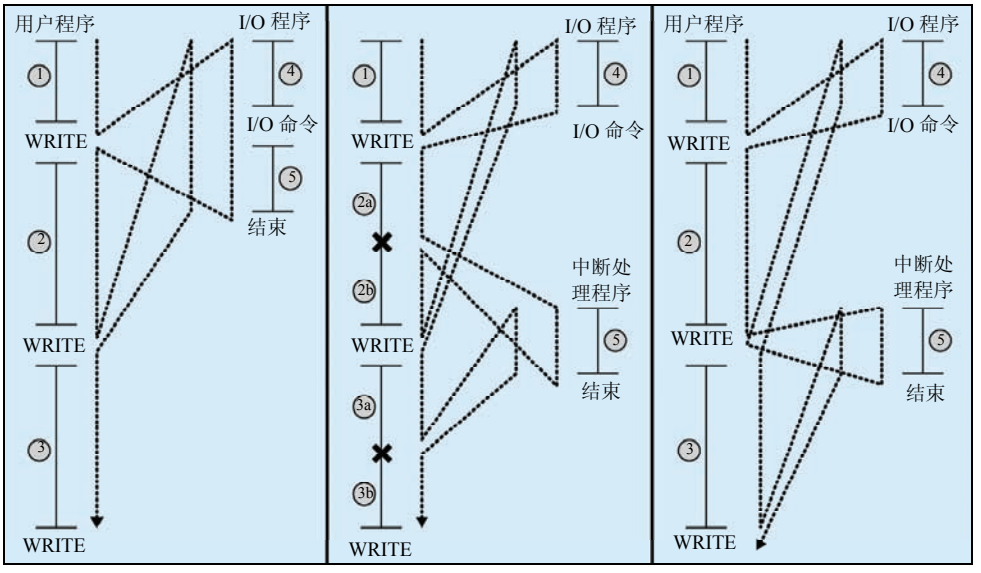


图 1.5　在有中断和无中断时程序的控制流

- 图中标记为 5 的指令序列，用于完成操作。包括设置一个表示操作成功或失败的标记。

虚线代表处理器执行的路径；也就是说，这条线显示了指令执行的顺序。当遇到第一条 WRITE 指令之后，用户程序被中断，I/O 程序开始执行。在 I/O 程序执行完成后，WRITE 指令之后的用户程序立即恢复执行。

由于完成 I/O 操作可能花费较长的时间，I/O 程序需要挂起等待操作完成，因此用户程序会在 WRITE 调用处停留相当长的一段时间。

1.4.1　中断和指令周期

利用中断功能，处理器可以在 I/O 操作的执行过程中执行其他指令。考虑图 1.5b 所示的控制流，和前面一样，用户程序到达系统调用 WRITE 处，但涉及的 I/O 程序仅包括准备代码和真正的 I/O 命令。在这些为数不多的几条指令执行后，控制返回到用户程序。在这期间，外部设备忙于从计算机存储器接收数据并打印。这种 I/O 操作和用户程序中指令的执行是并发的。

当外部设备做好服务的准备，也就是说，当它准备好从处理器接收更多的数据时，该外部设备的 I/O 模块给处理器发送一个**中断请求**信号。这时处理器会做出响应，暂停当前程序的处理，转去处理服务于特定 I/O 设备的程序，这个程序称做中断处理程序（interrupt handler）。在对该设备的服务响应完成后，处理器恢复原先的执行。图 1.5b 中用“✖”表示发生中断的点。注意，中断可以在主程序中的任何位置发生，而不是在一条指定的指令处。

从用户程序的角度看，中断打断了正常执行的序列。当中断处理完成后，再恢复执行（见图 1.6）。因此，用户程序并不需要为中断添加任何特殊的代码，处理器和操作系统负责挂起用户程

序，然后在同一个地方恢复执行。

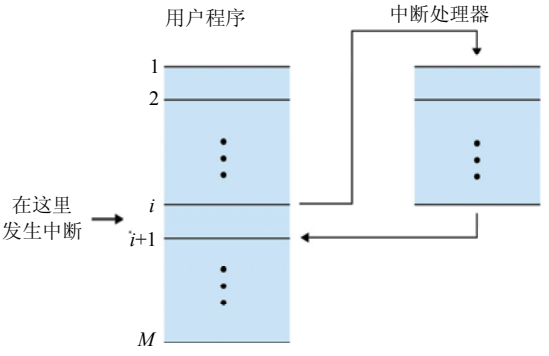


图 1.6　通过中断转移控制

为适应中断产生的情况，在指令周期中要增加一个**中断阶段**，如图 1.7 所示（与图 1.2 对照）。在中断阶段中，处理器检查是否有中断发生，即检查是否出现中断信号。如果没有中断，处理器继续运行，并在取指周期取当前程序的下一条指令；如果有中断，处理器挂起当前程序的执行，并执行一个**中断处理**程序。这个中断处理程序通常是操作系统的一部分，它确定中断的性质，并执行所需要的操作。例如，在前面的例子中，处理程序决定哪一个 I/O 模块产生中断，并转到往该 I/O 模块中写更多数据的程序。当中断处理程序完成后，处理器在中断点恢复对用户程序的执行。

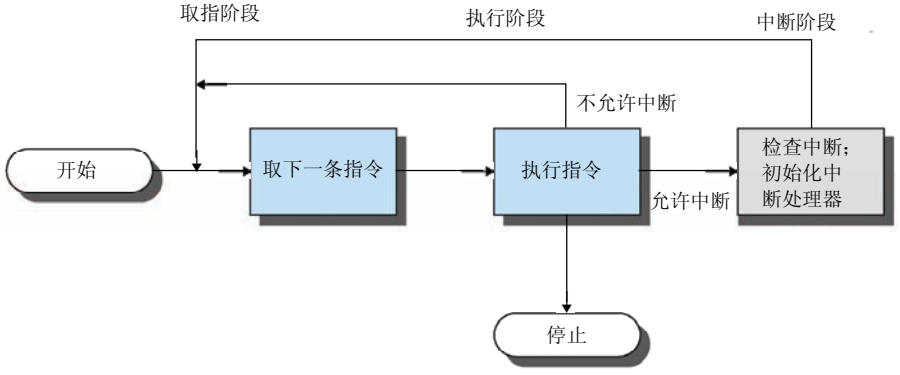
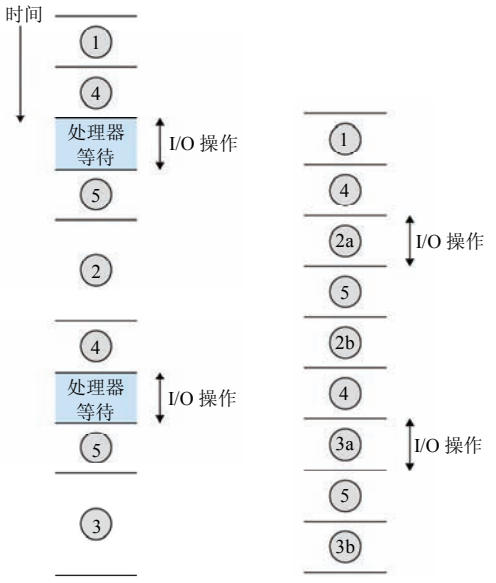


图 1.7　中断和指令周期

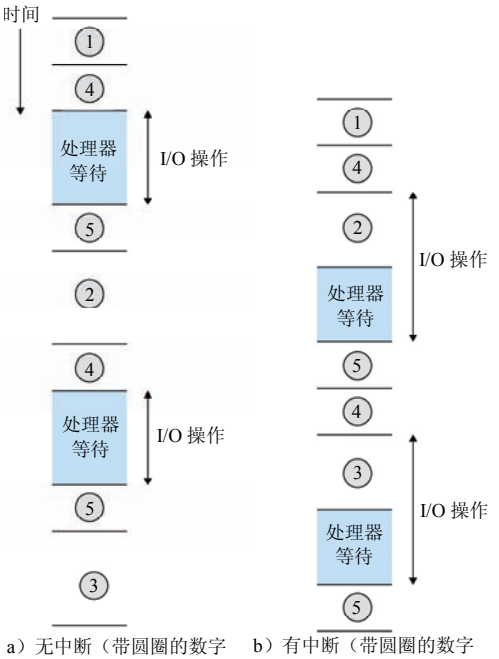
很显然，在这个处理中有一定的开销，在中断处理程序中必须执行额外的指令以确定中断的性质，并决定采用适当的操作。然而，如果简单地等待 I/O 操作的完成将花费更多的时间，因此使用中断能够更有效地使用处理器。

为进一步理解在效率上的提高，请参阅图 1.8，它是关于图 1.5a 和图 1.5b 中控制流的时序图。图 1.5b 和图 1.8 假设 I/O 操作的时间相当短，小于用户程序中写操作之间完成指令执行的时间。而更典型的情况是，特别是对比较慢的设备如打印机来说，I/O 操作比执行一系列用户指令的时间要长得多，图 1.5c 显示了这类事件状态。在这种情况下，用户程序在由第一次调用产生的 I/O 操作完成之前，就到达了第二次 WRITE 调用。结果是用户程序在这一点挂起，当前面的 I/O 操作完成后，才能继续新的 WRITE 调用，也才能开始一次新的 I/O 操作。图 1.9 给出了在这种情况下使用中断和不使用中断的时序图，我们可以看到 I/O 操作在未完成时与用户指令的执行有所重叠。由于这部分时间的存在，效率仍然有所提高。



a）无中断（带圆圈的数字指图 1.5a 中的数字）　b）有中断（带圆圈的数字指图 1.5b 中的数字）

图 1.8　程序时序：短 I/O 等待



a）无中断（带圆圈的数字指图 1.5a 中的数字）　b）有中断（带圆圈的数字指图 1.5c 中的数字）

图 1.9　程序时序：长 I/O 等待

1.4.2　中断处理

中断激活了很多事件，包括处理器硬件中的事件以及软件中的事件。图 1.10 显示了一个典型的序列，当 I/O 设备完成一次 I/O 操作时，发生下列硬件事件：

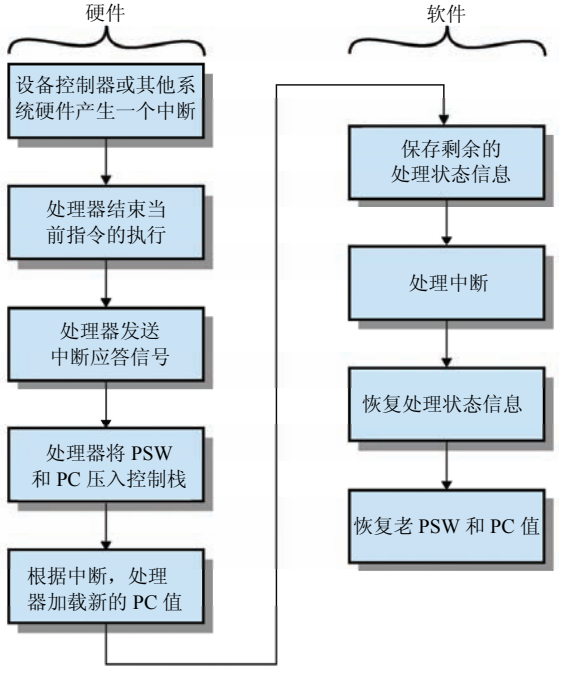


图 1.10　简单中断处理

1）设备给处理器发出一个中断信号。

2）处理器在响应中断前结束当前指令的执行，如图 1.7 所示。

3）处理器对中断进行测定，确定存在未响应的中断，并给提交中断的设备发送确认信号，确认信号允许该设备取消它的中断信号。

4）处理器需要为把控制权转移到中断程序中去做准备。首先，需要保存从中断点恢复当前程序所需要的信息，要求的最少信息包括程序状态字（PSW）和保存在程序计数器中的下一条要执行的指令地址，它们被压入系统控制栈（见附录 1B）中。

5）处理器把响应此中断的中断处理程序入口地址装入程序计数器中。可以针对每类中断有一个中断处理程序，也可以针对每个设备和每类中断各有一个中断处理程序，这取决于计算机系统结构和操作系统的设计。如果有多个中断处理程序，处理器就必须决定调用哪一个，这个信息可能已经包含在最初的中断信号中，否则处理器必须给发中断的设备发送请求，以获取含有所需信息的响应。

一旦完成对程序计数器的装入，处理器则继续到下一个指令周期，该指令周期也是从取指开始。由于取指是由程序计数器的内容决定的，因此控制被转移到中断处理程序，该程序的执行引起以下的操作：

6）在这一点，与被中断程序相关的程序计数器和 PSW 被保存到系统栈中，此外，还有一些其他信息被当做正在执行程序的状态的一部分。特别需要保存处理器寄存器的内容，因为中断处理程序可能会用到这些寄存器，因此所有这些值和任何其他的状态信息都需要保存。在典型情况下，中断处理程序一开始就在栈中保存所有的寄存器内容，其他必须保存的状态信息将在第 3 章中讲述。图 1.11a 给出了一个简单的例子。在这个例子中，用户程序在执行地址为 N 的存储单元中的指令之后被中断，所有寄存器的内容和下一条指令的地址（N+1），一共 M 个字，被压入控制栈中。栈指针被更新指向新的栈顶，程序计数器被更新指向中断服务程序的开始。

7）中断处理程序现在可以开始处理中断，其中包括检查与 I/O 操作相关的状态信息或其他引起中断的事件，还可能包括给 I/O 设备发送附加命令或应答。

8）当中断处理结束后，被保存的寄存器值从栈中释放并恢复到寄存器中，如图 1.11b 所示。

9）最后的操作是从栈中恢复 PSW 和程序计数器的值，其结果是下一条要执行的指令来自前面被中断的程序。

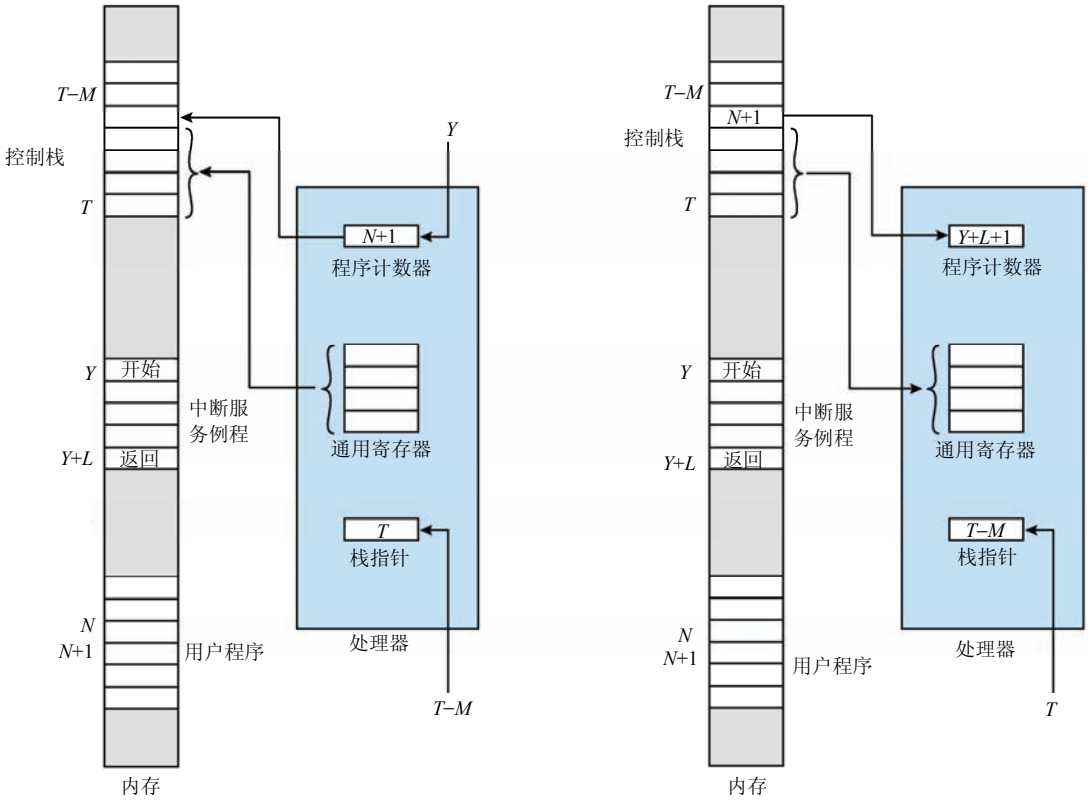
保存被中断程序的所有状态信息并在以后恢复这些信息，这是十分重要的，这是由于中断并不是程序调用的一个例程，它可以在任何时候发生，因而可以在用户程序执行过程中的任何一点上发生，它的发生是不可预测的。

1.4.3　多个中断

至此，我们已讨论了发生一个中断的情况。假设一下，当正在处理一个中断时，可以发生一个或者多个中断，例如，一个程序可能从一条通信线中接收数据并打印结果。每完成一个打印操作，打印机就会产生一个中断；每当一个数据单元到达，通信线控制器也会产生一个中断。数据单元可能是一个字符，也可能是连续的一块字符串，这取决于通信规则本身。在任何情况下，都有可能在处理打印机中断的过程中发生一个通信中断。

处理多个中断有两种方法。第一种方法是当正在处理一个中断时，禁止再发生中断。禁止中断的意思是处理器将对任何新的中断请求信号不予理睬。如果在这期间发生了中断，通常中断保持挂起，当处理器再次允许中断时，再由处理器检查。因此，当用户程序正在执行并且有一个中断发生时，立即禁止中断；当中断处理程序完成后，在恢复用户程序之前再允

许中断，并且由处理器检查是否还有中断发生。这个方法很简单，因为所有中断都严格按顺序处理（见图 1.12a）。



a）在存储单元 *N* 中的指令之后发生中断　　b）从中断返回

图 1.11　因中断而产生的存储器和寄存器中的变化

上述方法的缺点是没有考虑相对优先级和时间限制的要求。例如，当来自通信线的输入到达时，可能需要快速接收，以便为更多的输入让出空间。如果在第二批输入到达时第一批还没有处理完，就有可能由于 I/O 设备的缓冲区装满或溢出而丢失数据。

第二种方法是定义中断优先级，允许高优先级的中断打断低优先级的中断处理程序的运行（见图 1.12b）。第二种方法的例子如下，假设一个系统有三个 I/O 设备：打印机、磁盘和通信线，优先级依次为 2、4 和 5，图 1.13 给出了可能的顺序[TANE06]。用户程序在 t=0 时开始，在 t=10 时，发生一个打印机中断；用户信息被放置到系统栈中并开始执行打印机中断服务例程（Interrupt Service Routine，ISR）；当这个例程仍在执行时，在 t=15 时发生了一个通信中断，由于通信线的优先级高于打印机，必须处理这个中断，打印机 ISR 被打断，其状态被压入栈中，并开始执行通信 ISR；当这个程序正在执行时，又发生了一个磁盘中断（t=20），由于这个中断的优先级比较低，它被简单地挂起，通信 ISR 运行直到结束。

当通信 ISR 完成后（t=25），恢复以前关于执行打印机 ISR 的处理器状态。但是，在执行这个例程中的任何一条指令前，处理器必须完成高优先级的磁盘中断，这样控制权转移给磁盘 ISR。只有当这个例程也完成时（t=35），才恢复打印机 ISR。当打印机 ISR 完成时（t=40），控制最终返回到用户程序。

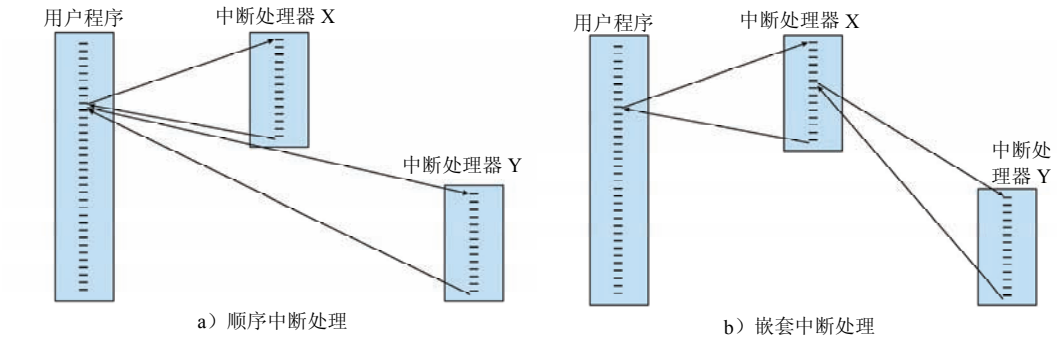


图 1.12　多中断中的控制转移

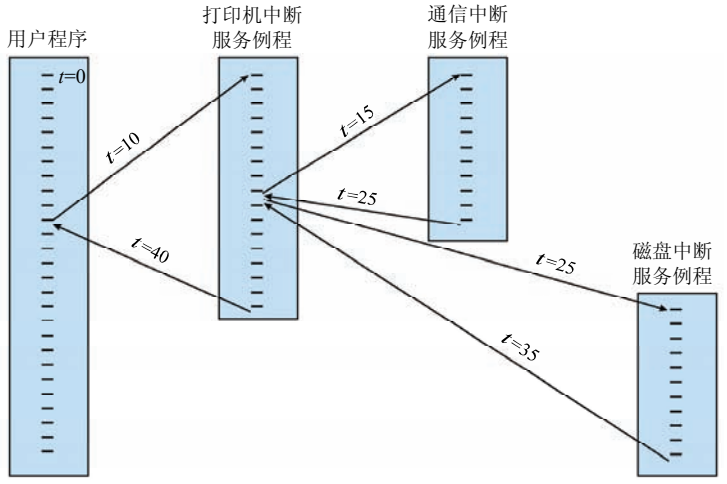


图 1.13　多中断的时间顺序

1.4.4　多道程序设计

即使使用了中断，处理器仍有可能未得到有效的利用，例如，图 1.9b 曾用于说明处理器在长 I/O 等待下的使用率，但如果完成 I/O 操作的时间远远大于 I/O 调用期间用户代码的执行时间（通常情况下），则在大部分时间处理器是空闲的。解决这个问题的方法是允许多道用户程序同时处于活动状态。

假设处理器执行两道程序。一道程序从存储器中读数据并放入外部设备中，另一道是包括大量计算的应用程序。处理器开始执行输出程序，给外部设备发送一个写命令，接着开始执行其他应用程序。当处理器处理很多程序时，执行顺序取决于它们的相对优先级以及它们是否正在等待 I/O。当一个程序被中断时，控制权转移给中断处理程序，一旦中断处理程序完成，控制权可能并不立即返回到这个用户程序，而可能转移到其他待运行的具有更高优先级的程序。最终，当原先被中断的用户程序变为最高的优先级时，它将被重新恢复执行。这种多道程序轮流执行的概念称做多道程序设计，第 2 章将进一步对此进行讨论。

1.5　存储器的层次结构

计算机存储器的设计目标可以归纳成三个问题：多大的容量？多快的速度？多贵的价格？“多大的容量”的问题从某种意义上来说是无止境的，存储器有多大的容量，就可能开发出

相应的应用程序使用它。“多快的速度”的问题相对易于回答，为达到最佳的性能，存储器的速度必须能够跟得上处理器的速度。换言之，当处理器正在执行指令时，我们不希望它会因为等待指令或操作数而暂停。最后一个问题也必须考虑，对一个实际的计算机系统，存储器的价格与计算机其他部件的价格相比应该是合理的。

应该认识到，存储器的这三个重要特性间存在着一定的折衷，即容量、存取时间和价格。在任何时候，实现存储器系统会用到各种各样的技术，但各种技术之间往往存在着以下关系：

- 存取时间越快，每一个“位”的价格越高。
- 容量越大，每一个“位”的价格越低。
- 容量越大，存取速度越慢。

设计者面临的困难是很明显的，由于需求是较大的容量和每一个“位”较低的价格，因而设计者通常希望使用能够提供大容量存储的存储器技术。但是为满足性能要求，又需要使用昂贵的、容量相对比较小而具有快速存取时间的存储器。

解决这个难题的方法是，不依赖于单一的存储组件或技术，而是使用**存储器的层次结构**。一种典型的层次结构如图 1.14 所示。当沿这个层次结构从上向下看，会得到以下情况：

a）每一个“位”的价格递减

b）容量递增

c）存取时间递增

d）处理器访问存储器的频率递减

因此，容量较大、价格较便宜的慢速存储器是容量较小、价格较贵的快速存储器的后备。这种存储器的层次结构能够成功的关键在于低层访问频率递减。在本章后面部分讲解高速缓存（cache）时，以及在本书后面讲解虚拟内存时，将详细分析这个概念，这里只给出简要说明。

假设处理器存取两级存储器，第一级存储器容量为 1 000 个字节，存取时间为 0.1μs；第二级存储器包含 100 000 个字节，存取时间为 1μs 。假如需要存取第一级存储器中的一个字节，则处理器可直接存取此字节；如果这个字节位于第二级存储器，则此字节首先需要转移到第一级存储器中，然后再由处理器存取。为简单起见，我们忽略了处理器用于确定这个字节是在第一级存储器还是在第二级存储器所需的时间。图 1.15 给出了反映这种模型的一般曲线形状。此图表示了二级存储器的平均存取时间是**命中率** H 的函数，H 定义为对较快存储器（如高速缓存）的访问次数与对所有存储器的访问次数的比值，T_1 是访问第一级存储器的存取时间，T_2 是访问第二级存储器的存取时间[⊖]。可以发现，当第一级存储器的存取次数所占比例较高时，总的平均存取时间更接近于第一级存储器的存取时间而不是第二级存储器的存取时间。

例如，假设有95%的存储器存取（H=0.95）发生在高速缓存中，则访问一个字节的平均存取时间可表示为：

$$（0.95）（0.1\mu s）+（0.05）（0.1\mu s+1\mu s）= 0.095 + 0.055 = 0.15\mu s$$

此结果非常接近于快速存储器的存取时间。因此仅当条件 a）到 d）适用时，原则上可以实现该策略。通过使用各种技术手段，现有的存储器系统满足条件 a）到 c），而且条件 d）通常也是有效的。

条件 d）有效的基础是访问的**局部性原理**［DENN68］。在执行程序期间，处理器的指令访存和数据访存呈现“簇”状（指一组数据集合）。典型的程序包含许多迭代循环和子程序，一旦程序进入一个循环或子程序执行，就会重复访问一个小范围的指令集合。同理，对表和数组的操作也涉及存取“一簇”数据。经过很长的一段时间，程序访问的“簇”会改变，但在较短的时间内，

⊖ 若在快速存储器中找到了存取的字，则定义为命中；若在快速存储器中未找到存取的字，则定义为不命中。

处理器主要访问存储器中固定的“簇”。

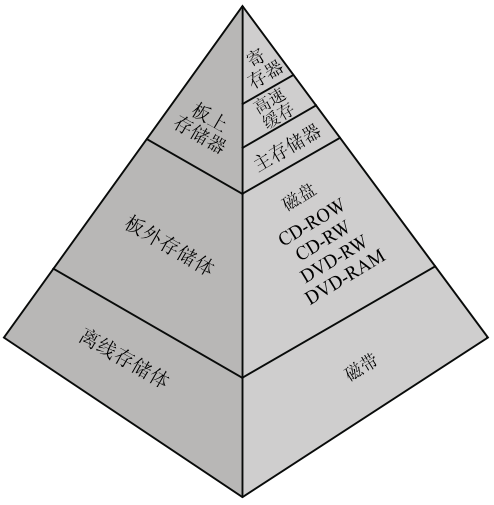


图 1.14　存储器的层次结构

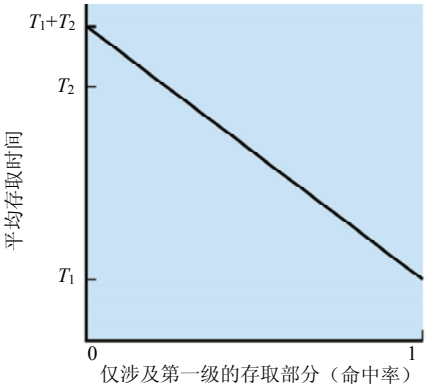


图 1.15　一个简单的二级存储器的性能

因此，可以通过层次组织数据，使得随着组织层次的递减，对各层次的访问比例也依次递减。考虑前面提到的二级存储器的例子，让第二级存储器包含所有的指令和数据，程序当前的访问“簇”暂时存放在第一级存储器中。有时第一级存储器中的某个簇要换出到第二级存储器中，以便为新的“簇”进入第一级存储器让出空间。但平均起来，大多数存储访问是对第一级存储器中的指令和数据的访问。

此原理可以应用于多级存储器组织结构中。最快、最小和最贵的存储器类型由位于处理器内部的寄存器组成。在典型情况下，一个处理器包含多个寄存器，某些处理器包含上百个寄存器。向下跳过两级存储器层次就到了内存层次，内存是计算机中主要的内部存储器系统。内存中的每个单元位置都有一个唯一的地址对应，而且大多数机器指令会访问一个或多个内存地址。内存通常是高速的、容量较小的高速缓存的扩展。高速缓存通常对程序员不可见，或者更确切地说，是对处理器不可见。高速缓存用于在内存和处理器的寄存器之间分段移动数据，以提高数据访问的性能。

前面描述的三种形式的存储器通常是易失的，并且采用的是半导体技术。半导体存储器有各种类型，它们的速度和价格各不相同。数据更多的是永久保存在外部海量存储设备中，通常是硬盘和可移动的储存介质，如可移动磁盘、磁带和光存储介质。外部的、非易失性的存储器也称为**二级存储器**（secondary memory）或**辅助存储器**（auxiliary memory），它们用于存储程序和数据文件，其表现形式是程序员可以看到的文件和记录，而不是单个的字节或字。硬盘还用作内存的扩展，即虚拟存储器（virtual memory），这方面的内容将在第 8 章讲述。

在软件中还可以有效地增加额外的存储层次。例如，一部分内存可以作为缓冲区（buffer），用于临时保存从磁盘中读出的数据。这种技术，有时称为磁盘高速缓存（将在第 11 章中详细讲述），可以通过两种方法提高性能：

- 磁盘成簇写。即采用次数少、数据量大的传输方式，而不是次数多、数据量小的传输方式。选择整批数据一次传输可以提高磁盘的性能，同时减少对处理器的影响。
- 一些注定要“写出”（write-out）的数据也许会在下一次存储到磁盘之前被程序访问到。在此情况下，数据能够迅速地从软件设置的磁盘高速缓存中取出，而不是从缓慢的磁盘中取回。

附录 1A 分析了多级存储器结构的性能。

1.6　高速缓存

尽管高速缓存对操作系统是不可见的，但它与其他存储管理硬件相互影响。此外，很多用于虚拟存储（将在第 8 章讲解）的原理也可以用于高速缓存。

1.6.1　动机

在全部指令周期中，处理器在取指令时至少访问一次存储器，而且通常还要多次访问存储器用于取操作数或保存结果。处理器执行指令的速度显然受存储周期（从存储器中读一个字或写一个字到存储器中所花的时间）的限制。长期以来，由于处理器和内存的速度不匹配，这个限制已经成为很严重的问题。近年来，处理器速度的提高一直快于存储器访问速度的提高，这需要在速度、价格和大小之间进行折衷。理想情况下，内存的构造技术可以采用与处理器中的寄存器相同的构造技术，这样主存的存储周期才跟得上处理器周期。但这样成本太高，解决的方法是利用局部性原理（principle of locality），即在处理器和内存之间提供一个容量小而速度快的存储器，称做高速缓存。

1.6.2　高速缓存原理

高速缓存试图使访问速度接近现有最快的存储器，同时保持价格便宜的大存储容量（以较为便宜的半导体存储器技术实现）。图 1.16 说明了这个概念，图中有一个相对容量大而速度比较慢的内存和一个容量较小且速度较快的高速缓存，高速缓存包含一部分内存数据的副本。当处理器试图读取存储器中的一个字节或字时，要进行一次检查以确定这个字节或字是否在高速缓存中。如果在，该字节或字从高速缓存传递给处理器；如果不在，则由固定数目的字节组成的一块内存数据先被读入高速缓存，然后该字节或字从高速缓存传递给处理器。由于访问局部性现象的存在，当一块数据被取入高速缓存以满足一次存储器访问时，很可能紧接着的多次访问的数据是该块中的其他字节。

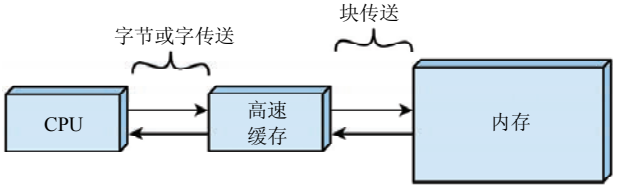


图 1.16　高速缓存和内存

图 1.17 描述了高速缓存/内存的系统结构。内存由 2^n 个可寻址的字组成，每个字有一个唯一的 n 位地址。为便于映射，此存储器可以看做是由一些固定大小的**块**组成，每块包含 K 个字，也就是说，一共有 $M=2^n/K$ 个块。高速缓存中有 C 个存储槽（slot，也称为 line），每个槽有 K 个字，槽的数目远远小于存储器中块的数目（$C<<M$）[⊖]。内存中块的某些子集驻留在高速缓存的槽中，如果读存储器中某一个块的某一个字，而这个块又不在槽中，则这个块被转移到一个槽中。由于块的数目比槽多，一个槽不可能唯一或永久对应于一个块。因此，每个槽中有一个标签，用以标识当前存储的是哪一个块。标签通常是地址中较高的若干位，表示以这些位开始的所有地址。

举一个简单的例子，假设我们有一个 6 位地址和 2 位标签。标签 01 表示由下列地址单元组成的块：010000、010001、010010、010011、010100、010101、010110、010111、011000、011001、011010、011011、011100、011101、011110、011111。

⊖　符号<<表示远远小于；类似地，符号>>表示远远大于。

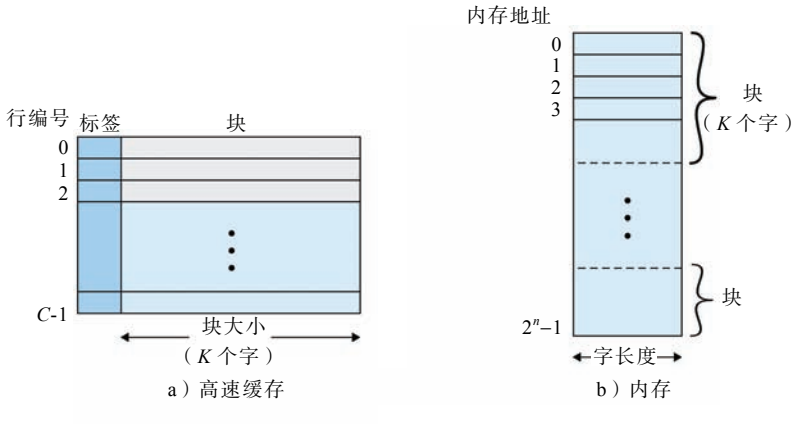


图 1.17　高速缓存/内存的结构

图 1.18 显示了读操作的过程。处理器生成要读的字的地址 RA，如果这个字在高速缓存中，它将被传递给处理器；否则，包含这个字的块将被装入高速缓存，然后这个字被传递给处理器。

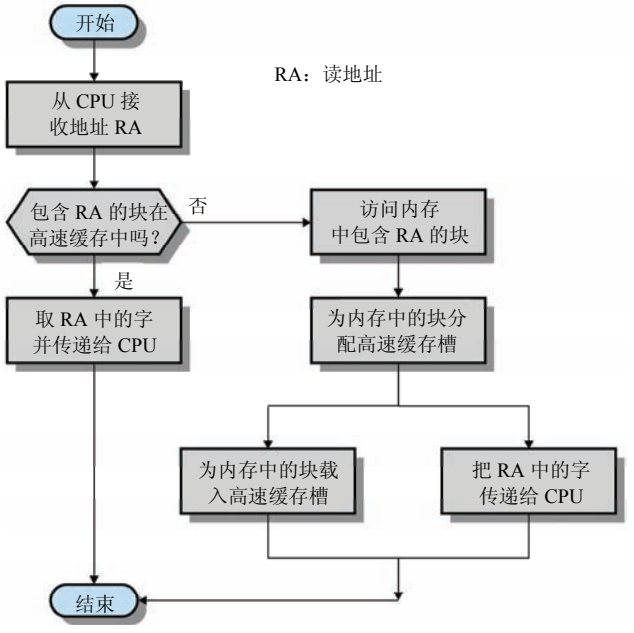


图 1.18　高速缓存读操作

1.6.3　高速缓存设计

有关高速缓存设计的详细内容已超出了本书的范围，这里只简单地概括主要的设计因素。我们将会看到在进行虚拟存储器和磁盘高速缓存设计时，也必须解决类似的设计问题。这些问题可分为：高速缓存大小、块大小、映射函数、替换算法、写策略。

前面已经讨论了**高速缓存大小**的问题，结论是适当小的高速缓存可以对性能产生显著的影响。另一个尺寸问题是关于**块大小**的，即高速缓存与内存间的数据交换单位。当块大小从很小增长到很大时，由于局部性原理，命中率首先会增加。局部性原理指的是位于被访问字附近的数据在近期被访问到的概率比较大。当块大小增大时，更多的有用数据被取到高速缓存中。但是，当

块变得更大时，新近取到的数据被用到的可能性开始小于那些必须移出高速缓存的数据再次被用到的可能性（移出高速缓存是为了给新块让出位置），这时命中率反而开始降低。

当一个新块被读入高速缓存中时，由**映射函数**确定这个块将占据哪个高速缓存单元。设计映射函数要考虑两方面的约束。首先，当读入一个块时，另一个块可能会被替换出高速缓存。替换方法应该能够尽量减小替换出的块在不久的将来还会被用到的可能性。映射函数设计得越灵活，就有更大的余地来设计出可以增大命中率的**替换算法**。其次，如果映射函数越灵活，则完成搜索以确定某个指定块是否位于高速缓存中的功能所需要的逻辑电路也就越复杂。

在映射函数的约束下，当一个新块加入到高速缓存中时，如果高速缓存中的所有存储槽都已被别的块占满，那么替换算法要选择替换不久的将来被访问的可能性最小的块。尽管不可能找到这样的块，但是合理且有效的策略是替换高速缓存中最长时间未被访问的块。这个策略称做最近最少使用（Least-Recently-Used，LRU）算法。标识最近最少使用的块需要硬件机制支持。

如果高速缓存中某个块的内容被修改，则需要在它被换出高速缓存之前把它写回内存。**写策略**规定何时发生存储器写操作。一种极端情况是每当块被更新后就发生写操作；而另一种极端情况是只有当块被替换时才发生写操作。后一种策略减少了存储器写操作的次数，但是使内存处于一种过时的状态，这会妨碍多处理器操作以及 I/O 模块的直接内存存取。

1.7　I/O 通信技术

对 I/O 操作有三种可能的技术：可编程 I/O、中断驱动 I/O、直接内存存取（DMA）。

1.7.1　可编程 I/O

当处理器正在执行程序并遇到一个与 I/O 相关的指令时，它通过给相应的 I/O 模块发命令来执行这个指令。使用可编程 I/O 操作时，I/O 模块执行请求的动作并设置 I/O 状态寄存器中相应的位，它并不进一步通知处理器，尤其是它并不中断处理器。因此处理器在执行 I/O 指令后，还要定期检查 I/O 模块的状态，以确定 I/O 操作是否已经完成。

如果使用这种技术，处理器负责从内存中提取数据以用于输出，并在内存中保存数据以用于输入。I/O 软件应该设计为由处理器执行直接控制 I/O 操作的指令，包括检测设备状态、发送读命令或写命令和传送数据，因此指令集中包括以下几类 I/O 指令：

- **控制**：用于激活外部设备，并告诉它做什么。例如，可以指示磁带倒退或前移一个记录。
- **状态**：用于测试与 I/O 模块及其外围设备相关的各种状态条件。
- **传送**：用于在存储器寄存器和外部设备间读数据或写数据。

图 1.19a 给出了使用可编程 I/O 的一个例子：从外部设备读取一块数据（如磁带中的一条记录）到存储器，每次读一个字（例如 16 位）的数据。对读入的每个字，处理器必须停留在状态检查周期，直到确定该字已经在 I/O 模块的数据寄存器中了。这个流程图说明了该技术的主要缺点：这是一个耗时的处理，处理器总是处于没有用的繁忙中。

