目 录

【程序员的自我修养】

第1部分 简介

温故而知新

1.1 从 Hello World 说起

毫无疑问，“Hello World”对于程序员来说肯定是如雷贯耳。就是这样一个简单的程序，带领了无数的人进入了程序的世界。简单的事物背后往往又蕴涵着复杂的机制，如果我们深入思考一个简单的“Hello World”程序，就会发现很多问题看似很简单，但实际上我们并没有一个非常清晰的思路；或者在我们脑海里有着模糊的印象，但真正到某些细节的时候可能又模糊不清了。比如对于C语言编写的Hello World程序：

```
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    return 0;
}
```

对于下面这些问题，你的脑子里能够马上反应出一个很清晰又很明确的答案吗？

- 程序为什么要被编译器编译了之后才可以运行？
- 编译器在把C语言程序转换成可以执行的机器码的过程中做了什么，怎么做的？
- 最后编译出来的可执行文件里面是什么？除了机器码还有什么？它们怎么存放的，怎么组织的？
- #include <stdio.h>是什么意思？把stdio.h包含进来意味着什么？C语言库又是什么？它怎么实现的？
- 不同的编译器（Microsoft VC、GCC）和不同的硬件平台（x86、SPARC、MIPS、ARM），以及不同的操作系统（Windows、Linux、UNIX、Solaris），最终编译出来的结果一样吗？为什么？
- Hello World程序是怎么运行起来的？操作系统是怎么装载它的？它从哪儿开始执行，到哪儿结束？main函数之前发生了什么？main函数结束以后又发生了什么？
- 如果没有操作系统，Hello World可以运行吗？如果要在一台没有操作系统的机器上运行Hello World需要什么？应该怎么实现？
- printf是怎么实现的？它为什么可以有不定数量的参数？为什么它能够在终端上输出字符串？
- Hello World程序在运行时，它在内存中是什么样子的？

对于上面的问题，如果你确信能够非常清楚地了解里面的各个细节，并且对其中的过程和机制都了如指掌，那么很遗憾，这本书不是为你准备的；如果你发现对其中一些问题并不

是很了解，甚至从来没有想到过一个 Hello World 还能引出这么多值得思考的问题，而你又想了解它们，那么恭喜你，这本书就是为你准备的。随着各个章节的逐步展开，我们会从最基本的编译、静态链接到操作系统如何装载程序、动态链接及运行库和标准库的实现，甚至一些操作系统的机制，力争深入浅出地将这些问题层层剥开，最终使得这些程序运行背后的机制形成一个非常清晰而流畅的脉络。

在开始进入庞大而又繁琐的系统软件之前，让我们先进行热身活动，那就是一起来回顾计算机系统的一些基本而又重要的概念。整个计算机系统回顾过程将分为两个部分，分别是硬件部分和软件部分。本书的主要目的不是介绍计算机系统结构，第 1 章的回顾只是巩固和总结计算机软硬件体系里面几个重要的概念，这些概念在我们后面的章节中将时时伴随着我们，失去了它们的支撑，后面的章节将会显得繁琐而又晦涩。如果你自认为这些基本概念很简单，那么你可以大概地浏览一遍几个知识点的标题，然后直接跳到第 2 章；反之，如果你觉得有些概念还不是很清楚，甚至从来没听说过这些概念，那么请你仔细阅读相关章节，相信这个过程对你阅读本书甚至对你深入了解计算机大有裨益。

1.2 万变不离其宗

计算机是个非常广泛的概念，大到占用数层楼的用于科学计算的超级计算机，小到手机上的嵌入式芯片都可以被称为计算机。虽然它们的外形、结构和性能都千差万别，但至少它们都有“计算”这个概念。在本书里面，我们将计算机的范围限定在最为流行、使用最广泛的 PC 机，更具体地讲是采用兼容 x86 指令集的 32 位 CPU 的个人计算机。原因很简单：因为笔者手上目前只有这种类型的计算机可供操作和实验，不过相信 90%以上的读者也是，所以在这一点上我们很快能达成共识。其实选择具体哪种平台并不是最关键的，虽然各种平台的软硬件差别很多，但是本质上它们的基本概念和工作原理都是一样的，只要我们能够掌握一种平台上的技术，那么其他的平台都是大同小异的，很轻松地可以举一反三。所以我们相信，只有你能够深刻地理解 x86 平台下的系统软件背后的机理，当有一天你需要在 MIPS 指令集的嵌入式平台上做开发，或者需要为 64 位的 Windows 或 Linux 开发应用程序的时候，你很快就能找到它们之间的相通之处。

撇开计算机硬件中纷繁复杂的各种设备、芯片及外围接口等，站在软件开发者的角度看，我们只须抓住硬件的几个关键部件。对于系统程序开发者来说，计算机多如牛毛的硬件设备中，有三个部件最为关键，它们分别是中央处理器 CPU、内存和 I/O 控制芯片，这三个部件几乎就是计算机的核心了；对于普通应用程序开发者来说，他们似乎除了要关心 CPU 以外，其他的硬件细节基本不用关心，对于一些高级平台的开发者来说（如 Java、.NET 或脚本语言开发者），连 CPU 都不需要关心，因为这些平台为它们提供了一个通用的抽象的计算机，

他们只要关心这个抽象的计算机就可以了。

早期的计算机没有很复杂的图形功能，CPU 的核心频率也不高，跟内存的频率一样，它们都是直接连接在同一个**总线**（Bus）上的。由于 I/O 设备诸如显示设备、键盘、软盘和磁盘等速度与 CPU 和内存相比还是慢很多，当时也没有复杂的图形设备，显示设备大多是只能输出字符的终端。为了协调 I/O 设备与总线之间的速度，也为了能够让 CPU 能够和 I/O 设备进行通信，一般每个设备都会有一个相应的 I/O 控制器。早期的计算机硬件结构如图 1-1 所示。

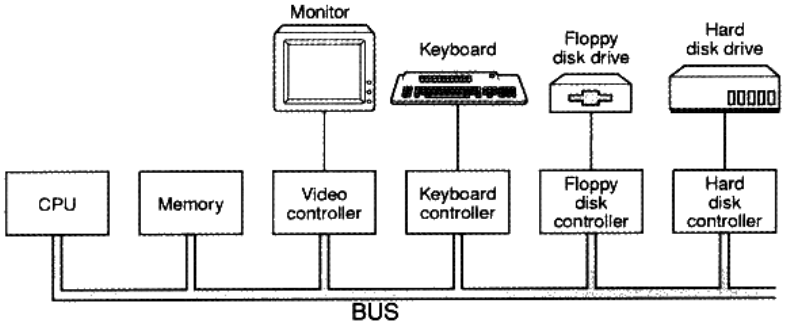

图 1-1 早期的计算机硬件结构

后来由于 CPU 核心频率的提升，导致内存跟不上 CPU 的速度，于是产生了与内存频率一致的系统总线，而 CPU 采用倍频的方式与系统总线进行通信。接着随着图形化的操作系统普及，特别是 3D 游戏和多媒体的发展，使得图形芯片需要跟 CPU 和内存之间大量交换数据，慢速的 I/O 总线已经无法满足图形设备的巨大需求。为了协调 CPU、内存和高速的图形设备，人们专门设计了一个高速的北桥芯片，以便它们之间能够高速地交换数据。

由于北桥运行的速度非常高，所有相对低速的设备如果全都直接连接在北桥上，北桥既须处理高速设备，又须处理低速设备，设计就会十分复杂。于是人们又设计了专门处理低速设备的**南桥**（Southbridge）芯片，磁盘、USB、键盘、鼠标等设备都连接在南桥上，由南桥将它们汇总后连接到北桥上。20 世纪 90 年代的 PC 机在系统总线上采用的是 PCI 结构，而在低速设备上采用的 ISA 总线，采用 PCI/ISA 及南北桥设计的硬件构架如图 1-2 所示。

位于中间是连接所有高速芯片的**北桥**（Northbridge，PCI Bridge），它就像人的心脏，连接并驱动身体的各个部位；它的左边是 CPU，负责所有的控制和运算，就像人的大脑。北桥还连接着几个高速部件，包括左边的内存和下面的 PCI 总线。

PCI 的速度最高为 133 MHz，它还是不能满足人们的需求，于是人们又发明了 AGP、

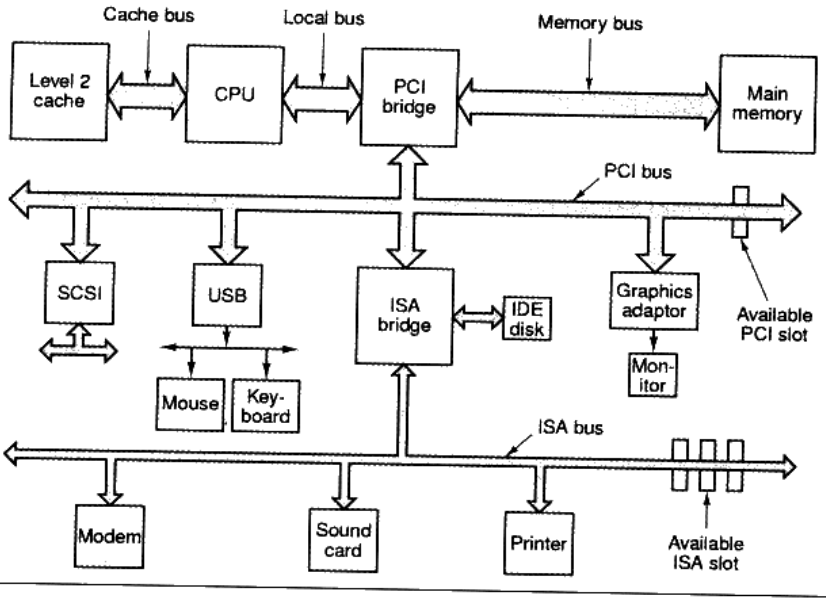

图 1-2 硬件结构框架

PCI Express 等诸多总线结构和相应控制芯片。虽然硬件结构看似越来越复杂，但实际上它还是没有脱离最初的 CPU、内存，以及 I/O 的基本结构。我们从程序开发的角度看待硬件时可以简单地将它看成最初的硬件模型。