1.7.2　中断驱动 I/O

可编程 I/O 的问题是处理器通常必须等待很长的时间，以确定 I/O 模块是否做好了接收或发送更多数据的准备。处理器在等待期间必须不断地询问 I/O 模块的状态，其结果是严重地降低了整个系统的性能。

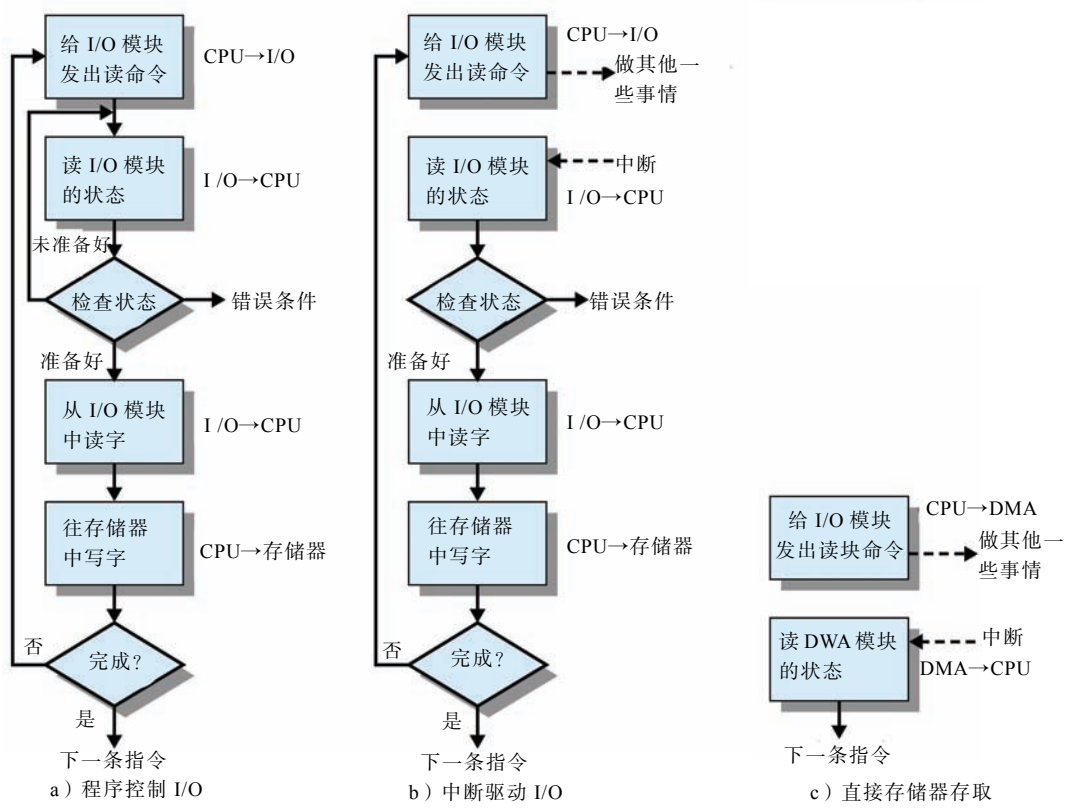


图 1.19　用于输入一块数据的三种技术

另一种选择是处理器给模块发送 I/O 命令，然后继续做其他一些有用的工作。当 I/O 模块准备好与处理器交换数据时，它将打断处理器的执行并请求服务。处理器和前面一样执行数据传送，然后恢复处理器以前的执行过程。

首先，从 I/O 模块的角度考虑这是如何工作的。对于输入操作，I/O 模块从处理器中接收一个 READ 命令，然后开始从相关的外围设备读数据。一旦数据被读入该模块的数据寄存器，模块通过控制线给处理器发送一个中断信号，然后等待直到处理器请求该数据。当处理器发出这个请求后，模块把数据放到数据总线上，然后准备下一次的 I/O 操作。

从处理器的角看度，输入操作的过程如下：处理器发一个 READ 命令，然后保存当前程序的上下文（如程序计数器和处理器寄存器），离开当前程序，去做其他事情（例如，处理器可以同时在几个不同的程序中工作）。在每个指令周期的末尾，处理器检查中断（见图 1.7）。当发生来自 I/O 模块的中断时，处理器保存当前正在执行的程序的上下文，开始执行中断处理程序（interrupt-handling program）处理此中断。在这个例子中，处理器从 I/O 模块中读取数据，并保存在存储器中。然后，恢复发出 I/O 命令的程序（或其他某个程序）的上下文并继续执行。

图 1.19b 给出了使用中断驱动 I/O 读数据块的例子。中断驱动 I/O 比可编程 I/O 更有效，这是因为它消除了不必要的等待。但是，由于数据中的每个字不论从存储器到 I/O 模块还是从 I/O 模块到存储器都必须通过处理器处理，这导致中断驱动 I/O 仍然会花费很多处理器时间。

计算机系统中不可避免有多个 I/O 模块，因此需要一定的机制，使得处理器能够确定中断是由哪个模块引发的，并且在多个中断产生的情况下处理器要决定先处理哪一个。在某些系统中有多条中断线，这样每个模块可在不同的线上发送中断信号，每个中断线有不同的优先级。

当然，也可能只有一个中断线，但要使用额外的线保存设备地址，而且不同的设备有不同的优先级。

1.7.3　直接内存存取

尽管中断驱动 I/O 比简单的可编程 I/O 更有效，但处理器仍然需要主动干预在存储器和 I/O 模块之间的数据传送，并且任何数据传送都必须完全通过处理器。因此这两种 I/O 形式都有两方面固有的缺陷：

1）I/O 传送速度受限于处理器测试设备和提供服务的速度。

2）处理器忙于管理 I/O 传送的工作，必须执行很多指令以完成 I/O 传送。

当需要移动大量的数据时，需要使用一种更有效的技术：直接内存存取（DMA）。DMA 功能可以由系统总线中一个独立的模块完成，也可以并入到一个 I/O 模块中。不论采用哪种形式，该技术的工作方式如下所示：当处理器要读或写一块数据时，它给 DMA 模块产生一条命令，发送以下信息：

- 是否请求一次读或写。
- 涉及的 I/O 设备的地址。
- 开始读或写的存储器单元。
- 需要读或写的字数。

之后处理器继续其他工作。处理器把这个操作委托给 DMA 模块，由该模块负责处理。DMA 模块直接与存储器交互，传送整个数据块，每次传送一个字。这个过程不需要处理器参与。当传送完成后，DMA 模块发一个中断信号给处理器。因此只有在开始传送和传送结束时处理器才会参与（见图 1.19c）。

DMA 模块需要控制总线以便与存储器进行数据传送。由于在总线使用中存在竞争，当处理器需要使用总线时要等待 DMA 模块。注意，这并不是一个中断，处理器没有保存上下文环境去做其他事情，而是仅仅暂停一个总线周期（在总线上传输一个字的时间）。其总的影响是在 DMA 传送过程中，当处理器需要访问总线时处理器的执行速度会变慢。尽管如此，对多字 I/O 传送来说，DMA 比中断驱动和程序控制 I/O 更有效。

1.8　推荐读物和网站

［STAL06］中更详细地讲述了本章中的所有主题，此外还有很多关于计算机组织与系统结构的其他书籍。更值得一读的有［PATT07］和［HENN07］，前者是一本全面的综述，后者是一本更高级的书籍，重点讲述设计的定量特征。

[DENN05]讲述了局部性原理的发展和应用，感兴趣的读者可以一读。

DENN05 Denning,P.*"The Locality Principle"Communications* of the ACM,July 2005.

HENN07 Hennessy,J.,and Patterson,D.Computer *Architecture:A Quantitative Approach.* San Mateo,CA: Morgan Kaufmann,2007.

PATT07 Patterson,D.,and Hennessy,J.Computer Organization and Design:*The Hardware/ Software Interface*.San Mateo,CA:Morgan Kaufmann,2007.

STAL06 Stallings,W.*Computer Organization and Architecture,*7th ed.Upper Saddle River,NJ:Prentice Hall,2006.

推荐网站

- WWW Computer Architecture Home Page：提供与计算机体系结构研究人员相关的信息索引，包括体系结构组和项目、技术组织、文献、招聘和商业信息。
- CPU Info Center：关于特定处理器的信息，包括论文、产品信息和最新通告。

1.9　关键术语、复习题和习题

地址寄存器	变址寄存器	局部性	辅助存储器
高速缓存	输入/输出（I/O）	内存	段指针
高速缓存槽	指令	多道程序设计	空间局部性
中央处理单元（CPU）	指令周期	处理器	栈
条件码	指令寄存器	程序计数器	栈帧
数据寄存器	中断	可编程 I/O	栈指针
直接内存存取（DMA）	中断驱动 I/O	可重入过程	系统总线
命中率	I/O 模块	寄存器	时间局部性

复习题

1.1　列出并简要地定义计算机的 4 个主要组成部分。
1.2　定义处理器寄存器的两种主要类别。
1.3　一般而言，一条机器指令能指定的 4 种不同的操作是什么？
1.4　什么是中断？
1.5　多中断的处理方式是什么？
1.6　内存层次的各个元素间的特征是什么？
1.7　什么是高速缓存？
1.8　列出并简要地定义 I/O 操作的三种技术。
1.9　空间局部性和时间局部性的区别是什么？
1.10　开发空间局部性和时间局部性的策略是什么？

习题

1.1　假设图 1.3 中的理想处理器还有两个 I/O 指令和一个减指令：

0011 ＝ 从 I/O 中载入 AC
0111 ＝ 把 AC 保存到 I/O 中
0100 ＝ 从 AC 中减去指定字的内容

在这种情况下，使用 12 位地址标识一个特殊的外部设备。请给出以下程序的执行过程（按照图 1.4 的格式）。

a）从设备 4 中载入 AC。
b）减去存储器单元 960 的内容。
c）把 AC 保存到设备 5 中。

假设从设备 5 中取到的下一个值为 10，960 单元中的值为 1。

1.2　本章中用 6 个步骤来描述图 1.4 中的程序执行情况，请使用 MAR 和 MBR 扩充这个描述。
1.3　假设有一个 32 位微处理器，其 32 位的指令由两个域组成：第一个字节包含操作码，其余部分为一个直接操作数或一个操作数地址。

a）最大可直接寻址的存储器能力为多少（以字节为单位）？
b）如果微处理器总线具有下面的情况，请分析对系统速度的影响：
　① 一个 32 位局部地址总线和一个 16 位局部数据总线，或者
　② 一个 16 位局部地址总线和一个 16 位局部数据总线
c）程序计数器和指令寄存器分别需要多少位？

1.4　假设有一个微处理器产生一个 16 位的地址（例如，假设程序计数器和地址寄存器都是 16 位）并且具有一个 16 位的数据总线。

a）如果连接到一个 16 位存储器上，处理器能够直接访问的最大存储器地址空间为多少？
b）如果连接到一个 8 位存储器上，处理器能够直接访问的最大存储器地址空间为多少？
c）处理访问一个独立的 I/O 空间需要哪些结构特征？

d）如果输入指令和输出指令可以表示 8 位 I/O 端口号，这个微处理器可以支持多少 8 位 I/O 端口？

1.5　考虑一个 32 位微处理器，它有一个 16 位外部数据总线，并由一个 8MHz 的输入时钟驱动。假设这个微处理器有一个总线周期，其最大持续时间等于 4 个输入时钟周期。请问该微处理器可以支持的最大数据传送速度为多少？外部数据总线增加到 21 位，或者外部时钟频率加倍，哪种措施可以更好地提高处理器性能？请叙述你的设想并解释原因。（提示：确定每个总线周期能够传送的字节数。）

1.6　考虑一个计算机系统，它包含一个 I/O 模块，用以控制一台简单的键盘/打印机电传打字设备。CPU 中包含下列寄存器，这些寄存器直接连接到系统总线上：

INPR：输入寄存器，8 位

OUTR：输出寄存器，8 位

FGI：输入标记，1 位

FGO：输出标记，1 位

IEN：中断允许，1 位

I/O 模块控制从打字机中输入击键，并输出到打印机中去。打字机可以把一个字母数字符号编码成一个 8 位字，也可以把一个 8 位字解码成一个字母数字符号。当 8 位字从打字机进入输入寄存器时，输入标记被置位；当打印一个字时，输出标记被置位。

a）描述 CPU 如何使用这 4 个寄存器实现与打字机间的输入/输出。

b）描述通过使用 IEN，如何提高执行效率？

1.7　实际上在所有包括 DMA 模块的系统中，DMA 访问内存的优先级总是高于处理器访问内存的优先级。这是为什么？

1.8　一个 DMA 模块从外部设备给内存传送字符，传送速度为 9600 位每秒（b/s）。处理器可以以每秒 100 万次的速度取指令，由于 DMA 活动，处理器的速度将会减慢多少？

1.9　一台计算机包括一个 CPU 和一台 I/O 设备 D，通过一条共享总线连接到内存 M，数据总线的宽度为 1 个字。CPU 每秒最多可执行 106 条指令，平均每条指令需要 5 个处理器周期，其中 3 个周期需要使用存储器总线。存储器读/写操作使用 1 个处理器周期。假设 CPU 正在连续不断地执行后台程序，并且需要保证 95%的指令执行速度，但没有任何 I/O 指令。假设 1 个处理器周期等于 1 个总线周期，现在要在 M 和 D 之间传送大块数据。

a）若使用程序控制 I/O，I/O 每传送 1 个字需要 CPU 执行两个指令，请估计通过 D 的 I/O 数据传送的最大可能速度。

b）如果使用 DMA 传送，请估计传送速度。

1.10　考虑以下代码：

```
for(i=0; i<5; i++)
     for(j=0; j<5; j++)
          for(k=0; k<5; k++)
               a[j]=a[j]+i*k;
```

a）定义访问局部性概念，并讨论在上面代码中的访问局部性。

b）对上面的代码，请举例说明在计算机存储体系中操作系统是如何应用访问局部性原理的。

1.11　请将附录 1A 中的式（1.1）和式（1.2）推广到 n 级存储器层次结构中。

1.12　考虑一个存储器系统，它具有以下参数：

$T_c = 100\text{ns}$　　$C_c = 0.005$ 分/位

$T_m = 1\ 000\text{ns}$　　$C_m = 0.0002$ 分/位

a）5MB 的内存价格为多少？

b）使用高速缓存技术，5MB 的内存价格为多少？

c）如果有效存取时间比高速缓存存取时间多 20%，命中率 H 为多少？

1.13　一台计算机包括高速缓存、内存和一个用作虚拟存储器的磁盘。如果要存取的字在高速缓存中，存取需要 15ns；如果该字在内存中而不在高速缓存中，把它载入高速缓存需要 45ns（包括初始检查高速缓存的时间），然后再重新开始存取；如果该字不在内存中，需要 10ms 从磁盘中取出该字，复制到高速缓存中还需要 45ns，然后再重新开始存取。高速缓存的命中率为 0.8，内存的命中率为 0.7，则该系统中存取一个字的平均存取时间是多少（单位：ns）？

1.14　假设处理器使用一个栈来管理过程调用和返回。请问可以取消程序计数器而用栈指针代替吗？

附录 1A　两级存储器的性能特征

在本章中，通过使用高速缓存作为内存和处理器间的缓冲器，建立了一个两级内部存储器。这个两级结构通过开发局部性，相对于一级存储器提供了更高的性能。本附录将探讨局部性。

内存高速缓存机制是计算机系统结构的一部分，它由硬件实现，通常对操作系统是不可见的。因此，本书不再讨论这个机制，但是还有其他两种两级存储器方法：虚拟存储器和磁盘高速缓存（见表 1.2）也使用了局部性，并且至少有一部分是由操作系统实现的。我们将在第 8 章和第 11 章分别讨论它们。本附录中将介绍对三种方法都适用的两级存储器的性能特征。

表 1.2　两级存储器的特征

类　别	内存高速缓存	虚拟存储器（页面调度）	磁盘高速缓存
典型的存取时间比	5:1	10^6:1	10^6:1
内存管理系统	由特殊的硬件实现	硬件和系统软件的结合	系统软件
典型的块大小	4 到 128 字节	64 到 4096 字节	64 到 4096 字节
处理器访问的第二级	直接访问	间接访问	间接访问

局部性

两级存储器提高性能的基础是局部性原理，这在 1.5 节中曾经提到过。这个原理声明存储器访问表现出簇聚性。在很长的一段时间中，使用的簇会变化，但在很短的时间内，处理器基本上只与存储器访问中的一个固定的簇打交道。

局部性原理是很有效的，原因如下：

1）除了分支和调用指令，程序执行都是顺序的，而这两类指令在所有程序指令中只占了一小部分。因此，大多数情况下，要取的下一条指令都是紧跟在取到的上一条指令之后的。

2）很少会出现很长的连续不断的过程调用序列，继而是相应的返回序列。相反，程序中过程调用的深度窗口限制在一个很小的范围内，因此在较短的时间中，指令的引用局限在很少的几个过程中。

3）大多数循环结构都由相对比较少的几个指令重复若干次组成的。在循环过程中，计算被限制在程序中一个很小的相邻部分中。

4）在许多程序中，很多计算都涉及处理诸如数组、记录序列之类的数据结构。在大多数情况下，对这类数据结构的连续引用是对位置相邻的数据项进行操作。

以上原因在很多研究中都得到了证实。关于第 1）点，有各种分析高级语言程序行为的研究，表 1.3 列出了在执行过程中各种语句类型出现频率的主要结果，这些结果来自下面的研究。Knuth［KNUT71］分析了用于学生实习的一组 FORTRAN 程序，这是最早关于程序设计语言行为的研究。Tanenbaum［TANE78］收集了用于操作系统程序的 300 多个过程，这些过程用支持结构化程序设计的语言（SAL）编写，并发表了测量结果。Patterson 和 Sequin［PATT82］分析了一组取自编译器和用于排版、计算机辅助设计（CAD）、排序和文件比较的程序的测量结果，还研究了程序设计语言 C 和 Pascal。Huck［HUCK83］分析了用于表示各种通用的科学计算的 4 个程序，包括快速傅立叶变换和各种微分方程。关于这些语言和应用的研究达成了一致的结果：在一个程序的生命周期中，分支和调用指令仅占了执行语句中的一小部分。因此，这些研究证实了前面给出的断言 1）。

表 1.3　高级语言操作中的相对动态频率

研究	［HUCK83］	［HNUT71］	［PATT82］		［TANE78］
语言	Pascal	FORTRAN	Pascal	C	SAL
工作量	科学计算	学生	系统	系统	系统
赋值	74	67	45	38	42
循环	4	3	5	3	4
调用	1	3	15	12	12

（续）

研究	［HUCK83］	［HNUT71］	［PATT82］		［TANE78］
语言	Pascal	FORTRAN	Pascal	C	SAL
工作量	科学计算	学生	系统	系统	系统
条件	20	11	29	43	36
转移	2	9	—	3	—
其他	—	7	6	1	6

关于断言 2），［PATT85］中的研究提供了证实，这可用图 1.20 说明。图 1.20 显示了调用-返回行为，每次调用以向下和向右的线表示，每次返回以向上和向右的线表示，图中定义的深度窗口等于 5。只有调用返回序列在任何一个方向上的移动为 6 时，才引起窗口移动。正如在图中所看到的，正在执行的程序在一个固定窗口中保留了很长的一段时间。同样，对 C 和 Pascal 程序的分析表明，对深度为 8 的窗口，只有 1%的调用或返回需要移动［TAMI83］。

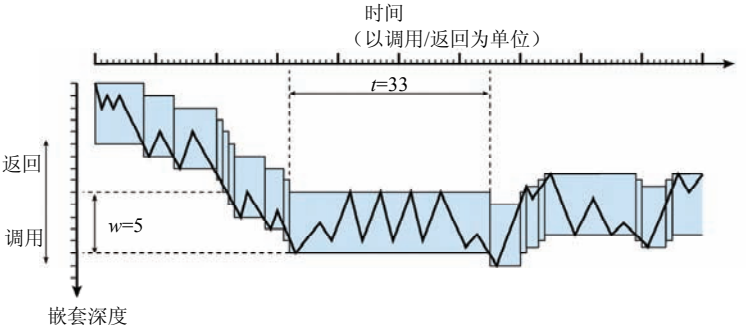


图 1.20　程序的调用返回行为示例

局部性原理在近期的很多研究中也不断得到证实。例如，图 1.21 显示了在一个站点上关于 Web 页访问模式的研究［BAEN97］。

空间局部性和时间局部性是有区别的。**空间局部性**（spatial locality）指执行涉及很多簇聚的存储器单元的趋势，这反映了处理器顺序访问指令的倾向，同时，也反映了程序顺序访问数据单元的倾向，如处理数据表。**时间局部性**（temporal locality）指处理器访问最近使用过的存储器单元的趋势，例如，当执行一个循环时，处理器重复执行相同的指令集合。

传统上，时间局部性是通过将近来使用的指令和数据值保存到高速缓存中并使用高速缓存的层次结构实现的。空间局部性通常是使用较大的高速缓存并将预取机制集成到高速缓存控制逻辑中实现的。近来，人们已经进行了许多研究，以优化这些技术，从而达到更好的性能，但基本的策略仍保持不变。

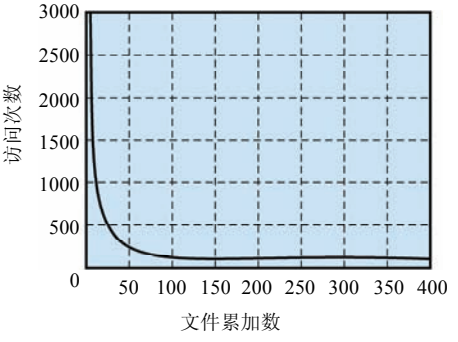


图 1.21　Web 页的访问局部性

两级存储器的操作

在两级存储器结构中也使用了局部性特性。上层存储器（M1）比下层存储器（M2）更小、更快、成本更高（每位），M1 用于临时存储空间较大的 M2 中的部分内容。当访问存储器时，首先试图访问 M1 中的项目，如果成功，就可以进行快速访问；如果不成功，则把一块存储器单元从 M2 中复制到 M1 中，再通过 M1 进行访问。由于局部性，当一个块被取到 M1 中时，将会有很多对块中单元的访问，从而加快整个服务。

为说明访问一项的平均时间，不仅要考虑两级存储器的速度，而且还包括能在 M1 中找到给定引用的概率。为此有：

$$\begin{aligned} T_s &= H \times T_1 + (1-H) \times (T_1 + T_2) \\ &= T_1 + (1-H) \times T_2 \end{aligned} \qquad (1.1)$$

其中，

T_s=（系统）平均访问时间，

T_1=M1（如高速缓存、磁盘高速缓存）的访问时间，

T_2=M2（如内存、磁盘）的访问时间，

H=命中率（访问可在 M1 中找到的次数比）。

图 1.15 显示了平均访问时间关于命中率的函数。可以看出，命中率越高，总的平均访问时间更接近 M1，而不是 M2。

性能

下面讨论与评价两级存储器机制相关的一些参数。首先考虑价格，有

$$C_s = \frac{C_1 S_1 + C_2 S_1}{S_1 + S_2} \tag{1.2}$$

其中，

C_s=两级存储器的平均每位价格，

C_1=上层存储器 M1 的平均每位价格，

C_2=下层存储器 M2 的平均每位价格，

S_1=M1 的大小，

S_2=M2 的大小。

我们希望 $C_s \approx C_2$，如果 $C_1 >> C_2$，则需要 $S_1 << S_2$。图 1.22 显示了这种关系。[⊖]

接下来考虑访问时间。为使一个两级存储器能够有重大的性能提高，需要使 T_s 近似等于 T_1（$T_s \approx T_1$）。如果 T_1 远远小于 T_2（$T_1 << T_2$），则需要命中率接近于 1。

因此，我们希望 M1 较小则可以降低价格，希望它较大则可以提高命中率，从而提高性能。是否存在能使这两种需求都在合理范围内的 M1 的大小呢？我们可以通过一系列子问题来回答这个问题：

- 满足性能要求需要多大的命中率？
- 为保证所需要的命中率，M1 的大小应为多少？
- 这个大小满足价格要求吗？

考虑值 T_1/T_s，它称做存取效率，用于衡量平均存取时间（T_s）与 M1 的存取时间（T_1）的接近程度。根据式（1.1）得到：

$$\frac{T_1}{T_s} = \frac{1}{1 + (1 - H)\frac{T_2}{T_1}} \tag{1.3}$$

图 1.23 中，T_1/T_s 被绘制成关于命中率 H 的函数，T_2/T_1 值为参数。因此，为满足性能要求，所需要的命中率为 0.8 到 0.9。

现在我们可以更准确地表达相对存储器大小的问题。对 $S_1 << S_2$，命中率为 0.8 或更高就一定合理吗？这取决于很多因素，包括正在执行软件的性质以及两级存储器的设计细节。当然，主要决定因素是局部性的程度。图 1.24 表现了局部性对命中率的影响。显然，如果 M1 和 M2 大小相等，则命中率为 1.0：所有 M2 中的项也总是存储在 M1 中。现在假设没有局部性，也就是说，访问是完全随机的。在这种情况下，如果 M1 的大小为 M2 的一半，任何时刻 M2 中有一半的项在 M1 中，因此命中率为 0.5。但是实际上，访问中总是存在某种程度的局部性，图 1.24 中给出了中等局部性和强局部性的影响。

因此，如果局部性比较强，就可能实现命中率比较大而容量相对比较小的上层存储器。例如，很多研究表明，不论内存大小为多少，小高速缓存都能产生 0.75 以上的命中率（例如［AGAR89］、［PRZY88］、［STRE83］和［SMIT82］）。通常，高速缓存大小在 1K 到 128K 个字之间都可以胜任，而内存的大小则通常在几吉字节范围内。在考虑虚拟存储器和磁盘高速缓存时，可以引用其他研究来证实同一种现象，即由于局部性，相对小的 M1 可以产生较大的命中率。

⊖ 注意两个轴都使用了对数标度。有关对数标度的基本回顾请参阅位于 WilliamStallings.com/studentSupport.html 中的 Computer Science Student Support Site 站点上的数据复习文档。

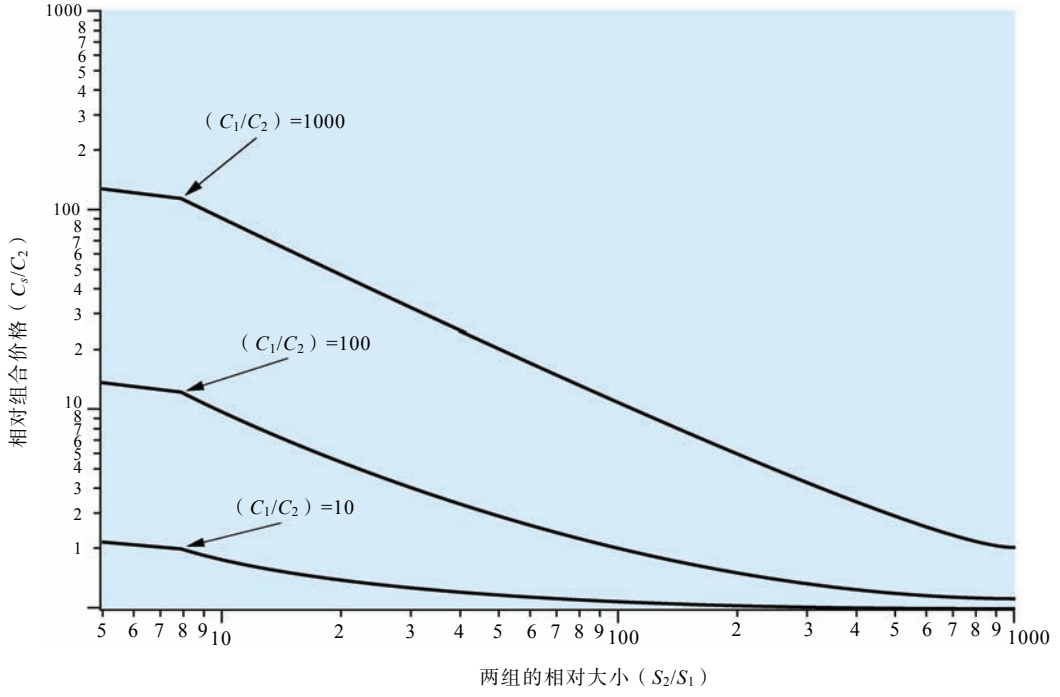


图 1.22　对一个两级存储器，存储器平均价格与存储器相对大小之间的关系

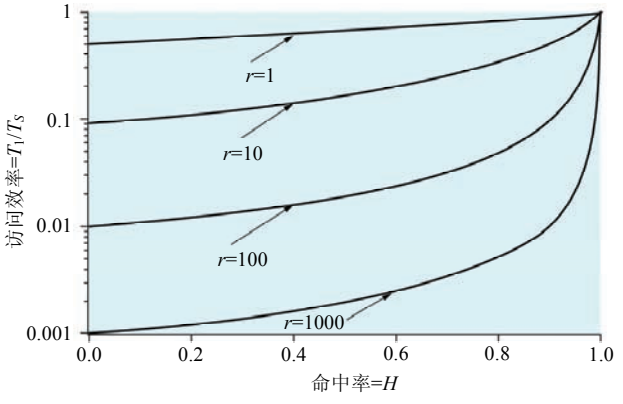


图 1.23　访问效率关于命中率的函数（$r=T_2/T_1$）

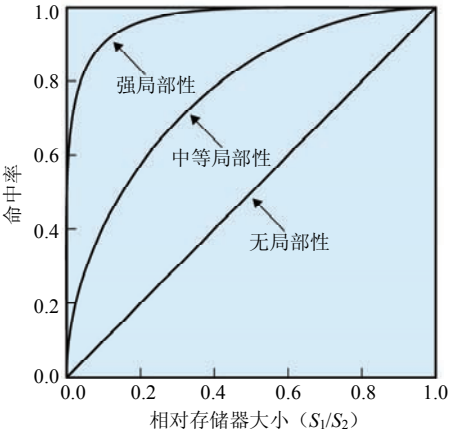


图 1.24　命中率关于相对存储器大小的函数

这就引出前面所列的最后一个问题：两个存储器的相对大小是否能满足价格要求？回答显然是肯定的。如果只需要一个相对比较小的上层存储器实现较好的性能，那么两级存储器平均每位的价格将接近比较便宜的下层存储器。

附录 1B　过程控制

控制过程调用和返回的最常用的技术是使用栈。本附录概述了栈最基本的特征，并给出了它们在过程控制中的使用方法。

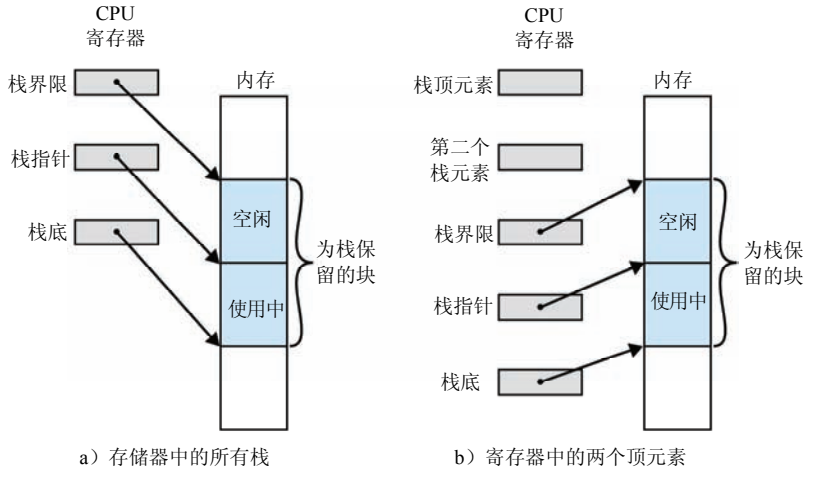
栈的实现

栈是一个有序的元素集合，一次只能访问一个元素，访问点称做栈顶。栈中的元素数目，或者说栈的长度是可变的。只可以在栈顶添加或删除数据项。基于这个原因，栈也称做下推表或后进先出（LIFO）表。

栈的实现需要有一些用于存储栈中元素的单元集合。图 1.25 给出一种典型的方法，在内存（或虚拟存储器）中为栈保留一块连续的单元。大多数时候，块中只有一部分填充着栈元素，剩余部分供栈增长时使用。正确操作需要三个地址，这些地址通常保存在处理器寄存器中。

- **栈指针**：包含栈顶地址。如果往栈中添加（PUSH）或删除（POP）一项，这个指针减 1 或加 1，以包含新的栈顶地址。
- **栈底**：包含保留块中最底层单元的地址。当往一个空栈中添加一项时，这是所用到的第一个单元。对一个空栈进行 POP 操作，则发生错误。
- **栈界限**：包含保留块中另一端，即顶端单元的地址。如果对一个满的栈进行 PUSH 操作，则发生错误。

传统上，以及在现在的大多数机器中，栈底是保留的栈块的高端地址，而栈界限是低端地址。因此，栈是从高端地址向低端地址增长的。



a）存储器中的所有栈　　b）寄存器中的两个顶元素

图 1.25　典型的栈结构

过程调用和返回

管理过程调用和返回的最常用的技术是使用栈。当处理器执行一个调用时，它将返回地址放在栈中；当执行一个返回时，它使用栈顶的地址。对于图 1.26 中的嵌套过程，图 1.27 显示了栈的使用情况。

在过程调用时，通常还需要传递参数，可以把它们传递到寄存器中。另一种可能的方法是把参数保存在存储器中的 Call 指令后，在这种情况下，参数后面必须是返回单元。这两种方法都有缺陷，如果使用寄存器，被调用程序和调用程序都必须被写入，以确保正确使用寄存器；而在存储器中保存参数，则很难交换可变数目的参数。

更灵活的参数传递方法是栈。当处理器执行一次调用时，不仅在栈中保存返回地址，而且保存传递给被调用过程的参数。被调用过程从栈中访问这些参数，在返回前，返回参数也可以放在栈中返回地址的下面。为一次过程调用保存的整个参数集合，包括返回地址，称做栈帧（stack frame）。

图 1.28 给出了一个例子，过程 P 中声明了局部变量 $x1$ 和 $x2$，过程 P 调用过程 Q，Q 中声明了局部变量 $y1$ 和 $y2$。存储在每个栈帧中的第一项指向前一帧的指针，如果参数的数量与长度是可变的，就需要用到该指针。第二项是相应于该栈帧的过程的返回点。最后，在栈帧的顶部为局部变量分配空间。局部变量可用于参数传递。例如，假设在 P 调用 Q 时，它会传递一个参数值，该参数值可存储在变量 $y1$ 中。因此，在高级语言中，P 例程中的一个指令看起来会如下所示：

CALL Q($y1$)

执行该调用时，会为 Q 创建一个新的栈帧（如图 1.28b 所示），这包含一个到 P 的栈帧的指针、到 P 的返回值以及 Q 的两个局部变量，其中一个被初始化为由 P 传递的参数。另一个局部变量 $y2$ 是由 Q 在计算过程中使用的局部变量。在栈帧中包含这样一个局部变量的目的将在后面讨论。

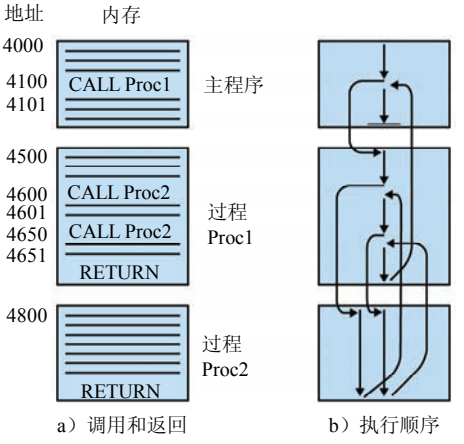


图 1.26　嵌套过程

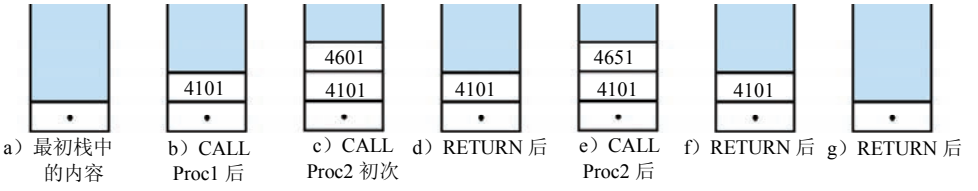


图 1.27　使用栈实现图 1.26 中的嵌套过程

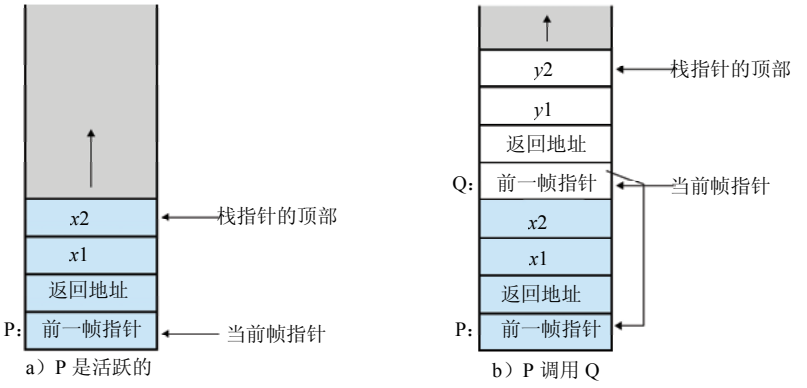


图 1.28　在使用示例过程 P 和 Q 时栈帧的生长

可重入过程

可重入过程是一个很有用的概念，特别是在同时支持多用户的系统中。可重入过程是指程序代码的一个副本在同一段时间内可以被多个用户共享使用。可重入有两个重要特征：程序代码不能修改其自身，每个用户的局部数据必须单独保存。一个可重入过程可以被中断，由一个正在中断的程序调用，在返回该过程时仍能正确执行。在共享系统中，可重入可以更有效地使用内存：程序代码的一个副本保留在内存中，有多个应用程序可以调用这个过程。

因此，可重入过程必须有一个永久不变的部分（组成过程的指令）和一个临时部分（指向调用程序的指针以及指向程序所使用的局部变量的存储地址指针）。过程的每个执行实例称做激活（activation），将执行永久部分的代码，但拥有自己的局部变量和参数的副本。与特定的激活相关联的临时部分称做**激活记录**（activation record）。

支持可重入过程最方便的方法是使用栈。当调用一个可重入过程时，该过程的激活记录保存在栈中。这样，激活记录就成为过程调用所创建的栈帧的一部分。

第 2 章　操作系统概述

本章简述操作系统的发展历史。这个历史本身很有趣且从中也可以大致了解操作系统的原理。首先在第一节介绍操作系统的目标和功能，然后讲述操作系统如何从原始的批处理系统演变成高级的多任务、多用户系统。本章的其余部分给出了两个操作系统的历史和总体特征，这两个系统将作为示例系统贯穿于本书。本章的所有内容将在后面作更深入的讲解。

2.1　操作系统的目标和功能

操作系统是控制应用程序执行的程序，并充当应用程序和计算机硬件之间的接口。它有下面三个目标：

- **方便**：操作系统使计算机更易于使用。
- **有效**：操作系统允许以更有效的方式使用计算机系统资源。
- **扩展能力**：在构造操作系统时，应该允许在不妨碍服务的前提下有效地开发、测试和引进新的系统功能。

接下来将依次介绍操作系统的这三个目标。

2.1.1　作为用户/计算机接口的操作系统

为用户提供应用的硬件和软件可以看做是一种层次结构，如图 2.1 所示。应用程序的用户，即终端用户，通常并不关心计算机的硬件细节。因此，终端用户把计算机系统看做是一组应用程序。一个应用程序可以用一种程序设计语言描述，并且由程序员开发而成。如果需要用一组完全负责控制计算机硬件的机器指令开发应用程序，将会是一件非常复杂的任务。为简化这个任务，需要提供一些系统程序，其中一部分称做实用工具，它们实现了在创建程序、管理文件和控制 I/O 设备中经常使用的功能。程序员在开发应用程序时将使用这些功能提供的接口；在应用程序运行时，将调用这些实用工具以实现特定的功能。最重要的系统程序是操作系统，操作系统为程序员屏蔽了硬件细节，并为程序员使用系统提供方便的接口。它可以作为中介，使程序员和应用程序更容易地访问和使用这些功能和服务。