SMP 与多核

人们总是希望计算机越来越快，这是毫无疑问的。在过去的 50 年里，CPU 的频率从几十 KHz 到现在的 4GHz，整整提高了数十万倍，基本上每 18 个月频率就会翻倍。但是自 2004 年以来，这种规律似乎已经失效，CPU 的频率自从那时开始再也没有发生质的提高。原因是人们在制造 CPU 的工艺方面已经达到了物理极限，除非 CPU 制造工艺有本质的突破，否则 CPU 的频率将会一直被目前 4GHz 的“天花板”所限制。

在频率上短期内已经没有提高的余地了，于是人们开始想办法从另外一个角度来提高 CPU 的速度，就是增加 CPU 的数量。一个计算机拥有多个 CPU 早就不是什么新鲜事了，很早以前就有了多 CPU 的计算机，其中最常见的一种形式就是**对称多处理器（SMP，Symmetrical Multi-Processing）**，简单地讲就是每个 CPU 在系统中所处的地位和所发挥的功能都是一样的，是相互对称的。理论上讲，增加 CPU 的数量就可以提高运算速度，并且理想情况下，速度的提高与 CPU 的数量成正比。但实际上并非如此，因为我们的程序并不是都能分解成若干个完全不相干的子问题。就比如一个女人可以花 10 个月生出一个孩子，但是 10 个女人并不能在一个月就生出一个孩子一样。

当然很多时候多处理器是非常有用的，最常见的情况就是在大型的数据库、网络服务器上，它们要同时处理大量的请求，而这些请求之间往往是相互独立的，所以多处理器就可以最大效能地发挥威力。

多处理器应用最多的场合也是这些商用的服务器和需要处理大量计算的环境。而在个人电脑中，使用多处理器则是比较奢侈的行为，毕竟多处理器的成本是很高的。于是处理器的厂商开始考虑将多个处理器“合并在一起打包出售”，这些“被打包”的处理器之间共享比较昂贵的缓存部件，只保留多个核心，并且以一个处理器的外包装进行出售，售价比单核心的处理器只贵了一点，这就是多核处理器（Multi-core Processor）的基本想法。多核处理器实际上就是 SMP 的简化版，当然它们在细节上还有一些差别，但是从程序员的角度来看，它们之间区别很小，逻辑上来看它们是完全相同的。只是多核和 SMP 在缓存共享等方面有细微的差别，使得程序在优化上可以有针对性地处理。简单地讲，除非想把 CPU 的每一滴油水都榨干，否则可以把多核和 SMP 看成同一个概念。

推荐阅读：“Free Lunch is Over”（免费午餐已经结束了）

http://www.gotw.ca/publications/concurrency-ddj.htm

随着 CPU 频率碰到了“天花板”，多核处理器越来越普及，对程序员开发程序的方式也将发生极大的变化，这篇文章很好地分析了将要到来的多核时代对程序开发的挑战和机遇。

1.3 站得高，望得远

系统软件这个概念其实比较模糊，传统意义上一般将用于管理计算机本身的软件称为系统软件，以区别普通的应用程序。系统软件可以分成两块，一块是平台性的，比如操作系统内核、驱动程序、运行库和数以千计的系统工具；另外一块是用于程序开发的，比如编译器、汇编器、链接器等开发工具和开发库。本书将着重介绍系统软件的一部分，主要是链接器和库（包括运行库和开发库）的相关内容。

计算机系统软件体系结构采用一种层的结构，有人说过一句名言：

“计算机科学领域的任何问题都可以通过增加一个间接的中间层来解决”[1]

“Any problem in computer science can be solved by another layer of indirection.”

这句话几乎概括了计算机系统软件体系结构的设计要点，整个体系结构从上到下都是按照严格的层次结构设计的。不仅是计算机系统软件整个体系是这样的，体系里面的每个组件

[1] 遗憾的是，这句经典的名言出处无从考证，据说是有人从图灵奖的获得者 Butler Lampson 的讲座上听来的；也有人说是 EDSAC 的发明者 David Wheeler 讲的；还有人指出这是 CMU 计算机系创始人 Alan Perlis 的名言。

比如操作系统本身，很多应用程序、软件系统甚至很多硬件结构都是按照这种层次的结构组织和设计的。系统软件体系结构中，各种软件的位置如图 1-3 所示。

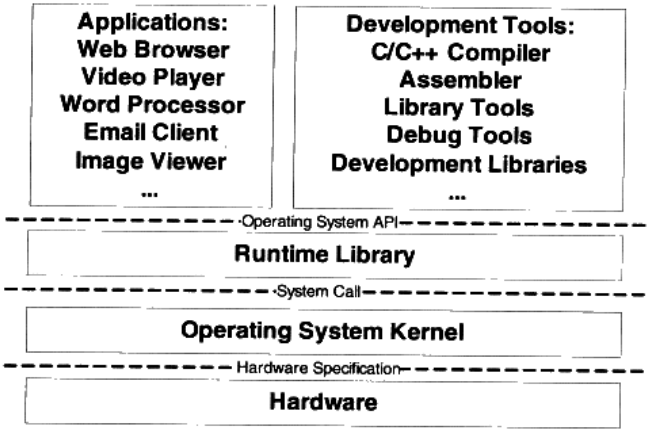

图 1-3 计算机软件体系结构

每个层次之间都须要相互通信，既然须要通信就必须有一个通信的协议，我们一般将其称为**接口**（Interface），接口的下面那层是接口的提供者，由它定义接口；接口的上面那层是接口的使用者，它使用该接口来实现所需要的功能。在层次体系中，接口是被精心设计过的，尽量保持稳定不变，那么理论上层次之间只要遵循这个接口，任何一个层都可以被修改或被替换。除了硬件和应用程序，其他都是所谓的中间层，每个中间层都是对它下面的那层的包装和扩展。正是这些中间层的存在，使得应用程序和硬件之间保持相对的独立，比如硬件和操作系统都日新月异地发展，但是最初为 80386 芯片和 DOS 系统设计的软件在最新的多核处理器和 Windows Vista 下还是能够运行的，这方面归功于硬件和操作系统本身保持了向后兼容性，另一方面不得不归功于这种层次结构的设计方式。最近开始流行的虚拟机技术更是在硬件和操作系统之间增加了一层虚拟层，使得一个计算机上可以同时运行多个操作系统，这也是层次结构带来的好处，在尽可能少改变甚至不改变其他层的情况下，新增加一个层次就可以提供前所未有的功能。

我们的软件体系中，位于最上层的是应用程序，比如我们平时用到的网络浏览器、Email 客户端、多媒体播放器、图片浏览器等。从整个层次结构上来看，开发工具与应用程序是属于同一个层次的，因为它们都使用一个接口，那就是操作系统**应用程序编程接口**（Application Programming Interface）。应用程序接口的提供者是运行库，什么样的运行库提供什么样的 API，比如 Linux 下的 Glibc 库提供 POSIX 的 API；Windows 的运行库提供 Windows API，最常见的 32 位 Windows 提供的 API 又被称为 Win32。

运行库使用操作系统提供的**系统调用接口**（System call Interface），系统调用接口在实

现中往往以软件中断（Software Interrupt）的方式提供，比如 Linux 使用 0x80 号中断作为系统调用接口，Windows 使用 0x2E 号中断作为系统调用接口（从 Windows XP Sp2 开始，Windows 开始采用一种新的系统调用方式）。

操作系统内核层对于硬件层来说是硬件接口的使用者，而硬件是接口的定义者，硬件的接口定义决定了操作系统内核，具体来讲就是驱动程序如何操作硬件，如何与硬件进行通信。这种接口往往被叫做硬件规格（Hardware Specification），硬件的生产厂商负责提供硬件规格，操作系统和驱动程序的开发者通过阅读硬件规格文档所规定的各种硬件编程接口标准来编写操作系统和驱动程序。

1.4 操作系统做什么

操作系统的一个功能是提供抽象的接口，另外一个主要功能是管理硬件资源。

计算机硬件的能力是有限的，比如一个 CPU 一秒钟能够执行的指令条数是 1 亿条或是 1GB 的内存能够最多同时存储 1GB 的数据。无论你是否使用它，资源总是那么多。当然我们不希望自己花钱买回来的硬件成为摆设，充分挖掘硬件的能力，使得计算机运行得更有效率，在更短的时间内处理更多的任务，才是我们的目标。这对于早期动辄数百万美元的古董计算机来说更是如此，人们挖空心思让计算机硬件发挥所有潜能。一个计算机中的资源主要分 CPU、存储器（包括内存和磁盘）和 I/O 设备，我们分别从这三个方面来看看如何挖掘它们的潜力。

1.4.1 不要让 CPU 打盹

在计算机发展早期，CPU 资源十分昂贵，如果一个 CPU 只能运行一个程序，那么当程序读写磁盘（当时可能是磁带）时，CPU 就空闲下来了，这在当时简直就是暴殄天物。于是人们很快编写了一个监控程序，当某个程序暂时无须使用 CPU 时，监控程序就把另外的正在等待 CPU 资源的程序启动，使得 CPU 能够充分地利用起来。这种被称为多道程序（Multiprogramming）的方法看似很原始，但是它当时的确大大提高了 CPU 的利用率。不过这种原始的多道程序技术存在最大的问题是程序之间的调度策略太粗糙。对于多道程序来说，程序之间不分轻重缓急，如果有些程序急需使用 CPU 来完成一些任务（比如用户交互的任务），那么很有可能很长时间后才有机会分配到 CPU。这对于有些响应时间要求高的程序来说是很致命的，想象一下你在 Windows 上面点击鼠标 10 分钟以后系统才有反应，那该是多么沮丧的事。

经过稍微改进，程序运行模式变成了一种协作的模式，即每个程序运行一段时间以后都

主动让出 CPU 给其他程序，使得一段时间内每个程序都有机会运行一小段时间。这对于一些交互式的任务尤为重要，比如点击一下鼠标或按下一个键盘按键后，程序所要处理的任务可能并不多，但是它需要尽快地被处理，使得用户能够立即看到效果。这种程序协作模式叫做**分时系统（Time-Sharing System）**，这时候的监控程序已经比多道程序要复杂多了，完整的操作系统雏形已经逐渐形成了。Windows 的早期版本（Windows 95 和 Windows NT 之前），Mac OS X 之前的 Mac OS 版本都是采用这种分时系统的方式来调度程序的。比如在 Windows 3.1 中，程序调用 Yield、GetMessage 或 PeekMessage 这几个系统调用时，Windows 3.1 操作系统会判断是否有其他程序正在等待 CPU，如果有，则可能暂停执行当前的程序，把 CPU 让出来给其他程序。如果一个程序在进行一个很耗时的计算，一直霸占着 CPU 不放，那么操作系统也没办法，其他程序都只有等着，整个系统看过去好像死机了一样。比如一个程序进入了一个 while(1)的死循环，那么整个系统都停止了。

这在现在看来是很荒唐的事，系统中的任何一个程序死循环都会导致系统死机，这是无法令人接受的。当然当时的 PC 硬件处理能力本身就很弱，PC 上的应用也大多是比较低端的应用，所以这种分时方式勉强也能应付一下当时的交互式环境了。此前在高端领域，非 PC 的大中小型机领域，其实已经在研究一种更为先进的操作系统模式了。这种模式就是我们现在很熟悉的**多任务（Multi-tasking）系统**，操作系统接管了所有的硬件资源，并且本身运行在一个受硬件保护的级别。所有的应用程序都以**进程（Process）**的方式运行在比操作系统权限更低的级别，每个进程都有自己独立的地址空间，使得进程之间的地址空间相互隔离。CPU 由操作系统统一进行分配，每个进程根据进程优先级的高低都有机会得到 CPU，但是，如果运行时间超出了一定的时间，操作系统会暂停该进程，将 CPU 资源分配给其他等待运行的进程。这种 CPU 的分配方式即所谓的**抢占式（Preemptive）**，操作系统可以强制剥夺 CPU 资源并且分配给它认为目前最需要的进程。如果操作系统分配给每个进程的时间都很短，即 CPU 在多个进程间快速地切换，从而造成了很多进程都在同时运行的假象。目前几乎所有现代的操作系统都是采用这种方式，比如我们熟悉的 UNIX、Linux、Windows NT，以及 Mac OS X 等流行的操作系统。

1.4.2 设备驱动

操作系统作为硬件层的上层，它是对硬件的管理和抽象。对于操作系统上面的运行库和应用程序来说，它们希望看到的是一个统一的硬件访问模式。作为应用程序的开发者，我们不希望在开发应用程序的时候直接读写硬件端口、处理硬件中断等这些繁琐的事情。由于硬件之间千差万别，它们的操作方式和访问方式都有区别。比如我们希望在显示器上画一条直线，对于程序员来说，最好的方式是不管计算机使用什么显卡、什么显示器，多少大小多少分辨率，我们都只要调用一个统一的 LineTo()函数，具体的实现方式由操作系统来完成。试想一下如果程序员需要关心具体的硬件，那么结果会是这样：对于 A 型号的显卡来说，需

要往 I/O 端口 0x1001 写一个命令 0x1111，然后从端口 0x1002 中读取一个 4 字节的显存地址，然后使用 DDA（一种画直线的图形算法）逐个地在显存上画点……如果是 B 型号的显卡，可能完全是另外一种方式。这简直就是灾难。不过在操作系统成熟之前，的确存在这样的情况，就是应用程序的程序员需要直接跟硬件打交道。

当成熟的操作系统出现以后，硬件逐渐被抽象成了一系列概念。在 UNIX 中，硬件设备的访问形式跟访问普通的文件形式一样；在 Windows 系统中，图形硬件被抽象成了 GDI，声音和多媒体设备被抽象成了 DirectX 对象；磁盘被抽象成了普通文件系统，等等。程序员逐渐从硬件细节中解放出来，可以更多地关注应用程序本身的开发。这些繁琐的硬件细节全都交给了操作系统，具体地讲是操作系统中的**硬件驱动（Device Driver）程序**来完成。驱动程序可以看作是操作系统的一部分，它往往跟操作系统内核一起运行在特权级，但它又与操作系统内核之间有一定的独立性，使得驱动程序有比较好的灵活性。因为 PC 的硬件多如牛毛，操作系统开发者不可能为每个硬件开发一个驱动程序，这些驱动程序的开发工作通常由硬件生产厂商完成。操作系统开发者为硬件生产厂商提供了一系列接口和框架，凡是按照这个接口和框架开发的驱动程序都可以在该操作系统上使用。让我们以一个读取文件为例子来看看操作系统和驱动程序在这个过程中扮演了什么样的角色。

提到文件的读取，那么不得不提到文件系统这个操作系统中最为重要的组成部分之一。文件系统管理着磁盘中文件的存储方式，比如我们在 Linux 系统下有一个文件“/home/user/test.dat”，长度为 8 000 个字节。那么我们在创建这个文件的时候，Linux 的 ext3 文件系统有可能将这个文件按照这样的方式存储在磁盘中：文件的前 4 096 字节存储在磁盘的 1000 号扇区到 1007 号扇区，每个扇区 512 字节，8 个扇区刚好 4 096 字节；文件的第 4 097 个字节到第 8 000 字节共 3 904 个字节，存储在磁盘的 2000 号扇区到 2007 号扇区，8 个扇区也是 4 096 字节，只不过只存储了 3 904 个有效的字节，剩下的 192 个字节无效。如果把这个文件的存储方式看作是一个链状的结构，它的结构如图 1-4 所示。

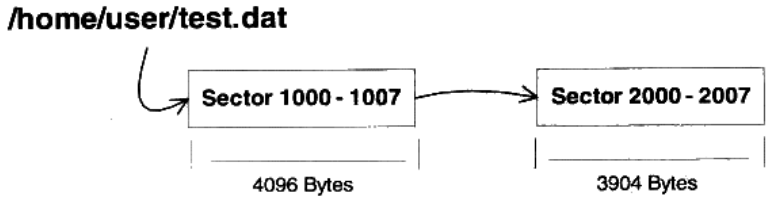

图 1-4 文件在磁盘中的结构

这里我们先穿插一个关于硬盘的结构介绍，关于硬盘结构可能很多读者已经有一个大概的了解，那就是硬盘基本存储单位为扇区（Sector），每个扇区一般为 512 字节。一

个硬盘往往有多个盘片，每个盘片分两面，每面按照同心圆划分为若干个磁道，每个磁道划分为若干个扇区。比如一个硬盘有 2 个盘片，每个盘面分 65 536 磁道，每个磁道分 1 024 个扇区，那么硬盘的容量就是 2 * 2 * 65 536 * 1 024 * 512 = 137 438 953 472 字节（128GB）。但是我们可以想象，每个盘面上同心圆的周长不一样，如果按照每个磁道都拥有相同数量的扇区，那么靠近盘面外围的磁道密度肯定比内圈更加稀疏，这样是比较浪费空间的。但是如果不同的磁道扇区数又不同，计算起来就十分麻烦。为了屏蔽这些复杂的硬件细节，现代的硬盘普遍使用一种叫做 LBA（Logical Block Address）的方式，即整个硬盘中所有的扇区从 0 开始编号，一直到最后一个扇区，这个扇区编号叫做**逻辑扇区号**。逻辑扇区号抛弃了所有复杂的磁道、盘面之类的概念。当我们给出一个逻辑的扇区号时，硬盘的电子设备会将其转换成实际的盘面、磁道等这些位置。

文件系统保存了这些文件的存储结构，负责维护这些数据结构并且保证磁盘中的扇区能够有效地组织和利用。那么当我们在 Linux 操作系统中，要读取这个文件的前 4 096 个字节时，我们会使用一个 read 的系统调用来实现。文件系统收到 read 请求之后，判断出文件的前 4 096 个字节位于磁盘的 1000 号逻辑扇区到 1007 号逻辑扇区。然后文件系统就向硬盘驱动发出一个读取逻辑扇区为 1000 号开始的 8 个扇区的请求，磁盘驱动程序收到这个请求以后就向硬盘发出硬件命令。向硬件发送 I/O 命令的方式有很多种，其中最为常见的一种就是通过读写 I/O 端口寄存器来实现。在 x86 平台上，共有 65 536 个硬件端口寄存器，不同的硬件被分配到了不同的 I/O 端口地址。CPU 提供了两条专门的指令“in”和“out”来实现对硬件端口的读和写。

对 IDE 接口来说，它有两个通道，分别为 IDE0 和 IDE1，每个通道上可以连接两个设备，分别为 Master 和 Slave，一个 PC 中最多可以有 4 个 IDE 设备。假设我们的文件位于 IDE0 的 Master 硬盘上，这也是正常情况下硬盘所在的位置。在 PC 中，IDE0 通道的 I/O 端口地址是 0x1F0～0x1F7 及 0x376～0x377。通过读写这些端口地址就能与 IDE 硬盘进行通信。这些端口的作用和操作方式十分复杂，我们以实现读取 1000 号逻辑扇区开始的 8 个扇区为例：

- 第 0x1F3～0x1F6 4 个字节的端口地址是用来写入 LBA 地址的，那么 1000 号逻辑扇区的 LBA 地址为 0x000003E8，所以我们需要往 0x1F3、0x1F4 写入 0x00，往 0x1F5 写入 0x03，往 0x1F6 写入 0xE8。
- 0x1F2 这个地址用来写入命令所需要读写的扇区数。比如读取 8 个扇区即写入 8。
- 0x1F7 这个地址用来写入要执行的操作的命令码，对于读取操作来说，命令字为 0x20。

所以我们要执行的指令为：

```
out 0x1F3, 0x00
out 0x1F4, 0x00
out 0x1F5, 0x03
out 0x1F6, 0xE8
```

```
out 0x1F2, 0x08
out 0x1F7, 0x20
```