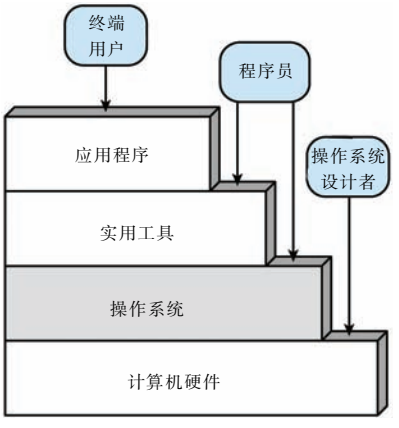


图 2.1　计算机系统的层次和视图

简单地说，操作系统通常提供了以下几个方面的服务：

- **程序开发**：操作系统提供各种各样的工具和服务，如编辑器和调试器，用于帮助程序员开发程序。通常，这些服务以实用工具程序的形式出现，严格来说并不属于操作系统核心的一部分；它们由操作系统提供，称做应用程序开发工具。
- **程序运行**：运行一个程序需要很多步骤，包括必须把指令和数据载入到内存、初始化 I/O 设备和文件、准备其他一些资源。操作系统为用户处理这些调度问题。

- **I/O 设备访问**：每个 I/O 设备的操作都需要特有的指令集或控制信号，操作系统隐藏这些细节并提供了统一的接口，因此程序员可以使用简单的读和写操作访问这些设备。
- **文件访问控制**：对操作系统而言，关于文件的控制不仅必须详细了解 I/O 设备（磁盘驱动器、磁带驱动器）的特性，而且必须详细了解存储介质中文件数据的结构。此外，对有多个用户的系统，操作系统还可以提供保护机制来控制对文件的访问。
- **系统访问**：对于共享或公共系统，操作系统控制对整个系统的访问以及对某个特殊系统资源的访问。访问功能模块必须提供对资源和数据的保护，以避免未授权用户的访问，还必须解决资源竞争时的冲突问题。
- **错误检测和响应**：计算机系统运行时可能发生各种各样的错误，包括内部和外部硬件错误，如存储器错误、设备失效或故障，以及各种软件错误，如算术溢出、试图访问被禁止的存储器单元、操作系统无法满足应用程序的请求等。对每种情况，操作系统都必须提供响应以清除错误条件，使其对正在运行的应用程序影响最小。响应可以是终止引起错误的程序、重试操作或简单地给应用程序报告错误。
- **记账**：一个好的操作系统可以收集对各种资源使用的统计信息，监控诸如响应时间之类的性能参数。在任何系统中，这个信息对于预测将来增强功能的需求以及调整系统以提高性能都是很有用的。对多用户系统，这个信息还可用于记账。

2.1.2　作为资源管理器的操作系统

一台计算机就是一组资源，这些资源用于对数据的移动、存储和处理，以及对这些功能的控制。而操作系统负责管理这些资源。

那么是否可以说是操作系统在控制数据的移动、存储和处理呢？从某个角度来看，答案是肯定的：通过管理计算机资源，操作系统控制计算机的基本功能，但是这个控制是通过一种不寻常的方式来实施的。通常，我们把控制机制想象成在被控制对象之外或者至少与被控制对象有一些差别和距离（例如，住宅供热系统是由自动调温器控制的，它完全不同于热产生和热发送装置）。但是，操作系统却不是这种情况，作为控制机制，它有两方面不同之处：

- 操作系统与普通的计算机软件作用相同，它也是由处理器执行的一段程序或一组程序。
- 操作系统经常会释放控制，而且必须依赖处理器才能恢复控制。

操作系统实际上不过是一组计算机程序，与其他计算机程序类似，它们都给处理器提供指令，主要区别在于程序的意图。操作系统控制处理器使用其他系统资源，并控制其他程序的执行时机。但是，处理器为了做任何一件这类事情，都必须停止执行操作系统程序，而去执行其他程序。因此，这时操作系统释放对处理器的控制，让处理器去做其他一些有用的工作，然后用足够长的时间恢复控制权，让处理器准备好做下一项工作。随着本章内容的深入，读者将逐渐明白所有这些机制。

图 2.2 显示了由操作系统管理的主要资源。操作系统中有一部分在内存中，其中包括**内核程序**（kernel，或称 nucleus）和当前正在使用的其他操作系统程序，内核程序包含操作系统中最常使用的功能。内存的其余部分包含用户程序和数据，它的分配由操作系统和处理器中的存储管理硬件联合控制。操作系统决定在程序运行过程中何时使用 I/O 设备，并控制文件的访问和使用。处理器自身也是一个资源，操作系统必须决定在运行一个特定的用户程序时，可以分配多少处理器时间，在多处理器系统中，这个决定要传到所有的处理器。

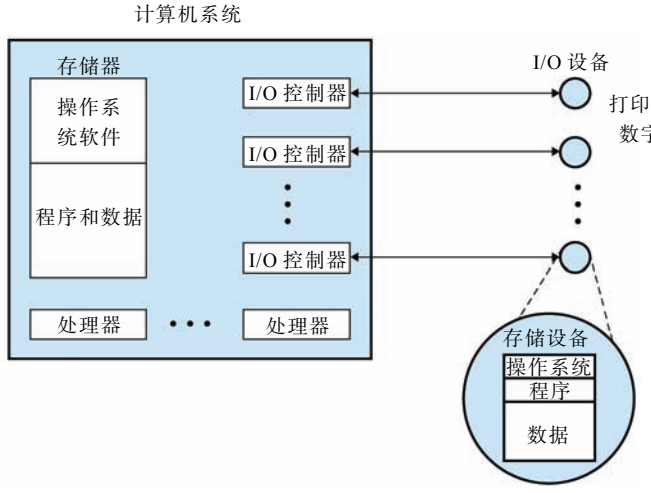


图 2.2　操作系统作为资源管理器

2.1.3　操作系统的易扩展性

一个重要的操作系统应该能够不断发展，其原因如下：

- **硬件升级和新型硬件的出现**：举一个例子，早期运行 UNIX 和 Macintosh 的处理器没有“分页”的硬件⊖，因此这两个操作系统也没有使用分页机制，而较新的版本经过修改，具备了分页功能。同样，图形终端和页面式终端替代了行滚动终端，这也将影响操作系统的设计，例如，图形终端允许用户通过屏幕上的“窗口”同时查看多个应用程序，这就要求在操作系统中提供更复杂的支持。
- **新的服务**：为适应用户的要求或满足系统管理员的需要，需要扩展操作系统以提供新的服务。例如，如果发现用现有的工具很难保持较好的性能，操作系统就必须增加新的度量和控制工具。
- **纠正错误**：任何一个操作系统都有错误，随着时间的推移这些错误逐渐被发现并会引入相应的补丁程序。当然，补丁本身也可能会引入新的错误。

操作系统经常性的变化对它的设计提出一定的要求。一个非常明确的观点是，在构造系统时应该采用模块化的结构，清楚地定义模块间的接口，并备有说明文档。对于像现代操作系统这样的大型程序，简单的模块化是不够的［DENN80a］，也就是说，不能只是简单地把程序划分为模块，还需要做更多的工作。在本章的后续部分将继续讨论这个问题。

2.2　操作系统的发展

了解这些年来操作系统的发展历史，有助于理解操作系统的关键性设计需求，也有助于理解现代操作系统基本特征的意义。

2.2.1　串行处理

对于早期的计算机，从 20 世纪 40 年代后期到 20 世纪 50 年代中期，程序员都是直接与计算机硬件打交道的，因为当时还没有操作系统。这些机器都是在一个控制台上运行的，控制台包括显示灯、触发器、某种类型的输入设备和打印机。用机器代码编写的程序通过输入设备（如卡片阅读机）载入计算机。如果一个错误使得程序停止，错误原因由显示灯指示。如果程序正常完成，输出结果将出现在打印机中。

⊖　分页将在本章后面简短讨论，详细讨论请参阅第 7 章。

这些早期系统引出了两个主要问题：

- **调度**：大多数装置都使用一个硬拷贝的登记表预订机器时间。通常，一个用户可以以半小时为单位登记一段时间。有可能用户登记了 1 小时，而只用了 45 分钟就完成了工作，在剩下的时间中计算机只能闲置，这时就会导致浪费。另一方面，如果用户遇到一个问题，没有在分配的时间内完成工作，在解决这个问题之前就会被强制停止。
- **准备时间**：一个程序称做**作业**，它可能包括往内存中加载编译器和高级语言程序（源程序），保存编译好的程序（目标程序），然后加载目标程序和公用函数并链接在一起。每一步都可能包括安装或拆卸磁带，或者准备卡片组。如果在此期间发生了错误，用户只能全部重新开始。因此，在程序运行前的准备需要花费大量的时间。

这种操作模式称做**串行处理**，反映了用户必须顺序访问计算机的事实。后来，为使串行处理更加有效，开发了各种各样的系统软件工具，其中包括公用函数库、链接器、加载器、调试器和 I/O 驱动程序，它们作为公用软件，对所有的用户来说都是可用的。

2.2.2　简单批处理系统

早期的计算机是非常昂贵的，同时由于调度和准备而浪费时间是难以接受的，因此最大限度地利用处理器是非常重要的。

为提高利用率，人们有了开发批处理操作系统的想法。第一个批处理操作系统（同时也是第一个操作系统）是 20 世纪 50 年代中期由 General Motors 开发的，用在 IBM 701 上［WEIZ81］。这个系统随后经过进一步的改进，被很多 IBM 用户在 IBM 704 中实现。在 20 世纪 60 年代早期，许多厂商为他们自己的计算机系统开发了批处理操作系统，用于 IBM 7090/7094 计算机的操作系统 IBSYS 最为著名，它对其他系统有着广泛的影响。

简单批处理方案的中心思想是使用一个称做**监控程序**的软件。通过使用这类操作系统，用户不再直接访问机器，相反，用户把卡片或磁带中的作业提交给计算机操作员，由他把这些作业按顺序组织成一批，并将整个批作业放在输入设备上，供监控程序使用。每个程序完成处理后返回到监控程序，同时，监控程序自动加载下一个程序。

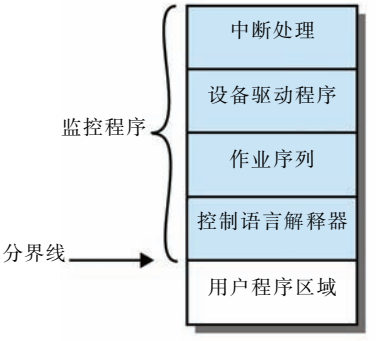


图 2.3　常驻监控程序的内存布局

为了理解这个方案如何工作，可以从以下两个角度进行分析：监控程序角度和处理器角度。

- **监控程序角度**：监控程序控制事件的顺序。为做到这一点，大部分监控程序必须总是处于内存中并且可以执行（见图 2.3），这部分称做**常驻监控程序**（resident monitor）。其他部分包括一些实用程序和公用函数，它们作为用户程序的子程序，在需要用到它们的作业开始执行时被载入。监控程序每次从输入设备（通常是卡片阅读机或磁带驱动器）中读取一个作业。读入后，当前作业被放置在用户程序区域，并且把控制权交给这个作业。当作业完成后，它将控制权返回给监控程序，监控程序立即读取下一个作业。每个作业的结果被发送到输出设备（如打印机），交付给用户。
- **处理器角度**：从某个角度看，处理器执行内存中存储的监控程序中的指令，这些指令读入下一个作业并存储到内存中的另一个部分。一旦已经读入一个作业，处理器将会遇到监控程序中的分支指令，分支指令指导处理器在用户程序的开始处继续执行。处理器继而执行用户程序中的指令，直到遇到一个结束指令或错误条件。不论哪种情况都将导致处理器从监控程序中取下一条指令。因此，“控制权交给作业”仅仅意味着处理器当前取和执行的都是用户程序中的指令，而“控制权返回给监控程序”的意思是处理器当前从监控程序中取指令并执行指令。

监控程序完成调度功能：一批作业排队等候，处理器尽可能迅速地执行作业，没有任何空闲时间。监控程序还改善了作业的准备时间，每个作业中的指令均以一种**作业控制语言**（Job Control Language，JCL）的基本形式给出。这是一种特殊类型的程序设计语言，用于为监控程序提供指令。举一个简单的例子，用户提交一个用 FORTRAN 语言编写的程序以及程序需要用到的一些数据，所有 FORTRAN 指令和数据在一个单独打孔的卡片中，或者是磁带中一个单独的记录。除了 FORTRAN 指令和数据行，作业中还包括作业控制指令，这些指令以“$”符号打头。作业的整体格式如下所示：

```
$JOB
$FTN
.
.      } FORTRAN 指令
.
$LOAD
$RUN
.
.      } 数据
.
$END
```

为执行这个作业，监控程序读$FTN 行，从海量存储器（通常为磁带）中载入合适的语言编译器。编译器将用户程序翻译成目标代码，并保存在内存或海量存储器中。如果保存在内存中，则操作称做“编译、加载和运行”。如果保存在磁带中，就需要$LOAD 指令。在编译操作之后监控程序重新获得控制权，此时监控程序读$LOAD 指令，启动一个加载器，并将控制权转移给它，加载器将目标程序载入内存（在编译器所占的位置中）。在这种方式中，有一大段内存可以由不同的子系统共享，但是每次只能运行一个子系统。

在用户程序的执行过程中，任何输入指令都会读入一行数据。用户程序中的输入指令导致调用一个输入例程，输入例程是操作系统的一部分，它检查输入以确保程序并不是意外读入一个 JCL 行。如果是这样，就会发生错误，控制权转移给监控程序。用户作业完成后，监控程序扫描输入行，直到遇到下一条 JCL 指令。因此，不管程序中的数据行太多或太少，系统都受保护。

可以看出，监控程序或者说批处理操作系统，只是一个简单的计算机程序。它依赖于处理器可以从内存的不同部分取指令的能力，以交替地获取或释放控制权。此外，还考虑到了其他硬件功能：

- **内存保护**：当用户程序正在运行时，不能改变包含监控程序的内存区域。如果试图这样做，处理器硬件将发现错误，并将控制转移给监控程序，监控程序取消这个作业，输出错误信息，并载入下一个作业。
- **定时器**：定时器用于防止一个作业独占系统。在每个作业开始时，设置定时器，如果定时器时间到，用户程序被停止，控制权返回给监控程序。
- **特权指令**：某些机器指令设计成特权指令，只能由监控程序执行。如果处理器在运行一个用户程序时遇到这类指令，则会发生错误，并将控制权转移给监控程序。I/O 指令属于特权指令，因此监控程序可以控制所有 I/O 设备，此外还可以避免用户程序意外地读到下一个作业中的作业控制指令。如果用户程序希望执行 I/O，它必须请求监控程序为自己执行这个操作。
- **中断**：早期的计算机模型并没有中断能力。这个特征使得操作系统在让用户程序放弃控制权或从用户程序获得控制权时具有更大的灵活性。

内存保护和特权指令引入了操作模式的概念。用户程序执行在**用户态**，在这个模式下，有些内存区域是受到保护的，特权指令也不允许执行。监控程序运行在系统态，也可以称为**内核态**，在这个模式下，可以执行特权指令，而且受保护的内存区域也是可以访问的。

当然，没有这些功能也可以构造操作系统。但是，计算机厂商很快认识到这样做会造成混乱，因此，即使是相对比较原始的批处理操作系统也提供这些硬件功能。

对批处理操作系统来说，用户程序和监控程序交替执行。这样做存在两方面的缺点：一部分内存交付给监控程序；监控程序消耗了一部分机器时间。所有这些都构成了系统开销，尽管存在系统开销，但是简单的批处理系统还是提高了计算机的利用率。

2.2.3　多道程序设计批处理系统

即便对由简单批处理操作系统提供的自动作业序列，处理器仍然经常是空闲的。问题在于I/O 设备相对于处理器速度太慢。图 2.4 详细列出了一个有代表性的计算过程，这个计算过程所涉及的程序用于处理一个记录文件，并且平均每秒处理 100 条指令。在这个例子中，计算机 96%的时间都是用于等待 I/O 设备完成文件数据传送。图 2.5a 显示了这种只有一个单独程序的情况，称做单道程序设计（uniprogramming）。处理器花费一定的运行时间进行计算，直到遇到一个 I/O 指令，这时它必须等到这个 I/O 指令结束后才能继续进行。

从文件中读一条记录	15 微秒
执行 100 条指令	1 微秒
往文件中写一条记录	15 微秒
总计	31 微秒
CPU 利用率= 1/31 = 0.032=3.2%	

图 2.4　系统利用率实例

这种低效率是可以避免的。内存空间可以保存操作系统（常驻监控程序）和一个用户程序。假设内存空间容得下操作系统和两个用户程序，那么当一个作业需要等待 I/O 时，处理器可以切换到另一个可能并不在等待 I/O 的作业（见图 2.5b）。进一步还可以扩展存储器以保存三个、四个或更多的程序，并且在它们之间进行切换（见图 2.5c）。这种处理称做**多道程序设计**（multiprogramming）或**多任务处理**（multitasking），它是现代操作系统的主要方案。

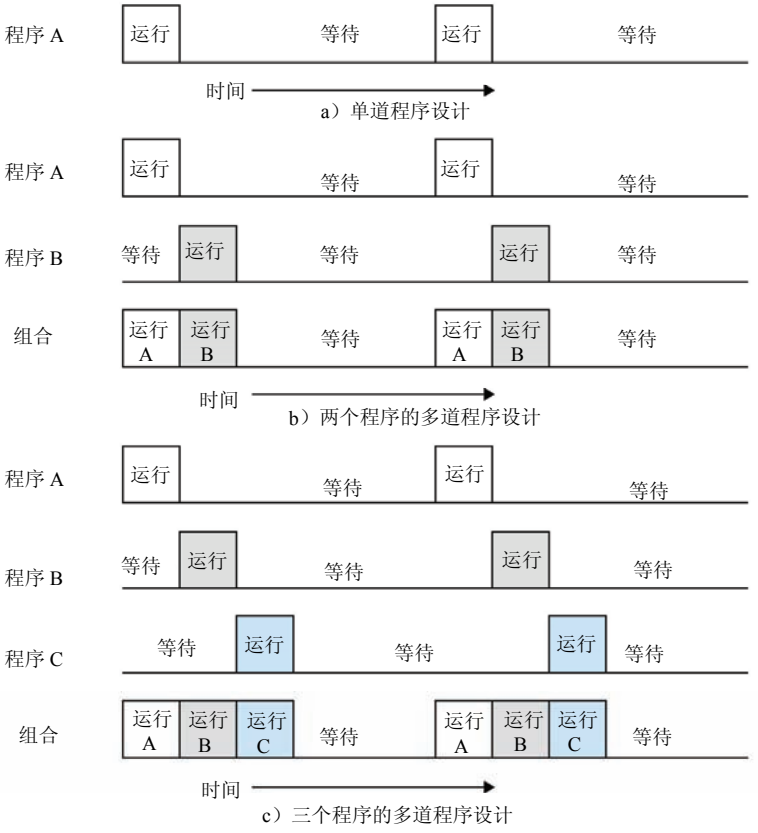


图 2.5　多道程序设计实例

这里给出一个简单的例子来说明多道程序设计的好处。考虑一台计算机，它有 250M 字节的可用存储器（没有被操作系统使用）、磁盘、终端和打印机，同时提交执行三个程序：JOB1、JOB2 和 JOB3，它们的属性在表 2.1 中列出。假设 JOB2 和 JOB3 对处理器只有最低的要求，JOB3 还要求连续使用磁盘和打印机。对于简单的批处理环境，这些作业将顺序执行。因此，JOB1 在 5 分钟后完成，JOB2 必须等待到这 5 分钟过后，然后在这之后 15 分钟完成，而 JOB3 则在 20 分钟后才开始，即从它最初被提交开始，30 分钟后才完成。表 2.2 中的单道程序设计列出了平均资源利用率、吞吐量和响应时间，图 2.6a 显示了各个设备的利用率。显然，在整个所需要的 30 分钟时间内，所有资源都没有得到充分使用。

现在假设作业在多道程序操作系统下并行运行。由于作业间几乎没有资源竞争，所有这三个作业都可以在计算机中同时存在其他作业的情况下，在几乎最小的时间内运行（假设 JOB2 和 JOB3 均分配到了足够的处理器时间，以保证它们的输入和输出操作处于活动状态）。JOB1 仍然需要 5 分钟完成，但这个时间末尾，JOB2 也完成了三分之一，而 JOB3 则完成了一半。所有这三个作业将在 15 分钟内完成。由图 2.6b 中的直方图可获得表 2.2 中多道程序设计中的那一列数据，从中可以看出性能的提高是很明显的。

表 2.1　示例程序执行属性

类　　别	JOB1	JOB2	JOB3
作业类型	大量计算	大量 I/O	大量 I/O
持续时间	5 分钟	15 分钟	10 分钟
需要的内存	50M	100M	75M
是否需要磁盘	否	否	是
是否需要终端	否	是	否
是否需要打印机	否	否	是

表 2.2　多道程序设计中的资源利用结果

类　　别	单道程序设计	多道程序设计
处理器使用	20%	40%
存储器使用	33%	67%
磁盘使用	33%	67%
打印机使用	33%	67%
总共运行时间	30 分钟	15 分钟
吞吐率	6 个作业/小时	12 个作业/小时
平均响应时间	18 分钟	10 分钟

和简单的批处理系统一样，多道程序批处理系统必须依赖于某些计算机硬件功能，对多道程序设计有用的最显著的辅助功能是支持 I/O 中断和直接存储器访问（DMA）的硬件。通过中断驱动的 I/O 或 DMA，处理器可以为一个作业发出 I/O 命令，当设备控制器执行 I/O 操作时，处理器执行另一个作业；当 I/O 操作完成时，处理器被中断，控制权被传递给操作系统中的中断处理程序，然后操作系统把控制权传递给另一个作业。

多道程序操作系统比单个程序或**单道程序**系统相对要复杂一些。对准备运行的多个作业，它们必须保留在内存中，这就需要**内存管理**（memory management）。此外，如果多个作业都准备运行，处理器必须决定运行哪一个，这需要某种调度算法。这些概念将在本章后面部分详细讲述。

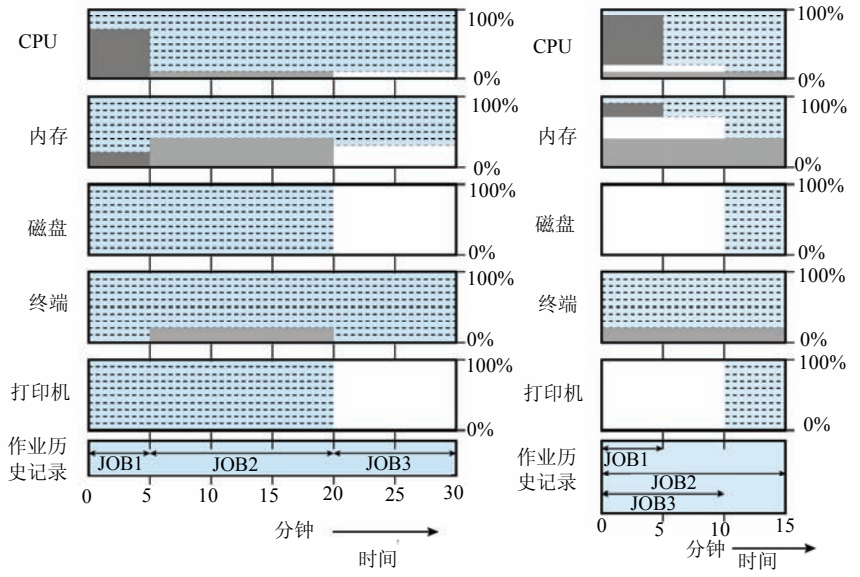


图 2.6　利用率直方图

2.2.4　分时系统

通过使用多道程序设计，可以使批处理变得更加有效。但是，对许多作业来说，需要提供一种模式，以使用户可以直接与计算机交互。实际上，对一些作业如事务处理，交互模式是必需的。

当今，通常使用专用的个人计算机或工作站来完成交互式计算任务，但这在 20 世纪 60 年代却是行不通的，当时大多数计算机都非常庞大而且昂贵，因而分时系统应运而生。

正如多道程序设计允许处理器同时处理多个批作业一样，它还可以用于处理多个交互作业。对后一种情况，由于多个用户分享处理器时间，因而该技术称做**分时**（time sharing）。在分时系统中，多个用户可以通过终端同时访问系统，由操作系统控制每个用户程序以很短的时间为单位交替执行。因此，如果有 n 个用户同时请求服务，若不计操作系统开销，每个用户平均只能得到计算机有效速度的 1/n。但是由于人的反应时间相对比较慢，所以一个设计良好的系统，其响应时间应该可以接近于专用计算机。

批处理和分时都使用了多道程序设计，其主要差别如表 2.3 所示。

第一个分时操作系统是由麻省理工学院（MIT）开发的兼容分时系统（Compatible Time-Sharing System，CTSS）[CORB62]，源于多路存取计算机项目（Machine-Aided Cognition 或 Multiple-Access Computers，Project MAC），该系统最初是在 1961 年为 IBM 709 开发的，后来又移植到 IBM 7094 中。

表 2.3　批处理多道程序设计和分时的比较

项　　目	批处理多道程序设计	分　　时
主要目标	充分使用处理器	减小响应时间
操作系统指令源	作业提供的作业控制语言命令	从终端键入的命令

与后来的系统相比，CTSS 是相当原始的。该系统运行在一台内存为 32 000 个 36 位字的机器上，常驻监控程序占用了 5000 个。当控制权被分配给一个交互用户时，该用户的程序和数据

被载入到内存剩余的 27 000 个字的空间中。程序通常在第 5000 个字单元处开始被载入，这简化了监控程序和内存管理。系统时钟以大约每 0.2 秒一个的速度产生中断，在每个时钟中断处，操作系统恢复控制权，并将处理器分配给另一位用户。因此，在固定的时间间隔内，当前用户被剥夺，另一个用户被载入。这项技术称为**时间片**（time slicing）技术。为了以后便于恢复，保留老的用户程序状态，在新的用户程序和数据被读入之前，老的用户程序和数据被写出到磁盘。随后，当获得下一次机会时，老的用户程序代码和数据被恢复到内存中。

为减小磁盘开销，只有当新来的程序需要重写用户存储空间时，用户存储空间才被写出。这个原理如图 2.7 所示。假设有 4 个交互用户，其存储器需求如下：

- JOB1：15 000
- JOB2：20 000
- JOB3：5000
- JOB4：10 000

最初，监控程序载入 JOB1 并把控制权转交给它，如图 2.7a 所示。稍后，监控程序决定把控制权转交给 JOB2，由于 JOB2 比 JOB1 需要更多的存储空间，JOB1 必须先被写出，然后载入 JOB2，如图 2.7b 所示。接下来，JOB3 被载入并运行，但是由于 JOB3 比 JOB2 小，JOB2 的一部分仍然留在存储器中，以减少写磁盘的时间，如图 2.7c 所示。稍后，监控程序决定把控制交回 JOB1，当 JOB1 载入存储器时，JOB2 的另外一部分将被写出，如图 2.7d 所示。当载入 JOB4 时，JOB1 的一部分和 JOB2 的一部分仍留在存储器中，如图 2.7e 所示。此时，如果 JOB1 或 JOB2 被激活，则只需要载入一部分。在这个例子中是 JOB2 接着运行，这就要求 JOB4 和 JOB1 留在存储器中的那一部分被写出，然后读入 JOB2 的其余部分，如图 2.7f 所示。

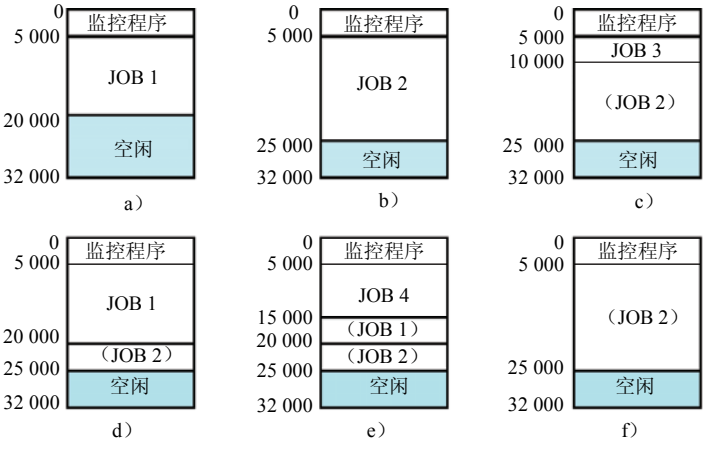


图 2.7　CTSS 操作

与当今的分时系统相比，CTSS 是一种原始的方法，但它可以工作。它非常简单，从而使监控程序最小。由于一个作业经常被载入到存储器中相同的单元，因而在载入时不需要重定位技术（在后面讲述）。这个技术仅仅写出必须的内容，可以减少磁盘的活动。在 7094 上运行时，CTSS 最多可支持 32 个用户。

分时和多道程序设计引发了操作系统中的许多新问题。如果内存中有多个作业，必须保护它们不相互干扰，例如不会修改其他作业的数据。有多个交互用户时，必须对文件系统进行保护，只有授权用户才可以访问某个特定的文件，还必须处理资源（如打印机和海量存储器）竞争问题。在本书中会经常遇到这样或那样的问题以及可能的解决方法。

2.3　主要的成就

操作系统是最复杂的软件之一，这反映在为了达到那些困难的甚至相互冲突的目标（方便、有效和易扩展性）而带来的挑战上。[DENN80a]提出在操作系统开发中的5个重要的理论进展：进程、内存管理、信息保护和安全、调度和资源管理、系统结构。

每个进展都是为了解决实际的难题，并由相关原理或抽象概念来描述的。这5个领域包括了现代操作系统设计和实现中的关键问题。本节给出对这5个领域的简单回顾，也可作为本书其余部分的综述。

2.3.1　进程

进程的概念是操作系统结构的基础，Multics的设计者在20世纪60年代首次使用了这个术语[DALE68]，它比作业更通用一些。存在很多关于进程的定义，如下所示：

- 一个正在执行的程序。
- 计算机中正在运行的程序的一个实例。
- 可以分配给处理器并由处理器执行的一个实体。
- 由单一的顺序的执行线程、一个当前状态和一组相关的系统资源所描述的活动单元。

后面将会对这个概念进行更清晰的阐述。

计算机系统的发展有三条主线：多道程序批处理操作、分时和实时事务系统，它们在时间安排和同步中所产生的问题推动了进程概念的发展。正如前面所讲的，多道程序设计是为了让处理器和I/O设备（包括存储设备）同时保持忙状态，以实现最大效率。其关键机制是：在响应表示I/O事务结束的信号时，操作系统将对内存中驻留的不同程序进行处理器切换。

发展的第二条主线是通用的分时。其主要设计目标是能及时响应单个用户的要求，但是由于成本原因，又要可以同时支持多个用户。由于用户反应时间相对比较慢，这两个目标是可以同时实现的。例如，如果一个典型用户平均需要每分钟2秒的处理时间，则可以有近30个这样的用户共享同一个系统，并且感觉不到互相的干扰。当然，在这个计算中，还必须考虑操作系统的开销因素。

发展的另一个重要主线是实时事务处理系统。在这种情况下，很多用户都在对数据库进行查询或修改，例如航空公司的预订系统。事务处理系统和分时系统的主要差别在于前者局限于一个或几个应用，而分时系统的用户可以从事程序开发、作业执行以及使用各种各样的应用程序。对于这两种情况，系统响应时间都是最重要的。

系统程序员在开发早期的多道程序和多用户交互系统时使用的主要工具是中断。一个已定义事件（如I/O完成）的发生可以暂停任何作业的活动。处理器保存某些上下文（如程序计数器和其他寄存器），然后跳转到中断处理程序中，处理中断，然后恢复用户被中断作业或其他作业的处理。

设计出一个能够协调各种不同活动的系统软件是非常困难的。在任何时刻都有许多作业在运行中，每个作业都包括要求按顺序执行的很多步骤，因此，分析事件序列的所有组合几乎是不可能的。由于缺乏能够在所有活动中进行协调和合作的系统级的方法，程序员只能基于他们对操作系统所控制的环境的理解，采用自己的特殊方法。然而这种方法是很脆弱的，尤其对于一些程序设计中的小错误，因为这些错误只有在很少见的事件序列发生时才会出现。由于需要从应用程序软件错误和硬件错误中区分出这些错误，因而诊断工作是很困难的。即使检测出错误，也很难确定其原因，因为很难再现错误产生的精确场景。一般而言，产生这类错误有4个主要原因[DENN80a]：

- **不正确的同步**：常常会出现这样的情况，即一个例程必须挂起，等待系统中其他地方的某一事件。例如，一个程序启动了一个 I/O 读操作，在继续进行之前必须等到缓冲区中有数据。在这种情况下，需要来自其他例程的一个信号，而设计不正确的信号机制可能导致信号丢失或接收到重复信号。
- **失败的互斥**：常常会出现多个用户或程序试图同时使用一个共享资源的情况。例如，两个用户可能试图同时编辑一个文件。如果不控制这种访问，就会发生错误。因此必须有某种互斥机制，以保证一次只允许一个例程对一部分数据执行事务处理。很难证明这类互斥机制的实现对所有可能的事件序列都是正确的。
- **不确定的程序操作**：一个特定程序的结果只依赖于该程序的输入，而并不依赖于共享系统中其他程序的活动。但是，当程序共享内存并且处理器控制它们交错执行时，它们可能会因为重写相同的内存区域而发生不可预测的相互干扰。因此，程序调度顺序可能会影响某个特定程序的输出结果。
- **死锁**：很可能有两个或多个程序相互挂起等待。例如，两个程序可能都需要两个 I/O 设备执行一些操作（如从磁盘复制到磁带）。一个程序获得了一个设备的控制权，而另一个程序获得了另一个设备的控制权，它们都等待对方释放自己想要的资源。这样的死锁依赖于资源分配和释放的时机安排。

解决这些问题需要一种系统级的方法监控处理器中不同程序的执行。进程的概念为此提供了基础。进程可以看做是由三部分组成的：

- 一段可执行的程序
- 程序所需要的相关数据（变量、工作空间、缓冲区等）
- 程序的执行上下文

最后一部分是根本。**执行上下文**（execution context）又称做**进程状态**（process state），是操作系统用来管理和控制进程所需的内部数据。这种内部信息和进程是分开的，因为操作系统信息不允许被进程直接访问。上下文包括操作系统管理进程以及处理器正确执行进程所需要的所有信息。包括了各种处理器寄存器的内容，如程序计数器和数据寄存器。它还包括操作系统使用的信息，如进程优先级以及进程是否在等待特定 I/O 事件的完成。

图 2.8 给出了一种进程管理的方法。两个进程 A 和 B，存在于内存的某些部分。也就是说，给每个进程（包含程序、数据和上下文信息）分配一块存储器区域，并且在由操作系统建立和维护的进程表中进行记录。进程表包含记录每个进程的表项，表项内容包括指向包含进程的存储块地址的指针，还包括该进程的部分或全部执行上下文。执行上下文的其余部分存放在别处，可能和进程自己保存在一起（如图 2.8 所示），通常也可能保存在内存里一块独立的区域中。进程索引寄存器（process index register）包含当前正在控制处理器的进程在进程表中的索引。程序计数器指向该进程中下一条待执行的指令。基址寄存器（base register）和界限寄存器（limit register）定义了该进程所占据的存储器区域：基址寄存器中保存了该存储器区域的开始地址，界限寄存器中保存了该区域的大小（以字节或字为单位）。程序计数器和所有的数据引用相对于基址寄存器被解释，并且不能超过界限寄存器中的值，这就可以保护内部进程间不会相互干涉。

在图 2.8 中，进程索引寄存器表明进程 B 正在执行。以前执行的进程被临时中断，在 A 中断的同时，所有寄存器的内容被记录在它的执行上下文环境中，以后操作系统就可以执行进程切换，恢复进程 A 的执行。进程切换过程包括保存 B 的上下文和恢复 A 的上下文。当在程序计数器中载入指向 A 的程序区域的值时，进程 A 自动恢复执行。

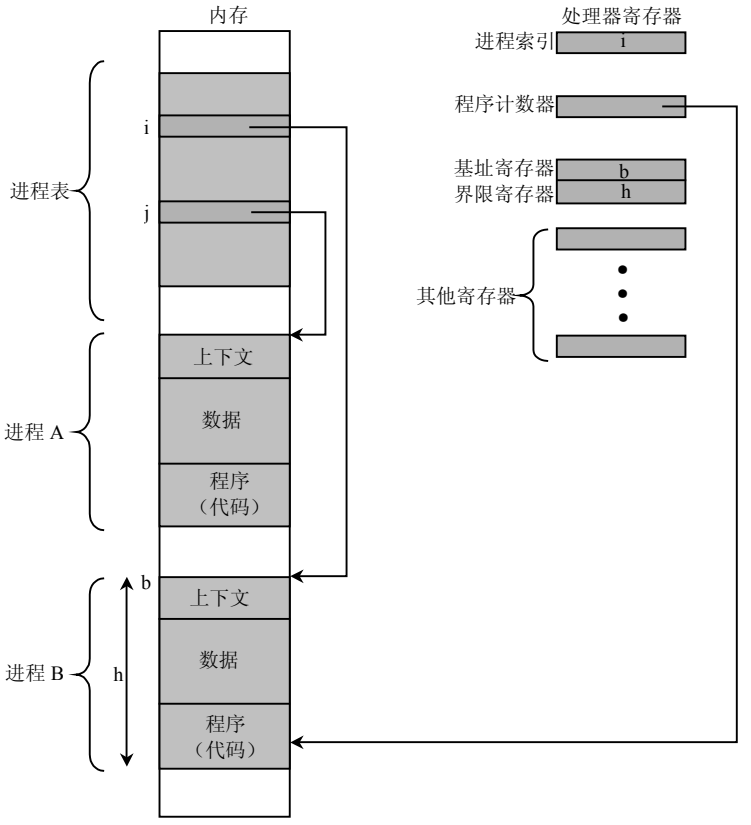


图 2.8　典型的进程实现方法

因此，进程被当做数据结构来实现。一个进程可以是正在执行，也可以是等待执行。任何时候整个进程**状态**都包含在它的上下文环境中。这个结构使得可以开发功能强大的技术，以确保在进程中进行协调和合作。在操作系统中可能会设计和并入一些新的功能（如优先级），这可以通过扩展上下文环境以包括支持这些特征的新信息。在本书中，将有很多关于使用进程结构解决在多道程序设计和资源共享中出现的问题的例子。

2.3.2　内存管理

通过支持模块化程序设计的计算环境和数据的灵活使用，用户的要求可以得到很好的满足。系统管理员需要有效且有条理地控制存储器分配。操作系统为满足这些要求，担负着 5 个基本的存储器管理责任：

- **进程隔离**：操作系统必须保护独立的进程，防止互相干涉各自的存储空间，包括数据和指令。
- **自动分配和管理**：程序应该根据需要在存储层次间动态地分配，分配对程序员是透明的。因此，程序员无需关心与存储限制有关的问题，操作系统有效地实现分配问题，可以仅在需要时才给作业分配存储空间。
- **支持模块化程序设计**：程序员应该能够定义程序模块，并且动态地创建、销毁模块，动态地改变模块大小。
- **保护和访问控制**：不论在存储层次中的哪一级，存储器的共享都会产生一个程序访问另一个程序存储空间的潜在可能性。当一个特定的应用程序需要共享时，这是可取的。但