在硬盘收到这个命令以后，它就会执行相应的操作，并且将数据读取到事先设置好的内存地址中（这个内存地址也是通过类似的命令方式设置的）。当然这里的例子中只是最简单的情况，实际情况比这个复杂得多，驱动程序须要考虑硬件的状态（是否忙碌或读取错误）、调度和分配各个请求以达到最高的性能等。

1.5 内存不够怎么办

上面一节中我们提到了进程的概念，进程的总体目标是希望每个进程从逻辑上来看都可以独占计算机的资源。操作系统的多任务功能使得 CPU 能够在多个进程之间很好地共享，从进程的角度看好像是它独占了 CPU 而不用考虑与其他进程分享 CPU 的事情。操作系统的 I/O 抽象模型也很好地实现了 I/O 设备的共享和抽象，那么唯一剩下的就是主存，也就是内存的分配问题了。

在早期的计算机中，程序是直接运行在物理内存上的，也就是说，程序在运行时所访问的地址都是物理地址。当然，如果一个计算机同时只运行一个程序，那么只要程序要求的内存空间不要超过物理内存的大小，就不会有问题。但事实上为了更有效地利用硬件资源，我们必须同时运行多个程序，正如前面的多道程序、分时系统和多任务中一样，当我们能够同时运行多个程序时，CPU 的利用率将会比较高。那么很明显的一个问题是，**如何将计算机上有限的物理内存分配给多个程序使用**。

假设我们的计算机有 128 MB 内存，程序 A 运行需要 10 MB，程序 B 需要 100 MB，程序 C 需要 20 MB。如果我们需要同时运行程序 A 和 B，那么比较直接的做法是将内存的前 10 MB 分配给程序 A，10 MB～110 MB 分配给 B。这样就能够实现 A 和 B 两个程序同时运行，但是这种简单的内存分配策略问题很多。

- **地址空间不隔离** 所有程序都直接访问物理地址，程序所使用的内存空间不是相互隔离的。恶意的程序可以很容易改写其他程序的内存数据，以达到破坏的目的；有些非恶意的、但是有臭虫的程序可能不小心修改了其他程序的数据，就会使其他程序也崩溃，这对于需要安全稳定的计算环境的用户来说是不能容忍的。用户希望他在使用计算机的时候，其中一个任务失败了，至少不会影响其他任务。
- **内存使用效率低** 由于没有有效的内存管理机制，通常需要一个程序执行时，监控程序就将整个程序装入内存中然后开始执行。如果我们忽然需要运行程序 C，那么这时内存空间其实已经不够了，这时候我们可以用的一个办法是将其他程序的数据暂时写到磁盘里面，等到需要用到的时候再读回来。由于程序所需要的空间是连续的，那么这个例子里面，如果我们将程序 A 换出到磁盘所释放的内存空间是不够的，所以只能

将 B 换出到磁盘，然后将 C 读入到内存开始运行。可以看到整个过程中有大量的数据在换入换出，导致效率十分低下。

- **程序运行的地址不确定** 因为程序每次需要装入运行时，我们都需要给它从内存中分配一块足够大的空闲区域，这个空闲区域的位置是不确定的。这给程序的编写造成了一定的麻烦，因为程序在编写时，它访问数据和指令跳转时的目标地址很多都是固定的，这涉及程序的**重定位**问题，我们在第 2 部分和第 3 部分还会详细探讨重定位的问题。

解决这几个问题的思路就是使用我们前文提到过的法宝：增加中间层，即使用一种间接的地址访问方法。整个想法是这样的，我们把程序给出的地址看作是一种**虚拟地址**（Virtual Address），然后通过某些映射的方法，将这个虚拟地址转换成实际的物理地址。这样，只要我们能够妥善地控制这个虚拟地址到物理地址的映射过程，就可以保证任意一个程序所能够访问的物理内存区域跟另外一个程序相互不重叠，以达到地址空间隔离的效果。

1.5.1 关于隔离

让我们回到程序的运行本质上来。用户程序在运行时不希望介入到这些复杂的存储器管理过程中，作为普通的程序，它需要的是一个简单的执行环境，有一个单一的地址空间、有自己的 CPU，好像整个程序占有整个计算机而不用关心其他的程序（当然程序间通信的部分除外，因为这是程序主动要求跟其他程序通信和联系）。所谓的地址空间是个比较抽象的概念，你可以把它想象成一个很大的数组，每个数组的元素是一个字节，而这个数组大小由地址空间的地址长度决定，比如 32 位的地址空间的大小为 2^32 = 4 294 967 296 字节，即 4GB，地址空间有效的地址是 0～4 294 967 295，用十六进制表示就是 0x00000000～0xFFFFFFFF。地址空间分两种：虚拟地址空间（Virtual Address Space）和物理地址空间（Physical Address Space）。物理地址空间是实实在在存在的，存在于计算机中，而且对于每一台计算机来说只有唯一的一个，你可以把物理空间想象成物理内存，比如你的计算机用的是 Intel 的 Pentium 4 的处理器，那么它是 32 位的机器，即计算机地址线有 32 条（实际上是 36 条地址线，不过我们暂时认为它只是 32 条），那么物理空间就有 4GB。但是你的计算机上只装了 512MB 的内存，那么其实物理地址的真正有效部分只有 0x00000000～0x1FFFFFFF，其他部分都是无效的（实际上还有一些外部 I/O 设备映射到物理空间的，也是有效的，但是我们暂时无视其存在）。虚拟地址空间是指虚拟的、人们想象出来的地址空间，其实它并不存在，每个进程都有自己独立的虚拟空间，而且每个进程只能访问自己的地址空间，这样就有效地做到了进程的隔离。

1.5.2 分段（Segmentation）

最开始人们使用的是一种叫做**分段**（Segmentation）的方法，基本思路是把一段与程

序所需要的内存空间大小的虚拟空间映射到某个地址空间。比如程序 A 需要 10 MB 内存，那么我们假设有一个地址从 0x00000000 到 0x00A00000 的 10MB 大小的一个假象的空间，也就是虚拟空间，然后我们从实际的物理内存中分配一个相同大小的物理地址，假设是物理地址 0x00100000 开始到 0x00B00000 结束的一块空间。然后我们把这两块相同大小的地址空间一一映射，即虚拟空间中的每个字节相对应于物理空间中的每个字节。这个映射过程由软件来设置，比如操作系统来设置这个映射函数，实际的地址转换由硬件完成。比如当程序 A 中访问地址 0x00001000 时，CPU 会将这个地址转换成实际的物理地址 0x00101000。那么比如程序 A 和程序 B 在运行时，它们的虚拟空间和物理空间映射关系可能如图 1-5 所示。

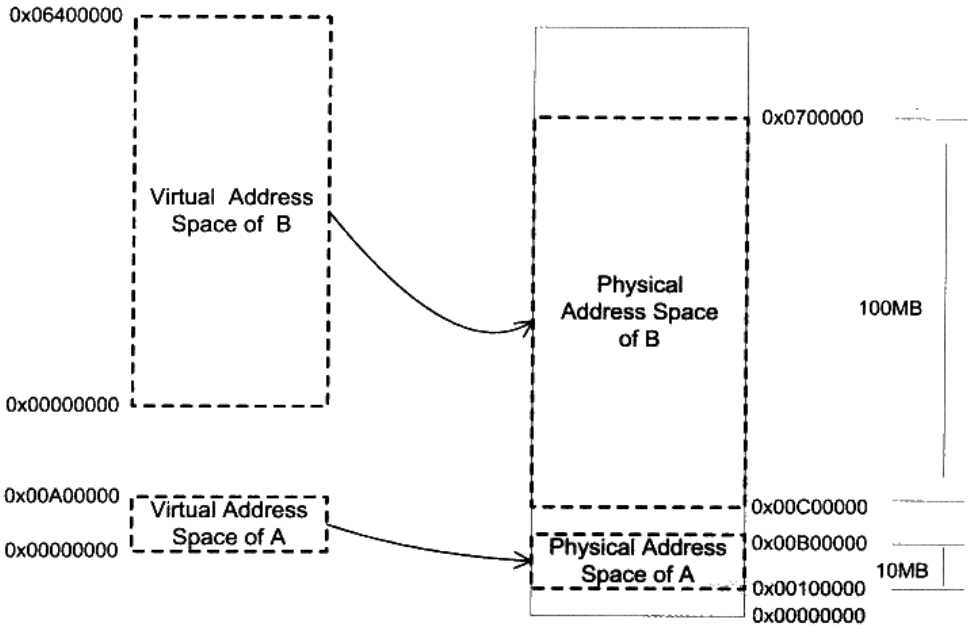

图 1-5 段映射机制

分段的方法基本解决了上面提到的 3 个问题中的第一个和第三个。首先它做到了地址隔离，因为程序 A 和程序 B 被映射到了两块不同的物理空间区域，它们之间没有任何重叠，如果程序 A 访问虚拟空间的地址超出了 0x00A00000 这个范围，那么硬件就会判断这是一个非法的访问，拒绝这个地址请求，并将这个请求报告给操作系统或监控程序，由它来决定如何处理。再者，对于每个程序来说，无论它们被分配到物理地址的哪一个区域，对于程序来说都是透明的，它们不需要关心物理地址的变化，它们只需要按照从地址 0x00000000 到 0x00A00000 来编写程序、放置变量，所以程序不再需要重定位。

但是分段的这种方法还是没有解决我们的第二个问题，即内存使用效率的问题。分段对内存区域的映射还是按照程序为单位，如果内存不足，被换入换出到磁盘的都是整个程序，这样势必会造成大量的磁盘访问操作，从而严重影响速度，这种方法还是显得粗糙，粒度比较大。事实上，根据程序的局部性原理，当一个程序在运行时，在某个时间段内，它只是频繁地用到了一小部分数据，也就是说，程序的很多数据其实在一个时间段内都是不会被用到的。人们很自然地想到了更小粒度的内存分割和映射的方法，使得程序的局部性原理得到充分的利用，大大提高了内存的使用率。这种方法就是**分页**（Paging）。

1.5.3 分页（Paging）

分页的基本方法是把地址空间人为地等分成固定大小的页，每一页的大小由硬件决定，或硬件支持多种大小的页，由操作系统选择决定页的大小。比如 Intel Pentium 系列处理器支持 4KB 或 4MB 的页大小，那么操作系统可以选择每页大小为 4KB，也可以选择每页大小为 4MB，但是在同一时刻只能选择一种大小，所以对整个系统来说，页就是固定大小的。目前几乎所有的 PC 上的操作系统都使用 4KB 大小的页。我们使用的 PC 机是 32 位的虚拟地址空间，也就是 4GB，那么按 4KB 每页分的话，总共有 1 048 576 个页。物理空间也是同样的分法。

下面我们来看一个简单的例子，如图 1-6 所示，每个虚拟空间有 8 页，每页大小为 1KB，那么虚拟地址空间就是 8KB。我们假设该计算机有 13 条地址线，即拥有 2^13 的物理寻址能力，那么理论上物理空间可以多达 8KB。但是出于种种原因，购买内存的资金不够，只买得起 6KB 的内存，所以物理空间其实真正有效的只是前 6KB。

那么，当我们把进程的虚拟地址空间按页分割，把常用的数据和代码页装载到内存中，把不常用的代码和数据保存在磁盘里，当需要用到的时候再把它从磁盘里取出来即可。以图 1-6 为例，我们假设有两个进程 Process1 和 Process2，它们进程中的部分虚拟页面被映射到了物理页面，比如 VP0、VP1 和 VP7 映射到 PP0、PP2 和 PP3；而有部分页面却在磁盘中，比如 VP2 和 VP3 位于磁盘的 DP0 和 DP1 中；另外还有一些页面如 VP4、VP5 和 VP6 可能尚未被用到或访问到，它们暂时处于未使用的状态。在这里，我们把虚拟空间的页就叫**虚拟页**（VP，Virtual Page），把物理内存中的页叫做**物理页**（PP，Physical Page），把磁盘中的页叫做**磁盘页**（DP，Disk Page）。图中的线表示映射关系，我们可以看到虚拟空间的有些页被映射到同一个物理页，这样就可以实现内存共享。

图 1-6 中 Process1 的 VP2 和 VP3 不在内存中，但是当进程需要用到这两个页的时候，硬件会捕获到这个消息，就是所谓的**页错误**（Page Fault），然后操作系统接管进程，负责将 VP2 和 VP3 从磁盘中读出来并且装入内存，然后将内存中的这两个页与 VP2 和 VP3 之间建立映射关系。以页为单位来存取和交换这些数据非常方便，硬件本身就支持这种以页为单位的操作方式。

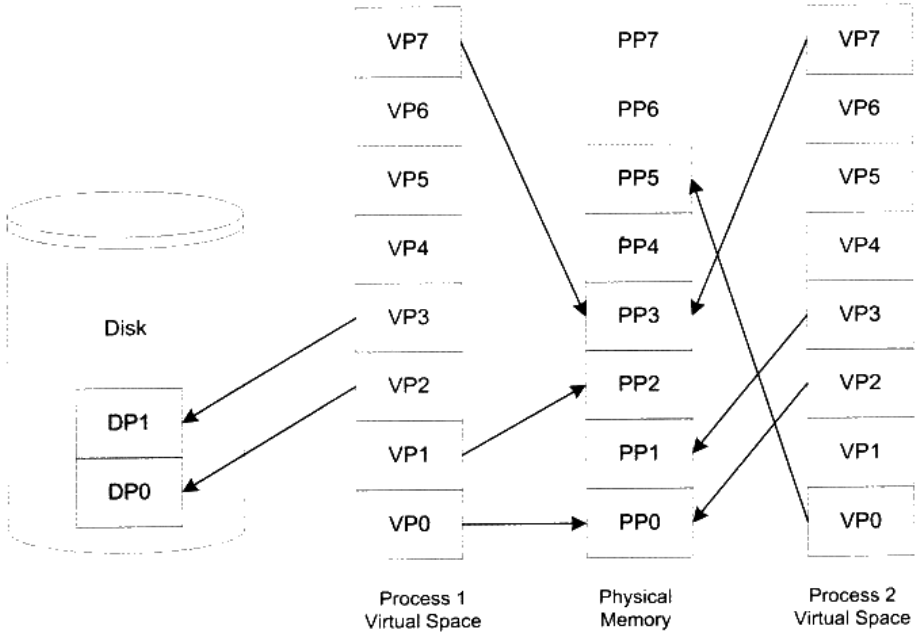

图 1-6 进程虚拟空间、物理空间和磁盘之间的页映射关系

保护也是页映射的目的之一，简单地说就是每个页可以设置权限属性，谁可以修改，谁可以访问等，而只有操作系统有权限修改这些属性，那么操作系统就可以做到保护自己和保护进程。对于保护，我们这里只是简单介绍，详细的介绍和为什么要保护我们将会在本书的第 2 部分再介绍。

虚拟存储的实现需要依靠硬件的支持，对于不同的 CPU 来说是不同的。但是几乎所有的硬件都采用一个叫 MMU（Memory Management Unit）的部件来进行页映射，如图 1-7 所示。

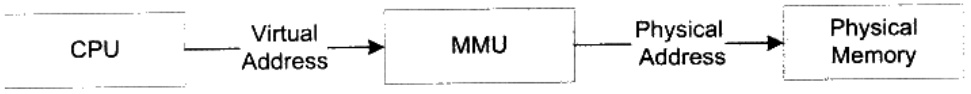

图 1-7 虚拟地址到物理地址的转换

在页映射模式下，CPU 发出的是 Virtual Address，即我们的程序看到的是虚拟地址。经过 MMU 转换以后就变成了 Physical Address。一般 MMU 都集成在 CPU 内部了，不会以独立的部件存在。

1.6 众人拾柴火焰高

1.6.1 线程基础

现代软件系统中，除了进程之外，线程也是一个十分重要的概念。特别是随着 CPU 频率增长开始出现停滞，而开始向多核方向发展。多线程，作为实现软件并发执行的一个重要的方法，也开始具有越来越重要的地位。我们将在这一节回顾线程相关的内容，包括线程的概念、线程的调度、线程安全、用户线程与内核线程之间的映射关系。虽然线程相关的概念与本书的内容并不是十分相关，但是我们相信深刻地理解线程对于更加深入地理解装载、动态链接和运行库，特别是运行库与多线程相关部分的内容会有很大的帮助。

什么是线程

线程（Thread），有时被称为**轻量级进程**（Lightweight Process, LWP），是程序执行流的最小单元。一个标准的线程由线程 ID、当前指令指针（PC）、寄存器集合和堆栈组成。通常意义上，一个进程由一个到多个线程组成，各个线程之间共享程序的内存空间（包括代码段、数据段、堆等）及一些进程级的资源（如打开文件和信号）。一个经典的线程与进程的关系如图 1-8 所示。

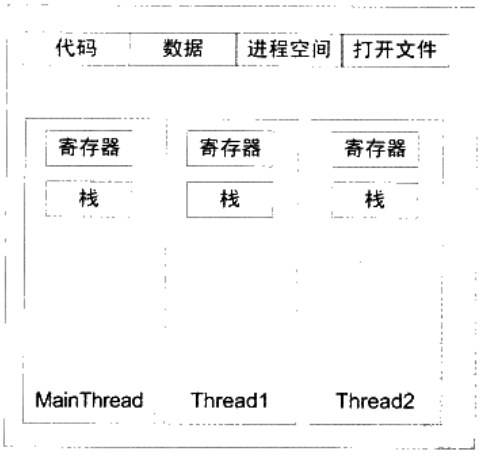

图 1-8 进程内的线程

大多数软件应用中，线程的数量都不止一个。多个线程可以互不干扰地并发执行，并共享进程的全局变量和堆的数据。那么，多个线程与单线程的进程相比，又有哪些优势呢？通常来说，使用多线程的原因有如下几点。

- 某个操作可能会陷入长时间等待，等待的线程会进入睡眠状态，无法继续执行。多线程执行可以有效利用等待的时间。典型的例子是等待网络响应，这可能要花费数秒甚至数十秒。
- 某个操作（常常是计算）会消耗大量的时间，如果只有一个线程，程序和用户之间的交互会中断。多线程可以让一个线程负责交互，另一个线程负责计算。
- 程序逻辑本身就要求并发操作，例如一个多端下载软件（例如 Bittorrent）。
- 多 CPU 或多核计算机（基本就是未来的主流计算机），本身具备同时执行多个线程的能力，因此单线程程序无法全面地发挥计算机的全部计算能力。
- 相对于多进程应用，多线程在数据共享方面效率要高很多。

线程的访问权限

线程的访问非常自由，它可以访问进程内存里的所有数据，甚至包括其他线程的堆栈（如果它知道其他线程的堆栈地址，那么这就是很少见的情况），但实际运用中线程也拥有自己的私有存储空间，包括以下几方面。

- 栈（尽管并非完全无法被其他线程访问，但一般情况下仍然可以认为是私有的数据）。
- 线程局部存储（Thread Local Storage, TLS）。线程局部存储是某些操作系统为线程单独提供的私有空间，但通常只具有很有限的容量。
- 寄存器（包括 PC 寄存器），寄存器是执行流的基本数据，因此为线程私有。

从 C 程序员的角度来看，数据在线程之间是否私有如表 1-1 所示。

表 1-1

线程私有	线程之间共享（进程所有）
● 局部变量 ● 函数的参数 ● TLS 数据	● 全局变量 ● 堆上的数据 ● 函数里的静态变量 ● 程序代码，任何线程都有权利读取并执行任何代码 ● 打开的文件，A 线程打开的文件可以由 B 线程读写

线程调度与优先级

不论是在多处理器的计算机上还是在单处理器的计算机上，线程总是“并发”执行的。当线程数量小于等于处理器数量时（并且操作系统支持多处理器），线程的并发是真正的并发，不同的线程运行在不同的处理器上，彼此之间互不相干。但对于线程数量大于处理器数量的情况，线程的并发会受到一些阻碍，因为此时至少有一个处理器会运行多个线程。