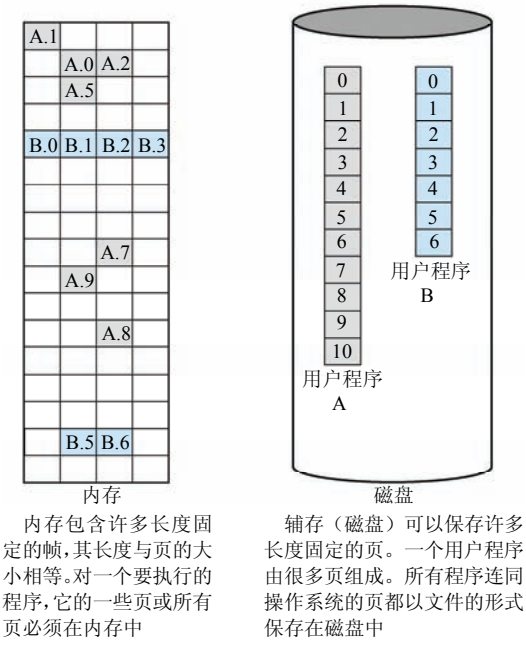
在别的时候，它可能威胁到程序的完整性，甚至威胁到操作系统自身。操作系统必须允许一部分内存可以由各种用户以各种方式进行访问。

- **长期存储**：许多应用程序需要在计算机关机后长时间保存信息。

在典型情况下，操作系统使用虚拟存储器和文件系统机制来满足这些要求。文件系统实现了长期存储，它在一个有名字的对象中保存信息，这个对象称做文件。对程序员来说，文件是一个很方便的概念；对操作系统来说，文件是访问控制和保护的一个有用单元。

虚拟存储器机制允许程序从逻辑的角度访问存储器，而不考虑物理内存上可用的空间数量。虚拟存储器的构想是为了满足有多个用户作业同时驻留在内存中的要求，这样，当一个进程被写出到辅助存储器中并且后继进程被读入时，在连续的进程执行之间将不会脱节。由于进程大小不同，如果处理器在很多进程间切换，则很难把它们紧密地压入内存中，因此引进了分页系统。在分页系统中，进程由许多固定大小的块组成，这些块称做页。程序通过**虚地址**（virtual address）访问字，虚地址由页号和页中的偏移量组成。进程的每一页都可以放置在内存中的任何地方，分页系统提供了程序中使用的虚地址和内存中的**实地址**（real address）或物理地址之间的动态映射。

有了动态映射硬件，下一逻辑步骤是消除一个进程的所有页同时驻留在内存中的要求。一个进程的所有页都保留在磁盘中，当进程执行时，一部分页在内存中。如果需要访问的某一页不在内存中，存储管理硬件可以检测到，然后安排载入这个缺页。这个配置称做**虚拟内存**，如图 2.9 所示。



内存包含许多长度固定的帧，其长度与页的大小相等。对一个要执行的程序，它的一些页或所有页必须在内存中

辅存（磁盘）可以保存许多长度固定的页。一个用户程序由很多页组成。所有程序连同操作系统的页都以文件的形式保存在磁盘中

图 2.9　虚拟存储器概念

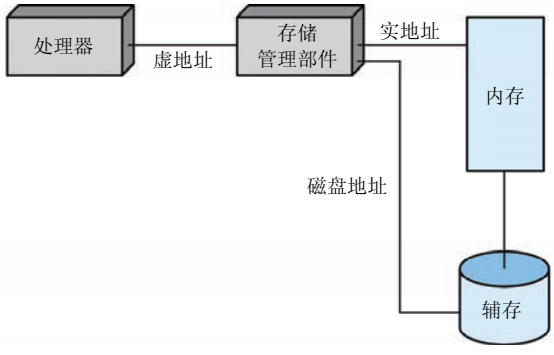


图 2.10　虚拟存储器寻址

处理器硬件和操作系统一起提供给用户“虚拟处理器”的概念，而“虚拟处理器”有对虚拟存储器的访问权。这个存储器可以是一个线性地址空间，也可以是段的集合，而段是可变长度的连续地址块。不论哪种情况，程序设计语言的指令都可以访问虚拟存储器区域中的程序和数据。可以通过给每个进程一个唯一的不重叠的虚拟存储器空间来实现进程隔离；可以通过使两个虚拟

存储器空间的一部分重叠来实现内存共享；文件可用于长期存储，文件或其中一部分可以复制到虚拟存储器中供程序操作。

图 2.10 显示了虚拟存储器方案中的寻址关系。存储器由内存和低速的辅助存储器组成，内存可直接访问到（通过机器指令），外存则可以通过把块载入内存间接访问到。地址转换硬件（映射器）位于处理器和内存之间。程序使用虚地址访问，虚地址将映射成真实的内存地址。如果访问的虚地址不在实际内存中，实际内存中的一部分内容将换到外存中，然后换入所需要的数据块。在这个活动过程中，产生这个地址访问的进程必须被挂起。操作系统设计者的任务是开发开销很少的地址转换机制，以及可以减小各级存储器级间交换量的存储分配策略。

2.3.3　信息保护和安全

信息保护是在使用分时系统时提出的，近年来计算机网络进一步关注和发展了这个问题。由于环境不同，涉及一个组织的威胁的本质也不同。但是，有一些通用工具可以嵌入支持各种保护和安全机制的计算机和操作系统内部。总之，我们关心对计算机系统的控制访问和其中保存的信息。

大多数与操作系统相关的安全和保护问题可以分为 4 类：

- **可用性**：保护系统不被打断。
- **保密性**：保证用户不能读到未授权访问的数据。
- **数据完整性**：保护数据不被未授权修改。
- **认证**：涉及用户身份的正确认证和消息或数据的合法性。

2.3.4　调度和资源管理

操作系统的一个关键任务是管理各种可用资源（内存空间、I/O 设备、处理器），并调度各种活动进程使用这些资源。任何资源分配和调度策略都必须考虑三个因素：

- **公平性**：通常希望给竞争使用某一特定资源的所有进程提供几乎相等和公平的访问机会。对同一类作业，也就是说有类似请求的作业，更是需要如此。
- **有差别的响应性**：另一方面，操作系统可能需要区分有不同服务要求的不同作业类。操作系统将试图做出满足所有要求的分配和调度决策，并且动态地做出决策。例如，如果一个进程正在等待使用一个 I/O 设备，操作系统会尽可能迅速地调度这个进程，从而释放这个设备以方便其他进程使用。
- **有效性**：操作系统希望获得最大的吞吐量和最小的响应时间，并且在分时的情况下，能够容纳尽可能多的用户。这些标准互相矛盾，在给定状态下寻找适当的平衡是操作系统中一个正在进行研究的问题。

调度和资源管理任务是一个基本的操作系统研究问题，并且可以应用数学研究成果。此外，系统活动的度量对监视性能并进行调节是非常重要的。

图 2.11 给出了多道程序设计环境中涉及进程调度和资源分配的操作系统主要组件。操作系统中维护着多个队列，每个队列代表等待某些资源的进程的简单列表。短期队列由在内存中（或至少最基本的一小部分在内存中）并等待处理器可用时随时准备运行的进程组成。任何一个这样的进程都可以在下一步使用处理器，究竟选择哪一个取决于短期调度器，或者称为分派器（dispatcher）。一个常用的策略是依次给队列中的每个进程一定的时间，这称为**时间片轮转**（round-robin）技术，时间片轮转技术使用了一个环形队列。另一种策略是给不同的进程分配不同的优先级，根据优先级进行调度。

长期队列是等待使用处理器的新作业的列表。操作系统通过把长期队列中的作业转移到短期队列中，实现往系统中添加作业，这时内存的一部分必须分配给新到来的作业。因此，操作系统

要避免由于允许太多的进程进入系统而过量使用内存或处理时间。每个 I/O 设备都有一个 I/O 队列，可能有多个进程请求使用同一个 I/O 设备。所有等待使用一个设备的进程在该设备的队列中排队，同时操作系统必须决定把可用的 I/O 设备分配给哪一个进程。

如果发生了一个中断，则操作系统在中断处理程序入口得到处理器的控制权。进程可以通过服务调用明确地请求某些操作系统的服务，如 I/O 设备处理服务。在这种情况下，服务调用处理程序是操作系统的入口点。在任何情况下，只要处理中断或服务调用，就会请求短期调度器选择一个进程执行。

前面所述的是一个功能描述，关于操作系统这部分的细节和模块化的设计，在各种系统中各不相同。操作系统中这方面的研究大多针对选择算法和数据结构，其目的是提供公平性、有差别的响应性和有效性。

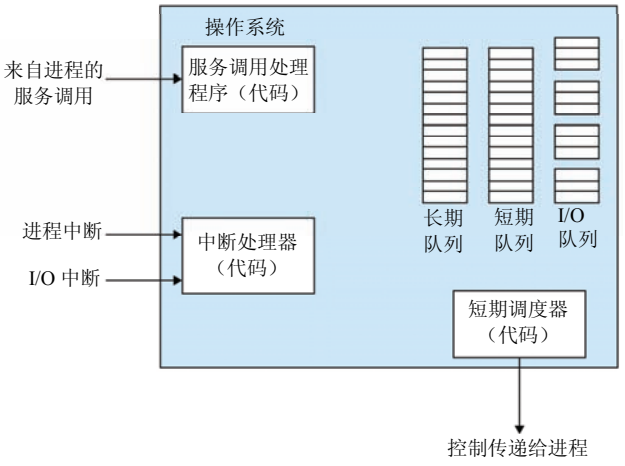


图 2.11　用于多道程序设计的操作系统的主要组件

2.3.5　系统结构

随着操作系统中增加了越来越多的功能，并且底层硬件变得更强大、更加通用，导致操作系统的大小和复杂性也随着增加。MIT 在 1963 年投入使用的 CTSS，大约包含 32 000 个 36 位字；一年后，IBM 开发的 OS/360 有超过 100 万条的机器指令；1975 年 MIT 和 Bell 实验室开发的 Multics 系统增长到了 2 000 万条机器指令。近年来，确实也针对一些小型系统引入过比较简单的操作系统，但是随着底层硬件和用户需求的增长，它们也不可避免地变得越来越复杂。因此，当今的 UNIX 系统要比那些天才的程序员在 20 世纪 70 年代早期开发的那个小系统复杂得多，而简单的 MS-DOS 让位于具有更多更复杂能力的 OS/2 和 Windows 操作系统。例如，Windows NT 4.0 包含 1 600 万行代码，而 Windows 2000 的代码量则超过这个数目的两倍。

一个功能完善的操作系统的大小和它所处理的任务的困难性，导致了 4 个让人遗憾但又普遍存在的问题。第一，操作系统在交付使用时就习惯性地表现出落后，这就要求有新的操作系统或升级老的系统。第二，随着时间的推移会发现越来越多潜在的缺陷，这些缺陷必须及时修复。第三，总是难以达到期望的性能。第四，理论表明，不可能开发出既复杂的又不易受各种包括病毒、蠕虫和未授权访问之类的安全性攻击的操作系统。

为管理操作系统的复杂性并克服这些问题，多年来操作系统的软件结构得到广泛的重视。有几点是显而易见的：软件必须是模块化的，这有助于组织软件开发过程、限定诊断范围和修正错误；模块相互之间必须有定义很好的接口，接口必须尽可能简单，这不但可以简化程序设计任务，

还可以使系统的扩展更加容易。通过模块间简单清楚的接口，当一个模块改变时，对其他模块的影响可以减到最小。

对于运行数百万到数千万条代码的大型操作系统，仅仅有模块化程序设计是不够的，软件体系结构和信息抽象的概念正得到越来越广泛的使用。现代操作系统的层次结构按照复杂性、时间刻度、抽象级进行功能划分。我们可以把系统看做是一系列的层。每一层执行操作系统所需功能的相关子集。它依赖于下一个较低层，较低层执行更为原始的功能并隐藏这些功能的细节。它还给相邻的较高层提供服务。在理想情况下，可以通过定义层使得改变一层时不需要改变其他层。因此我们把一个问题分解成几个更易于处理的子问题。

通常情况下，较低层的处理时间很短。操作系统的某些部分必须直接与计算机硬件交互，这里，事件的时间刻度仅为几十亿分之一秒。而另一端，操作系统的某些功能直接与用户交互，用户发出命令的频率则要小得多，可能每隔几秒钟发一次命令。使用层次结构可以很好地与这种频率差别场景保持一致。

这些原理的应用方式在不同的操作系统中有很大的不同。但是，为了获得操作系统的一个概观，这里给出一个层次操作系统模型是很有用的。让我们来看一下［BROW84］和［DENN84］中提出的模型，尽管该模型没有对应于特定的操作系统，但它提供了一个从高层来看待操作系统结构的视角。模型的定义见表 2.4，由下面几层组成：

表 2.4　操作系统设计层次

层	名　称	对　象	示例操作
13	shell	用户程序设计环境	shell 语言中的语句
12	用户进程	用户进程	退出、终止、挂起和恢复
11	目录	目录	创建、销毁、连接、分离、查找和列表
10	设备	外部设备，如打印机、显示器和键盘	打开、关闭、读和写
9	文件系统	文件	创建、销毁、打开、关闭、读和写
8	通信	管道	创建、销毁、打开、关闭、读和写
7	虚拟存储器	段、页	读、写和取
6	本地辅助存储器	数据块、设备通道	读、写、分配和空闲
5	原始进程	原始进程、信号量、准备就绪列表	挂起、恢复、等待和发信号
4	中断	中断处理程序	调用、屏蔽、去屏蔽和重试
3	过程	过程、调用栈、显示	标记栈、调用、返回
2	指令集合	计算栈、微程序解释器、标量和数组数据	加载、保存、加操作、减操作、转移
1	电路	寄存器、门、总线等	清空、传送、激活、求反

注：阴影部分表示硬件。

- **第 1 层**：由电路组成，处理的对象是寄存器、存储单元和逻辑门。定义在这些对象上的操作是动作，如清空寄存器或读取存储单元。
- **第 2 层**：处理器指令集合。该层定义的操作是机器语言指令集合允许的操作，如加、减、加载和保存。
- **第 3 层**：增加了过程或子程序的概念，以及调用/返回操作。
- **第 4 层**：引入了中断，能导致处理器保存当前环境、调用中断处理程序。

前面这 4 层并不是操作系统的一部分，而是构成了处理器的硬件。但是，操作系统的一些元素开始在这些层出现，如中断处理程序。从第 5 层开始，才真正到达了操作系统，并开始出现和多道程序设计相关的概念。

- **第 5 层**：在这一层引入了进程的概念，用来表示程序的执行。操作系统运行多个进程的

基本要求包括挂起和恢复进程的能力，这就要求保存硬件寄存器，使得可以从一个进程切换到另一个。此外，如果进程需要合作，则需要一些同步方法。操作系统设计中一个最简单的技术和重要的概念是信号量，简单信号机制将在第 5 章讲述。

- **第 6 层**：处理计算机的辅助存储设备。在这一层出现了定位读/写头和实际传送数据块的功能。第 6 层依赖于第 5 层对操作的调度和当一个操作完成后通知等待进程该操作已完成的能力。更高层涉及对磁盘中所需数据的寻址，并向第 5 层中的设备驱动程序请求相应的块。
- **第 7 层**：为进程创建一个逻辑地址空间。这一层把虚地址空间组织成块，可以在内存和外存之间移动。比较常用的有三个方案：使用固定大小的页、使用可变长度的段或两者都用。当所需要的块不在内存中时，这一层的逻辑将请求第 6 层的传送。

至此，操作系统处理的都是单处理器的资源。从第 8 层开始，操作系统处理外部对象，如外围设备、网络和网络中的计算机。这些位于高层的对象都是逻辑对象，命名对象可以在同一台计算机或在多台计算机间共享。

- **第 8 层**：处理进程间的信息和消息通信。尽管第 5 层提供了一个原始的信号机制，用于进程间的同步，但这一层处理更丰富的信息共享。用于此目的的最强大的工具之一是管道（pipe），它是为进程间的数据流提供的一个逻辑通道。一个管道定义成它的输出来自一个进程，而它的输入是到另一个进程中去。它还可用于把外部设备或文件链接到进程。这个概念将在第 6 章中讲述。
- **第 9 层**：支持称为文件的长期存储。在这一层，辅助存储器中的数据可以看做是一个抽象的可变长度的实体。这与第 6 层辅助存储器中面向硬件的磁道、簇和固定大小的块形成对比。
- **第 10 层**：提供访问外部设备的标准接口。
- **第 11 层**：负责维护系统资源和对象的外部标识符与内部标识符间的关联。外部标识符是应用程序和用户使用的名字；内部标识符是一个地址或可被操作系统低层使用、用于定位和控制一个对象的其他指示符。这些关联在目录中维护，目录项不仅包括外部/内部映射，而且包括诸如访问权之类的特性。
- **第 12 层**：提供了一个支持进程的功能完善的软件设施，这和第 5 层中所提供的大不相同。第 5 层只维护与进程相关的处理器寄存器内容和用于调度进程的逻辑，而第 12 层支持进程管理所需的全部信息，这包括进程的虚地址空间、可能与进程发生交互的对象和进程的列表以及对交互的约束、在进程创建后传递给进程的参数和操作系统在控制进程时可能用到的其他特性。
- **第 13 层**：为用户提供操作系统的一个界面。它之所以称做**命令行解释器**（shell），是因为它将用户和操作系统细节分离开，而简单地把操作系统作为一组服务的集合提供给用户。命令行解释器接受用户命令或作业控制语句，对它们进行解释，并在需要时创建和控制进程。例如，这一层的界面可以用图形方式实现，即通过菜单提供用户可以使用的命令，并输出结果到一个特殊设备（如显示器）来显示。

这个操作系统的假设模型提供了一个有用的可描述结构，可以用作实现操作系统的指南。在本书后面讲述某个特定的设计问题时，可返回参考此结构。

2.4　现代操作系统的特征

在过去数年中，操作系统的结构和功能得到逐步发展。但是近年来，许多新的设计要素引入到新操作系统以及现有操作系统的新版本中，使操作系统产生了本质性的变化。这些现代操作系

统针对硬件中的新发展、新的应用程序和新的安全威胁。促使操作系统发展的硬件因素主要有：包含多处理器的计算机系统、高速增长的机器速度、高速网络连接和容量不断增加的各种存储设备。多媒体应用、Internet 和 Web 访问、客户/服务器计算等应用领域也影响着操作系统的设计。在安全性方面，互联网的访问增加了潜在的威胁和更加复杂的攻击，例如病毒、蠕虫和黑客技术，这些都对操作系统的设计产生了深远的影响。

对操作系统要求上的变化速度之快不仅需要修改和增强现有的操作系统体系结构，而且需要有新的操作系统组织方法。在实验用和商用操作系统中有很多不同的方法和设计要素，大致可以分为：微内核体系结构、多线程、对称多处理、分布式操作系统、面向对象设计。

至今，大多数操作系统都有一个**单体内核**（monolithic kernel），大多数认为是操作系统应该提供的功能由这些大内核提供，包括调度、文件系统、网络、设备驱动器、存储管理等。典型情况下，这个大内核是作为一个进程实现的，所有元素都共享相同的地址空间。**微内核体系结构**只给内核分配一些最基本的功能，包括地址空间、进程间通信（InterProcess Communication，简称 IPC）和基本的调度。其他的操作系统服务都是由运行在用户态下且与其他应用程序类似的进程提供，这些进程可根据特定的应用和环境需求进行定制，有时也称这些进程为服务器。这种方法把内核和服务程序的开发分离开，可以为特定的应用程序或环境要求定制服务程序。微内核方法可以使系统结构的设计更加简单、灵活，很适合于分布式环境。实质上，微内核可以以相同的方式与本地和远程的服务进程交互，使分布式系统的构造更为方便。

多线程技术是指把执行一个应用程序的进程划分成可以同时运行的多个线程。线程和进程有以下差别：

- **线程**：可分派的工作单元。它包括处理器上下文环境（包含程序计数器和栈指针）和栈中自己的数据区域（为允许子程序分支）。线程顺序执行，并且是可中断的，这样处理器可以转到另一个线程。
- **进程**：一个或多个线程和相关系统资源（如包含数据和代码的存储器空间、打开的文件和设备）的集合。这紧密对应于一个正在执行的程序的概念。通过把一个应用程序分解成多个线程，程序员可以在很大程度上控制应用程序的模块性和应用程序相关事件的时间安排。

多线程对执行许多本质上独立、不需要串行处理的应用程序是很有用的，例如监听和处理很多客户请求的数据库服务器。在同一个进程中运行多个线程，在线程间来回切换所涉及的处理器开销要比在不同进程间进行切换的开销少。线程对构造进程是非常有用的，进程作为操作系统内核的一部分，将在第 3 章中讲述。

到现在为止，大多数单用户的个人计算机和工作站基本上都只包含一个通用的微处理器。随着性能要求的不断增加以及微处理器价格的不断降低，计算机厂商引进了拥有多个微处理器的计算机。为实现更高的有效性和可靠性，可使用**对称多处理**（Symmetric MultiProcessing，SMP）技术。对称多处理不仅指计算机硬件结构，而且指反映该硬件结构的操作系统行为。对称多处理计算机可以定义为具有以下特征的一个独立的计算机系统：

1）有多个处理器。

2）这些处理器共享同一个内存和 I/O 设备，它们之间通过通信总线或别的内部连接方案互相连接。

3）所有处理器都可以执行相同的功能（因此称为对称）。

近年来，在单芯片上的多处理器系统（也称为单片多处理器系统）已经开始广泛应用。无论是单片多处理器还是多片对称多处理器（SMP），许多设计要点是一样的。

对称多处理操作系统可调度进程或线程到所有的处理器运行。对称多处理器结构比单处理器

结构具有更多的潜在优势，如下所示：

- **性能**：如果计算机完成的工作可组织为让一部分工作可以并行完成，那么有多个处理器的系统将比只有一个同类型处理器的系统产生更好的性能，如图 2.12 所示。对多道程序设计而言，一次只能执行一个进程，此时所有别的进程都在等待处理器。对多处理系统而言，多个进程可以分别在不同的处理器上同时运行。
- **可用性**：在对称多处理计算机中，由于所有的处理器都可以执行相同的功能，因而单个处理器的失败并不会使机器停止。相反，系统可以继续运行，只是性能有所降低。

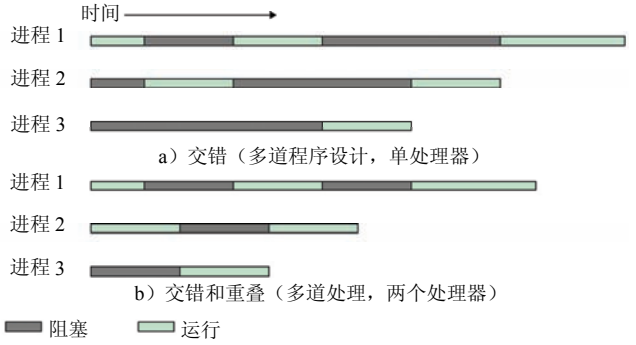


图 2.12　多道程序设计和多道处理

- **增量增长**：用户可以通过添加额外的处理器来增强系统的功能。
- **可扩展性**：生产商可以根据系统配置的处理器的数量，提供一系列不同价格和性能特征的产品。

特别需要注意的是，这些只是潜在的优点，而不是确定的。操作系统必须提供发掘对称多处理计算机系统中并行性的工具和功能。

多线程和对称多处理总是被放在一起讨论，但它们是两个独立的概念。即使在单处理器机器中，多线程对结构化的应用程序和内核进程也是很有用的。由于多个处理器可以并行运行多个进程，因而对称多处理计算机对非线程化的进程也是有用的。但是，这两个设施是互补的，一起使用将会更有效。

对称多处理技术一个很具有吸引力的特征是多处理器的存在对用户是透明的。操作系统负责在多个处理器中调度线程或进程，并且负责处理器间的同步。本书讲述了给用户提供单系统外部特征的调度和同步机制。另外一个不同的问题是给一群计算机（多机系统）提供单系统外部特征。在这种情况下，需要处理的是一群实体（计算机），每一个都有自己的内存、外存和其他 I/O 模块。**分布式操作系统**使用户产生错觉，使多机系统好像具有一个单一的内存空间、外存空间以及其他的统一存取措施，如分布式文件系统。尽管集群正变得越来越流行，市场上也有很多集群产品，但是，分布式操作系统的技术发展水平落后于单处理器操作系统和对称多处理操作系统。我们将在第八部分分析这类系统。

操作系统设计的另一个改革是使用面向对象技术。面向对象设计的原理用于给小内核增加模块化的扩展上。在操作系统一级，基于对象的结构使程序员可以定制操作系统，而不会破坏系统的完整性。面向对象技术还使得分布式工具和分布式操作系统的开发变得更容易。

2.5　微软的 Windows 概述

2.5.1　历史

Windows 的故事从一个完全不同的操作系统开始，这个操作系统是由微软公司为第一台 IBM 个人计算机开发的，称为 MS-DOS 或者 PC-DOS。最初的版本 DOS 1.0 是在 1981 年 8 月发行的，它由 4000 行汇编语言源代码组成，使用 Intel 8086 微处理器运行在 8KB 的内存中。

当 IBM 研制基于硬盘的个人计算机 PC XT 时，微软公司在 1983 年发布了 DOS 2.0，它包含

对硬盘的支持，并提供了层次目录。在此之前，磁盘只能包含一个目录，最多支持 64 个文件。这对软盘来说是足够了，但对硬盘来说就太受限制了，而且一个目录的限制过于死板。新版本允许目录包含子目录和文件，还在操作系统中嵌入了内容丰富的命令集，提供外部程序必须执行的功能，这些外部程序在第 1 版中是以实用程序的形式提供的。在增加的功能中有一些类似 UNIX 的特征，如 I/O 重定向和后台打印。I/O 重定向是指为给定的应用程序改变输入输出的能力。驻留内存部分则增长到 24KB。

当 IBM 在 1984 年发布 PC AT 时，微软发布了 DOS 3.0。AT 包含 Intel 80286 处理器，它提供扩充访问和内存保护功能，而 DOS 中却没有使用这些新功能。为保证与以前版本的兼容性，操作系统仅仅把 80286 简单地当做一个“快速的 8086”。操作系统提供了对新键盘和硬盘外围设备的支持。即便如此，对存储器的要求还是增长到了 36KB。3.0 版本有几个比较著名的升级，1984 年发布的 DOS 3.1 支持 PC 间的联网，常驻部分的大小没有改变，这是通过增加操作系统交换量实现的；1987 年发布的 DOS 3.3 提供了对新型 IBM 机器 PS/2 的支持。同样，这个版本没有使用 PS/2 中的处理器能力，这些是由 80286 和 32 位的 80386 芯片提供的。常驻部分最少增长到了 46KB，如果选择了某些可选的扩展功能，则还会需要更多的空间。

此时，DOS 所使用的环境已远远超出了它的能力。随着 80486 的引入，Intel Pentium 芯片提供的能力和功能已经不能用简单的 DOS 来开发。在此期间，从 20 世纪 80 年代早期开始，微软开始开发一种可以放入用户和 DOS 之间的图形化用户界面（GUI），其目的是为了与 Macintosh 竞争。Macintosh 的操作系统无比好用。1990 年，微软有了一个 GUI 版本，称做 Windows 3.0，其用户友好性接近 Macintosh，但是，它仍然需要运行在 DOS 之上。

在微软为 IBM 研制下一代操作系统的过程中，它希望开发新的微处理器的能力，并结合 Windows 易于使用的特点。在这种尝试失败后，微软终于超越了自我，研制出一种全新的操作系统 Windows NT。Windows NT 充分利用当代微处理器的能力，提供单用户环境或多用户环境下的多任务。

第一版 Windows NT（3.1）在 1993 年发布，它是同 Windows 3.1 有着相同 GUI 的另一个微软操作系统（Windows 3.0 的后继）。但是，Windows NT 3.1 是一个新的 32 位操作系统，具有支持老的 DOS 和 Windows 应用程序的能力，并提供了对 OS/2 的支持。

在几个 Windows NT 3.x 版本之后，微软发布了 Windows NT 4.0。Windows NT 4.0 本质上与 Windows 3.x 具有相同的内部结构，最显著的外部变化是 Windows NT 4.0 提供了与 Windows 95 相同的用户界面。主要的结构变化是在 Windows 3.x 中作为 Win32 子系统一部分的几个图形组件（在用户态下运行）被移到 Windows NT 执行体（在内核态下运行）中。这个变化的优点是加速了这些重要函数的操作，而潜在的缺陷是这些图形函数现在可以使用低层系统服务，这可能会对操作系统的可靠性产生影响。

在 2000 年，微软引进了下一个重要的升级版本，现在称为 Windows 2000。同样，底层的执行体和内核结构与 Windows NT 4.0 在根本上是相同的，但是还增加了一些新特点。Windows 2000 的重点是增加支持分布处理的服务和功能，Windows 2000 新特征的核心元素是活动目录（Active Directory），这是一个分布目录服务，能够将任意对象名映射到关于这些对象的任意类型的信息上。Windows 2000 也增加了即插即用和电源管理工具，这些特性已经在 Windows 98 中存在，继承于 Windows 95。这些特性对笔记本电脑特别重要，因为笔记本电脑经常使用扩展设备（docking station）和电池供电。

关于 Windows 2000 的最后一点是 Windows 2000 Server 和 Windows 2000 desktop 的区别。本质上，内核、执行程序结构和服务保持相同，但 Server 还包括一些用作网络服务器时所需的服务。

在 2001 年，微软发布了最新的桌面操作系统 Windows XP，同时提供了家用和商业工作站两种版本。同样在 2001 年发布了 64 位版本的 Windows XP。2003 年，微软又发布了新的服务器

版本 Windows Server 2003，包括 32 位和 64 位两种。Windows Server 2003 主要为 Intel 的 Itanium 硬件设计。在为 Sever 2003 升级的第一个服务包中，微软在桌面版和服务器版中都引入了对 AMD64 处理器结构的支持。

2007 年，Windows Vista 作为最新的桌面版本的 Windows 操作系统发布。Vista 对 Intel x86 和 AMD x64 结构都提供支持。发布版主要的改动细节在于图形用户界面（GUI）的变化，以及许多对安全性的改进。对应的服务器版本是 Windows Server 2008。

2.5.2　单用户多任务

Windows（从 Windows 2000 以后）是微机操作系统新潮流的一个重要例子（其他例子有 Linux 和 MacOS）。Windows 的动因是开发当今的 32 位和 64 位微处理器处理能力的需求，在速度、硬件完善度和存储能力几个方面与大型机和小型机进行竞争。

这些新操作系统的一个最重要的特征是，尽管它们仍然希望支持一个单独的交互用户，但它们的确是多任务操作系统。两个主要的发展因素引发了个人计算机、工作站和服务器中的多任务需求。首先，随着微处理器的速度和存储能力不断增长以及虚拟存储器的支持，应用程序变得更加复杂且相关性更强。例如，用户可能希望同时使用文字处理器、画图程序和电子表格应用程序来产生文件。如果没有多任务，用户要创建一幅图并把它粘贴到字处理文件中，则需要以下步骤：

1）打开画图程序。

2）创建一幅图并保存在一个文件中或一个临时的剪贴板中。

3）关闭画图程序。

4）打开文字处理程序。

5）在正确的位置插入这幅图。

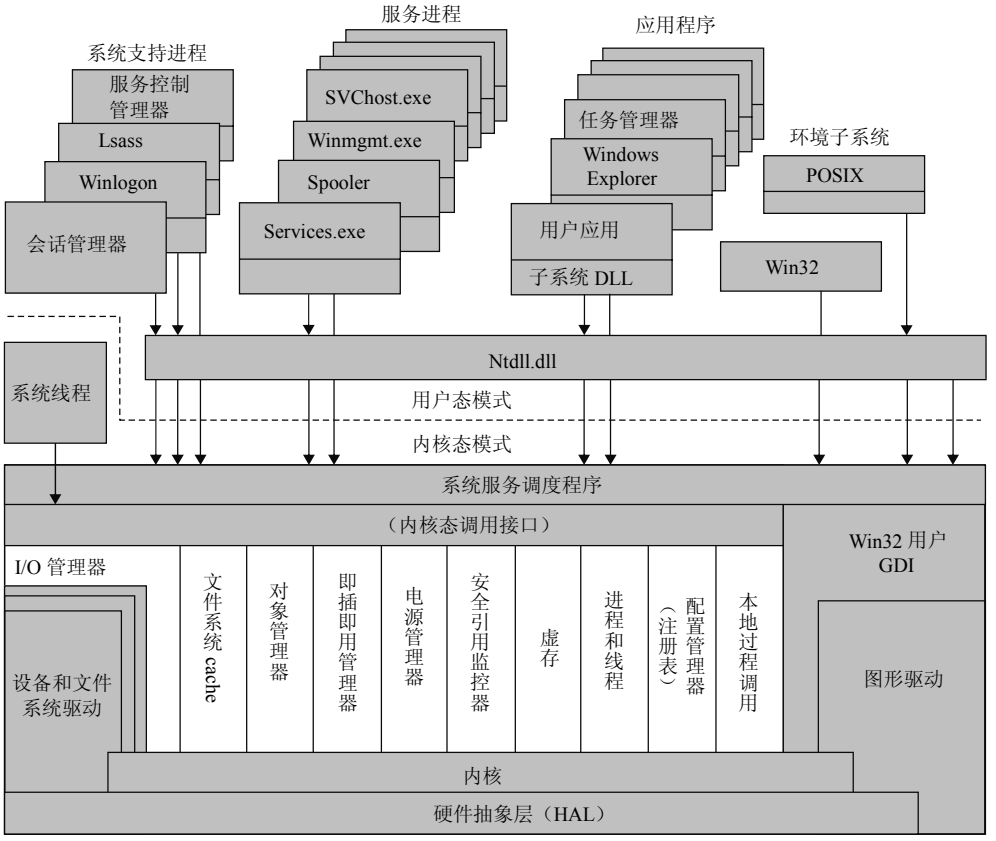
如果需要有任何修改，用户必须关闭文字处理程序，打开画图程序，编辑这个图形并保存，关闭画图程序，打开文字处理程序，插入这幅修改后的图。这很快就会变得单调乏味。随着用户可以得到的服务和能力越来越强大、种类越来越多，单任务环境变得更加笨拙、对用户不够友好。在多任务环境中，用户打开所需要的每个应用程序，并让它保持打开状态。信息可以在这些应用程序间很容易地来回移动。每个应用程序有一个或多个打开的窗口，图形界面和诸如鼠标之类的指示设备使得用户可以在环境中迅速地定位。

多任务的第二个动机是客户/服务器计算的发展。在客户/服务器计算中，个人计算机或工作站（客户）和主机系统（服务器）联合使用，以实现特定的应用。它们两个被连接在一起，每一个都被分配给一部分与其能力相适应的作业。客户/服务器可以在个人计算机和服务器的局域网中实现，或者可以通过用户系统和一台大主机（如大型机）间的连接实现。一个应用程序可能涉及一台或多台个人计算机以及一个或多个服务器设备。为提供所需的响应性，操作系统需要支持复杂的实时通信硬件和相关的通信协议以及数据传送结构，同时还应支持正在进行的用户交互。

前面是对 Windows 桌面版本的讨论。Windows Server 版本也是多任务的，但可以支持多个用户，它支持多个终端服务器连接，并提供网络中多个用户使用的共享服务。作为一个 Internet 服务器，Windows 可以支持数千个同时发生的 Web 连接。

2.5.3　体系结构

图 2.13 显示了 Windows 2000 的整体结构，以后的各种版本的 Windows，包括 Vista，在这一层本质上都有相同的结构。模块化结构给 Windows 2000 提供了相当大的灵活性，它可以在各种硬件平台上执行，可运行为各种别的操作系统编写的应用程序。截至本书写作时，Windows 仅在 Intel x86 和 AMD64 硬件平台上实现，服务器版也支持 Intel IA64（Itanium）。



Lsass 表示本地安全认证服务器　　POSIX 表示可移植操作系统接口
GDI 表示图形设备接口　　DLL 表示动态链接库
白色部分表示可执行的

图 2.13　Windows 和 Windows Vista 体系结构[RUSS05]

和几乎所有操作系统一样，Windows 把面向应用的软件和操作系统核心软件分开，后者包括在内核态下运行的执行体、内核、设备驱动器和硬件抽象层。内核模块软件可以访问系统数据和硬件，在用户态运行的其余软件被限制访问系统数据。

操作系统组织

Windows 的体系结构是高度模块化的。每个系统函数都正好由一个操作系统部件管理，操作系统的其余部分和所有应用程序通过相应的部件使用标准接口访问这个函数。关键的系统数据只能通过相应的函数访问。从理论上讲，任何模块都可以移动、升级或替换，而不需要重写整个系统或它的标准应用程序接口（API）。

Windows 的内核态组件包括以下类型：

- **执行体**：包括操作系统基础服务，例如内存管理、进程和线程管理、安全、I/O 和进程间通信。
- **内核**：控制处理器的执行。内核管理包括线程调度、进程切换、异常和中断处理、多处理器同步。跟执行体和用户级的其他部分不同，内核本身的代码并不在线程内执行。因此，内核是操作系统中唯一不可抢占或分页的一部分。
- **硬件抽象层**（Hardware Abstraction Layer，HAL）：在通用的硬件命令和响应与某一特定平台专用的命令和响应之间进行映射，它将操作系统从与平台相关的硬件差异中隔离

出来。HAL 使得每个机器的系统总线、直接存储器访问（DMA）控制器、中断控制器、系统计时器和存储器模块对内核来说看上去都是相同的。它还对对称多处理（SMP）提供支持，随后对这部分进行解释。

- **设备驱动**：用来扩展执行体的动态库。这些库包括硬件设备驱动程序，可以将用户 I/O 函数调用转换为特定的硬件设备 I/O 请求，动态库还包括一些软件构件，用于实现文件系统、网络协议和其他必须运行在内核态中的系统扩展功能。
- **窗口和图形系统**：实现图形用户界面函数（简称 GUI 函数），例如处理窗口、用户界面控制和画图。

Windows 执行体包括一些特殊的系统函数模块，并为用户态的软件提供 API。以下是对每个执行体模块的简单描述：

- **I/O 管理器**：提供了应用程序访问 I/O 设备的一个框架，还负责为进一步的处理分发合适的设备驱动程序。I/O 管理器实现了所有的 Windows I/O API，并实施了安全性、设备命名和文件系统（使用对象管理器）。Windows I/O 将在第 11 章中讲述。
- **高速缓存管理器**：通过使最近访问过的磁盘数据驻留在内存中以供快速访问，以及在更新后的数据发送到磁盘之前，通过在内存中保持一段很短的时间以延迟磁盘写操作，来提高基于文件的 I/O 性能。
- **对象管理器**：创建、管理和删除 Windows 执行体对象和用于表示诸如进程、线程和同步对象等资源的抽象数据类型。为对象的保持、命名和安全性设置实施统一的规则。对象管理器还创建对象句柄，对象句柄是由访问控制信息和指向对象的指针组成的。本节稍后将进一步讨论 Windows 对象。
- **即插即用管理器**：决定并加载为支持一个特定的设备所需的驱动。
- **电源管理器**：调整各种设备间的电源管理，并且可以把处理器置为休眠状态以达到节电的目的，甚至可以将内存中的内容写入磁盘，然后切断整个系统的电源。
- **安全访问监控程序**：强制执行访问确认和审核产生的规则。Windows 面向对象模型允许统一一致的安全视图，直到组成执行体的基本实体。因此，Windows 为所有受保护对象的访问确认和审核检查使用相同的例程，这些受保护对象包括文件、进程、地址空间和 I/O 设备。Windows 的安全性将在第 15 章讲述。
- **虚拟内存管理器**：管理虚拟地址、物理地址和磁盘上的页面文件。控制内存管理硬件和相应的数据结构，把进程地址空间中的虚地址映射到计算机内存中的物理页。Windows 虚拟内存管理将在第 8 章讲述。
- **进程/线程管理器**：创建、管理和删除对象，跟踪进程和线程对象。Windows 进程和线程管理将在第 4 章讲述。
- **配置管理器**：负责执行和管理系统注册表，系统注册表是保存系统和每个用户参数设置的数据仓库。
- **本地过程调用（Local Procedure Call，LPC）机制**：为本地进程实现服务和子系统间的通信，而实现的一套高效的跨进程的过程调用机制。类似于分布处理中远程过程调用（Remote Procedure Call，RPC）的方式。

用户态进程

Windows 支持 4 种基本的用户进程类型：

- **特殊系统进程**：需进行管理系统的用户态服务，如会话管理程序、认证子系统、服务管理程序和登录进程等。

- **服务进程**：打印机后台管理程序、事件记录器、与设备驱动协作的用户态构件、不同的网络服务程序以及许多这样的程序。微软和外部的软件开发者都要使用它们来扩展系统的功能，因为这些服务是在 Windows 系统中后台运行用户态活动的唯一方法。
- **环境子系统**：提供不同的操作系统的个性化设置（环境）。支持的子系统有 Win32/WinFX 和 POSIX。每个环境子系统包括一个在所有子系统应用程序中都会共享的子系统进程和把用户应用程序调用转换成本地过程调用（LPC）和/或原生 Windows 调用的动态链接库（DLL）。
- **用户应用程序**：可执行程序（EXE）和动态链接库（DLL）向用户提供使用系统的功能。EXE 和 DLL 一般是针对特定的环境子系统，尽管这其中有一些作为操作系统组成部分的程序使用了原生系统接口（NTAPI）。同样，也支持为 Windows 3.1 或 MS-DOS 写的 16 位程序。

Windows 支持为多操作系统特性写的应用程序，Windows 利用一组在环境子系统保护之下的通用的内核态构件来提供这种支持。每个子系统在执行时都包括一个独立的进程，该进程包含共享的数据结构、优先级和需要实现特定的个性化的执行对象的句柄。当第一个这种类型的应用程序启动时，Windows 会话管理器会启动上述的进程。子系统进程作为系统用户运行，因此执行体会保护其地址空间免受普通用户进程的影响。

受保护的子系统提供一个图形或命令行用户界面，为用户定义操作系统的外观。另外，每个受保护的子系统为那个特定的操作环境提供 API，这就意味着为那些特定的操作环境创建的应用程序可以在 Windows 下不用改变即可运行，其原因是它们看到的操作系统接口与编写它们时的接口是相同的。

最重要的子系统是 Win32。Win32 是在 Windows NT 和 Windows 95 及后继版本和 Windows 9x 中都实现了的 API。许多为 Windows 9x 系统操作系统写的 Win32 应用程序可以不用改变就能在 NT 系统上运行。在 Windows XP 版本中，微软重点改进了与 Windows 9x 的兼容性，这样他们就可以终止对 9x 系列的支持而专注于 NT 了。

最新的 Windows API 是 WinFX，基于微软的.NET 编程模型。WinFX 在 Windows 中是作为 win32 的更高一层来实现的，而不是一个独立的子系统类型。

2.5.4　客户/服务器模型

Windows 操作系统服务、受保护子系统和应用程序都采用客户/服务器计算模型构造，客户/服务器模型是分布式计算中的一种常用模型，将在本书的第六部分讲述。正如 Windows 的设计一样，在单个系统内部也可以采用相同的结构。

NT 原生 API 是一套基于内核的服务，提供一些核心抽象供系统使用，如进程、线程、虚拟内存、I/O 和通信。通过使用客户/服务器模型，Windows 在用户态进程中提供了一系列丰富的服务。环境子系统和 Windows 用户态服务都是以进程的形式实现，通过 RPC 与客户端进行通信。每个服务器进程等待客户的一个服务请求（例如，存储服务、进程创建服务或处理器调度服务）。客户可以是应用程序或另一个操作系统模块，它通过发送消息请求服务。消息从执行体发送到适当的服务器，服务器执行所请求的操作，并通过另一条消息返回结果或状态信息，再由执行体发送回客户。

客户/服务器结构的优点如下：

- 简化了执行体。可以在用户态服务器中构造各种各样的 API，而不会有任何冲突或重复；可以很容易地加入新的 API。
- 提高了可靠性。每个新的服务运行在内核之外，有自己的存储空间，这样可以免受其他服务的干扰，单个客户的失败不会使操作系统的其余部分崩溃。

- 为应用程序与服务间通过 RPC 调用进行通信提供了一致的方法，且没有限制其灵活性。函数桩（function stub）把消息传递进程对客户应用程序隐藏起来，函数桩是为了包装 RPC 调用的一小段代码。当通过一个 API 访问一个环境子系统或服务时，位于客户端应用程序中的函数桩把调用参数包作为一个消息发送给一个服务器子系统执行。
- 为分布式计算提供了适当的基础。典型地，分布式计算使用客户/服务器模块，通过分布的客户和服务器模块以及客户与服务器间的消息交换实现远程过程调用。对于 Windows，本地服务器可以代表本地客户应用程序给远程服务器传递一条消息，客户不需要知道请求是在本地还是在远程得到服务的。实际上，一条请求是在本地还是远程得到服务，可以基于当前负载条件和动态配置的变化而动态变化。

2.5.5　线程和 SMP

Windows 的两个重要特征是支持线程和支持对称多处理（SMP），这些在 2.4 节都曾经讲述过。[RUSS05] 列出了 Windows 支持线程和 SMP 的下列特征：

- 操作系统例程可以在任何可以得到的处理器上运行，不同的例程可以在不同的处理器上同时执行。
- Windows 支持在单个进程的执行中使用多个线程。同一个进程中的多线程可以在不同的处理器上同时执行。
- 服务器进程可以使用多线程，以处理多个用户同时发出的请求。
- Windows 提供在进程间共享数据和资源的机制以及灵活的进程间通信的能力。

2.5.6　Windows 对象

Windows 大量使用面向对象设计的概念。面向对象方法简化了进程间资源和数据的共享，便于保护资源免受未经许可的访问。Windows 使用的面向对象重要概念如下：

- **封装**：一个对象由一个或多个称做属性的数据项组成，在这些数据上可以执行一个或多个称做服务的过程。访问对象中数据的唯一方法是引用对象的一个服务，因此，对象中的数据可以很容易地保护起来，避免未经授权的使用和不正确的使用（例如试图执行不可执行的数据片）。
- **对象类和实例**：一个对象类是一个模板，它列出了对象的属性和服务，并定义了对象的某些特性。操作系统可以在需要时创建对象类的特定实例，例如，每个当前处于活动状态的进程只有一个进程对象类和一个进程对象。这种方法简化了对象的创建和管理。
- **继承**：尽管要靠手工编码实现，但执行体使用继承通过添加新的特性来扩展对象类。每个执行体类都基于一个基类，这个基类定义虚方法，以便支持创建、命名、安全保护和删除对象。调度程序对象是继承事件对象属性的执行体对象，因此，它们能使用常规的同步方法。其他特定的对象类型（如设备类），允许这些面向特定设备的类从基类中继承，增加额外的数据和方法。
- **多态性**：Windows 内部使用通用的 API 函数集操作任何类型的对象，这正是本节附录 B 中定义的多态性的一个特征。但是，由于许多 API 是特定的对象类型所特有的，因此 Windows 并不是完全多态的。

对面向对象概念不熟悉的读者可以参阅本书最后的附录 B。

Windows 中的所有实体并非都是对象。当数据对用户态的访问是开放的，或者当数据访问是共享的或受限制的时都使用对象。对象表示的实体有文件、进程、线程、信号、计时器和窗口。Windows 通过对象管理器以一致的方法创建和管理所有的对象类型，对象管理器代表应用程序负责创建和销毁对象，并负责授权访问对象的服务和数据。

执行体中的每个对象有时称为内核对象（以区分执行体并不关心的用户级对象），作为内核分配的内存块存在，并且只能被内核访问。数据结构的一些元素（例如对象名、安全参数、使用计数）对所有的对象类型都是相同的，而其余的元素是某一特定对象所特有的（例如线程对象的优先级）。因为这些对象的数据结构位于只能由内核来访问的进程地址空间中，应用程序不可能引用这些数据结构并且直接地读写。实际上，应用程序通过一组执行体支持的对象操作函数间接地操作对象。当对象创建后，请求创建的应用程序得到该对象的句柄，句柄实质上是指向被引用对象的指针。句柄可以被同一个进程中的任何线程使用，来访问可操作此对象的 Win32 函数，或者被复制到其他进程中。

对象可以有与之相关联的安全信息，以安全描述符（Security Descriptor，SD）的形式表示。安全信息可以用于限制对对象的访问，例如，一个进程可以创建一个命名信号量对象，使得只有某些用户可以打开和使用这个信号。信号对象的安全描述符可以列出那些允许（或不允许）访问信号对象的用户，以及允许访问的类型（读、写、改变等）。

在 Windows 中，对象可以是命名的，也可以是未命名的。当一个进程创建了一个未命名对象，对象管理器返回这个对象的句柄，而句柄是访问该对象的唯一途径。命名对象有一个名字，其他进程可以使用这个名字获得对象的句柄。例如，如果进程 A 希望与进程 B 同步，则它可以创建一个命名事件对象，并把事件名传递给 B，进程 B 打开并使用这个事件对象；但是，如果 A 仅仅希望使用事件同步它自己内部的两个线程，则它将创建一个未命名事件对象，因为其他进程不需要使用这个事件。

作为 Windows 管理的对象的一个例子，下面列出了微内核管理的两类对象：

- **分派器对象**：是执行体对象的子集，线程可以在该类对象上等待，用来控制基于线程的系统操作的分派和同步。这些将在第 6 章讲述。
- **控制对象**：被内核组件用来管理不受普通线程调度控制的处理器操作。表 2.5 列出了内核控制对象。

Windows 不是一个成熟的面向对象操作系统，它不是用面向对象语言实现的，完全位于执行体组件中的数据结构没有表示成对象。然而，Windows 展现了面向对象技术的能力，表明了这种技术在操作系统设计中不断增长的趋势。

表 2.5　Windows 内核控制对象

控制对象	说　明
异步过程调用	用于打断一个特定线程的执行，以一种特定的处理器模式调用过程
延迟过程调用	用于延迟中断处理，以避免延迟中断硬件处理。也可以用于实现定时器和进程间通信
中断	通过中断分派表 IDT（Interrupt Dispatch Table）中的项，把中断源连接到中断服务例程上。每个处理器有一个 IDT，用于分发该处理器中发生的中断
进程	表示虚地址空间和一组线程对象执行时所需要的控制信息。一个进程包括指向地址映射的指针，包含线程对象的一列就绪线程、属于进程的一列线程、在进程中执行的所有线程的累加时间和基本优先级
线程	表示线程对象，包括调度优先权和数量，以及该运行在哪个处理器上
分布图	用于衡量一块代码中的运行时间分布。用户代码和系统代码都可以建立分布图

2.6　传统的 UNIX 系统

2.6.1　历史

UNIX 的历史是一个经常谈论到的神话，这里就不再重复了，而是提供一个简单的概述。

UNIX 最初是在贝尔实验室开发的，1970 年在 PDP-7 上开始运行。贝尔实验室的部分人员参与了 MIT 的 MAC 项目中的分时工作。这个项目导致开发了第一个 CTSS，然后是 Multics。尽管通常说 UNIX 是 Multics 的一个缩小了的版本，但是，实际上 UNIX 的开发者声称更多地受到 CTSS 的影响［RITC78］。尽管如此，UNIX 吸收了 Multics 的许多思想。

在贝尔实验室，以及后来其他地方关于 UNIX 的工作产生了一系列的 UNIX 版本。第一个著名的里程碑是把 UNIX 系统从 PDP-7 上移植到 PDP-11 上，第一次暗示了 UNIX 将成为所有计算机上的操作系统；下一个重要的里程碑是用 C 语言重写 UNIX，这在当时是一个前所未闻的策略。通常人们认为像操作系统这样需要处理时间限制事件的复杂系统，必须完全用汇编语言编写。产生这种看法的原因如下：

- 按照今天的标准，内存（包括 RAM 和二级存储器）容量小且价格贵，因此高效使用内存很重要。这就包括了不同的内存覆盖（overlay）技术，如使用不同的代码段和数据段，以及自修改代码。
- 尽管自 20 世纪 50 年代起就开始使用编译器，计算机业一直对自动生成的代码的质量持有怀疑。在资源空间很小的情况下，时间上和空间上都高效的代码就很有必要了。
- 处理器和总线速度相对较慢，因此，节省时钟周期会使得运行时间上有很大的改进。

C 语言实现证明了对大部分而不是全部系统代码使用高级语言的优点。现在，实际上所有的 UNIX 实现都是用 C 语言编写的。

这些 UNIX 的早期版本在贝尔实验室中是非常流行的。1974 年，UNIX 系统第一次出现在一本技术期刊中［RITC74］，这大大地激发了人们对该系统的兴趣，UNIX 的许可证提供给了商业机构和大学。第一个在贝尔实验室外可以使用的版本是 1976 年的第 6 版，接着 1978 年发行的第 7 版是大多数现代 UNIX 系统的先驱。最重要的非 AT&T 系统是加利福尼亚大学伯克利分校开发的 UNIX BSD，最初在 PDP 上运行，后来在 VAX 机上运行。AT&T 继续开发改进系统，1982 年，贝尔实验室将 UNIX 的多个 AT&T 变种合并成一个系统，即商业销售的 UNIX System III。后来在操作系统中又增加了很多功能组件，产生了 UNIX Sytem V。

2.6.2　描述

图 2.14 给出了对 UNIX 结构的概述。底层硬件被操作系统软件包围，操作系统通常称做系统内核，或简单地称做内核，以强调它与用户和应用程序的隔离。本书中举的 UNIX 的例子，主要关注的是 UNIX 内核。但是，UNIX 拥有许多用户服务和接口，它们也被看做是系统的一部分，可以分为命令解释器、其他接口软件和 C 编译器部分（编译器、汇编器和加载器），它们的外层由用户应用程序和到 C 编译器的用户接口组成。

图 2.15 提供了对内核的更深入描述。用户程序可以直接调用操作系统服务，也可以通过库程序调用。系统调用接口是内核和用户的边界，它允许高层软件使用特定的内核函数。另一方面，操作系统包含直接与硬件交互的原子例程（primitive routine）。在这两个接口之间，系统被划分成两个主要部分，一个关心进程控制，另一个关心文件管理和 I/O。进程控制子系统负责内存管理、进程的调度和分发、进程的同步以及进程间的通信。文件系统按字符流或块的形式在内存和外部设备间交换数据，为实现这一点，需要用到各种设备驱动程序。对面向块的传送，使用磁盘高速缓存方法：内存中的一个系统缓冲区介于用户地址空间和外部设备之间。

本节描述的是传统的 UNIX 系统，［VAHA96］使用这个术语表示 System V 版本 3（简称 SVR3）、4.3BSD 以及更早的版本。下面是关于传统 UNIX 系统的综述：它被设计成在单一处理器上运行，缺乏保护其数据结构避免被多个处理器同时访问的能力；它的内核不是通用的，只支

持一种文件系统、进程调度策略和可执行文件格式。传统 UNIX 的内核没有设计成可扩展的，几乎没有代码重用的设施。其结果是，当往不同的 UNIX 版本中增加新功能时，必须增加很多新的代码，因而产生了一个膨胀的、非模块化的内核。

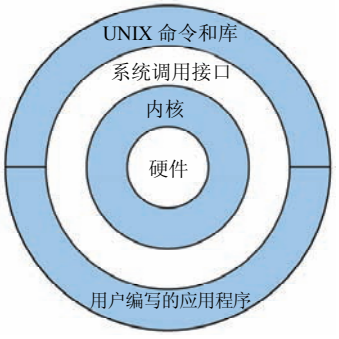


图 2.14　UNIX 的一般体系结构

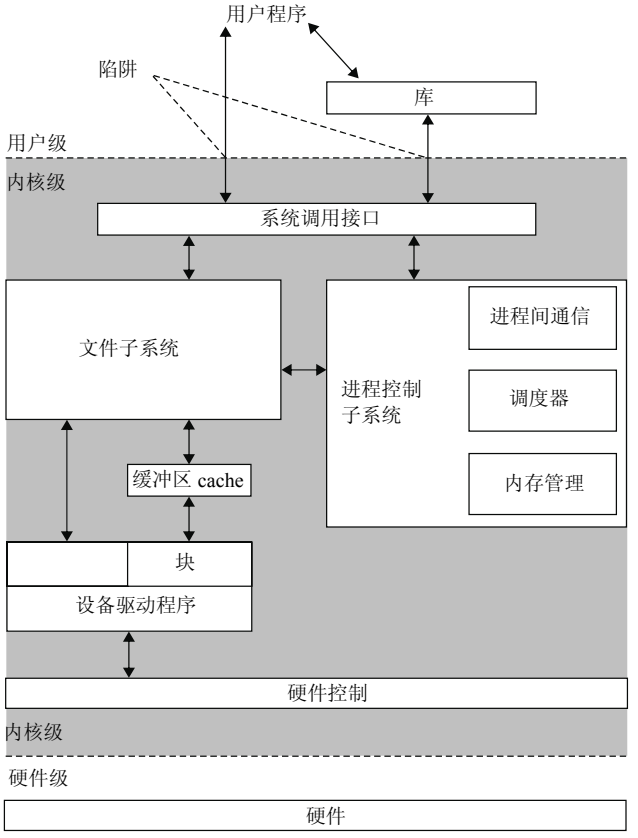


图 2.15　传统 UNIX 内核

2.7　现代 UNIX 系统

随着 UNIX 的发展，出现了很多不同的实现版本，每种实现版本都提供了一些有用的功能。这就需要产生一种新的实现版本，以合并许多重要的技术革新，增加其他现代操作系统设计特征，创造出一种模块化更好的结构。典型的现代 UNIX 内核具有如图 2.16 所示的结构。有一个小的核心软件，它以模块化的风格编写，提供许多操作系统进程所需要的功能和服务；每个外部圆圈表示相应的功能和以多种方式实现的接口。

下面给出现代 UNIX 系统的一些例子。

2.7.1　系统 V 版本 4（SVR4）

由 AT&T 和 Sun Microsystem 联合开发的 SVR4 结合了 SVR3、4.3BSD、Microsoft Xenix System V 和 SunOS 的特点。它几乎完全重写了系统 V 的内核，产生了一个简洁的、有些复杂的实现版本。这个版本中的新特点包括对实时处理的支持、进程调度类、动态分配数据结构、虚拟内存管理、虚拟文件系统和可以剥夺的内核。

SVR4 同时汲取了商业设计者和学院设计者的成果，为商业 UNIX 的部署提供统一的平台。它已经实现了这个目标，是现有的最重要的 UNIX 变种。它合并了在任何 UNIX 系统中曾经开发

过的大多数重要特征，并以一种完整的、有商业生存力的方式实现这些特征。SVR4 可以在从 32 位微处理器到超级计算机的很广范围内的处理器上运行。

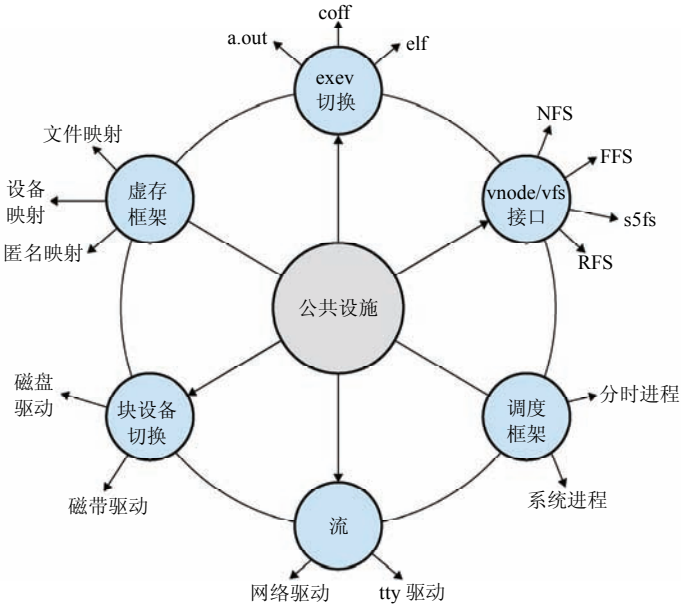


图 2.16　现代 UNIX 内核

2.7.2　BSD

UNIX 版本的 BSD（Berkeley Software Distribution）系列在操作系统设计原理的发展中扮演着重要的角色。4.xBSD 广泛用于学院版的 UNIX 系统，并且成为许多商业 UNIX 产品的基础。可以肯定地说，BSD 对 UNIX 的普及起着主要作用，大多数 UNIX 的增强功能首先出现在 BSD 版本中。

4.4BSD 是 Berkeley 最后发布的 BSD 版本，随后其设计和实现组织就解散了。它是 4.3BSD 的一个重要升级，包含新的虚拟内存系统、对内核结构所做的改变以及对一系列其他特征的增强。

应用最为广泛的且文档最好的一个 BSD 版本是 FreeBSD。FreeBSD 在基于因特网的服务器和防火墙中最常用到，还应用在许多嵌入式系统中。

最新版本的 Macintosh 操作系统 Mac OS X 是基于 FreeBSD 5.0 和 Mach 3.0 微内核的。

2.7.3　Solaris 10

Solaris 是 Sun 基于 SVR4 的 UNIX 版本，最新版本是 10。Solaris 的实现提供了 SVR4 的所有特征以及许多更高级的特征，如完全可抢占、支持多线程的内核、完全支持 SMP 以及文件系统的面向对象接口。Solaris 是使用最为广泛、最成功的商业 UNIX 实现版本。

2.8　Linux 操作系统

Windows/Linux 比较	
Windows Vista	Linux
概　　述	
商业操作系统，受到 VAX/VMS 的巨大影响，要求与多个操作系统兼容，比如 DOS/Windows、POSIX 以及最初的 OS/2	一种 UNIX 的开源实现，注重简单和效率，可运行在多种处理器架构上

（续）

影响基础设计决策的环境	
32 位程序地址空间	16 位程序地址空间
MB 级物理内存	KB 级物理内存
虚拟内存	基于内存映射的交换系统
多处理器（4 路）	单处理器
基于 I/O 设备的微控制器	基于 I/O 设备的状态机
客户/服务器分布式计算	独立交互系统
大量、多样的用户群	少量友好的用户

与当今环境比较：

64 位地址

GB 级物理内存

虚拟内存、虚拟处理器

多处理器（64 ~ 128）

高速互联网/内部网、Web 服务

单用户，易受世界各地的黑客攻击

尽管 Windows 和 Linux 都在不断适应环境并做出改变，但是初始设计环境（也就是在 1989 年和 1973 年）还是严重影响了其设计选择：

并发单元：线程与进程　[地址空间，单处理器]

进程创建：CreateProcess()与 fork()　[地址空间，交换操作]

I/O：　异步与同步　[交换操作，I/O 设备]

安全性：　自由访问与 uid/gid　[用户群]

系统结构	
模块化内核，通过组件显式发布数据结构和接口 三层： • 硬件抽象层管理处理器、中断和 DMA、BIOS 详细信息 • 内核层管理 CPU 调度、中断和同步 • 执行体层在完全线程化、通常是抢占式的环境中实现操作系统的主要功能	单体内核
动态数据结构和内核地址空间组织；初始化代码在启动后被抛弃。大部分内核代码和数据是可分页的。不可分页的内核代码和数据采用大页面，以提高 TLB 效率。	内核代码和数据被静态分配给不可分页内存
文件系统、网络、设备是可加载/不可加载的驱动程序（动态链接库），使用可扩展的 I/O 系统接口	对加载/卸载内核模块的广泛支持，比如设备驱动程序和文件系统
动态加载的驱动程序可以提供可分页和不可分页的区域	模块不能被分页，但是可以卸载
名字空间的根目录是虚拟的，文件系统挂载在下面；可以很容易地扩展系统对象类型，影响统一命名、引用、生命周期管理、安全性和基于句柄的同步	
OS 个性化作为用户态子系统实现。原生 NT API 基于通用内核句柄/对象架构并允许跨进程操作虚拟内存、线程和其他内核对象	名字空间根目录位于文件系统中；添加新命名的系统对象需要文件系统更改或映射到设备模型上实现了与 POSIX 兼容的类似 UNIX 的接口；内核 API 比 Windows 的 API 简单得多；能够理解各种类型的可执行文件
自由访问控制、分散的特权、审计	用户/组 ID；类似 NT 特权的权能与进程关联

2.8.1　历史

Linux 开始是用于 IBM PC（Intel 80386）结构的一个 UNIX 变种，最初的版本是由芬兰一名

计算机科学专业的学生 Linus Torvalds 写的。1991 年 Torvalds 在 Internet 上公布了最早的 Linux 版本，从那以后，很多人通过在 Internet 上的合作，为 Linux 的发展做出了贡献，所有这些都在 Torvalds 的控制下。由于 Linux 是自由的，并且可以得到源代码，因而它成为其他诸如 Sun 公司和 IBM 公司提供的 UNIX 工作站的较早的替代产品。当今，Linux 是具有全面功能的 UNIX 系统，可以在所有这些平台甚至更多平台上运行，包括 Intel Pentium 和 Itanium、Motorola/IBM PowerPC。

Linux 成功的关键在于它是由自由软件基金会（Free Software Foundation，FSF）赞助的自由软件包。FSF 的目标是稳定的、与平台无关的软件，它必须是自由的、高质量的、为用户团体所接受的。FSF 的 GNU 项目[⊖] 为软件开发者提供了工具，而 GNU Public License（GPL）是 FSF 批准标志。Torvalds 在开发内核的过程中使用了 GNU 工具，后来他在 GPL 之下发布了这个内核。这样，我们今天所见到的 Linux 发行版本是 FSF 的 GNU 项目、Torvald 的个人努力以及遍布世界的很多合作者们共同的产品。

除了由很多个程序员使用以外，Linux 已经明显地渗透到了业界，这并不是因为自由软件的缘故，而是因为 Linux 内核的质量。很多天才的程序员对当前版本都有贡献，产生了这一在技术上给人留下深刻印象的产品；而且，Linux 是高度模块化和易于配置的，这使得它很容易在各种不同的硬件平台上显示出最佳的性能；另外，由于可以获得源代码，销售商可以调整应用程序和使用方法以满足特定的要求。本书将提供基于最新的 Linux 2.6 版本的内核的细节。

2.8.2　模块结构

大多数 UNIX 内核是单体的。前面已经讲过，单体内核是指在一大块代码中实际上包含了所有操作系统功能，并作为一个单一进程运行，具有唯一地址空间。内核中的所有功能部件可以访问所有的内部数据结构和例程。如果对典型的单体式操作系统的任何部分进行了改变，在变化生效前，所有的模块和例程都必须重新链接、重新安装，系统必须重新启动。其结果是，任何修改（如增加一个新的设备驱动程序或文件系统函数）都是很困难的。这个问题在 Linux 中尤其尖锐，Linux 的开发是全球性的，是由独立的程序员组成的联系松散的组织完成的。

尽管 Linux 没有采用微内核的方法，但是由于它特殊的模块结构，也具有很多微内核方法的优点。Linux 的结构是一个模块的集合，这些模块可以根据需要自动地加载和卸载。这些相对独立的块称做**可加载模块**（loadable module）[GOYE99]。实质上，一个模块就是内核在运行时可以链接或断开链接的一个对象文件。典型地，一个模块实现一些特定的功能，例如文件系统、设备驱动或是内核上层的一些特征。尽管模块可以因为各种目的而创建内核线程，但是它不作为自身的进程或线程执行。当然，模块会代表当前进程在内核态下执行。

因此，虽然 Linux 被认为是单体内核，但是它的模块结构克服了在开发和发展内核过程中所遇到的困难。

Linux 可加载模块有两个重要特征：

- **动态链接**：当内核已经在内存中并正在运行时，内核模块可以被加载和链接到内核。模块也可以在任何时刻被断开链接，从内存中移出。
- **可堆栈模块**：模块按层次排列，当被高层的客户模块访问时，它们作为库；当被低层模块访问时，它们作为客户。

动态链接 [FRAN97] 简化了配置任务，节省了内核所占的内存空间。在 Linux 中，用户程序或用户可以使用 insmod 和 rmmod 命令显式地加载和卸载内核模块，内核自身监视对于特定函数的需求，并可以根据需求加载和卸载模块。通过可堆栈模块可以定义模块间的依赖关系，这有

⊖ GNU 是 GNU's Not UNIX 的首字母简写。GNU 项目是一系列免费的软件，包括为开发类 UNIX 操作系统的软件包和工具，它常常使用 Linux 内核。

两个好处：

1）对一组相似的模块的相同的代码（例如相似硬件的驱动程序）可以移入一个模块，以减少重复。

2）内核可以确保所需要的模块都存在，避免卸载其他正在运行的模块仍然依赖着的模块，并且当加载一个新模块时，加载任何所需要的附加模块。

图 2.17 举例说明了 Linux 管理模块的结构，该图显示了当只有两个模块 FAT 和 VFAT 被加载后内核模块的列表。每个模块由两个表定义，即模块表和符号表。模块表包括以下元素：

- ***next**：指向后面的模块。所有模块被组织到一个链表中，链表以一个伪模块开始（图 2.17 中没有显示）。
- ***name**：指向模块名的指针。
- **size**：模块大小，以内存页计。
- **usecount**：模块引用计数器。当操作系统引用的模块函数开始时计数器增加，终止时减少。
- **flags**：模块标志。
- **nsyms**：输出的符号数。
- **ndeps**：引用的模块数。
- ***syms**：指向这个模块符号表的指针。
- ***deps**：指向被这个模块引用的模块列表的指针。
- ***refs**：指向使用这个模块的模块列表的指针。

符号表定义了该模块控制的符号，它们将在别的地方使用到。

图 2.17 显示了 VFAT 模块在 FAT 模块后被加载，并且它依赖于 FAT 模块。

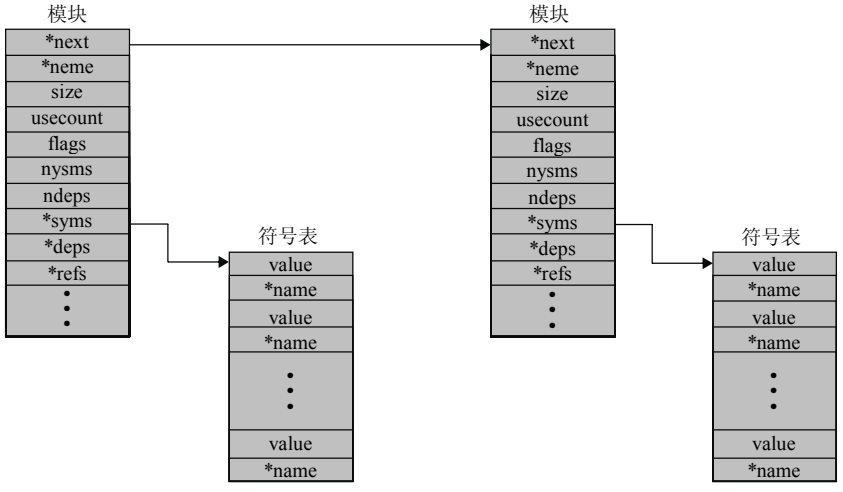


图 2.17　Linux 内核模块列表示例

2.8.3　内核组件

图 2.18 摘自[MOSB02]显示了基于 IA-64 体系结构（例如 Intel Itanium）的 Linux 内核的主要组件。图中显示了运行在内核之上的一些进程，每个方框表示一个进程，每条带箭头的曲线表示一个正在执行的线程⊖。内核本身包括一组相互关联的组件，箭头表示主要的关联。底层的硬

⊖ 在 Linux 中，进程与线程的概念相同。但是，Linux 中的多线程可以按这样一种方法来有效地组合在一起，即单个进程由多个线程组成。这些内容将在第 4 章中详细探讨。

件也是一个组件集，箭头表示硬件组件被哪一个内核组件使用或控制。当然所有的内核组件都在 CPU 上执行，但是为了简洁，没有显示它们的关系。

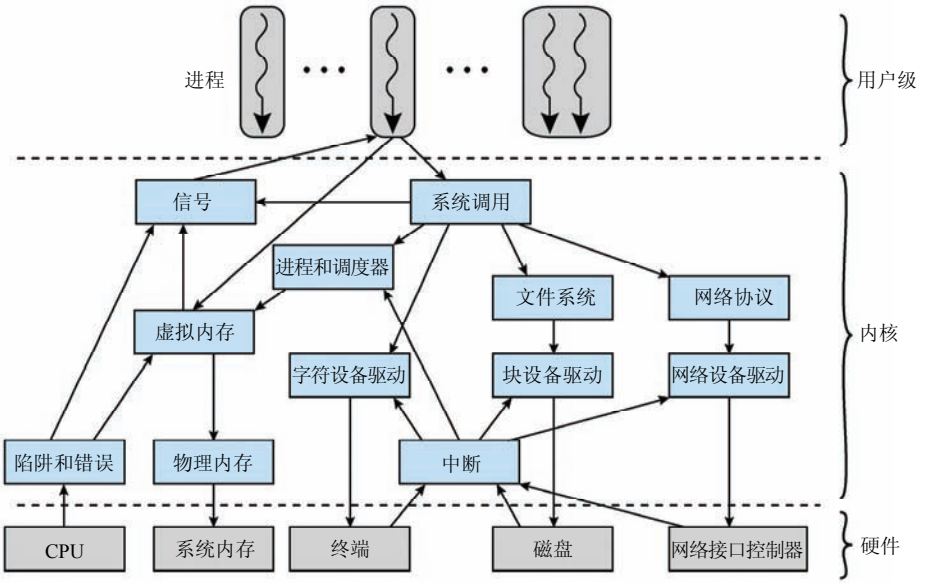


图 2.18　Linux 内核组件

主要的内核组件简要介绍如下：

- **信号**：内核通过信号通知进程。例如，信号用来通知进程某些错误，比如被 0 除错误。表 2.6 给出了一些信号的例子。
- **系统调用**：进程是通过系统调用来请求系统服务的。一共有几百个系统调用，可以粗略地分为 6 类：文件系统、进程、调度、进程间通信，套接字（网络）和其他。表 2.7 分别给出了每个类的一些例子。
- **进程和调度器**：创建、管理和调度进程。
- **虚拟内存**：为进程分配和管理虚拟内存。
- **文件系统**：为文件、目录和其他文件相关的对象提供一个全局的、分层次的命名空间，还提供文件系统函数。
- **网络协议**：为用户的 TCP/IP 协议套件提供套接字接口。
- **字符设备驱动**：管理向内核一次发送或接收一个字节数据的设备，比如终端、调制解调器和打印机。
- **块设备驱动**：管理以块为单位向内核发送和接收数据的设备，比如各种形式的外存（磁盘、CD-ROM 等）。
- **网络设备驱动**：对网络接口卡和通信端口提供管理，它们负责连接到网桥或路由之类的网络设备。
- **陷阱和错误**：处理 CPU 产生的陷阱和错误，例如内存错误。
- **物理内存**：管理实际内存中的内存页池和为虚拟内存分配内存页。
- **中断**：处理来自外设的中断。

表 2.6　一些 Linux 信号

信　号	说　明	信　号	说　明
SIGHUP	终端挂起	SIGCONT	继续

（续）

信　号	说　明	信　号	说　明
SIGQUIT	键盘退出	SIGTSTP	键盘停止
SIGTRAP	跟踪陷阱	SIGTTOU	终端写
SIGBUS	总线错误	SIGXCPU	超出 CPU 限制
SIGKILL	Kill 信号	SIGVTALRM	虚拟告警器时钟
SIGSEGV	段错误	SIGWINCH	窗口大小没有改变
SIGPIPT	坏的管道	SIGPWR	电源错误
SIGTERM	终止	SIGRTMIN	第一个实时信号
SIGCHLD	子进程状态未改变	SIGRTMAX	最后一个实时信号

表 2.7　一些 Linux 系统调用

文件系统相关	
close	关闭文件描述符
link	为文件起新名
open	打开或者创建一个文件或设备
read	从文件描述符中读
write	往文件描述符中写
进程相关	
execve	执行程序
exit	终止调用的进程
getpid	获得进程标志
setuid	设置当前进程的用户标志
prtrace	为父进程提供一种方法，使之可以监视和控制另一个进程的执行，并且检查和修改它的核心映像和寄存器
调度相关	
Sched_getparam	根据进程的标志 pid 设置与调度策略相关的调度参数
Sched_get_priority_max	返回最大的优先级值，这个值可能被用于由 policy 确定的调度算法
Sched_setscheduler	根据进程 pid 设置调度策略（如 FIFO）和相关参数
Sched_rr_get_interval	根据进程 pid 把轮转时间量写入到 timespec 结构中，这个结构由参数 tp 表示
Sched_yield	通过这个系统调用，一个进程可以自动地释放处理器。这个进程将会因为静态优先级被移动到队列尾部，同时一个新的进程开始运行
进程间通信（IPC）相关	
msgrcv	为接收消息分配的消息缓冲结构。系统调用根据 msqid 把一个消息从消息队列读取到新创建的消息缓冲区中
semctl	根据 cmd 对信号量集 semid 执行控制操作
semop	对信号量集 semid 选定的成员执行操作
shmat	把由 shmid 标识的共享内存段附加到调用进程的数据段
shmctl	允许用户用一个共享的内存段接收信息，设置共享内存段的所有者、组和权限，或者销毁一个段
套接字（网络）相关	
bind	为套接字分配一个本地的 IP 地址和端口，成功返回 0，失败返回−1
connect	在给定的套接字和远程套接字间建立连接，远程套接字需要与套接字地址相关联
gethostname	返回本地主机名称
send	把*msg 指向的缓冲区中数据按字节发送给定的套接字
setsockopt	设置套接字属性

（续）

其　他	
create_module	试图创建一个可加载的模块入口，为加载这个模块预定所需的内核内存空间
Fsync	把文件中的所有核心部分复制到硬盘中，并且等待设备报告所有的部分都被写入到存储器中
query_module	查询内核中可加载模块相关的信息
time	返回从 1970 年 1 月 1 日起的时间，以秒为单位
vhangup	在当前终端模拟挂起操作。这个调用在其他用户登录时提供一个“干净”的 tty

2.9　推荐读物和网站

[BRIN01]收集了近年来关于操作系统主要进展的优秀论文。[SWAI07]是一篇有趣的关于未来操作系统的短文。

［VAHA96］是讲述 UNIX 内部结构的一本优秀书籍，它提供了很多 UNIX 变种间的比较。［GOOD94］提供了关于 UNIX SVR4 的丰富的技术细节。对于流行的开源 FreeBSD，[MCKU05]值得强烈推荐。[MCDO07]较好地讲述了 Solaris 内部结构。[BOVE06]和[LOVE05]是讲述 Linux 内部结构的两本好书。

尽管有很多关于 Windows 各种版本的书，但是于内部结构相关的内容却非常少。要推荐的书是[RUSS05]，它的内容只涵盖了 Windows Server 2003 的内容，但大部分内容对 Vista 也是正确的。

BOVE06 Bovet,D.,and Cesati,M.*Understanding the Linux Kernel*.Sebastopol, CA : O'Reilly,2006.

BRIN01 Brinch Hansen,P.*Classic Operating Systems:From Batch Processing to Distributed Systems*. New York:Springer-Verlag,2001.

GOOD94 Goodheart,B.,and Cox,J.*The Magic Garden Explained:The Internals of UNIX System V Release 4*.Englewood Cliffs,NJ:Prentice Hall,1994.

LOVE05 Love,R.*Linux Kernel Development*.Waltham,MA:Novell Press,2005.

MCDO07 McDougall,R.,and Mauro,J.*Solaris Internals:Solaris 10 and OpenSolaris Kernel Architecture*. Palo Alto,CA:Sun Microsystems Press,2007.

MCKU05 McKusick,M.,and Neville-Neil,J.*The Design and Implementation of the FreeBSD Operating System*.Reading,MA:Addison-Wesley,2005.

RUSS05 Russinovich,M.,and Solomon,D.*Microsoft Windows Internals:Microsoft Windows Server（TM）2003,Windows XP,and Windows 2000*.Redmond,WA:Microsoft Press,2005.

SWAI07 Swaine,M.“Wither Operating Systems?”*Dr.Dobb's Journal*,March 2007.

VAHA96 Vahalia,U.*UNIX Internals:The New Frontiers*.Upper Saddle River,NJ:Prentice Hall,1996.