在单处理器对应多线程的情况下，并发是一种模拟出来的状态。操作系统会让这些多线程程序轮流执行，每次仅执行一小段时间（通常是几十到几百毫秒），这样每个线程就“看起来”在同时执行。这样的一个不断在处理器上切换不同的线程的行为称之为**线程调度**（Thread Schedule）。在线程调度中，线程通常拥有至少三种状态，分别是：

- **运行**（Running）：此时线程正在执行。
- **就绪**（Ready）：此时线程可以立刻运行，但 CPU 已经被占用。
- **等待**（Waiting）：此时线程正在等待某一事件（通常是 I/O 或同步）发生，无法执行。

处于运行中线程拥有一段可以执行的时间，这段时间称为**时间片**（Time Slice），当时间片用尽的时候，该进程将进入就绪状态。如果在时间片用尽之前进程就开始等待某事件，那么它将进入等待状态。每当一个线程离开运行状态时，调度系统就会选择一个其他的就绪线程继续执行。在一个处于等待状态的线程所等待的事件发生之后，该线程将进入就绪状态。这 3 个状态的转移如图 1-9 所示。

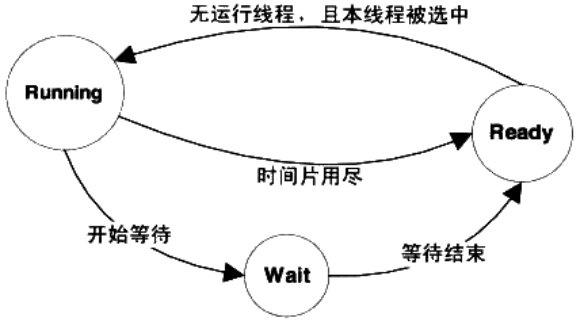

图 1-9 线程状态切换

线程调度自多任务操作系统问世以来就不断地被提出不同的方案和算法。现在主流的调度方式尽管各不相同，但都带有**优先级调度**（Priority Schedule）和**轮转法**（Round Robin）的痕迹。所谓轮转法，即是之前提到的让各个线程轮流执行一小段时间的方法。这决定了线程之间交错执行的特点。而优先级调度则决定了线程按照什么顺序轮流执行。在具有优先级调度的系统中，线程都拥有各自的**线程优先级**（Thread Priority）。具有高优先级的线程会更早地执行，而低优先级的线程常常要等待到系统中已经没有高优先级的可执行的线程存在时才能够执行。在 Windows 中，可以通过使用：

```
BOOL WINAPI SetThreadPriority(HANDLE hThread, int nPriority);
```

来设置线程的优先级，而 Linux 下与线程相关的操作可以通过 pthread 库来实现。

在 Windows 和 Linux 中，线程的优先级不仅可以由用户手动设置，系统还会根据不同

线程的表现自动调整优先级，以使得调度更有效率。例如通常情况下，频繁地进入等待状态（进入等待状态，会放弃之后仍然可占用的时间份额）的线程（例如处理 I/O 的线程）比频繁进行大量计算、以至于每次都要把时间片全部用尽的线程要受欢迎得多。其实道理很简单，频繁等待的线程通常只占用很少的时间，CPU 也喜欢先捏软柿子。我们一般把频繁等待的线程称之为 **IO 密集型线程**（IO Bound Thread），而把很少等待的线程称为 **CPU 密集型线程**（CPU Bound Thread）。IO 密集型线程总是比 CPU 密集型线程容易得到优先级的提升。

在优先级调度下，存在一种**饿死**（Starvation）的现象，一个线程被饿死，是说它的优先级较低，在它执行之前，总是有较高优先级的线程试图执行，因此这个低优先级线程始终无法执行。当一个 CPU 密集型的线程获得较高的优先级时，许多低优先级的进程就很可能饿死。而一个高优先级的 IO 密集型线程由于大部分时间都处于等待状态，因此相对不容易造成其他线程饿死。为了避免饿死现象，调度系统常常会逐步提升那些等待了过长时间的得不到执行的线程的优先级。在这样的手段下，一个线程只要等待足够长的时间，其优先级一定会提高到足够让它执行的程度。

让我们总结一下，在优先级调度的环境下，线程的优先级改变一般有三种方式。

- 用户指定优先级。
- 根据进入等待状态的频繁程度提升或降低优先级。
- 长时间得不到执行而被提升优先级。

可抢占线程和不可抢占线程

我们之前讨论的线程调度有一个特点，那就是线程在用尽时间片之后会被强制剥夺继续执行的权利，而进入就绪状态，这个过程叫做**抢占**（Preemption），即之后执行的别的线程抢占了当前线程。在早期的一些系统（例如 Windows 3.1）里，线程是不可抢占的。线程必须手动发出一个放弃执行的命令，才能让其他的线程得到执行。在这样的调度模型下，线程必须主动进入就绪状态，而不是靠时间片用尽来被强制进入。如果线程始终拒绝进入就绪状态，并且也不进行任何的等待操作，那么其他的线程将永远无法执行。在不可抢占线程中，线程主动放弃执行无非两种情况。

- 当线程试图等待某事件时（I/O 等）。
- 线程主动放弃时间片。

因此，在不可抢占线程执行的时候，有一个显著的特点，那就是线程调度的时机是确定的，线程调度只会发生在线程主动放弃执行或线程等待某事件的时候。这样可以避免一些因为抢占式线程里调度时机不确定而产生的问题（见下一节：线程安全）。但即使如此，非抢占式线程在今日已经十分少见。

Linux 的多线程

Windows 对进程和线程的实现如同教科书一般标准，Windows 内核有明确的线程和进程的概念。在 Windows API 中，可以使用明确的 API：CreateProcess 和 CreateThread 来创建进程和线程，并且有一系列的 API 来操纵它们。但对于 Linux 来说，线程并不是一个通用的概念。

Linux 对多线程的支持颇为贫乏，事实上，在 Linux 内核中并不存在真正意义上的线程概念。Linux 将所有的执行实体（无论是线程还是进程）都称为**任务（Task）**，每一个任务概念上都类似于一个单线程的进程，具有内存空间、执行实体、文件资源等。不过，Linux 下不同的任务之间可以选择共享内存空间，因而在实际意义上，共享了同一个内存空间的多个任务构成了一个进程，这些任务也就成了这个进程里的线程。在 Linux 下，用以下方法可以创建一个新的任务，如表 1-2 所示。

表 1-2

系统调用	作　用
fork	复制当前进程
exec	使用新的可执行映像覆盖当前可执行映像
clone	创建子进程并从指定位置开始执行

fork 函数产生一个和当前进程完全一样的新进程，并和当前进程一样从 fork 函数里返回。例如如下代码：

```
pid_t pid;
if (pid = fork())
{
    ....
}
```

在 fork 函数调用之后，新的任务将启动并和本任务一起从 fork 函数返回。但不同的是本任务的 fork 将返回新任务 pid，而新任务的 fork 将返回 0。

fork 产生新任务的速度非常快，因为 fork 并不复制原任务的内存空间，而是和原任务一起共享一个**写时复制（Copy on Write, COW）**的内存空间（见图 1-10）。所谓写时复制，指的是两个任务可以同时自由地读取内存，但任意一个任务试图对内存进行修改时，内存就会复制一份提供给修改方单独使用，以免影响到其他的任务使用。

fork 只能够产生本任务的镜像，因此须要使用 exec 配合才能够启动别的新任务。exec 可以用新的可执行映像替换当前的可执行映像，因此在 fork 产生了一个新任务之后，新任务可以调用 exec 来执行新的可执行文件。fork 和 exec 通常用于产生新任务，而如果要产生新线程，则可以使用 clone。clone 函数的原型如下：

```
int clone(int (*fn)(void*), void* child_stack, int flags, void* arg);
```

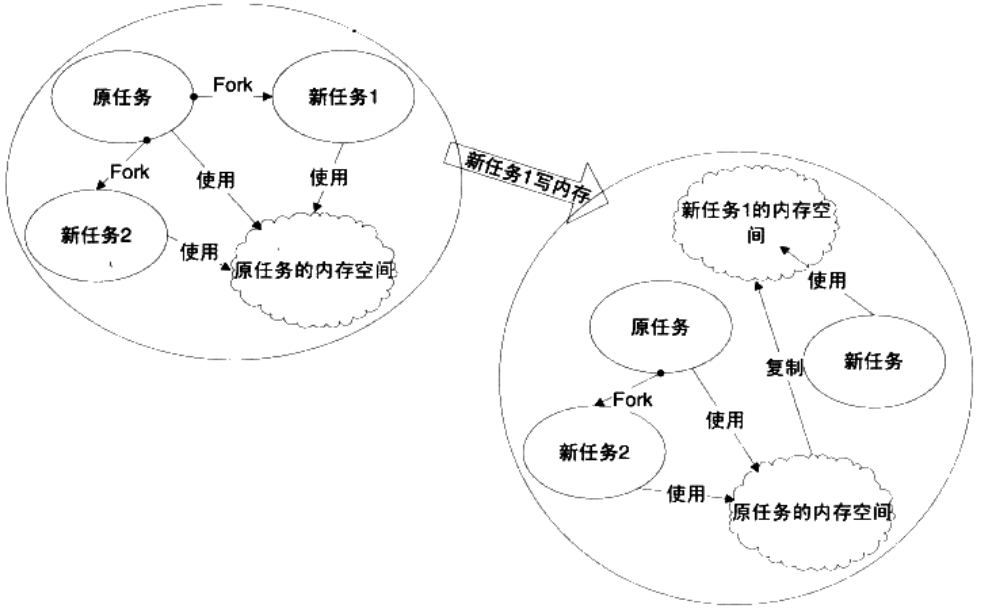

图 1-10 写时复制（Copy-On-Write）

使用 clone 可以产生一个新的任务，从指定的位置开始执行，并且（可选的）共享当前进程的内存空间和文件等。如此就可以在实际效果上产生一个线程。

1.6.2 线程安全

多线程程序处于一个多变的环境当中，可访问的全局变量和堆数据随时都可能被其他的线程改变。因此多线程程序在并发时数据的一致性变得非常重要。

竞争与原子操作

多个线程同时访问一个共享数据，可能造成很恶劣的后果。下面是一个著名的例子，假设有两个线程分别要执行如表 1-3 所示的 C 代码。

表 1-3

线程 1	线程 2
i=1; ++i;	--i;

在许多体系结构上，++i 的实现方法会如下：

（1）读取 i 到某个寄存器 X。

（2）X++。

（3）将 X 的内容存储回 i。

由于线程 1 和线程 2 并发执行，因此两个线程的执行序列很可能如下（注意，寄存器 X 的内容在不同的线程中是不一样的，这里用 $X^{[1]}$和 $X^{[2]}$分别表示线程 1 和线程 2 中的 X），如表 1-4 所示。

表 1-4

执行序号	执行指令	语句执行后变量值	线程
1	i=1	i=1，$X^{[1]}$=未知	1
2	$X^{[1]}$=i	i=1，$X^{[1]}$=1	1
3	$X^{[2]}$=i	i=1，$X^{[2]}$=1	2
4	$X^{[1]}$++	i=1，$X^{[1]}$=2	1
5	$X^{[2]}$--	i=1，$X^{[2]}$=0	2
6	i=$X^{[1]}$	i=2，$X^{[1]}$=2	1
7	i=$X^{[2]}$	i=0，$X^{[2]}$=0	2

从程序逻辑来看，两个线程都执行完毕之后，i 的值应该为 1，但从之前的执行序列可以看到，i 得到的值是 0。实际上这两个线程如果同时执行的话，i 的结果有可能是 0 或 1 或 2。可见，两个程序同时读写同一个共享数据会导致意想不到的后果。

很明显，自增（++）操作在多线程环境下会出现错误是因为这个操作被编译为汇编代码之后不止一条指令，因此在执行的时候可能执行了一半就被调度系统打断，去执行别的代码。我们把单指令的操作称为**原子的**（Atomic），因为无论如何，单条指令的执行是不会被打断的。为了避免出错，很多体系结构都提供了一些常用操作的原子指令，例如 i386 就有一条 inc 指令可以直接增加一个内存单元值，可以避免出现上例中的错误情况。在 Windows 里，有一套 API 专门进行一些原子操作（见表 1-5），这些 API 称为 Interlocked API。

表 1-5

Windows API	作用
InterlockedExchange	原子地交换两个值
InterlockedDecrement	原子地减少一个值
InterlockedIncrement	原子地增加一个值
InterlockedXor	原子地进行异或操作

使用这些函数时，Windows 将保证是原子操作的，因此可以不用担心出现问题。遗憾的是，尽管原子操作指令非常方便，但是它们仅适用于比较简单特定的场合。在复杂的场合下，

比如我们要保证一个复杂的数据结构更改的原子性，原子操作指令就力不从心了。这里我们需要更加通用的手段：锁。

同步与锁

为了避免多个线程同时读写同一个数据而产生不可预料的后果，我们需要将各个线程对同一个数据的访问**同步**（Synchronization）。所谓同步，既是指在一个线程访问数据未结束的时候，其他线程不得对同一个数据进行访问。如此，对数据的访问被原子化了。

同步的最常见方法是使用**锁**（Lock）。锁是一种非强制机制，每一个线程在访问数据或资源之前首先试图**获取**（Acquire）锁，并在访问结束之后**释放**（Release）锁。在锁已经被占用的时候试图获取锁时，线程会等待，直到锁重新可用。

二元信号量（Binary Semaphore）是最简单的一种锁，它只有两种状态：占用与非占用。它适合只能被唯一一个线程独占访问的资源。当二元信号量处于非占用状态时，第一个试图获取该二元信号量的线程会获得该锁，并将二元信号量置为占用状态，此后其他的所有试图获取该二元信号量的线程将会等待，直到该锁被释放。

对于允许多个线程并发访问的资源，多元信号量简称**信号量**（Semaphore），它是一个很好的选择。一个初始值为 N 的信号量允许 N 个线程并发访问。线程访问资源的时候首先获取信号量，进行如下操作：

- 将信号量的值减 1。
- 如果信号量的值小于 0，则进入等待状态，否则继续执行。

访问完资源之后，线程释放信号量，进行如下操作：

- 将信号量的值加 1。
- 如果信号量的值小于 1，唤醒一个等待中的线程。

互斥量（Mutex）和二元信号量很类似，资源仅同时允许一个线程访问，但和信号量不同的是，信号量在整个系统可以被任意线程获取并释放，也就是说，同一个信号量可以被系统中的一个线程获取之后由另一个线程释放。而互斥量则要求哪个线程获取了互斥量，哪个线程就要负责释放这个锁，其他线程越俎代庖去释放互斥量是无效的。

临界区（Critical Section）是比互斥量更加严格的同步手段。在术语中，把临界区的锁的获取称为进入临界区，而把锁的释放称为离开临界区。临界区和互斥量与信号量的区别在于，互斥量和信号量在系统的任何进程里都是可见的，也就是说，一个进程创建了一个互斥量或信号量，另一个进程试图去获取该锁是合法的。然而，临界区的作用范围仅限于本进程，其他的进程无法获取该锁。除此之外，临界区具有和互斥量相同的性质。

读写锁（Read-Write Lock）致力于一种更加特定的场合的同步。对于一段数据，多个线程同时读取总是没有问题的，但假设操作都不是原子型，只要有任何一个线程试图对这个数据进行修改，就必须使用同步手段来避免出错。如果我们使用上述信号量、互斥量或临界区中的任何一种来进行同步，尽管可以保证程序正确，但对于读取频繁，而仅仅偶尔写入的情况，会显得非常低效。读写锁可以避免这个问题。对于同一个锁，读写锁有两种获取方式，**共享的**（Shared）或**独占的**（Exclusive）。当锁处于自由的状态时，试图以任何一种方式获取锁都能成功，并将锁置于对应的状态。如果锁处于共享状态，其他线程以共享的方式获取锁仍然会成功，此时这个锁分配给了多个线程。然而，如果其他线程试图以独占的方式获取已经处于共享状态的锁，那么它将必须等待锁被所有的线程释放。相应地，处于独占状态的锁将阻止任何其他线程获取该锁，不论它们试图以哪种方式获取。读写锁的行为可以总结如表 1-6 所示。

表 1-6

读写锁状态	以共享方式获取	以独占方式获取
自由	成功	成功
共享	成功	等待
独占	等待	等待

条件变量（Condition Variable）作为一种同步手段，作用类似于一个栅栏。对于条件变量，线程可以有两种操作，首先线程可以等待条件变量，一个条件变量可以被多个线程等待。其次，线程可以唤醒条件变量，此时某个或所有等待此条件变量的线程都会被唤醒并继续支持。也就是说，使用条件变量可以让许多线程一起等待某个事件的发生，当事件发生时（条件变量被唤醒），所有的线程可以一起恢复执行。

可重入（Reentrant）与线程安全

一个函数被重入，表示这个函数没有执行完成，由于外部因素或内部调用，又一次进入该函数执行。一个函数要被重入，只有两种情况：

（1）多个线程同时执行这个函数。

（2）函数自身（可能是经过多层调用之后）调用自身。

一个函数被称为可重入的，表明该函数被重入之后不会产生任何不良后果。举个例子，如下面这个 sqr 函数就是可重入的：

```
int sqr(int x)
{
    return x * x;
}
```

一个函数要成为可重入的，必须具有如下几个特点：

- 不使用任何（局部）静态或全局的非 const 变量。
- 不返回任何（局部）静态或全局的非 const 变量的指针。
- 仅依赖于调用方提供的参数。
- 不依赖任何单个资源的锁（mutex 等）。
- 不调用任何不可重入的函数。

可重入是并发安全的强力保障，一个可重入的函数可以在多线程环境下放心使用。

过度优化

线程安全是一个非常烫手的山芋，因为即使合理地使用了锁，也不一定能保证线程安全，这是源于落后的编译器技术已经无法满足日益增长的并发需求。很多看似无错的代码在优化和并发面前又产生了麻烦。最简单的例子，让我们看看如下代码：

```
x = 0;
Thread1   Thread2
lock();     lock();
x++;      x++;
unlock();   unlock();
```

由于有 lock 和 unlock 的保护，x++的行为不会被并发所破坏，那么 x 的值似乎必然是 2 了。然而，如果编译器为了提高 x 的访问速度，把 x 放到了某个寄存器里，那么我们知道不同线程的寄存器是各自独立的，因此如果 Thread1 先获得锁，则程序的执行可能会呈现如下的情况：

- [Thread1]读取 x 的值到某个寄存器 R[1]（R[1]=0）。
- [Thread1]R[1]++（由于之后可能还要访问 x，因此 Thread1 暂时不将 R[1]写回 x）。
- [Thread2]读取 x 的值到某个寄存器 R[2]（R[2]=0）。
- [Thread2]R[2]++(R[2]=1)。
- [Thread2]将 R[2]写回至 x(x=1)。
- [Thread1]（很久以后）将 R[1]写回至 x(x=1)。

可见在这样的情况下即使正确地加锁，也不能保证多线程安全。下面是另一个例子：

```
x = y = 0;
Thread1  Thread2
x = 1;      y = 1;
r1 = y;     r2 = x;
```

很显然，r1 和 r2 至少有一个为 1，逻辑上不可能同时为 0。然而，事实上 r1=r2=0 的情况确实可能发生。原因在于早在几十年前，CPU 就发展出了动态调度，在执行程序的时候为了提高效率有可能交换指令的顺序。同样，编译器在进行优化的时候，也可能为了效率而

交换毫不相干的两条相邻指令（如 x=1 和 r1=y）的执行顺序。也就是说，以上代码执行的时候可能是这样的：

```
x = y = 0;
Thread1  Thread2
r1 = y;       y = 1;
x = 1;       r2 = x;
```

那么 r1=r2=0 就完全可能了。我们可以使用 volatile 关键字试图阻止过度优化，volatile 基本可以做到两件事情：

（1）阻止编译器为了提高速度将一个变量缓存到寄存器内而不写回。

（2）阻止编译器调整操作 volatile 变量的指令顺序。

可见 volatile 可以完美地解决第一个问题，但是 volatile 是否也能解决第二个问题呢？答案是不能。因为即使 volatile 能够阻止编译器调整顺序，也无法阻止 CPU 动态调度换序。