推荐网站

- The Operating System Resource Center：收集了许多关于操作系统的有用的文档和论文。
- Review of Operating Systems：关于商业的、免费的、研究的和业余爱好的各种操作系统的全面的综述。
- Operating System Technical Comparison：包括各种操作系统的大量翔实的信息。
- ACM Special Interest Group on Operating Systems：关于 SIGOPS 出版物和会议的信息。
- IEEE Technical Committee on Operating Systems and Application Enviroments：包括在线的新闻和其他网站的链接。
- The comp.os.research FAQ：很多关于操作系统设计的有价值的 FAQ。
- UNIX Guru Universe：关于 UNIX 源码的非常好的信息。
- Linux Documentation Project：名字描述了这个站点的内容。
- IBM's Linux Web site：提供了广泛的 Linux 技术和用户信息。其上许多内容是针对 IBM 产品的，但是有很多有价值的通用的技术信息。
- Windows Development：Windows 内部结构方面的很好的信息源。

2.10　关键术语、复习题和习题

关键术语

批处理	管程	物理地址	串行处理
批处理系统	单体内核	特权指令	对称多处理
执行上下文	多道批处理系统	进程	任务
中断	多道程序设计	进程状态	线程
作业	多任务	实地址	分时
作业控制语言	多线程	常驻监控程序	分时系统
内核	内核	时间片轮转	单道程序设计
内存管理	操作系统（OS）	调度	虚地址
微内核			

复习题

2.1　操作系统设计的三个目标是什么？
2.2　什么是操作系统的内核？
2.3　什么是多道程序设计？
2.4　什么是进程？
2.5　操作系统是怎么使用进程上下文的？
2.6　列出并简要介绍五种典型的操作系统的存储管理职责。
2.7　解释实地址和虚地址的区别。
2.8　描述时间片轮转调度技术。
2.9　解释单体内核和微内核的区别。
2.10　什么是多线程？

习题

2.1　假设我们有一台多道程序的计算机，每个作业有相同的特征。在一个计算周期 T 中，一个作业有一半时间花费在 I/O 上，另一半用于处理器的活动。每个作业一共运行 N 个周期。假设使用简单的时间片轮转调度，并且 I/O 操作可以与处理器操作重叠。定义以下量：

- 时间周期＝完成任务的实际时间
- 吞吐量＝每个时间周期 T 内平均完成的作业数目
- 处理器利用率＝处理器活跃（不是处于等待）的时间的百分比

当周期 T 分别按下列方式分布时，对 1 个、2 个和 4 个同时发生的作业，请计算这些量：

a）前一半用于 I/O，后一半用于处理器

b）前四分之一和后四分之一用于 I/O，中间部分用于处理器

2.2　I/O 密集型的程序是指如果单独运行，则花费在等待 I/O 上的时间比使用处理器的时间要多的程序。处理器密集型的程序则相反。假设短期调度算法偏爱那些在近期使用处理器时间较少的程序，请解释为什么这个算法偏爱 I/O 密集型的程序，但是并不是永远不受理处理器密集型程序所需的处理器时间？

2.3　a）解释操作系统从简单批处理系统发展为多道批处理系统的原因。

b）解释操作系统从多道批处理系统发展为分时系统的原因。

2.4　为什么用户态和内核态的设计被认为是好的操作系统设计？举例说明一个进程从用户态切换到内核态，然后又返回到用户态的过程。

2.5　在 IBM 的主机操作系统 OS/390 中，内核中的一个重要模块是系统资源管理程序（System Resource Manager，SRM），它负责地址空间（进程）之间的资源分配。SRM 使得 OS/390 在操作系统中具有特殊性，没有任何其他的主机操作系统，当然也没有任何其他类型的操作系统可以比得上 SRM 所实现

的功能。资源的概念包括处理器、实存和 I/O 通道，SRM 累加器、I/O 通道和各种重要数据结构的利用率，它的目标是基于性能监视和分析提供最优的性能，其安装设置了以后的各种性能目标作为 SRM 的指南，这会基于系统的利用率动态地修改安装和作业性能特点。SRM 依次提供报告，允许受过训练的操作员改进配置和参数设置，以改善用户服务。

现在关注 SRM 活动的一个实例。实存被划分为成千上万个大小相等的块，称做帧。每个帧可以保留一块称做页的虚拟内存。SRM 每秒大约接收 20 次控制，并在互相之间以及每个页面之间进行检查。如果页没有被引用或被改变，计数器增 1。一段时间后，SRM 求这些数的平均值，以确定系统中一个页面未曾被触及的平均秒数。这样做的目的是什么？SRM 将采取什么动作？

第二部分　进　　程

现代操作系统最基础的任务就是进程管理。操作系统必须为进程分配资源，使进程间可以交换信息，保护各个进程的资源不被其他进程占用，并且使进程可以同步。为了达到这些要求，操作系统必须为每一个进程维护一个数据结构，这个数据结构描述进程的状态和资源所有权，这样才能使操作系统进行进程控制。

在单处理器多道程序系统中，多个进程的执行可以在同一时间交叉进行。在多处理器系统中，不仅多个进程可以交叉执行，而且可以同步执行。交叉执行和同步执行都属于并发执行，这将给应用程序员和操作系统带来一些难题。

线程概念的引入，也给许多当代的操作系统中的进程管理带来困难。在一个多线程系统中，进程保留着资源所有权的属性，而多个并发执行流是执行在进程中运行的线程。

第二部分导读

第 3 章　进程描述和控制

传统的操作系统的主要任务是进程管理。每一个进程在任何时间内都处于一组执行状态中的一种情况：就绪态、运行态和阻塞态。操作系统跟踪这些执行状态，并且管理进程在这些状态间的转换过程。为了达到这个目的，操作系统通过相当精细的数据结构来表述每个进程。操作系统必须实现调度功能，并且为进程间的共享和同步提供便利。第 3 章讲述了典型操作系统中进程管理所使用到的数据结构和技术。

第 4 章：线程、对称多处理（SMP）和微内核

第 4 章涵盖了三个领域，这三个领域是许多现代操作系统的主要特征，并且是比传统操作系统更先进的标志。在许多操作系统中，传统的进程概念被分为两部分：一部分负责管理资源所有权（进程）；另一个部分负责指令流的执行（线程）。一个单独的进程可能包含多个线程。使用多线程的组织方法对程序的结构化和性能方面都有很大帮助。第 4 章还讲述了对称多处理机（SMP），SMP 是一个拥有多处理器的计算机系统，其中的每一个处理器都可以执行所有程序和系统代码。SMP 的组织方法增强了系统的性能和可靠性。SMP 通常和多线程机制一起使用，即使没有多线程也能大幅提高系统性能。最后，第 4 章将讲述微内核，微内核是操作系统为了减少运行在内核态的代码量的一种设计方式，并且分析了这种方法的优点。

第 5 章：并发：互斥和同步

当今操作系统的两个中心主题是多道程序和分布式处理，并发是这两个主题的基础，同时也是操作系统设计技术的基础。第 5 章讲述了并发控制的两个方面：互斥和同步。互斥是多进程（或

多线程）共享代码、资源或数据并使得在一个时间内只允许一个进程访问共享对象的一种能力。与互斥相关联的是同步：多进程根据信息的交换协调它们活动的能力。第 5 章以一个关于并发设计的讨论开始，提出了关于并发的一些设计问题。这一章还对并发的硬件支持进行了讨论，还有支持并发最重要的机制：信号量、管程和消息传递。

第 6 章：并发：死锁和饥饿

第 6 章讲述了并发控制的另外两个方面。死锁是这样一种情况：一组进程中的两个或多个进程要等待该组中的其他成员完成一个操作后才能继续运行，但是没有成员可以继续。死锁是一个很难预测的现象，并且没有比较容易的通用解决方法。第 6 章将提出处理死锁问题的三个主要手段：预防、避免和检测。饥饿是一个准备运行的进程由于其他进程的运行而一直不能访问处理器的情况。从大的方面说，饥饿是被当成调度问题来处理的，第四部分将会讲述。尽管第 6 章的中心是死锁，但是也在解决死锁的内容中提到了饥饿，因为解决死锁问题要避免带来饥饿问题。

第 3 章　进程描述和控制

操作系统的设计必须反映某些一般性的要求。所有多道程序操作系统，从诸如 Windows 98 的单用户系统到诸如 IBM z/OS 的可支持成千上万个用户的主机系统，它们的创建都围绕着进程的概念。因此，操作系统必须满足的大多数需求表示都涉及进程：

- 操作系统必须交替执行多个进程，在合理的响应时间范围内使处理器的利用率最大。
- 操作系统必须按照特定的策略（例如某些函数或应用程序具有较高的优先级）给进程分配资源，同时避免死锁[⊖]。
- 操作系统可以支持进程间的通信和用户创建进程，它们对构造应用程序很有帮助。

现在开始从分析操作系统表示和控制进程的方式来深入地学习操作系统。首先介绍进程的概念，讨论进程状态，进程状态描述了进程的行为特征；接着着眼于操作系统表示每个进程的状态所需要的数据结构，和操作系统为实现其目标所需要的进程的其他特征；然后看操作系统是如何使用这些数据结构控制进程的执行的；最后，讨论了 UNIX SVR4 中的进程管理。第 4 章提供了更多的现在操作系统（如 Solaris、Windows 和 Linux）的进程管理的例子。

注意，在本章中，偶尔会引用到虚拟内存。大多数时候，在处理进程时可以忽略这个概念，但某些地方需要考虑虚拟内存的概念。虚拟内存在第 8 章才会详细讲述，但在第 2 章中曾有过简单的概述。

3.1　什么是进程

3.1.1　背景

在给进程下定义之前，首先总结一下第 1 章和第 2 章介绍的一些概念：

1）一个计算机平台包括一组硬件资源，比如处理器、内存、I/O 模块、定时器和磁盘驱动器等。

2）计算机程序是为执行某些任务而开发的。在典型的情况下，它们接受外来的输入，做一些处理之后，输出结果。

3）直接根据给定的硬件平台写应用程序效率是低下的，主要原因如下：

a）针对相同的平台可以开发出很多应用程序，所以开发出这些应用程序访问计算机资源的通用例程是很有意义的。

b）处理器本身只能对多道程序设计提供有限的支持，需要用软件去管理处理器和其他资源同时被多个程序共享。

c）如果多个程序在同一时间都是活跃的，那么需要保护每个程序的数据、I/O 使用和其他资源不被其他程序占用。

4）开发操作系统是为了给应用程序提供一个方便、安全和一致的接口。操作系统是计算机硬件和应用程序之间的一层软件（如图 2.1 所示），对应用程序和工具提供了支持。

⊖ 有关死锁的内容将在第 6 章讲述。从本质上看，如果两个进程为了继续进行而需要相同的两个资源，而它们每人都拥有其中的一个资源，这时就会发生死锁。每个进程都将无限地等待自己没有的那个资源。

5）可以把操作系统想象为资源的统一抽象表示，可以被应用程序请求和访问。资源包括内存、网络接口和文件系统等。一旦操作系统为应用程序创建了这些资源的抽象表示，就必须管理它们的使用，例如一个操作系统可以允许资源共享和资源保护。

有了应用程序、系统软件和资源的概念，就可以讨论操作系统怎样以一个有序的方式管理应用程序的执行，以达到以下目的：

- 资源对多个应用程序是可用的。
- 物理处理器在多个应用程序间切换以保证所有程序都在执行中。
- 处理器和 I/O 设备能得到充分的利用。

所有现代操作系统采用的方法都是依据对应于一个或多个进程存在的应用程序执行的一种模型。

3.1.2　进程和进程控制块

第 2 章给进程下了以下几个定义：

- 正在执行的程序。
- 正在计算机上执行的程序实例。
- 能分配给处理器并由处理器执行的实体。
- 具有以下特征的活动单元：一组指令序列的执行、一个当前状态和相关的系统资源集。

也可以把进程当成由一组元素组成的实体，进程的两个基本的元素是**程序代码**（可能被执行相同程序的其他进程共享）和代码相关联的**数据集**。假设处理器开始执行这个程序代码，且我们把这个执行实体叫做进程。在进程执行时，任意给定一个时间，进程都可以唯一地被表征为以下元素：

- **标识符**：跟这个进程相关的唯一标识符，用来区别其他进程。
- **状态**：如果进程正在执行，那么进程处于运行态。
- **优先级**：相对于其他进程的优先级。
- **程序计数器**：程序中即将被执行的下一条指令的地址。
- **内存指针**：包括程序代码和进程相关数据的指针，还有和其他进程共享内存块的指针。
- **上下文数据**：进程执行时处理器的寄存器中的数据。
- **I/O 状态信息**：包括显式的 I/O 请求、分配给进程的 I/O 设备（例如磁带驱动器）和被进程使用的文件列表等。
- **记账信息**：可能包括处理器时间总和、使用的时钟数总和、时间限制、记账号等。

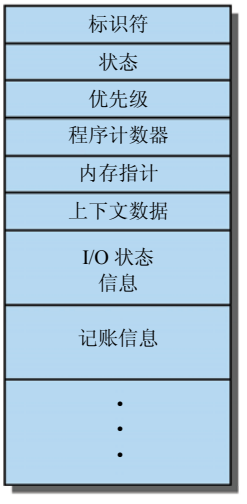


图 3.1　简化的进程控制块

前述的列表信息存放在一个叫做**进程控制块**（如图 3.1 所示）的数据结构中，该控制块由操作系统创建和管理。比较有意义的一点是，进程控制块包含了充分的信息，这样就可以中断一个进程的执行，并且在后来恢复执行进程时就好像进程未被中断过。进程控制块是操作系统能够支持多进程和提供多处理的关键工具。当进程被中断时，操作系统会把程序计数器和处理器寄存器（上下文数据）保存到进程控制块中的相应位置，进程状态也被改变为其他的值，例如阻塞态或就绪态（后面将讲述）。现在操作系统可以自由地把其他进程设置为运行态，把其他进程的程序计数器和进程上下文数据加载到处理器寄存器中，这样其他进程就可以开始执行了。

因此，可以说进程是由程序代码和相关数据还有进程控制块组成。对于一个单处理器计算机，

在任何时间都最多只有一个进程在执行，正在运行的这个进程的状态为*运行态*。

3.2　进程状态

正如前面所提到的，对一个被执行的程序，操作系统会为该程序创建一个进程或任务。从处理器的角度看，它在指令序列中按某种顺序执行指令，这个顺序根据程序计数器寄存器中不断变化的值来指示，程序计数器可能指向不同进程中不同部分的程序代码；从程序自身的角度看，它的执行涉及程序中的一系列指令。

可以通过列出为该进程执行的指令序列来描述单个进程的行为，这样的序列称做进程的**轨迹**。可以通过给出各个进程的轨迹是如何被交替的来描述处理器的行为。

考虑一个非常简单的例子，图 3.2 给出了三个进程在内存中的布局，为简化讨论，假设没有使用**虚拟内存**，因此所有三个进程都由完全载入内存中的程序表示，此外，有一个小的**分派器**[⊖]使处理器从一个进程切换到另一个进程。图 3.3 给出了这三个进程在执行过程早期的轨迹，给出了进程 A 和 C 中最初执行的 12 条指令，进程 B 执行 4 条指令，假设第 4 条指令调用了进程必须等待的 I/O 操作。

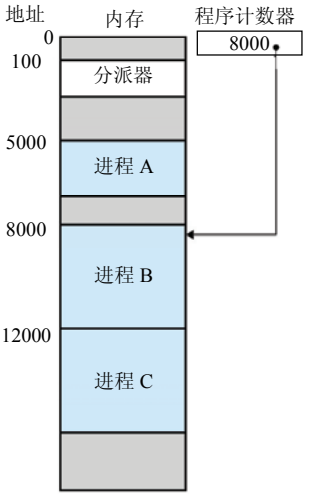


图 3.2　在指令周期 13 时的执行快照（如图 3.4 所示）

现在从处理器的角度看这些轨迹。图 3.4 给出了最初的 52 个指令周期中交替的轨迹（为方便起见，指令周期都给出了编号）。在图中，阴影部分代表由分配器执行的代码。在每个实例中由分派器执行的指令顺序是相同的，因为是分派器的同一个功能在执行。假设操作系统仅允许一个进程最多连续执行 6 个指令周期，在此之后将被中断，这避免了任何一个进程独占处理器时间。如图 3.4 所示，进程 A 最初的 6 条指令被执行，接下来是一个超时并执行分派器的某些代码，在控制转移给进程 B 之前分派器执行了 6 条指令[⊜]。在进程 B 的 4 条指令被执行后，进程 B 请求一个它必须等待的 I/O 动作，因此，处理器停止执行进程 B，并通过分派器转移到进程 C。在超时后，处理器返回进程 A，当这次处理超时时，进程 B 仍然等待那个 I/O 操作的完成，因此分派器再次转移到进程 C。

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

a）进程 A 的轨迹　　b）进程 B 的轨迹　　c）进程 C 的轨迹

5000 表示进程 A 的程序起始地址
8000 表示进程 B 的程序起始地址
12 000 表示进程 C 的程序起始地址

图 3.3　图 3.2 中进程的轨迹

⊖ 分派器即为调度器。——译者注

⊜ 进程只执行了很少的几条指令，并且分派器也是超常的低速，这样的设想完全是为了简化讨论。

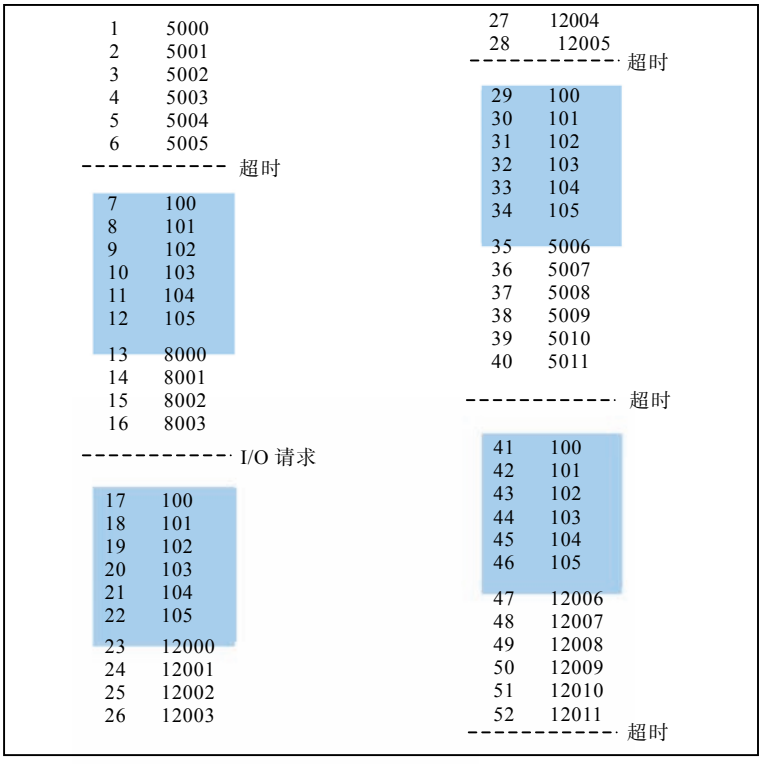


图 3.4　图 3.2 中进程的组合轨迹

3.2.1　两状态进程模型

操作系统的基本职责是控制进程的执行，这包括确定交替执行的方式和给进程分配资源。在设计控制进程的程序时，第一步就是描述进程所表现出的行为。

通过观察可知，在任何时刻，一个进程要么正在执行，要么没有执行，因而可以构造最简单的模型。一个进程可以处于以下两种状态之一：运行态或未运行态，如图 3.5a 所示。当操作系统创建一个新进程时，它将该进程以未运行态加入到系统中，操作系统知道这个进程是存在的，并正在等待执行机会。当前正在运行的进程不时地被中断，操作系统中的分派器部分将选择一个新进程运行。前一个进程从运行态转换到未运行态，另外一个进程转换到运行态。

从这个简单的模型可以意识到操作系统的一些设计元素。必须用某种方式来表示每个进程，使得操作系统能够跟踪它，也就是说，必须有一些与进程相关的信息，包括进程在内存中的当前状态和位置，即进程控制块。未运行的进程必须保持在某种类型的队列中，并等待它们的执行时机。图 3.5b 给出了一个结构，结构中有一个队列，队列中的每一项都指向某个特定进程的指针，或队列可以由数据块构成的链表组成，每个数据块表示一个进程。

可以用该排队图描述分派器的行为。被中断的进程转移到等待进程队列中，或者，如果进程已经结束或取消，则被销毁（离开系统）。在任何一种情况下，分派器均从队列中选择一个进程来执行。

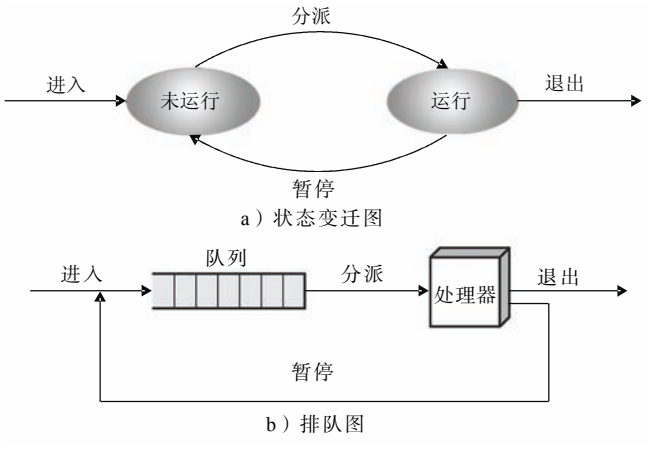


图 3.5　两状态进程模型

3.2.2　进程的创建和终止

在对简单的两状态模型进行改进之前，有必要讨论一下进程的创建和终止。无论使用哪种进程行为模型，进程的生存期都围绕着进程的创建和终止。

进程的创建

当一个新进程添加到那些正在被管理的进程集合中去时，操作系统需要建立用于管理该进程的数据结构（见 3.3 节），并在内存中给它分配地址空间。我们将在 3.3 节中讲述这些数据结构，这些行为构成了一个新进程的创建过程。

通常有 4 个事件会导致创建一个进程，如表 3.1 所示。在批处理环境中，响应作业提交时会创建进程；在交互环境中，当一个新用户试图登录时会创建进程。不论在哪种情况下，操作系统都负责新进程的创建，操作系统也可能会代表应用程序创建进程。例如，如果用户请求打印一个文件，则操作系统可以创建一个管理打印的进程，进而使请求进程可以继续执行，与完成打印任务的时间无关。

表 3.1　导致进程创建的原因

事　　件	说　　明
新的批处理作业	通常位于磁带或磁盘中的批处理作业控制流被提供给操作系统。当操作系统准备接纳新工作时，它将读取下一个作业控制命令
交互登录	终端用户登录到系统
操作系统因为提供一项服务而创建	操作系统可以创建一个进程，代表用户程序执行一个功能，使用户无需等待（如控制打印的进程）
由现有的进程派生	基于模块化的考虑，或者为了开发并行性，用户程序可以指示创建多个进程

传统地，操作系统创建进程的方式对用户和应用程序都是透明的，这在当代操作系统中也很普遍。但是，允许一个进程引发另一个进程的创建将是很有用的。例如，一个应用程序进程可以产生另一个进程，以接收应用程序产生的数据，并将数据组织成适合以后分析的格式。新进程与应用程序并行地运行，并当得到新的数据时被激活。这个方案对构造应用程序是非常有用的，例如，服务器进程（如打印服务器、文件服务器）可以为它处理的每个请求产生一个新进程。当操作系统为另一个进程的显式请求创建一个进程时，这个动作称为**进程派生**。

当一个进程派生另一个进程时，前一个称做**父进程**，被派生的进程称做**子进程**。在典型的情况下，相关进程需要相互之间的通信和合作。对程序员来说，合作是一个非常困难的任务，相关主题将在第 5 章讲述。

进程终止

表 3.2 概括了进程终止的典型原因。任何一个计算机系统都必须为进程提供表示其完成的方法，批处理作业中应该包含一个 Halt 指令或用于终止的操作系统显式服务调用来终止。在前一种情况下，Halt 指令将产生一个中断，警告操作系统一个进程已经完成。对交互式应用程序，用户的行为将指出何时进程完成，例如，在分时系统中，当用户退出系统或关闭自己的终端时，该用户的进程将被终止。在个人计算机或工作站中，用户可以结束一个应用程序（如字处理或电子表格）。所有这些行为最终导致发送给操作系统的一个服务请求，以终止发出请求的进程。

此外，很多错误和故障条件会导致进程终止。表 3.2 列出了一些最常见的识别条件[⊖]。

最后，在有些操作系统中，进程可以被创建它的进程终止，或当父进程终止时而终止。

表 3.2　导致进程终止的原因

事　件	说　明
正常完成	进程自行执行一个操作系统服务调用，表示它已经结束运行
超过时限	进程运行时间超过规定的时限。可以测量很多种类型的时间，包括总的运行时间（“挂钟时间”）、花费在执行上的时间以及对于交互进程从上一次用户输入到当前时刻的时间总量
无可用内存	系统无法满足进程需要的内存空间
越界	进程试图访问不允许访问的内存单元
保护错误	进程试图使用不允许使用的资源或文件，或者试图以一种不正确的方式使用，如往只读文件中写
算术错误	进程试图进行被禁止的计算，如除以零或者存储大于硬件可以接纳的数字
时间超出	进程等待某一事件发生的时间超过了规定的最大值
I/O 失败	在输入或输出期间发生错误，如找不到文件、在超过规定的最多努力次数后仍然读/写失败（例如当遇到了磁带上的一个坏区时）或者无效操作（如从行式打印机中读）
无效指令	进程试图执行一个不存在的指令（通常是由于转移到了数据区并企图执行数据）
特权指令	进程试图使用为操作系统保留的指令
数据误用	错误类型或未初始化的一块数据
操作员或操作系统干涉	由于某些原因，操作员或操作系统终止进程（例如，如果存在死锁）
父进程终止	当一个父进程终止时，操作系统可能会自动终止该进程的所有后代进程
父进程请求	父进程通常具有终止其任何后代进程的权力

3.2.3　五状态模型

如果所有进程都做好了执行准备，则图 3.5b 所给出的排队原则是有效的。队列是“先进先出”（first-in-first-out）的表，对于可运行的进程处理器以一种**轮转**（round-robin）方式操作（依次给队列中的每个进程一定的执行时间，然后进程返回队列，阻塞情况除外）。但是，即使对前面描述的简单例子，这个实现都是不合适的：存在着一些处于非运行状态但已经就绪等待执行的进程，而同时存在另外的一些处于阻塞状态等待 I/O 操作结束的进程。因此，如果使用单个队列，

⊖ 在某些情况下，一个宽松的操作系统可能会允许用户从错误中恢复而不结束进程。例如，如果用户请求访问文件失败，操作系统可能仅仅告知访问被拒绝并且允许进程继续运行。

分派器不能只考虑选择队列中最老的进程，相反，它应该扫描这个列表，查找那些未被阻塞且在队列中时间最长的进程。

解决这种情况的一种比较自然的方法是将非运行状态分成两个状态：就绪（ready）和阻塞（blocked），如图 3.6 所示。此外还应该另外增加两个已经证明很有用的状态。新图中的 5 个状态如下：

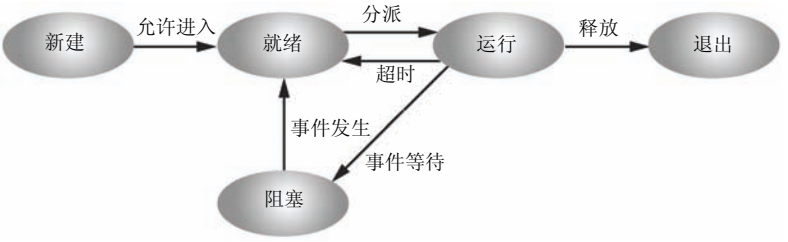


图 3.6　五状态进程模型

- **运行态**：该进程正在执行。在本章中，假设计算机只有一个处理器，因此一次最多只有一个进程处于这个状态。
- **就绪态**：进程做好了准备，只要有机会就开始执行。
- **阻塞/等待态**[⊖]：进程在某些事件发生前不能执行，如 I/O 操作完成。
- **新建态**：刚刚创建的进程，操作系统还没有把它加入到可执行进程组中。通常是进程控制块已经创建但还没有加载到内存中的新进程。
- **退出态**：操作系统从可执行进程组中释放出的进程，或者是因为它自身停止了，或者是因为某种原因被取消。

新建态和退出态对进程管理是非常有用的。新建状态对应于刚刚定义的进程。例如，如果一位新用户试图登录到分时系统中，或者一个新的批作业被提交执行，那么操作系统可以分两步定义新进程。首先，操作系统执行一些必需的辅助工作，将标识符关联到进程，分配和创建管理进程所需要的所有表。此时，进程处于新建状态，这意味着操作系统已经执行了创建进程的必需动作，但还没有执行进程。例如，操作系统可能基于性能或内存局限性的原因，限制系统中的进程数量。当进程处于新建态时，操作系统所需要的关于该进程的信息保存在内存中的进程表中，但进程自身还未进入内存，就是即将执行的程序代码不在内存中，也没有为与这个程序相关的数据分配空间。当进程处于新建态时，程序保留在外存中，通常是磁盘中[⊜]。

类似地，进程退出系统也分为两步。首先，当进程到达一个自然结束点时，由于出现不可恢复的错误而取消时，或当具有相应权限的另一个进程取消该进程时，进程被终止；终止使进程转换到退出态，此时，进程不再被执行了，与作业相关的表和其他信息临时被操作系统保留起来，这给辅助程序或支持程序提供了提取所需信息的时间。一个实用程序为了分析性能和利用率，可能需要提取进程的历史信息，一旦这些程序都提取了所需要的信息，操作系统就不再需要保留任何与该进程相关的数据，该进程将从系统中删除。

图 3.6 显示了导致进程状态转换的事件类型。可能的转换如下：

- **空→新建**：创建执行一个程序的新进程。这个事件在表 3.1 中所列出的原因下都会发生。
- **新建→就绪**：操作系统准备好再接纳一个进程时，把一个进程从新建态转换到就绪态。

⊖ “等待态”作为一个进程状态，经常用于替换术语“阻塞态”。一般情况下，我们常用“阻塞态”，但是这两个术语是可以互换的。

⊜ 在这一段的讨论中，忽略了虚拟内存的概念。在支持虚拟内存的系统中，当进程从新建态转换到就绪态时，它的程序代码和数据被加载到虚拟内存中。虚拟内存的简单介绍见第 2 章，详细内容请参阅第 8 章。

大多数系统基于现有的进程数或分配给现有进程的虚拟内存数量设置一些限制，以确保不会因为活跃进程的数量过多而导致系统的性能下降。

- **就绪→运行**：需要选择一个新进程运行时，操作系统选择一个处于就绪态的进程，这是调度器或分派器的工作。进程的选择问题将在第四部分探讨。
- **运行→退出**：如果当前正在运行的进程表示自己已经完成或取消，则它将被操作系统终止，见表 3.2。
- **运行→就绪**：这类转换最常见的原因是，正在运行的进程到达了“允许不中断执行”的最大时间段；实际上所有多道程序操作系统都实行了这类时间限定。这类转换还有很多其他原因，例如操作系统给不同的进程分配不同的优先级，但这不是在所有的操作系统中都实现了的。假设，进程 A 在一个给定的优先级运行，且具有更高优先级的进程 B 正处于阻塞态。如果操作系统知道进程 B 等待的事件已经发生了，则将 B 转换到就绪态，然后因为优先级的原因中断进程 A 的执行，将处理器分派给进程 B，我们说操作系统**抢占**了进程 A[⊖]。最后一种情况是，进程自愿释放对处理器的控制，例如一个周期性地进行记账和维护的后台进程。
- **运行→阻塞**：如果进程请求它必须等待的某些事件，则进入阻塞态。对操作系统的请求通常以系统服务调用的形式发出，也就是说，正在运行的程序请求调用操作系统中一部分代码所发生的过程。例如，进程可能请求操作系统的一个服务，但操作系统无法立即予以服务，它也可能请求了一个无法立即得到的资源，如文件或虚拟内存中的共享区域；或者也可能需要进行某种初始化的工作，如 I/O 操作所遇到的情况，并且只有在该初始化动作完成后才能继续执行。当进程互相通信，一个进程等待另一个进程提供输入时，或者等待来自另一个进程的信息时，都可能被阻塞。
- **阻塞→就绪**：当所等待的事件发生时，处于阻塞态的进程转换到就绪态。
- **就绪→退出**：为了清楚起见，状态图中没有表示这种转换。在某些系统中，父进程可以在任何时刻终止一个子进程。如果一个父进程终止，与该父进程相关的所有子进程都将被终止。
- **阻塞→退出**：前面一项提供了注释。

再回到前面的简单例子，图 3.7 显示了每个进程在状态间的转换，图 3.8a 给出了可能实现的排队规则，有两个队列：就绪队列和阻塞队列。进入系统的每个进程被放置在就绪队列中，当操作系统选择另一个进程运行时，将从就绪队列中选择。对于没有优先级的方案，这可以是一个简单的先进先出队列。当一个正在运行的进程被移出处理器时，它根据情况或者被终止，或者被放置在就绪或阻塞队列中。最后，当一个事件发生时，所有位于阻塞队列中等待这个事件的进程都被转换到就绪队列中。

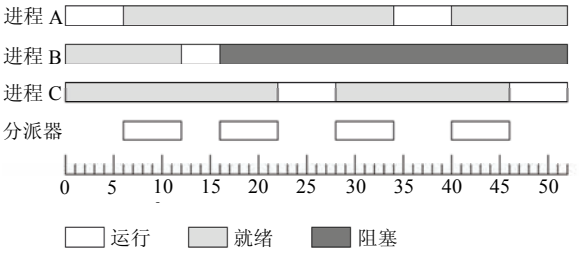
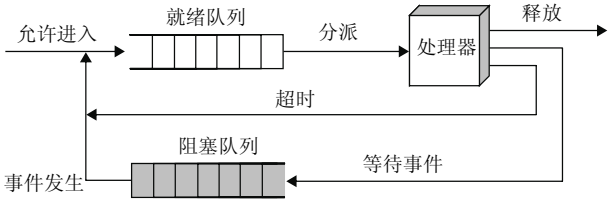


图 3.7　图 3.4 中的进程状态

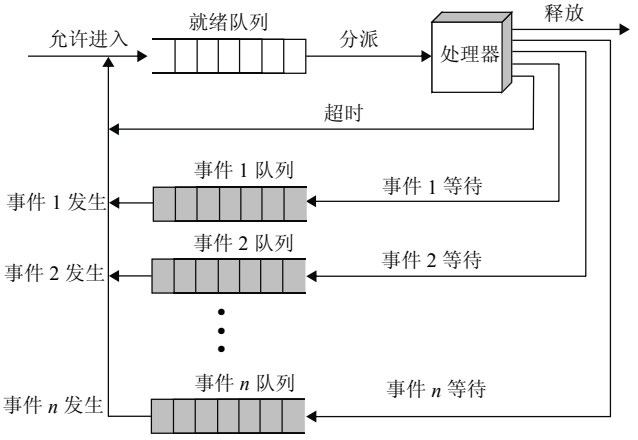
后一种方案意味着当一个事件发生时，操作系统必须扫描整个阻塞队列，搜索那些等待该事件的进程。在大型操作系统中，队列中可能有几百甚至几千个进程，因此，拥有多个队列将会很有效，一个事件可以对应一个队列。那么，

⊖ 一般来说，抢占这个术语被定义为收回一个进程正在使用的资源。在这种情况下，资源就是处理器本身。进程正在执行并且可以继续执行，但是由于其他进程需要执行而被抢占。

当事件发生时，相应队列中的所有进程都转换到就绪态（见图 3.8b）。



a）单一的阻塞队列



b）多条阻塞队列

图 3.8　图 3.6 的排队模型

最后还有一种改进是，如果按照优先级方案分派进程，维护多个就绪队列（每个优先级一个队列）将会带来很多的便利。操作系统可以很容易地确定哪个就绪进程具有最高的优先级且等待时间最长。

3.2.4　被挂起的进程

交换的需要

前面描述的三个基本状态（就绪态、运行态和阻塞态）提供了一种为进程行为建立模型的系统方法，并指导操作系统的实现。许多实际的操作系统都是按照这样的三种状态进行具体构造的。

但是，可以证明往模型中增加其他状态也是合理的。为了说明加入新状态的好处，考虑一个没有使用虚拟内存的系统，每个被执行的进程必须完全载入内存，因此，图 3.8b 中，所有队列中的所有进程必须驻留在内存中。

所有这些设计机制的原因都是由于 I/O 活动比计算速度慢很多，因此在单道程序系统中的处理器在大多数时候是空闲的。但是图 3.8b 的方案并没有完全解决这个问题。在这种情况下，内存保存有多个进程，当一个进程正在等待时，处理器可以转移到另一个进程，但是处理器比 I/O 要快得多，以至于内存中所有的进程都在等待 I/O 的情况很常见。因此，即使是多道程序设计，大多数时候处理器仍然可能处于空闲状态。

一种解决方法是内存可以被扩充以适应更多的进程，但是这种方法有两个缺陷。首先是内存的价格问题，当内存大小增加到兆位及千兆位时，价格也会随之增加；再者，程序对内存空间需求的增长速度比内存价格下降的速度快。因此，更大的内存往往导致更大的进程，而不是更多的进程。

另一种解决方案是交换，包括把内存中某个进程的一部分或全部移到磁盘中。当内存中没有

处于就绪状态的进程时，操作系统就把被阻塞的进程换出到磁盘中的“挂起队列”（suspend queue），这是暂时保存从内存中被“驱逐”出的进程队列，或者说是被挂起的进程队列。操作系统在此之后取出挂起队列中的另一个进程，或者接受一个新进程的请求，将其纳入内存运行。

“交换”（swapping）是一个 I/O 操作，因而也可能使问题更加恶化。但是由于磁盘 I/O 一般是系统中最快的 I/O（相对于磁带或打印机 I/O），所以交换通常会提高性能。

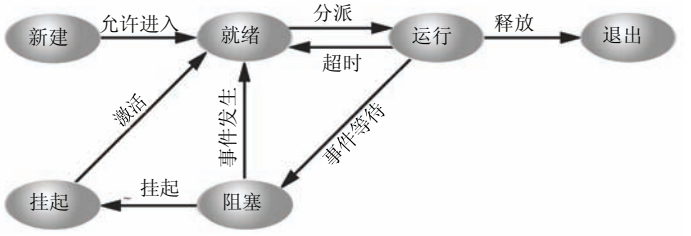
为使用前面描述的交换，在我们的进程行为模型（见图 3.9a）中必须增加另一个状态：挂起态。当内存中的所有进程都处于阻塞态时，操作系统可以把其中的一个进程置于挂起态，并将它转移到磁盘，内存中释放的空间可被调入的另一个进程使用。

当操作系统已经执行了一个换出操作，它可以有两种将一个进程取到内存中的选择：可以接纳一个新近创建的进程，或调入一个以前被挂起的进程。显然，通常比较倾向于调入一个以前被挂起的进程，给它提供服务，而不是增加系统中的负载总数。

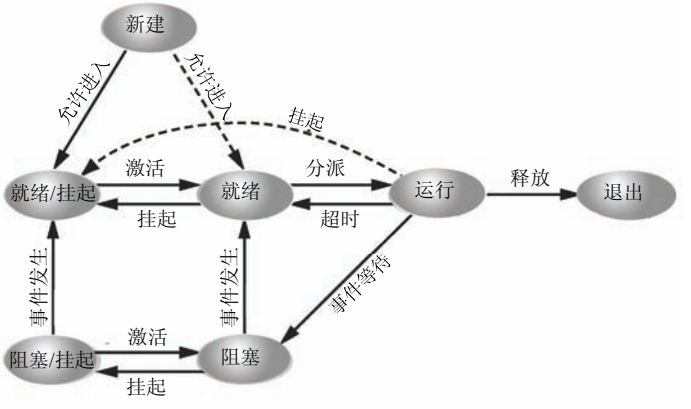
但是，这个推理也带来了一个难题，所有已经挂起的进程在挂起时都处于阻塞态。显然，这时把被阻塞的进程取回内存没有任何意义，因为它仍然没有准备好执行。但是，考虑到挂起状态中的每个进程最初是阻塞在一个特定的事件上，当这个事件发生时，进程就不再阻塞，可以继续执行。