另一个颇为著名的与换序有关的问题来自于 Singleton 模式的 double-check。一段典型的 double-check 的 singleton 代码是这样的（不熟悉 Singleton 的读者可以参考《设计模式：可复用面向对象软件的基础》，但下面所介绍的内容并不真正需要了解 Singleton）：

```
volatile T* pInst = 0;
T* GetInstance()
{
if (pInst == NULL)
{
    lock();
    if (pInst == NULL)
        pInst = new T;
    unlock();
}
return pInst;
}
```

抛开逻辑，这样的代码乍看是没有问题的，当函数返回时，PInst 总是指向一个有效的对象。而 lock 和 unlock 防止了多线程竞争导致的麻烦。双重的 if 在这里另有妙用，可以让 lock 的调用开销降低到最小。读者可以自己揣摩。

但是实际上这样的代码是有问题的。问题的来源仍然是 CPU 的乱序执行。C++里的 new 其实包含了两个步骤：

（1）分配内存。

（2）调用构造函数。

所以 pInst = new T 包含了三个步骤：

（1）分配内存。

（2）在内存的位置上调用构造函数。

（3）将内存的地址赋值给 pInst。

在这三步中，（2）和（3）的顺序是可以颠倒的。也就是说，完全有可能出现这样的情况：pInst 的值已经不是 NULL，但对象仍然没有构造完毕。这时候如果出现另外一个对 GetInstance 的并发调用，此时第一个 if 内的表达式 pInst==NULL 为 false，所以这个调用会直接返回尚未构造完全的对象的地址（pInst）以提供给用户使用。那么程序这个时候会不会崩溃就取决于这个类的设计如何了。

从上面两个例子可以看到 CPU 的乱序执行能力让我们对多线程的安全保障的努力变得异常困难。因此要保证线程安全，阻止 CPU 换序是必需的。遗憾的是，现在并不存在可移植的阻止换序的方法。通常情况下是调用 CPU 提供的一条指令，这条指令常常被称为 barrier。一条 barrier 指令会阻止 CPU 将该指令之前的指令交换到 barrier 之后，反之亦然。换句话说，barrier 指令的作用类似于一个拦水坝，阻止换序“穿透”这个大坝。

许多体系结构的 CPU 都提供 barrier 指令，不过它们的名称各不相同，例如 POWERPC 提供的其中一条指令名叫 lwsync。我们可以这样来保证线程安全：

```
#define barrier() __asm__ volatile ("lwsync")
volatile T* pInst = 0;
T* GetInstance()
{
    if (!pInst)
    {
        lock();
        if (!pInst)
        {
            T* temp = new T;
            barrier();
            pInst = temp;
        }
        unlock();
    }
    return pInst;
}
```

由于 barrier 的存在，对象的构造一定在 barrier 执行之前完成，因此当 pInst 被赋值时，对象总是完好的。

1.6.3 多线程内部情况

三种线程模型

线程的并发执行是由多处理器或操作系统调度来实现的。但实际情况要更为复杂一些：大多数操作系统，包括 Windows 和 Linux，都在内核里提供线程的支持，内核线程（注：这

里的内核线程和 Linux 内核里的 kernel_thread 并不是一回事）和我们之前讨论的一样，由多处理器或调度来实现并发。然而用户实际使用的线程并不是内核线程，而是存在于用户态的用户线程。用户态线程并不一定在操作系统内核里对应同等数量的内核线程，例如某些轻量级的线程库，对用户来说如果有三个线程在同时执行，对内核来说很可能只有一个线程。本节我们将详细介绍用户态多线程库的实现方式。

1. 一对一模型

对于直接支持线程的系统，一对一模型始终是最为简单的模型。对一对一模型来说，一个用户使用的线程就唯一对应一个内核使用的线程（但反过来不一定，一个内核里的线程在用户态不一定有对应的线程存在），如图 1-11 所示。

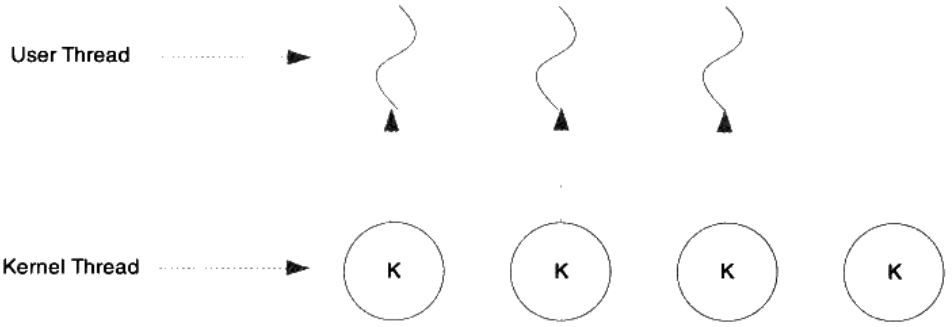

图 1-11 一对一线程模型

这样用户线程就具有了和内核线程一致的优点，线程之间的并发是真正的并发，一个线程因为某原因阻塞时，其他线程执行不会受到影响。此外，一对一模型也可以让多线程程序在多处理器的系统上有更好的表现。

一般直接使用 API 或系统调用创建的线程均为一对一的线程。例如在 Linux 里使用 clone（带有 CLONE_VM 参数）产生的线程就是一个一对一线程，因为此时在内核有一个唯一的线程与之对应。下列代码演示了这一过程：

```
int thread_function(void*)
{ ....}
char thread_stack[4096];

void foo
{
    clone(thread_function, thread_stack, CLONE_VM, 0);
}
```

在 Windows 里，使用 API CreateThread 即可创建一个一对一的线程。

一对一线程缺点有两个：

- 由于许多操作系统限制了内核线程的数量，因此一对一线程会让用户的线程数量受到限制。
- 许多操作系统内核线程调度时，上下文切换的开销较大，导致用户线程的执行效率下降。

2. 多对一模型

多对一模型将多个用户线程映射到一个内核线程上，线程之间的切换由用户态的代码来进行，因此相对于一对一模型，多对一模型的线程切换要快速许多。多对一的模型示意图如图 1-12 所示。

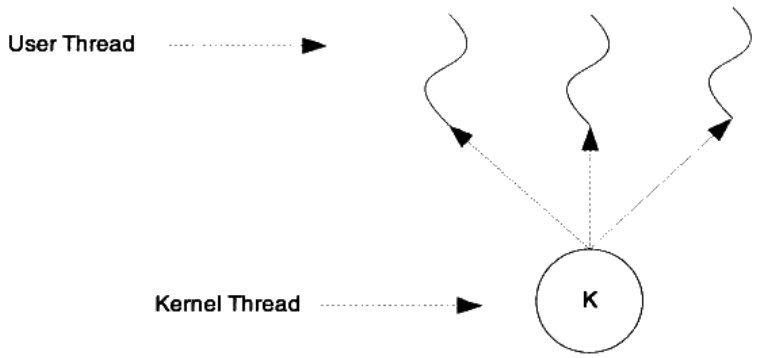

图 1-12 多对一线程模型

多对一模型一大问题是，如果其中一个用户线程阻塞，那么所有的线程都将无法执行，因为此时内核里的线程也随之阻塞了。另外，在多处理器系统上，处理器的增多对多对一模型的线程性能也不会有明显的帮助。但同时，多对一模型得到的好处是高效的上下文切换和几乎无限制的线程数量。

3. 多对多模型

多对多模型结合了多对一模型和一对一模型的特点，将多个用户线程映射到少数但不止一个内核线程上，如图 1-13 所示。

在多对多模型中，一个用户线程阻塞并不会使得所有的用户线程阻塞，因为此时还有别的线程可以被调度来执行。另外，多对多模型对用户线程的数量也没什么限制，在多处理器系统上，多对多模型的线程也能得到一定的性能提升，不过提升的幅度不如一对一模型高。

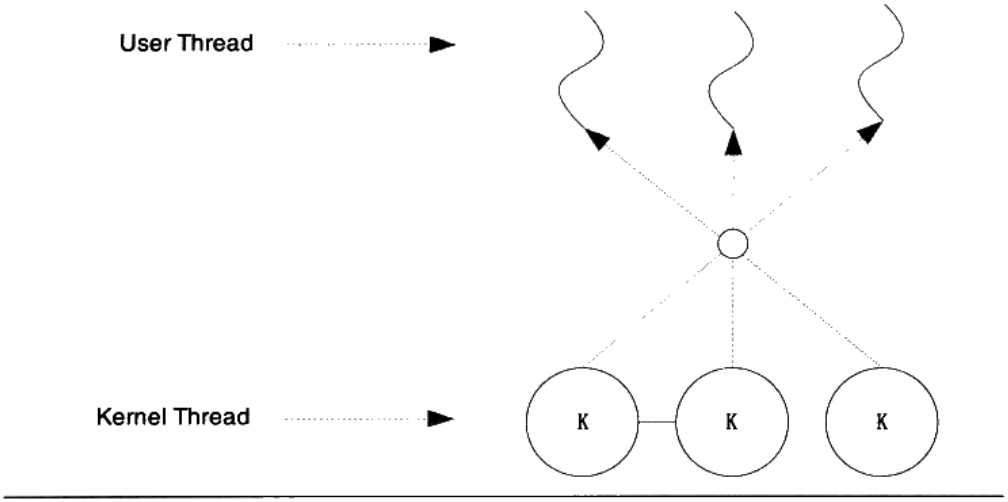

图 1-13 多对多线程模型

1.7 本章小结

在这一章中，我们对整个计算机的软硬件基本结构进行了回顾，包括 CPU 与外围部件的连接方式、SMP 与多核、软硬件层次体系结构、如何充分利用 CPU 及与系统软件十分相关的设备驱动、操作系统、虚拟空间、物理空间、页映射和线程的基础概念。虽然这些概念都是大家所了解的，但是我们认为还是有必要回顾一下，它们跟本书后面章节介绍的内容息息相关。正所谓温故而知新，这就是本章的目的。