因此，我们需要重新考虑设计方式。这里有两个独立的概念：进程是否在等待一个事件（阻塞与否）以及进程是否已经被换出内存（挂起与否）。为适应这种 2×2 的组合，需要 4 个状态：

- **就绪态**：进程在内存中并可以执行。
- **阻塞态**：进程在内存中并等待一个事件。
- **阻塞/挂起态**：进程在外存中并等待一个事件。
- **就绪/挂起态**：进程在外存中，但是只要被载入内存就可以执行。



a）有一个挂起态



b）有两个挂起态

图 3.9　有挂起态的进程状态转换图

在查看包含两个新挂起状态的状态转换图之前，必须提到另一点。到现在为止的论述都假设没有使用虚拟内存，进程或者都在内存中，或者都在内存之外。在虚拟内存方案中，可能会执行到只有一部分内容在内存中的进程，如果访问的进程地址不在内存中，则进程的相应部分可以被调入内存。虚拟内存的使用看上去会消除显式交换的需要，这是因为通过处理器中的存储管理硬件，任何期望的进程中的任何期望的地址都可以移入或移出内存。但是，正如在第 8 章中将会看到的，如果有足够多的活动进程，并且所有进程都有一部分在内存中，则有可能导致虚拟内存系统崩溃。因此，即使在虚拟存储系统中，操作系统也需要不时地根据执行情况显式地、完全地换出进程。

现在来看图 3.9b 中我们已开发的状态转换模型(图中的虚线表示可能但并不是必需的转换)。比较重要的新的转换如下：

- **阻塞→阻塞/挂起**：如果没有就绪进程，则至少一个阻塞进程被换出，为另一个没有阻塞的进程让出空间。如果操作系统确定当前正在运行的进程，或就绪进程为了维护基本的性能要求而需要更多的内存空间，那么，即使有可用的就绪态进程也可能出现这种转换。
- **阻塞/挂起→就绪/挂起**：如果等待的事件发生了，则处于阻塞/挂起状态的进程可以转换到就绪/挂起状态。注意，这要求操作系统必须能够得到挂起进程的状态信息。
- **就绪/挂起→就绪**：如果内存中没有就绪态进程，操作系统需要调入一个进程继续执行。此外，当处于就绪/挂起态的进程比处于就绪态的任何进程的优先级都要高时，也可以进行这种转换。这种情况的产生是由于操作系统设计者规定调入高优先级的进程比减少交换量更重要。
- **就绪→就绪/挂起**：通常，操作系统更倾向于挂起阻塞态进程而不是就绪态进程，因为就绪态进程可以立即执行，而阻塞态进程占用了内存空间但不能执行。但如果释放内存以得到足够空间的唯一方法是挂起一个就绪态进程，那么这种转换也是必需的。并且，如果操作系统确信高优先级的阻塞态进程很快将会就绪，那么它可能选择挂起一个低优先级的就绪态进程，而不是一个高优先级的阻塞态进程。

还需要考虑的几种其他转换有：

- **新建→就绪/挂起以及新建→就绪**：当创建一个新进程时，该进程或者加入到就绪队列，或者加入到就绪/挂起队列中。不论哪种情况，操作系统都必须建立一些表以管理进程，并为进程分配地址空间。操作系统可能更倾向于在初期执行这些辅助工作，这使得它可以维护大量的未阻塞的进程。通过这个策略，内存中经常会没有足够的空间分配给新进程，因此使用了（新建→就绪/挂起）转换。另一方面，我们可以证明创建进程的适时（just-in-time）原理，即尽可能推迟创建进程以减少操作系统的开销，并在系统被阻塞态进程阻塞时允许操作系统执行进程创建任务。
- **阻塞/挂起→阻塞**：这种转换在设计中比较少见，如果一个进程没有准备好执行，并且不在内存中，调入它又有什么意义？但是考虑到下面的情况：一个进程终止，释放了一些内存空间，阻塞/挂起队列中有一个进程比就绪/挂起队列中的任何进程的优先级都要高，并且操作系统有理由相信阻塞进程的事件很快就会发生，这时，把阻塞进程而不是就绪进程调入内存是合理的。
- **运行→就绪/挂起**：通常当分配给一个运行进程的时间期满时，它将转换到就绪态。但是，如果由于位于阻塞/挂起队列的具有较高优先级的进程变得不再被阻塞，操作系统抢占这个进程，也可以直接把这个运行进程转换到就绪/挂起队形中，并释放一些内存空间。
- **各种状态→退出**：在典型情况下，一个进程在运行时终止，或者是因为它已经完成，或者是因为出现了一些错误条件。但是，在某些操作系统中，一个进程可以被创建它的进

程终止，或当父进程终止时终止。如果允许这样，则进程在任何状态时都可以转换到退出态。

挂起的其他用途

到目前为止，挂起进程的概念与不在内存中的进程概念是等价的。一个不在内存中的进程，不论它是否在等待一个事件，都不能立即执行。

我们可以总结一下挂起进程的概念。首先，按照以下特点定义挂起进程：

1）进程不能立即执行。

2）进程可能是或不是正在等待一个事件。如果是，阻塞条件不依赖于挂起条件，阻塞事件的发生不会使进程立即被执行。

3）为阻止进程执行，可以通过代理把这个进程置于挂起状态，代理可以是进程自己，也可以是父进程或操作系统。

4）除非代理显式地命令系统进行状态转换，否则进程无法从这个状态中转移。

表 3.3 列出了进程的一些挂起原因。已经讨论过的一个原因是提供更多的内存空间，这样可以调入一个就绪/挂起态进程，或者增加分配给其他就绪态进程的内存。操作系统因为其他动机而挂起一个进程，例如，记账或跟踪进程可能用于监视系统的活动，可以使用进程记录各种资源（处理器、内存、通道）的使用情况以及系统中用户进程的进行速度。在操作员控制下的操作系统可以不时地打开或关闭这个进程。如果操作系统发现或怀疑有问题，它可以挂起进程。死锁就是一个例子，将在第 6 章讲述。另一个例子是，如果在进程测试时检测到通信线路中的问题，操作员让操作系统挂起使用该线路的进程。

另外一些原因关系到交互用户的行为。例如，如果用户怀疑程序中有缺陷，他（她）可以挂起执行程序并进行调试，检查并修改程序或数据，然后恢复执行；或者可能有一个收集记录或记账的后台程序，用户可能希望能够打开或关闭这个程序。

表 3.3　导致进程挂起的原因

事　件	说　明
交换	操作系统需要释放足够的内存空间，以调入并执行处于就绪状态的进程
其他 OS 原因	操作系统可能挂起后台进程或工具程序进程，或者被怀疑导致问题的进程
交互式用户请求	用户可能希望挂起一个程序的执行，目的是为了调试或者与一个资源的使用进行连接
定时	一个进程可能会周期性地执行（例如记账或系统监视进程），而且可能在等待下一个时间间隔时被挂起
父进程请求	父进程可能会希望挂起后代进程的执行，以检查或修改挂起的进程，或者协调不同后代进程之间的行为

时机的选择也会导致一个交换决策。例如，如果一个进程周期性地被激活，但大多数时间是空闲的，则在它在两次使用之间应该被换出。监视使用情况或用户活动的程序就是一个例子。

最后，父进程可能会希望挂起一个后代进程。例如，进程 A 可以生成进程 B，以执行文件读操作；随后，进程 B 在读文件的过程中遇到错误，并报告给进程 A；进程 A 挂起进程 B，调查错误的原因。

在所有这些情况中，挂起进程的活动都是由最初请求挂起的代理请求的。

3.3　进程描述

操作系统控制计算机系统内部的事件，它为处理器执行进程而进行调度和分派，给进程分配

资源，并响应用户程序的基本服务请求。因此，我们可以把操作系统看做是管理系统资源的实体。

这个概念如图 3.10 所示。在多道程序设计环境中，在虚拟内存中有许多已经创建了的进程（P_1，…，P_n），每个进程在执行期间，需要访问某些系统资源，包括处理器、I/O 设备和内存。在图中，进程 P_1 正在运行，该进程至少有一部分在内存中，并且还控制着两个 I/O 设备；进程 P_2 也在内存中，但由于正在等待分配给 P_1 的 I/O 设备而被阻塞；进程 P_n 已经被换出，因此是挂起的。

以后几章中将探讨操作系统代表进程管理这些资源的细节。这里关心的是一些最基本的问题：操作系统为了控制进程和管理资源需要哪些信息？

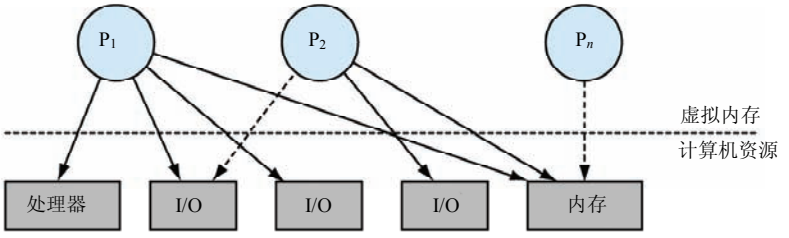


图 3.10　进程和资源（某一时刻的资源分配）

3.3.1　操作系统的控制结构

操作系统为了管理进程和资源，必须掌握关于每个进程和资源当前状态的信息。普遍使用的方法是：操作系统构造并维护它所管理的每个实体的信息表。图 3.11 给出了这种方法的一般概念，操作系统维护着 4 种不同类型的表：内存、I/O、文件和进程。尽管不同的操作系统中的实现细节不同，但基本上所有操作系统维护的信息都可以分为这 4 类。

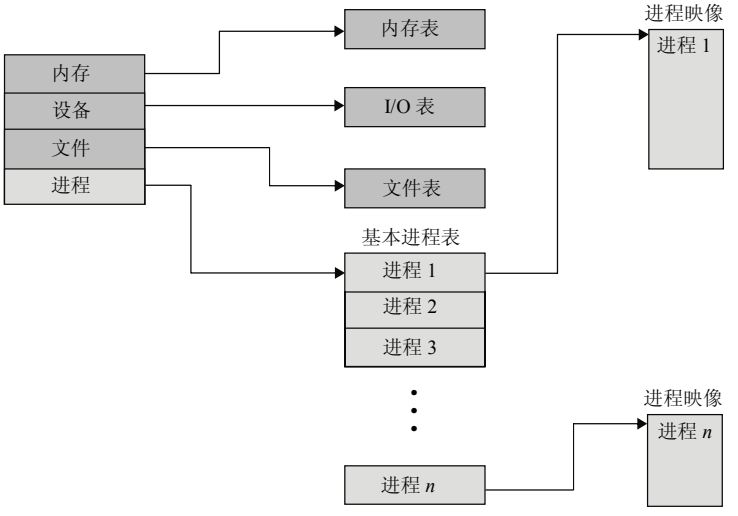


图 3.11　操作系统控制表的通用结构

内存表用于跟踪内（实）存和外存（虚拟内存）。内存的某些部分为操作系统而保留，剩余部分是进程可以使用的，保存在外存中的进程使用某种类型的虚拟内存或简单的交换机制。内存表必须包括以下信息：

- 分配给进程的内存。
- 分配给进程的外存。

- 内存块或虚拟内存块的任何保护属性，如哪些进程可以访问某些共享内存区域。
- 管理虚拟内存所需要的任何信息。

第三部分将详细讲述用于内存管理的信息结构。

操作系统使用 **I/O 表**管理计算机系统中的 I/O 设备和通道。在任何给定的时刻，一个 I/O 设备或者是可用的，或者已分配给某个特定的进程，如果正在进行 I/O 操作，则操作系统需要知道 I/O 操作的状态和作为 I/O 传送的源和目标的内存单元。在第 11 章将详细讲述 I/O 管理。

操作系统还维护着**文件表**，这些表提供关于文件是否存在、文件在外存中的位置、当前状态和其他属性的信息。大部分信息（不是全部信息）可能由文件管理系统维护和使用。在这种情况下，操作系统只有一点或者没有关于文件的信息；在其他操作系统中，很多文件管理的细节由操作系统自己管理。这方面的内容将在第 12 章讲述。

最后，操作系统为了管理进程必须维护**进程表**，本节的剩余部分将着重讲述所需的进程表。在此之前需要先明确两点：首先，尽管图 3.11 给出了 4 种不同的表，但是这些表必须以某种方式链接起来或交叉引用。内存、I/O 和文件是代表进程而被管理的，因此进程表中必须有对这些资源的直接或间接引用。文件表中的文件可以通过 I/O 设备访问，有时它们也位于内存中或虚拟内存中。这些表自身必须可以被操作系统访问到，因此它们受制于内存管理。

其次，操作系统最初如何知道创建表？显然，操作系统必须有基本环境的一些信息，如有多少内存空间、I/O 设备是什么以及它们的标识符是什么等。这是一个配置问题，也就是说，当操作系统初始化后，它必须可以使用定义基本环境的某些配置数据，这些数据必须在操作系统之外，通过人的帮助或一些自动配置软件而产生。

3.3.2　进程控制结构

操作系统在管理和控制进程时，首先必须知道进程的位置，再者，它必须知道在管理时所必需的进程属性（如进程 ID、进程状态）。

进程位置

在处理进程定位问题和进程属性问题之前，首先需要解决一个更基本的问题：进程的物理表示是什么？进程最少必须包括一个或一组被执行的程序，与这些程序相关联的是局部变量、全局变量和任何已定义常量的数据单元。因此，一个进程至少包括足够的内存空间，以保存该进程的程序和数据；此外，程序的执行通常涉及用于跟踪过程调用和过程间参数传递的栈（见附录 1B）。最后，与每个进程相关联的还有操作系统用于控制进程的许多属性，通常，属性的集合称做**进程控制块**（process control block）[⊖]。程序、数据、栈和属性的集合称做**进程映像**（process image）（见表 3.4）。

表 3.4　进程映像中的典型元素

项　目	说　明
用户数据	用户空间中的可修改部分，可以包括程序数据、用户栈区域和可修改的程序
用户程序	将被执行的程序
系统栈	每个进程有一个或多个后进先出（LIFO）系统栈。栈用于保存参数、过程调用地址和系统调用地址
进程控制块	操作系统控制进程所需要的数据（见表 3.5）

进程映像的位置依赖于使用的内存管理方案。对于最简单的情况，进程映像保存在邻近的或

⊖ 这个数据结构的其他一些常用的名字有任务控制块、进程描述符和任务描述符。

连续的存储块中。该存储块位于外存（通常是磁盘）中。因此，如果操作系统要管理进程，其进程映像至少有一部分必须位于内存中，为执行该进程，整个进程映像必须载入内存中或至少载入虚拟内存中。因此，操作系统需要知道每个进程在磁盘中的位置，并且对于内存中的每个进程，需要知道其在内存中的位置。来看一下第 2 章中 CTSS 操作系统关于这个方案的一个稍微复杂些的变体。在 CTSS 中，当进程被换出时，部分进程映像可能保留在内存中。因此，操作系统必须跟踪每个进程映像的哪一部分仍然在内存中。

现代操作系统假定分页硬件允许用不连续的物理内存来支持部分常驻内存的进程㊀。在任何给定的时刻，进程映像的一部分可以在内存中，剩余部分可以在外存中㊁。因此，操作系统维护的进程表必须表明每个进程映像中每页的位置。

图 3.11 描绘了位置信息的结构。有一个主进程表，每个进程在表中都有一个表项，每一项至少包含一个指向进程映像的指针。如果进程映像包括多个块，则这个信息直接包含在主进程表中，或可以通过交叉引用内存表中的项得到。当然，这个描述是一般性描述，特定的操作系统将按自己的方式组织位置信息。

进程属性

复杂的多道程序系统需要关于每个进程的大量信息，正如前面所述，该信息可以保留在进程控制块中。不同的系统以不同的方式组织该信息，本章和下一章的末尾都有很多这样的例子。目前先简单研究操作系统将会用到的信息，而不详细考虑信息是如何组织的。

表 3.5 列出了操作系统所需要的每个进程信息的简单分类。读者看到所需要的信息量时可能会有些惊讶。随着读者对操作系统进一步的理解，这个列表看起来会更加合理。

可以把进程控制块信息分成三类：

- 进程标识信息
- 处理器状态信息
- 进程控制信息

表 3.5　进程控制块中的典型元素

进程标识信息	
标识符	存储在进程控制块中的数字标识符，包括 • 此进程的标识符（Process ID，简称进程 ID） • 创建这个进程的进程（父进程）标识符 • 用户标识符（User ID，简称用户 ID）
处理器状态信息	
用户可见寄存器	用户可见寄存器是处于用户态的处理器执行的机器语言可以访问的寄存器。通常有 8 到 32 个此类寄存器，而在一些 RISC 实现中有超过 100 个此类寄存器
控制和状态寄存器	用于控制处理器操作的各种处理器寄存器，包括： • 程序计数器：包含将要取的下一条指令的地址 • 条件码：最近的算术或逻辑运算的结果（例如符号、零、进位、等于、溢出） • 状态信息：包括中断允许/禁用标志、异常模式
栈指针	每个进程有一个或多个与之相关联的后进先出（LIFO）系统栈。栈用于保存参数和过程调用或系统调用的地址，栈指针指向栈顶

㊀ 关于页、段和虚拟内存的概念，我们已在 2.3 节中进行了简要的描述。

㊁ 这个简要的讨论回避了一些细节，特别在使用虚拟内存的系统中，所有活动的进程映像都存储在外存中。当进程映像的一部分加载到内存中时，其加载过程是复制而并非移动。因此，外存保留了进程映像中的所有段（或页）的拷贝。但是当位于内存中的部分进程映像被修改时，在内存的更新部分被复制到外存以前，外存中的进程映像内容是过时的。

（续）

进程控制信息	
调度和状态信息	这是操作系统执行其调度功能所需要的信息，典型的信息项包括： • 进程状态：定义将被调度执行的进程的准备情况（例如运行态、就绪态、等待态、停止态） • 优先级：用于描述进程调度优先级的一个或多个域。在某些系统中，需要多个值（例如默认、当前、最高许可） • 调度相关信息：这取决于所使用的调度算法。例如进程等待的时间总量和进程在上一次运行时执行时间总量 • 事件：进程在继续执行前等待的事件标识
数据结构	进程可以以队列、环或者别的结构形式与其他进程进行链接。例如，所有具有某一特定优先级且处于等待状态的进程可链接在一个队列中；进程还可以表示出与另一个进程的父子（创建者-被创建者）关系。进程控制块为支持这些结构需要包含指向其他进程的指针
进程间通信	与两个独立进程间的通信相关联的各种标记、信号和信息。进程控制块中维护着某些或全部此类信息
进程特权	进程根据其可以访问的内存空间以及可以执行的指令类型被赋予各种特权。此外，特权还用于系统实用程序和服务的使用
存储管理	这一部分包括指向描述分配给该进程的虚拟内存空间的段表和页表的指针
资源的所有权和使用情况	进程控制的资源可以表示成诸如一个打开的文件，还可能包括处理器或其他资源的使用历史；调度器会需要这些信息

实际上在所有的操作系统中，对于**进程标识符**，每个进程都分配了一个唯一的数字标识符。进程标识符可以简单地表示为主进程表（图 3.11 所示）中的一个索引；否则，必须有一个映射，使得操作系统可以根据进程标识符定位相应的表。这个标识符在很多地方都是很有用的，操作系统控制的许多其他表可以使用进程标识符交叉引用进程表。例如，内存表可以组织起来以便提供一个关于内存的映射，指明每个区域分配给了哪个进程。I/O 表和文件表中也会有类似的引用。当进程相互之间进行通信时，进程标识符可用于通知操作系统某一特定通信的目标；当允许进程创建其他进程时，标识符可用于指明每个进程的父进程和后代进程。

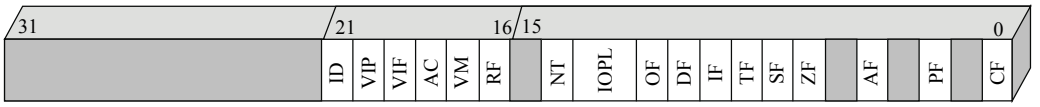
除了进程标识符，还给进程分配了一个用户标识符，用于标明拥有该进程的用户。

处理器状态信息包括处理器寄存器的内容。当一个进程正在运行时，其信息当然在寄存器中。当进程被中断时，所有的寄存器信息必须保存起来，使得进程恢复执行时这些信息都可以被恢复。所涉及的寄存器的种类和数目取决于处理器的设计。在典型情况下，寄存器组包括用户可见寄存器、控制和状态寄存器和栈指针，这些在第 1 章中都曾介绍过。

特别需要注意的是，所有的处理器设计都包括一个或一组通常称做程序状态字（Program Status Word, PSW）的寄存器，它包含状态信息。PSW 通常包含条件码和其他状态信息。Pentium 处理器中的处理器状态字就是一个很好的例子，它称做 EFLAGS 寄存器（如图 3.12 和表 3.6 所示），可被运行在 Pentium 处理器上的任何操作系统（包括 UNIX 和 Windows）使用。

进程控制块中第三个主要的信息类可以称做**进程控制信息**，这是操作系统控制和协调各种活动进程所需要的额外信息。表 3.5 的最后一部分表明了这类信息的范围，在随后的章节中将进一步详细分析操作系统的功能，表中所列出的各项用途也会逐渐明了。

图 3.13 给出了虚拟内存中进程映像的结构。每个进程映像包括一个进程控制块、用户栈、进程的专用地址空间以及与别的进程共享的任何其他地址空间。在这个图中，每个进程映像表现为一段地址相邻的区域。在实际的实现中可能不是这种情况，这取决于内存管理方案和操作系统组织控制结构的方法。



ID 表示标识符
VIP 表示虚拟中断挂起
VIF 表示虚拟中断标志
AC 表示对齐检查
VM 表示虚拟 8086 模式
RF 表示恢复标志
NT 表示嵌套任务标志
IOPL 表示 I/O 特权级
OF 表示溢出标志
DF 表示方向标志
IF 表示中断允许标志
TF 表示陷阱标志
SF 表示符号标志
ZF 表示零标志
AF 表示辅助进位标志
PF 表示奇偶校验标志
CF 表示进位标志

图 3.12　Pentium Ⅱ EFLAGS 寄存器

表 3.6　Pentium EFLAGS 寄存器位

控制位	
AC（对齐检查）	如果一个字或双字被定位在一个非字或非双字边界时置位
ID（标识符位）	如果这一位可以被置位和清除，处理器支持 CPUID 指令，这个指令可提供关于厂商、产品系列和模型的信息
RF（恢复标志）	允许程序员禁止调试异常，因此在一个调试异常后指令可以重新启动，不会立即引起另一个调试异常
IOPL（I/O 特权级）	当置位时，导致在保护模式操作期间，处理器在访问 I/O 设备时控制位产生一个异常
DF（方向标志）	确定字符串处理指令是否增长或缩减到 16 位的半个寄存器 SI 和 DI（用于 16 位操作）或 32 位寄存器 ESI 和 EDI（用于 32 位操作）
IF（中断允许标志）	当置位时，处理器将识别外部中断
TF（陷阱标志）	当置位时，每个指令执行后会引发一个中断。可用于调试
操作模式位	
NT（嵌套任务标志）	表明当前任务嵌套在运行于保护模式操作下的另一个任务中
VM（虚拟 8086 模式）	允许程序员启用或禁止虚拟 8086 模式，这决定了处理器是否像 8086 机那样运行
VIP（虚拟中断挂起）	用于虚拟 8086 模式下，表明一个或多个中断正在等待服务
VIF（虚拟中断标志）	用于虚拟 8086 模式下，代替 IF
条件码	
AF（辅助进位标志）	表示在一个使用 AL 寄存器的 8 位算术或逻辑运算中半个字节间的进位或借位
CF（进位标志）	表明在一个算术运算后，最左位的进位或借位情况。也可被一些移位或循环操作改变
OF（溢出标志）	表明在一次加法或减法后的算术溢出情况
PF（奇偶校验标志）	表明算术或逻辑运算结果的奇偶情况。1 表示偶数奇偶校验，0 表示奇数奇偶校验
SF（符号标志）	表明算术或逻辑运算结果的符号位情况
ZF（零标志）	表明算术或逻辑运算的结果是否为零的情况

正如表 3.5 中所指出的，进程控制块还可能包含构造信息，包括将进程控制块链接起来的指针。因此，前一节中所描述的队列可以由进程控制块的链表实现，例如，图 3.8a 中的排队结构可以按图 3.14 中的方式实现。

进程控制块的作用

进程控制块是操作系统中最重要的数据结构。每个进程控制块包含操作系统所需要的关于进程的所有信息。实际上，操作系统中的每个模块，包括那些涉及调度、资源分配、中断处理、性能监控和分析的模块，都可能读取和修改它们。可以说，资源控制块集合定义了操作系统的状态。

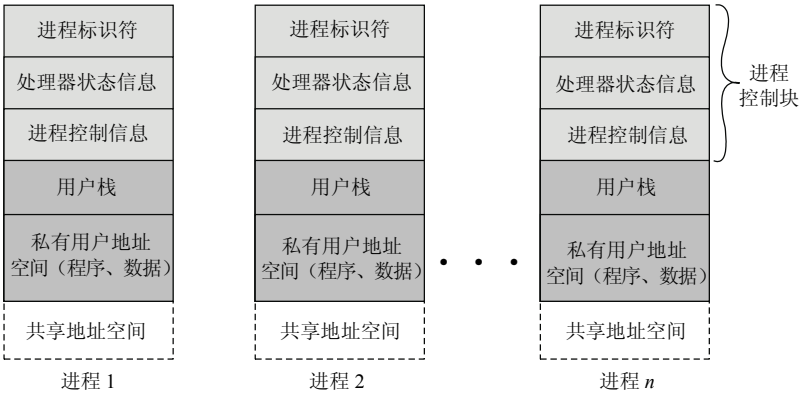


图 3.13　虚拟内存中的用户进程

这就带来了一个重要的设计问题。操作系统中的很多例程都需要访问进程控制块中的信息，直接访问这些表并不难，每个进程都有一个唯一 ID 号，可用作进程控制块指针表的索引。困难的不是访问而是保护，具体表现为下面两个问题：

- 一个例程（如中断处理程序）中有错误，可能会破坏进程控制块，进而破坏了系统对受影响进程的管理能力。
- 进程控制块的结构或语义的设计变化可能会影响到操作系统中的许多模块。

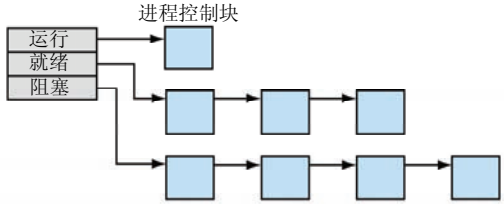


图 3.14　进程链表结构

这些问题可以通过要求操作系统中的所有例程都通过一个处理例程来专门处理，处理例程的任务仅仅是保护进程控制块，它是读写这些块的唯一的仲裁程序。使用这类进程，需要权衡性能问题和对系统软件剩余部分正确性的信任程度。

3.4　进程控制

3.4.1　执行模式

在继续讨论操作系统管理进程的方式之前，需要区分通常与操作系统相关联的以及与用户程序相关联的处理器执行模式。大多数处理器至少支持两种执行模式。某些指令只能在特权态下运行，包括读取或改变诸如程序状态字之类控制寄存器的指令、原始 I/O 指令和与内存管理相关的指令。另外，有部分内存区域仅在特权态下可以被访问到。

非特权态常称做**用户态**，这是因为用户程序通常在该模式下运行；特权态可称做**系统态**、**控制态**或**内核态**，内核态指的是操作系统的内核，这是操作系统中包含重要系统功能的部分。表 3.7 列出了操作系统内核中通常可以找到的功能。

使用两种模式的原因是很显然的，它可以保护操作系统和重要的操作系统表（如进程控制块）不受用户程序的干涉。在内核态下，软件具有对处理器以及所有指令、寄存器和内存的控制能力，这一级的控制对用户程序不是必需的，并且为了安全起见也不是用户程序可访问的。

这样产生了两个问题：处理器如何知道它正在什么模式下执行以及如何改变这一模式。对第一个问题，程序状态字中有一位表示执行模式，这一位应某些事件的要求而改变。在典型情况下，

当用户调用一个操作系统服务或中断触发系统例程的执行时，执行模式被设置成内核态，当从系统服务返回到用户进程时，执行模式被设置为用户态。例如 64 位 IA-64 体系结构的 Intel Itanium 处理器，有一个处理器状态寄存器（PSR），包含 2 位的 CPL（当前特权级别）域，级别 0 是最高特权级别，级别 3 是最低特权级别。大多数操作系统，如 Linux，使用级别 0 作为内核态，使用另一个级别作为用户态。当中断发生时，处理器清空大部分 PSR 中的位，包括 CPL 域，这将自动把 CPL 设置为 0。在中断处理例程结束时，最后的一个指令是 irt（中断返回），这条指令使处理器恢复中断程序的 PSR 值，也就是恢复了程序的特权级别。当应用程序调用一个系统调用时，会发生类似的情况。对于 Itanium，应用程序使用系统调用是通过以下方式实现的：把系统调用标识符和参数放在一个预定义的区域，然后通过执行一个特殊的指令中断用户态程序的执行，并把控制权交给内核。

表 3.7　操作系统内核的典型功能

进程管理
• 进程的创建和终止
• 进程的调度和分派
• 进程切换
• 进程同步以及对进程间通信的支持
• 进程控制块的管理
内存管理
• 给进程分配地址空间
• 交换
• 页和段的管理
I/O 管理
• 缓冲区管理
• 给进程分配 I/O 通道和设备
支持功能
• 中断处理
• 记账
• 监视

3.4.2　进程创建

3.2 节讲述了导致创建一个新进程的事件。在讨论了与进程相关的数据结构之后，现在可以简单描述实际创建进程时包含的步骤。

一旦操作系统决定基于某种原因（见表 3.1）创建一个新进程，它就可以按以下步骤继续进行：

1）**给新进程分配一个唯一的进程标识符**。此时，在主进程表中增加一个新表项，表中的每个新表项对应着一个进程。

2）**给进程分配空间**。这包括进程映像中的所有元素。因此，操作系统必须知道私有用户地址空间（程序和数据）和用户栈需要多少空间。可以根据进程的类型使用默认值，也可以在作业创建时根据用户请求设置。如果一个进程是由另一个进程生成的，则父进程可以把所需的值作为进程创建请求的一部分传递给操作系统。如果任何现有的地址空间被这个新进程共享，则必须建立正确的连接。最后，必须给进程控制块分配空间。

3）**初始化进程控制块**。进程标识符部分包括进程 ID 和其他相关的 ID，如父进程的 ID 等；处理器状态信息部分的大多数项目通常初始化成 0，除了程序计数器（被置为程序入口点）和系统栈指针（用来定义进程栈边界）；进程控制信息部分的初始化基于标准默认值和为该进程所请求的属性。例如，进程状态在典型情况下被初始化成就绪或就绪/挂起；除非显式地请求更高的优先级，否则优先级的默认值为最低优先级；除非显式地请求或从父进程处继承，否则进程最初不拥有任何资源（I/O 设备、文件）。

4）**设置正确的连接**。例如，如果操作系统把每个调度队列都保存成链表，则新进程必须放置在就绪或就绪/挂起链表中。

5）**创建或扩充其他数据结构**。例如，操作系统可能为每个进程保存着一个记账文件，可用于编制账单和/或进行性能评估。

3.4.3 进程切换

从表面看，进程切换的功能是很简单的。在某一时刻，一个正在运行的进程被中断，操作系统指定另一个进程为运行态，并把控制权交给这个进程。但是这会引发若干问题。首先，什么事件触发进程的切换？另一个问题是必须认识到模式切换与进程切换之间的区别。最后，为实现进程切换，操作系统必须对它控制的各种数据结构做些什么？

何时切换进程

进程切换可以在操作系统从当前正在运行的进程中获得控制权的任何时刻发生。表 3.8 给出了可能把控制权交给操作系统的事件。

首先考虑系统中断。实际上，大多数操作系统区分两种类型的系统中断。一种称为中断，另一种称为陷阱。前者与当前正在运行的进程无关的某种类型的外部事件相关，如完成一次 I/O 操作；后者与当前正在运行的进程所产生的错误或异常条件相关，如非法的文件访问。对于普通**中断**，控制首先转移给中断处理器，它做一些基本的辅助工作，然后转到与已经发生的特定类型的中断相关的操作系统例程。参见以下例子：

- **时钟中断**：操作系统确定当前正在运行的进程的执行时间是否已经超过了最大允许时间段（**时间片**，即进程在被中断前可以执行的最大时间段），如果超过了，进程必须切换到就绪态，调入另一个进程。
- **I/O 中断**：操作系统确定是否发生了 I/O 活动。如果 I/O 活动是一个或多个进程正在等待的事件，操作系统就把所有相应的阻塞态进程转换到就绪态（阻塞/挂起态进程转换到就绪/挂起态），操作系统必须决定是继续执行当前处于运行态的进程，还是让具有高优先级的就绪态进程抢占这个进程。
- **内存失效**：处理器访问一个虚拟内存地址，且此地址单元不在内存中时，操作系统必须从外存中把包含这个引用的内存块（页或段）调入内存中。在发出调入内存块的 I/O 请求之后，操作系统可能会执行一个进程切换，以恢复另一个进程的执行，发生内存失效的进程被置为阻塞态，当想要的块调入内存中时，该进程被置为就绪态。

表 3.8　进程执行的中断机制

机　制	原　因	使　用
中断	当前指令的外部执行	对异步外部事件的反应
陷阱	与当前指令的执行相关	处理一个错误或异常条件
系统调用	显式请求	调用操作系统函数

对于**陷阱**，操作系统确定错误或异常条件是否是致命的。如果是，当前正在运行的进程被转换到退出态，并发生进程切换；如果不是，操作系统的动作取决于错误的种类和操作系统的设计，其行为可以是试图恢复或通知用户，操作系统可能会进行一次进程切换或者继续执行当前正在运行的进程。

最后，操作系统可能被来自正在执行的程序的**系统调用**激活。例如，一个用户进程正在运行，并且正在执行一条请求 I/O 操作的指令，如打开文件，这个调用导致转移到作为操作系统代码一部分的一个例程上执行。通常，使用系统调用会导致把用户进程置为阻塞态。

模式切换

第 1 章讲述了中断阶段是指令周期的一部分。在中断阶段，处理器检查是否发生了任何中断，这通过中断信号来表示。如果没有未处理的中断，处理器继续取指令周期，即取当前进程中的下一条指令，如果存在一个未处理的中断，处理器需要做以下工作：

1）把程序计数器置成中断处理程序的开始地址。

2）从用户态切换到内核态，使得中断处理代码可以包含有特权的指令。

处理器现在继续取指阶段，并取中断处理程序的第一条指令，它将给中断提供服务。此时，被中断的进程上下文保存在被中断程序的进程控制块中。

现在的问题是，保存的上下文环境包括什么？答案是它必须包括所有中断处理可能改变的信息和恢复被中断程序时所需要的信息。因此，必须保存称做处理器状态信息的进程控制块部分，这包括程序计数器、其他处理器寄存器和栈信息。

还需要做些其他工作吗？这取决于下一步会发生什么。中断处理程序通常是执行一些与中断相关的基本任务的小程序。例如，它重置表示出现中断的标志或指示器。可能给产生中断的实体如 I/O 模块发送应答。它还做一些与产生中断的事件结果相关的基本辅助工作。例如，如果中断与 I/O 事件有关，中断处理程序将检查错误条件；如果发生了错误，中断处理程序给最初请求 I/O 操作的进程发一个信号。如果是时钟中断，处理程序将控制移交给分派器，当分配给当前正在运行进程的时间片用尽时，分派器将控制转移给另一个进程。

进程控制块中的其他信息如何处理？如果中断之后是切换到另一个应用程序，则需要做一些工作。但是，在大多数操作系统中，中断的发生并不是必须伴随着进程切换的。可能是中断处理器执行之后，当前正在运行的进程继续执行。在这种情况下，所需要做的是当中断发生时保存处理器状态信息，当控制返回给这个程序时恢复这些信息。在典型情况下，保存和恢复功能由硬件实现。

进程状态的变化

显然，模式切换与进程切换是不同的[⊖]。发生模式切换可以不改变正处于运行态的进程状态，在这种情况下，保存上下文环境和以后恢复上下文环境只需要很少的开销。但是，如果当前正在运行的进程被转换到另一个状态（就绪、阻塞等），则操作系统必须使其环境产生实质性的变化，完整的进程切换步骤如下：

1）保存处理器上下文环境，包括程序计数器和其他寄存器。

2）更新当前处于运行态进程的进程控制块，包括将进程的状态改变到另一状态（就绪态、阻塞态、就绪/挂起态或退出态）。还必须更新其他相关域，包括离开运行态的原因和记账信息。

3）将进程的进程控制块移到相应的队列（就绪、在事件 *i* 处阻塞、就绪/挂起）。

4）选择另一个进程执行，这方面的内容将在本书的第四部分探讨。

5）更新所选择进程的进程控制块，包括将进程的状态变为运行态。

6）更新内存管理的数据结构，这取决于如何管理地址转换，这方面的内容将在第三部分探讨。

7）恢复处理器在被选择的进程最近一次切换出运行状态时的上下文环境，这可以通过载入程序计数器和其他寄存器以前的值来实现。

因此，进程切换涉及状态变化，因而比模式切换需要做更多的工作。

3.5　操作系统的执行

第 2 章给出了关于操作系统的两个特殊事实：

⊖ 上下文切换这个术语经常出现在一些操作系统的文献和课本中。遗憾的是，尽管大部分文献把这个术语当做进程切换来使用，但是其他一些资料把它当成模式切换或者线程切换，线程切换将在后面的章节讲述。为了防止产生歧义，本书中不使用这个术语。

- 操作系统与普通的计算机软件以同样的方式运行，也就是说，它也是由处理器执行的一个程序。
- 操作系统经常释放控制权，并且依赖于处理器恢复控制权。

如果操作系统仅仅是一组程序，并且像其他程序一样由处理器执行，那么操作系统是一个进程吗？如果是，如何控制它？这些有趣的问题列出了大量的设计方法，图 3.15 给出了在当代各种操作系统中使用的各种方法。

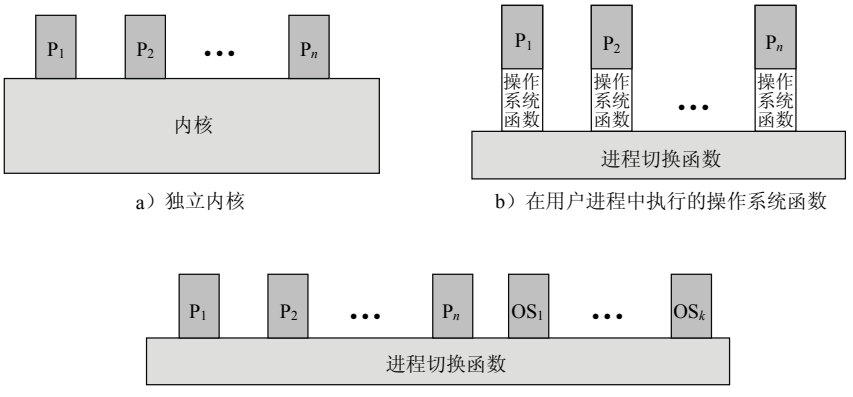
3.5.1　无进程的内核

在许多老操作系统中，非常传统和通用的一种方法是在所有的进程之外执行操作系统内核（见图 3.15a）。通过这种方法，在当前正运行的进程被中断或产生一个系统调用时，该进程的模式上下文环境被保存起来，控制权转交给内核。操作系统有自己的内存区域和系统栈，用于控制过程调用和返回。操作系统可以执行任何预期的功能，并恢复被中断进程的上下文，这将导致被中断的用户进程重新继续执行。或者，操作系统可以完成保存进程环境的功能，并继续调度和分派另一个进程，是否这样做取决于中断的原因和当前的情况。

无论哪种情况，其关键点是进程的概念仅仅适用于用户程序，操作系统代码作为一个在特权模式下工作的独立实体被执行。

3.5.2　在用户进程中执行

在较小的机器（PC 机、工作站）的操作系统中，常见的方法实际上是在用户进程的上下文中执行几乎所有操作系统软件。其观点是操作系统从根本上说是用户调用的一组例程，在用户进程环境中执行，用于实现各种功能，如图 3.15b 所示。在任何时刻，操作系统管理着 n 个进程映像，每个映像不仅包括图 3.13 中列出的区域，而且还包括内核程序的程序、数据和栈区域。



a）独立内核　　b）在用户进程中执行的操作系统函数

c）作为一个独立进程执行的操作系统函数

图 3.15　操作系统和用户进程的关系

图 3.16 给出了这个策略下的一个典型的进程映像结构。当进程在内核态下时，独立的内核栈用于管理调用/返回。操作系统代码和数据位于共享地址空间中，被所有的用户进程共享。

当发生一个中断、陷阱或系统调用时，处理器被置于内核态，控制权转交给操作系统。为了将控制从用户程序转交给操作系统，需要保存模式上下文环境并进行模式切换，然后切换到一个操作系统例程，但此时仍然是在当前用户进程中继续执行，因此，不需要执行进程切换，仅在同一个进程中进行模式切换。

如果操作系统完成其操作后，确定需要继续运行当前进程，则进行一次模式切换，在当前进

程中恢复被中断的程序。该方法的重要优点之一是，一个用户程序被中断以使用某些操作系统例程，然后被恢复，所有这些不用以牺牲两次进程切换为代价。如果确定需要发生进程切换而不是返回到先前执行的程序，则控制权被转交给进程切换例程，这个例程可能在当前进程中执行，也可能不在当前进程中执行，这取决于系统的设计。在某些特殊的情况下，如当前进程必须置于非运行态，而另一个进程将指定为正在运行的进程。为方便起见，这样一个转换过程在逻辑上可以看做是在所有进程之外的环境中被执行的。

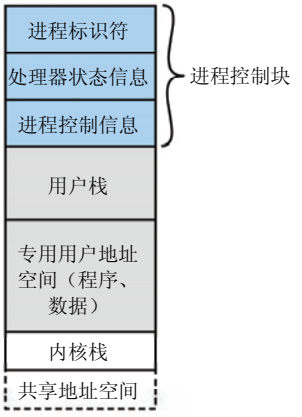


图 3.16　进程映像：操作系统在用户空间中执行

在某种程度上，对操作系统的这种看法是非常值得注意的。在某些时候，一个进程可以保存它的状态信息，从就绪态进程中选择另一个进程，并把控制权释放给这个进程。之所以说这是一种混杂的情况，是因为在关键时候，在用户进程中执行的代码是共享的操作系统代码而不是用户代码。基于用户态和内核态的概念，即使操作系统例程在用户进程环境中执行，用户也不能篡改或干涉操作系统例程。这进一步说明进程和程序的概念是不同的，它们之间不是一对一的关系。在一个进程中，用户程序和操作系统程序都有可能执行，而在不同用户进程中执行的操作系统程序是相同的。

3.5.3　基于进程的操作系统

图 3.15c 中显示的是另一种选择，即把操作系统作为一组系统进程来实现。与其他选择一样，作为内核一部分的软件在内核态下执行。不过在这种情况下，主要的内核函数被组织成独立的进程，同样，还可能有一些在任何进程之外执行的进程切换代码。

这种方法有几个优点。它利用程序设计原理，促使使用模块化操作系统，并且模块间具有最小的、简明的接口。此外，一些非关键的操作系统函数可简单地用独立的进程实现，例如，很早就曾经提到过的用于记录各种资源（处理器、内存、通道）的使用程度和系统中用户进程的执行速度的监控程序。由于这个程序没有为任何活动进程提供特定的服务，它只能被操作系统调用。作为一个进程，这个函数可以在指定的优先级上运行，并且在分派器的控制下与其他进程交替执行。最后，把操作系统作为一组进程实现，在多处理器或多机环境中都是十分有用的，这时一些操作系统服务可以传送到专用处理器中执行，以提高性能。

3.6　安全问题

操作系统对于每个进程都关联了一套权限。这些权限规定了进程可以获取哪些资源，包括内存区域、文件和特权系统指令等。典型的是，一个进程的运行代表着一个用户拥有操作系统认证的权限。在配置的时候，一个系统或者是一个实用进程就被分配了权限。

在一个典型的系统中，最高级别的权限指的是管理员、超级用户或根用户的访问权限[⊖]。根用户的访问权限提供了对一个操作系统所有的功能和服务的访问。一个有着根用户访问权限的进程可以安全地控制一个系统，可以增加或者改变程序和文件，对其他进程进行监控，发送和接收网络流量和改变权限。

设计任何操作系统的一个关键问题是阻止或者至少是探测一个用户或者是一种恶意软件（恶意程序）获得系统授权的权限的企图，尤其是从根用户获取。本节，我们将简短地总结关

⊖　在 UNIX 系统中，管理员或超级用户，这种账户称为根用户；因此有术语根用户访问。

于这种安全问题的威胁和对策。在第七部分将对其做更加详细的阐述。

3.6.1　系统访问威胁

系统访问威胁分为两大类：入侵者和恶意软件。

入侵者

对于安全，一个最普通的威胁就是入侵者（另外一个是病毒），通常是指黑客和解密高手。在早期的一项对入侵的重要研究中，Anderson [ANDE80] 定义了三种类型的入侵者：

冒充者：没有授权的个人去使用计算机和通过穿透系统的访问控制去使用一个合法用户的账号。

滥用职权者：一个合法的用户访问没有授权的数据、程序或资源，或者用户具有这种访问的授权，但是滥用了他的权限。

秘密用户：一个用户获得了系统的管理控制，然后使用这种控制来逃避审计和访问控制，或者废止审查收集。

冒充者有可能就是外部人员；滥用职权者一般都是内部人员；秘密用户可能是外部人员也可能是内部人员。

入侵者的攻击有良性的也有严重的。在良性阶段，许多人仅仅是想浏览互联网并想知道在互联网上到底有什么。在严重阶段，这些人尝试着去读权限数据，修改未授权的数据或者是破坏系统。

入侵者的目的是获得一个系统的访问权限，或是增加一个系统的权限获取的范围。最初许多攻击是利用了系统或软件的漏洞，这些漏洞可以让用户执行可以开启系统后门的代码。当程序以一定的权限运行，入侵者可以利用如缓冲溢出区攻击来获得系统的访问。将在 7 章介绍缓冲区溢出攻击。

此外，入侵者也可以尝试获取那些已经被保护的信息。在一些情况下，信息就是在表框里面的用户密码。如果知道了一些用户的密码，那么入侵者可以登录一个系统，然后运行合法用户的所有权限。

恶意软件

计算机系统最为复杂的威胁可能就是利用计算机系统漏洞的程序。这些威胁被称为**恶意软件**，或者是**恶意程序**。在这一部分，我们将关注如编辑器、编译器和内核级程序等应用程序以及通用程序的威胁。

恶意软件分为两大类：一种需要宿主程序，一种则是独立的。对于前者，也被称为**寄生**，其本质上是一些不能独立于实际应用程序、通用程序或系统程序而存在的片段，例如病毒、逻辑炸弹和后门。后者则是独立并可以被操作系统调度和运行的程序，例如蠕虫和僵尸程序。

我们也可以对这些软件威胁进行以下的区分：一种不能进行复制，而另外一种则可以。前者是通过触发器激活的程序或者程序片段，例如，逻辑炸弹、后门和僵尸程序。后者由一个程序片段或者一个独立的程序构成，当其运行后，可能产生一个或者多个它自己的副本，这些副本将在同一系统或其他系统中被激活，例如病毒和蠕虫。

比较而言，恶意软件可能是无害的，或者表现为一个或多个有害的操作，这些有害的操作包括毁坏内存里的文件和数据，通过绕开控制而获得权限访问和为入侵者提供一种绕开访问控制的方法。

3.6.2　对抗措施

入侵检测

RFC2828（网络安全术语表）对入侵检测的定义如下：入侵检测是一种安全服务，通过监控

和分析系统事件发现试图通过未经授权的方法访问系统资源的操作，并对这种操作提供实时或者准实时的警告。

入侵检测系统（IDS）可以分为如下几类：

- **基于宿主的 IDS**：对于可疑活动，监控单个宿主的特征和发生在宿主上的事件。
- **基于网络的 IDS**：监控特定的网络段或者设备网络流量，分析网络、传输和应用协议来辨别可疑活动。

IDS 由三个部分组成：

- **感应器**：感应器负责收集数据。感应器的输入可能是系统的任一部分，输入可能包含了侵扰的证据。典型的感应器输入包括网络包、日志文件和系统调用记录。感应器收集这些信息，并把这些信息发送给分析器。
- **分析器**：分析器从一个或多个感应器或者其他分析器接受输入数据。分析器负责确定入侵是否发生。这部分的输出表明了一个入侵已经发生。输出可能包含了支持入侵发生的结论的证据。分析器可能提供对于入侵结果采取何种操作的指导。
- **用户界面**：IDS 的用户界面可以让用户查看系统的输出和控制系统的行为。在一些系统中，用户界面可以等同于负责人、控制器或者控制台部分。

入侵检测系统用来侦测人类入侵者行为，以及恶意软件的行为。

认证

在许多计算机安全内容中，用户认证是一个主要的构建模块和最初防线。用户认证是许多种访问控制和用户责任的主要部分。RFC2828 对用户认证做了如下定义：

> 系统实体定义了验证和确认的过程。认证过程包括以下两步：
> - 确认步骤：对于安全系统，提出了标识符。（应小心地分配标识符，对于访问控制服务等其他安全服务，认证定义是基本部分。）
> - 验证步骤：提出或产生认证信息，用来证实在实体与标识符之间的绑定。

例如，用户 Alice Toklas 拥有一个用户标识符 ABTOKLAS。标识符信息可能被存储在 Alice 想要使用的任意一台服务器或者计算机系统中，而且系统管理员和其他用户可能知道这些信息。一种典型的与用户 ID 相关联的认证信息项就是密码，密码是用于保守秘密的（秘密仅仅被 Alice 和系统所知）。如果没有人得到或猜出 Alice 的密码，管理员可以通过 Alice 的用户 ID 和密码的结合建立 Alice 的访问许可，并审核 Alice 的操作。由于 Alice 的 ID 不是秘密，系统用户可以给她发送电子邮件，但是由于她的密码是保密的，所以没有人可以冒充成 Alice。

本质上，识别是这样一种方法：用户向系统提供一个声明的身份；用户认证就是确认声明的合法性的方法。

一共有 4 种主要的认证用户身份的方法，它们既可以单独使用，也可以联合使用：

- **个人知道的一些事物**：例如包括密码、个人身份号码（PIN）或者是预先安排的一套问题的答案。
- **个人拥有的一些事物**：例如包括电子通行卡、智能卡、物理钥匙。这种类型的身份验证称为令牌。
- **个人自身的事物（静态生物识别技术）**：例如包括指纹、虹膜和人脸的识别。
- **个人要做的事物（动态生物识别技术）**：例如包括语音模式、笔迹特征和输入节奏的识别。

适当的实现和使用所有的这些方法，可以提供可靠的用户认证。但是每个方法都有问题，使得对手能够猜测或盗取密码。类似地，用户能够伪造或盗取令牌。用户可能忘记密码或丢失令牌。而且，对于管理系统上的密码、令牌信息和保护系统上的这些信息，还存在显著的管理开销。对

于生物识别技术，也有各种各样的问题，包括误报和假否定、用户接受程度、费用和便利与否。

访问控制

访问控制实现了一种安全策略：指定谁或何物（例如进程的情况）可能有权使用特定的系统资源和在每个场景下被允许的访问类型。

访问控制机制调节了用户（或是代表用户执行的进程）与系统资源之间的关系，系统资源包括应用程序、操作系统、防火墙、路由器、文件和数据库。系统首先要对寻求访问的用户进行认证。通常，认证功能完全决定了用户是否被允许访问系统。然后访问控制功能决定了是否允许用户的特定访问要求。安全管理员维护着一个授权数据库，对于允许用户对何种资源采用什么样的访问方式，授权数据库做了详细说明。访问控制功能参考这个数据库来决定是否准予访问。审核功能监控和记录了用户对于系统资源的访问。

防火墙

对于保护本地系统或系统网络免于基于网络的安全威胁，防火墙是一种有效的手段，并且防火墙同时提供了经过广域网和互联网对外部网络的访问。传统上，防火墙是一种专用计算机，是计算机与外部网络的接口；防火墙内部建立了特殊的安全预防措施用以保护网络中计算机上的敏感文件。其被用于服务外部网络，尤其是互联网、连接和拨号线路。使用硬件或软件来实现，并且与单一的工作站或者 PC 连接的个人防火墙也很常见。

[BELL94]列举了如下防火墙的设计目标：

1）从内部到外部的通信必须通过防火墙，反之亦然。通过对除经过防火墙之外本地网络的所有访问都进行物理阻塞来达到目的。可以对其进行各种各样的配置，将在本章的后面部分进行阐述。

2）仅仅允许本地安全策略定义的授权通信通过。使用的不同类型防火墙是通过不同类型的安全策略实现的。

3）防火墙本身对于渗透是免疫的。这意味着在一个安全的操作系统上使用强固系统。值得信赖的计算机系统适合用作防火墙，并且在管理应用中也被要求使用。

3.7　UNIX SVR4 进程管理

UNIX 系统 V 使用了一种简单但是功能强大的进程机制，且对用户可见。UNIX 采用图 3.15b 中的模型，其中大部分操作系统在用户进程环境中执行。UNIX 使用两类进程，即系统进程和用户进程。系统进程在内核态下运行，执行操作系统代码以实现管理功能和内部处理，如内存空间的分配和进程交换；用户进程在用户态下运行以执行用户程序和实用程序，在内核态下运行以执行属于内核的指令。当产生异常（错误）或发生中断或用户进程发出系统调用时，用户进程可进入内核态。

3.7.1　进程状态

UNIX 操作系统中共有 9 种进程状态，如表 3.9 所示。图 3.17（基于［BACH86］中的图）是相应的状态转换图，它与图 3.9b 类似，有两个 UNIX 睡眠状态对应于图 3.9b 中的两个阻塞状态，其区别可简单概括如下：

- UNIX 采用两个运行态表示进程在用户态下执行还是在内核态下执行。
- UNIX 区分内存中就绪态和被抢占态这两个状态。从本质上看，它们是同一个状态，如图中它们之间的虚线所示，之所以区分这两个状态是为了强调进入被抢占状态的方式。当一个进程正在内核态下运行时（系统调用、时钟中断或 I/O 中断的结果），内核已经完成了其任务并准备把控制权返回给用户程序时，就可能会出现抢占的时机。这时，内核可能决定抢占当前进程，支持另一个已经就绪并具有较高优先级的进程。在这种情况下，

当前进程转换到被抢占态，但是为了分派处理，处于被抢占态的进程和处于内存中就绪态的进程构成了一条队列。

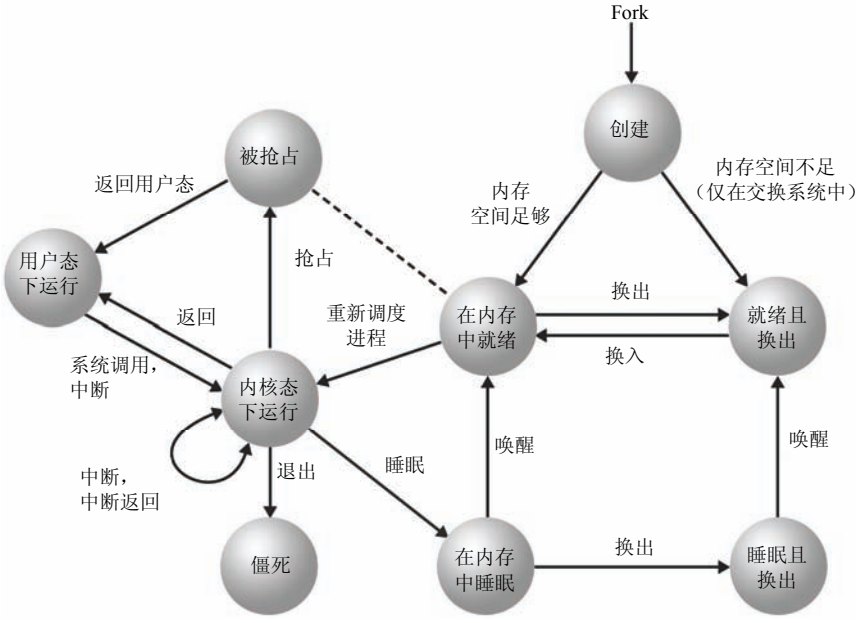


图 3.17　UNIX 进程状态转换图

只有当进程准备从内核态移到用户态时才可能发生抢占，进程在内核态下运行时是不会被抢占的，这使得 UNIX 不适用于实时处理。有关实时处理需求的讨论请参见第 10 章。

UNIX 中有两个独特的进程。进程 0 是一个特殊的进程，是在系统启动时创建的。实际上，这是预定义的一个数据结构，在启动时刻被加载，是交换进程。此外，进程 0 产生进程 1，称做初始进程，进程 1 是系统中的所有其他进程的祖先。当新的交互用户登录到系统时，由进程 1 为该用户创建一个用户进程。随后，用户进程可以创建子进程，从而构成一棵分支树，因此，任何应用程序都是由一组相关进程组成的。

表 3.9　UNIX 进程状态

进程状态	说　明
用户运行	在用户态下执行
内核运行	在内核态下执行
就绪，并驻留在内存中	只要内核调度到就立即准备运行
睡眠，并驻留在内存中	在某事件发生前不能执行，且进程在内存中（一种阻塞态）
就绪，被交换	进程已经就绪，但交换程序必须把它换入内存，内核才能调度它去执行
睡眠，被交换	进程正在等待一个事件，并且被交换到外存中（一种阻塞态）
被抢占	进程从内核态返回到用户态，但是内核抢占它，并做了进程切换，以调度另一个进程
创建	进程刚被创建，还没有做好运行的准备
僵死	进程不再存在，但是它留下一个记录，该记录可由其父进程收集

3.7.2　进程描述

UNIX 中的进程是一组相当复杂的数据结构，它给操作系统提供管理进程和分派进程所需要的所有信息。表 3.10 概括了进程映像中的元素，它们被组织成三部分：用户上下文、寄存器上

下文和系统级上下文。

表 3.10　UNIX 进程映像

用户级上下文	
进程正文	程序中可执行的机器指令
进程数据	由这个进程的程序可访问的数据
用户栈	包含参数、局部变量和在用户态下运行的函数指针
共享内存区	与其他进程共享的内存区，用于进程间的通信
寄存器上下文环境	
程序计数器	将要执行的下一条指令地址，该地址是内核中或用户内存空间中的内存地址
处理器状态寄存器	包含在抢占时的硬件状态，其内容和格式取决于硬件
栈指针	指向内核栈或用户栈的栈顶，取决于当前的运行模式
通用寄存器	与硬件相关
系统级上下文环境	
进程表项	定义了进程的状态，操作系统总是可以取到这个信息
U（用户）区	含有进程控制信息，这些信息只需要在该进程的上下文环境中存取
本进程区表	定义了从虚地址到物理地址的映射，还包含一个权限域，用于指明进程允许的访问类型：只读、读写或读–执行
内核栈	当进程在内核态下执行时，它含有内核过程的栈帧

用户级上下文包括用户程序的基本成分，可以由已编译的目标文件直接产生。用户程序被分成正文和数据两个区域，正文区是只读的，用于保存程序指令。当进程正在执行时，处理器使用用户栈进行过程调用和返回以及参数传递。共享内存区是与其他进程共享的数据区域，它只有一个物理副本，但是通过使用虚拟内存，对每个共享进程来说，共享内存区看上去好像在它们各自的地址空间中一样。当进程没有运行时，处理器状态信息保存在**寄存器上下文**中。

系统级上下文包含操作系统管理进程所需要的其余信息，它由静态部分和动态部分组成，静态部分的大小是固定的，贯穿于进程的生命周期；动态部分在进程的生命周期中大小可变。静态部分的一个成分是进程表项，这实际上是由操作系统维护的进程表的一部分，每个进程对应于表中的一项。进程表项包含对内核来说总是可以访问到的进程控制信息。因此，在虚拟内存系统中，所有的进程表项都在内存中，表 3.11 中列出了进程表项的内容。用户区，即 U 区，包含内核在进程的上下文环境中执行时所需要的额外的进程控制信息，当进程调入或调出内存时也会用到它。表 3.12 给出了这个表的内容。

表 3.11　UNIX 进程表项

项　目	说　明
进程状态	进程的当前状态
指针	指向 U 区和进程内存区（文本、数据和栈）
进程大小	使操作系统知道给进程分配多少空间
用户标识符	**实用户 ID** 标识负责正在运行的进程的用户；**有效用户 ID** 标识可被进程使用，以获得与特定程序相关的临时特权，当该程序作为进程的一部分执行时，进程以有效用户 ID 的权限进行操作
进程标识符	该进程的 ID 和父进程的 ID。这一项是在系统调用 fork 期间，当进程进入新建态时设置的
事件描述符	当进程处于睡眠态时有效。当事件发生时，该进程转换到就绪态
优先级	用于进程调度

（续）

项　　目	说　　明
信号	列举发送到进程但还没有处理的信号
定时器	包括进程执行时间、内核资源使用和用户设置的用于给进程发送警告信号的计时器
p_link	指向就绪队列中的下一个链接（进程就绪时有效）
内存状态	指明进程映像是在内存中还是已被换出。如果在内存中，该域还指出它是否可能被换出，或者是临时锁定在内存中

表 3.12　UNIX 的 U 区

项　　目	说　　明
进程表指针	指明对应于 U 区的表项
用户标识符	实用户 ID 和有效用户 ID，用于确定用户的权限
定时器	记录进程（以及它的后代）在用户态下执行的时间和在内核态下执行的时间
信号处理程序数组	对系统中定义的每类信号，指出进程收到信号后将做出什么反应（退出、忽略、执行特定的用户函数）
控制终端	如果有该进程的登录终端时，则指明它
错误域	记录在系统调用时遇到的错误
返回值	包含系统调用的结果
I/O 参数	描述传送的数据量、源（或目标）数据数组在用户空间中的地址和用于 I/O 的文件偏移量
文件参数	描述进程的文件系统环境的当前目录和当前根
用户文件描述符表	记录进程已打开的文件
限度域	限制进程的大小和可以写入的文件大小
容许模式域	屏蔽在由进程创建的文件中设置的模式

进程表项和 U 区的区别反映出 UNIX 内核总是在某些进程的上下文环境中执行，大多数时候，内核都在处理与该进程相关的部分，但是，某些时候，如当内核正在执行一个调度算法，准备分派另一个进程时，它需要访问其他进程的相关信息。当给定进程不是当前进程时，可以访问进程控制表中的信息。

系统级上下文静态部分的第三项是本进程区表，它由内存管理系统使用。最后，内核栈是系统级上下文环境的动态部分，当进程正在内核态下执行时需要使用这个栈，它包含当发生过程调用或中断时必须保存和恢复的信息。

3.7.3　进程控制

UNIX 中的进程创建是通过内核系统调用 fork()实现的。当一个进程产生一个 fork 请求时，操作系统执行以下功能［BACH86］：

1）为新进程在进程表中分配一个空项。

2）为子进程赋一个唯一的进程标识符。

3）做一个父进程上下文的逻辑副本，不包括共享内存区。

4）增加父进程拥有的所有文件的计数器，以表示有一个另外的进程现在也拥有这些文件。

5）把子进程置为就绪态。

6）向父进程返回子进程的进程号；对子进程返回零。

所有这些操作都在父进程的内核态下完成。为当内核完成这些功能后可以继续下面三种操作之一，它们可以认为是分派器例程的一个部分：

- 在父进程中继续执行。控制返回用户态下父进程进行 fork 调用处。
- 处理器控制权交给子进程。子进程开始执行代码，执行点与父进程相同，也就是说在 fork 调用的返回处。
- 控制转交给另一个进程。父进程和子进程都置于就绪状态。

很难想象这种创建进程的方法中父进程和子进程都执行相同的代码。其区别在于：当从 fork 中返回时，测试返回参数，如果值为零，则它是子进程，可以转移到相应的用户程序中继续执行；如果值不为零，则它是父进程，继续执行主程序。

3.8　小结

现代操作系统中最基本的构件是进程，操作系统的基本功能是创建、管理和终止进程。当进程处于活跃状态时，操作系统必须设法使每个进程都分配到处理器执行时间，并协调它们的活动、管理有冲突的请求、给进程分配系统资源。

为执行进程管理功能，操作系统维护着对每个进程的描述，或者称为进程映像，它包括执行进程的地址空间和一个进程控制块。进程控制块含有操作系统管理进程所需要的所有信息，包括它的当前状态、分配给它的资源、优先级和其他相关数据。

在整个生命周期中，进程总是在一些状态之间转换。最重要的状态有就绪态、运行态和阻塞态。一个就绪态进程是指当前没有执行但已做好了执行准备的进程，只要操作系统调度到它就立即可以执行；运行态进程是指当前正在被处理器执行的进程，在多处理器系统中，会有多个进程处于这种状态；阻塞态进程正在等待某一事件的完成，如一次 I/O 操作。

一个正在运行的进程可被一个在进程外发生且被处理器识别的中断事件打断，或者被执行操作系统的系统调用打断。不论哪种情况，处理器都执行一次模式切换，把控制转交给操作系统例程。操作系统在完成必需的操作后，可以恢复被中断的进程或者切换到别的进程。

3.9　推荐读物

关于 UNIX 进程管理的详细描述请参阅［GOOD94］和［GRAY97］。［NEHM75］中有关于进程状态的讨论以及分派所需要的操作系统原语。

GOOD94 Goodheart, B., and Cox, J. *The Magic Garden Explained*: *The Internals of UNIX System V Release 4*. Englewood Cliffs, NJ: Prentice Hall, 1994.

GRAY97 Gray, J. *Interprocess Communications in UNIX: The Nooks and Crannies*. Upper Saddle River, NJ: Prentice Hall, 1997.

NEHM75 Nehmer, J. "Dispatcher Primitives for the Construction of Operation System Kernels" *Acta Informatica*, vol. 5, 1975.

3.10　关键术语、复习题和习题

关键术语

阻塞态	父进程	进程切换	交换
子进程	抢占	程序状态字	系统态
退出态	特权态	就绪态	任务
中断	进程	轮转	跟踪
内核态	进程控制块	运行态	陷阱
模式切换	进程映像	挂起态	用户态
新建态			

复习题

3.1　什么是指令跟踪？
3.2　通常有哪些事件会导致创建一个进程？
3.3　对于图 3.6 中的进程模型，请简单定义每个状态。
3.4　抢占一个进程是什么意思？
3.5　什么是交换，其目的是什么？
3.6　为什么图 3.9b 中有两个阻塞态？
3.7　列出挂起进程的 4 个特点。
3.8　对于哪类实体，操作系统为了管理它而维护其信息表？
3.9　列出进程控制块中的三类信息。
3.10　为什么需要两种模式（用户态和内核态）？
3.11　操作系统创建一个新进程所执行的步骤是什么？
3.12　中断和陷阱有什么区别？
3.13　举出中断的三个例子。
3.14　模式切换和进程切换有什么区别？

习题

3.1　给出操作系统进行进程管理时的 5 种主要活动，并简单描述为什么需要它们。
3.2　假设一个计算机有 A 个输入输出设备和 B 个处理器，在任何时候内存最多容纳 C 个进程。假设 A<B<C，试问：
a）任一时刻，处于就绪态、运行态、阻塞态、就绪挂起态、阻塞挂起态的最大进程数目各是多少？
b）处在就绪态、运行态、阻塞态、就绪挂起态、阻塞挂起态的最小进程数目各是多少？
3.3　图 3.9b 包含 7 个状态。假设我们让系统有两个就绪状态：一般就绪状态 READY 和 SYSTEM READY 状态。系统当中处于 SYSTEM READY 状态的进程拥有最高优先级，并且按轮转方式执行；CPU 调度算法让处于这种状态的进程拥有特权并且它们从不被交换出内存。请画出对应的状态转换图，标识出每一个状态转换。
3.4　假设我们是一个操作系统的开发人员，该操作系统采用五状态进程模型（如图 3.5 所示）。在该操作系统中，所有进程按照轮转方式执行。再假设我们正在为运行 I/O 密集型任务的计算机开发新的操作系统。为了保证这些任务在需要时（即当一个突发 I/O 完成时）能快速获得处理器，我们在进程模型中考虑两个 READY 状态：READY-CPU 状态（为 CPU 密集型进程）和 READY-I/O 状态（为 I/O 密集型进程）。系统中处于 READY- I/O 状态的进程拥有最高权限访问 CPU，并且以 FCFS 方式执行。处于 READY-CPU 状态的进程以轮转方式执行。当进程进入系统时，它们被分类为 CPU 密集型或者 I/O 密集型。
a）讨论用如上所描述的两种就绪状态实现操作系统的优点和缺点。
b）讨论如何通过五状态进程模型并修改进程调度算法来获得六状态进程模型的好处。
3.5　对于图 3.9b 中给出的 7 状态进程模型，请仿照图 3.8b 画出它的排队图。
3.6　考虑图 3.9b 中的状态转换图。假设操作系统正在分派进程，有进程处于就绪态和就绪/挂起态，并且至少有一个处于就绪/挂起态的进程比处于就绪态的所有进程的优先级都高。有两种极端的策略：1）总是分派一个处于就绪状态的进程，以减少交换；2）总是把机会给具有最高优先级的进程，即使会导致在不需要交换时进行交换。请给出一种能均衡考虑优先级和性能的中间策略。
3.7　表 3.9 展示了 UNIX SVR4 操作系统的 9 个进程状态。
a）列出表 3.9 中的 9 个状态并且指出这 9 个状态与图 3.9b 中 7 个状态之间的关系。
b）给出 UNIX SVR4 操作系统中存在两个运行状态的理由。
3.8　VAX/VMS 操作系统采用了四种处理器访问模式，以促进系统资源在进程间的保护和共享。访问模式确定：
- **指令执行特权**：处理器将执行什么指令。
- **内存访问特权**：当前指令可能访问虚拟内存中的哪个单元。

4 种模式如下：

- **内核模式**：执行 VMS 操作系统的内核，包括内存管理、中断处理和 I/O 操作。
- **执行模式**：执行许多操作系统服务调用，包括文件（磁盘和磁带）和记录管理例程。
- **管理模式**：执行其他操作系统服务，如响应用户命令。
- **用户模式**：执行用户程序和诸如编译器、编辑器、链接程序、调试器之类的实用程序。

在较少特权模式执行的进程通常需要调用在较多特权模式下执行的过程，例如，一个用户程序需要一个操作系统服务。这个调用通过使用一个改变模式（简称 CHM）指令来实现，该指令将引发一个中断，把控制转交给处于新的访问模式下的例程，并通过执行 REI（Return from Exception or Interrupt，从异常或中断中返回）指令返回。

a）很多操作系统有两种模式，内核态和用户态，那么提供 4 种模式有什么优点和缺点？

b）你可以举出一种有 4 种以上模式的情况吗？

表 3.13　VAX/VMS 的进程状态

进程状态	说　　明
当前正在执行	运行进程
可计算（驻留）	就绪并驻留在内存中
可计算（换出）	就绪但换出内存
页面失效等待	进程引用了不在内存中的页，必须等待读入该页
页冲突等待	程序引用了另一个正处于页面失效等待的进程所等待的共享页，或者引用了进程正在读入或写出的私有页
一般事件等待	等待共享事件标志（事件标志是单位进程间的信号机制）
自由页等待	等待内存中的一个自由页被加入到用于该进程的页集合中（进程的工作页面组）
休眠等待（驻留）	进程把自己置于等待状态
休眠等待（换出）	休眠进程被换出内存
本地事件等待（驻留）	进程在内存中，并正在等待局部事件标志（通常是 I/O 完成）
本地事件等待（换出）	处于本地事件等待状态的进程被换出内存
挂起等待（驻留）	进程被另一个进程置于等待状态
挂起等待（换出）	挂起进程被换出内存
资源等待	进程正在等待各种系统资源

3.9　在前面的问题中讨论的 VMS 方案常常称做环状保护结构，如图 3.18 所示。3.3 节所描述的简单的内核/用户方案是一种两环结构，[SILB04] 指出了这种方法的问题：

环状（层次）结构的主要缺点是它不允许我们实施须知原理，特别地，如果一个对象必须在域 D_j 中可访问，但在域 D_i 中不可访问，则必须有 $j<i$。这意味着在 D_i 中可访问的每个段在 D_j 中都可以访问。

a）请清楚地解释上面引文中提出的问题。

b）请给出环状结构操作系统解决这个问题的一种方法。

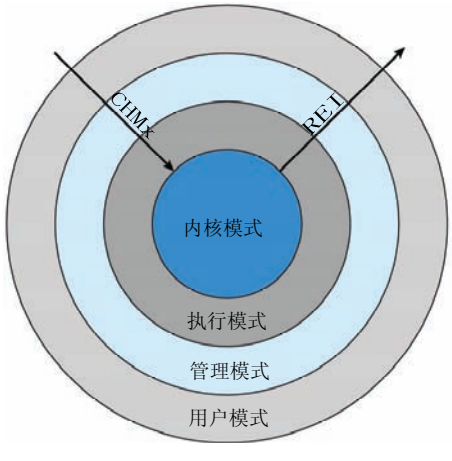


图 3.18　VAX/VMS 的访问模式

3.10　图 3.8b 表明一个进程每次只能在一个事件队列中。

a）是否能够允许进程同时等待一个或多个事件？请举例说明。

b）在这种情况下，如何修改图中的排队结构以支持这个新特点？

3.11　所有现代操作系统都有系统中断。

a）请解释中断如何支持多道程序设计。

b）请解释中断如何支持错误处理。

c）对于单线程进程，举例说明一个能够引起中断并且导致进程切换的情景。另外请举例说明能够引

起中断但是没有进程切换的情景。

3.12　如 3.7 节所述，在 UNIX 中进程是通过 fork()系统调用创建的。在处理 fork()请求的过程中，操作系统把子进程的 ID 返回给父进程。换言之，fork()系统调用创建了一个包含父子进程的进程树，其中父进程指向子进程。画出由下面三个连续的 fork()系统调用产生的进程树：

```
fork(); //A
fork(); //B
fork(); //C
```

用合适的 fork()语句标出每个被创建的进程。

编程项目 1：开发一个 shell 程序

shell 或者命令行解释器是操作系统中最基本的用户接口。第一个项目是写一个简单的 shell 程序——**myshell**，它具有以下属性：

1．这个 shell 程序必须支持以下内部命令：

a）**cd <directory>**——把当前默认目录改变为**<directory>**。如果没有**<directory>**参数，则显示当前目录。如果该目录不存在，会出现合适的错误信息。这个命令也可以改变 **PWD** 环境变量。

b）**clr**——清屏。

c）**dir <directory>**——列出目录**<directory>**的内容。

d）**environ**——列出所有的环境变量。

e）**echo <comment>**——在屏幕上显示**<comment>**并换行（多个空格和制表符可能被缩减为一个空格）。

f）**help**——显示用户手册，并且使用 **more** 命令过滤。

g）**pause**——停止 shell 操作直到按下回车键。

h）**quit**——退出 shell。

i）shell 的环境变量应该包含 **shell=<pathname>/myshell**，其中**<pathname>/myshell** 是可执行程序 shell 的完整路径（不是你的目录下的硬连线路径，而是它执行的路径）。

2．其他的命令行输入被解释为程序调用，shell 创建并执行这个程序，并作为自己的子进程。程序的执行的环境变量包含一下条目：

parent=<pathname>/myshell，其中 **<pathname>/myshell** 已经在 1.ix.中讲述过。

3．shell 必须能够从文件中提取命令行输入，例如 shell 使用以下命令行被调用：

myshell batchfile

这个批处理文件应该包含一组命令集，当到达文件结尾时 shell 退出。很明显，如果 shell 被调用时没有使用参数，它会在屏幕上显示提示符请求用户输入。

4．shell 必须支持 i/o 重定向，stdin 和 stdout，或者其中之一，例如命令行为：

programname arg1 arg2<inputfile>outputfile

使用 **arg1** 和 **arg2** 执行程序 **programname**，输入文件流被替换为 **inputfile**，输出文件流被替换为 **outputfile**。

stdout 重定向应该支持以下内部命令：**dir**、**environ**、**echo**、& **help**。

使用输出重定向时，如果重定向字符是**>**，则创建输出文件，如果存在则覆盖之；如果重定向字符为**>>**，也会创建输出文件，如果存在则添加到文件尾。

5．shell 必须支持后台程序执行。如果在命令行后添加&字符，在加载完程序后需要立刻返回命令行提示符。

6．命令行提示符必须包含当前路径。

提示：你可以假定所有命令行参数（包括重定向字符**<**、**>**、**>>**和后台执行字符&）和其他命令行参数用空白空间分开，空白空间可以为一个或多个空格或制表符（见上面 4 中的命令行）。

项目要求

1．设计一个简单的命令行 shell，满足上面的要求并且在指定的 UNIX 平台上执行。

2．写一个关于如何使用 shell 的简单的用户手册，用户手册应该包含足够的细节以方便 UNIX 初学者使用。例如，你应该解释 I/O 重定向、程序环境和后台程序执行。用户手册必须命名为 **readme**，必须是一个

标准文本编辑器可以打开的简单文本文档。

举一个描述的类型和深度的例子，你应该看一下 **csh** 和 **tcsh** 的在线帮助（**man csh**，**man tcsh**）。这两个 shell 显然比你的 shell 有更多的功能，你的用户手册没必要做这么大。不要包括编译指示—文件列表或源码，我们可以从你提交的其他文件中找出来。这应该是操作员手册而非程序员手册。

3. 源码必须有很详细的注释，并且有很好的组织结构以方便别人阅读和维护。结构和注释好的程序更加易于理解，并且可以保证批改你作业的人不用很费劲地去读你的代码。
4. 在截止期之前，要提供很详细的提交过程。
5. 提交内容为源码文件，包括文件、**makefile**（全部小写）和 **readme**（全部小写）文件。批改作业者会重新编译源码，如果编译不通过将没办法打分。
6. **makefile**（全部小写）必须产生二进制文件 **myshell**（全部小写）。一个 **makefile** 的例子如下：

```
# Joe Citizen, s1234567 - Operating Systems Project 1
# CompLab1/01 tutor:Fred Bloggs
myshell:myshell.c utility.c myshell.h
gcc -Wall myshell.c utility.c -o myshell
```

在命令提示符下键入 **make** 就会产生 **myshell** 程序。

提示：上面 **makefile** 的第 4 行必须以制表符开始。

7. 根据上面提供的实例，提交的目录应该包含以下文件：

```
makefile
myshell.c
utility.c
myshell.h
readme
```

提交

需要 **makefile** 文件，所有提交的文件将会被复制到一个目录下，所以不要在 **makefile** 中包含路径。**makefile** 中应该包含编译程序所需的依赖关系，如果包含了库文件，**makefile** 也会编译这个库文件的。

为了清楚起见，再重复一次：不要提交二进制或者目标代码文件。所需的只是源码、**makefile** 和 **readme** 文件。提交之前测试一下，把源码复制到一个空目录下，键入 **make** 命令编译。

我们将使用一个 shell 脚本把你的文件复制到测试目录下，删除已经存在的 **myshell**、***.a** 或***.o** 文件，执行 **make** 命令，复制一组测试文件到测试目录下，然后用一个标准的测试脚本通过 stdin 和命令行参数测试你的 shell 程序。如果这个过程因为名字错误、名字大小写错误、源码版本错误导致不能编译和文件不存在错误等而停止的话，那么打分过程也会停止。在这种情况下，所得的分数将基于已经完成的测试部分，还有源码和用户手册。

所需的文档

首先，源码和用户手册都将被评估和打分，源码需要注释，用户手册可以是你自己选择的形式（但要能被简单文本编辑器打开）。其次，手册应该包含足够的细节以方便 UNIX 初学者使用，例如，你应该解释 I/O 重定向、程序环境和后台程序执行。用户手册必须命名为 **readme**（全部小写，没有 **.txt** 后缀）。