计算机系统漫游

计算机系统是由硬件和系统软件组成的，它们共同工作来运行应用程序。虽然系统的具体实现方式随着时间不断变化，但是系统内在的概念却没有改变。所有计算机系统都有相似的硬件和软件组件，它们执行着相似的功能。一些程序员希望深入了解这些组件是如何工作的，以及这些组件是如何影响程序的正确性和性能的，以此来提高自身的技能。本书便是为这些读者而写的。

现在就要开始一次有趣的漫游历程了。如果你全力投身学习本书中的概念，完全理解底层计算机系统以及它对应用程序的影响，那么你将会逐渐成为凤毛麟角的“权威”程序员。

你将会学习一些实践技巧，比如如何避免由计算机表示数字的方式导致的奇怪的数字错误。你将学会怎样通过一些聪明的小窍门来优化你的 C 代码，以充分利用现代处理器和存储器系统的设计。你将了解到编译器是如何实现过程调用的，以及如何利用这些知识避免缓冲区溢出错误带来的安全漏洞，这些弱点会给网络和因特网软件带来了巨大的麻烦。你将学会如何识别和避免链接时那些令人讨厌的错误，它们困扰着普通的程序员。你将学会如何编写自己的 Unix 外壳、自己的动态存储分配包，甚至是自己的 Web 服务器。你会认识到并发带来的希望和陷阱，当单个芯片上集成了多个处理器核时，这个主题变得越来越重要。

在关于 C 编程语言的经典文献 [58] 中，Kernighan 和 Ritchie 通过图 1-1 所示的 `hello` 程序来向读者介绍 C 语言。尽管 `hello` 程序非常简单，但是为了让它完成运行，系统的每个主要组成部分都需要协调工作。从某种意义上来说，本书的目的就是要帮助你了解当你在系统上执行 `hello` 程序时，系统发生了什么以及为什么会这样。

code/intro/hello.c

```
1  #include <stdio.h>
2
3  int main()
4  {
5      printf("hello, world\n");
6  }
```

code/intro/hello.c

图 1-1　`hello` 程序

我们通过跟踪 `hello` 程序的生命周期来开始对系统的学习——从它被程序员创建，到在系统上运行，输出简单的消息，然后终止。我们将沿着这个程序的生命周期，简要地介绍一些逐步出现的关键概念、专业术语和组成部分。后面的章节将围绕这些内容展开。

1.1　信息就是位 + 上下文

`hello` 程序的生命周期是从一个源程序（或者说源文件）开始的，即程序员利用编辑器创建并保存的文本文件，文件名是 `hello.c`。源程序实际上就是一个由值 0 和 1 组成的位（bit）序列，8 个位被组织成一组，称为字节。每个字节表示程序中某个文本字符。

大部分的现代系统都使用 ASCII 标准来表示文本字符，这种方式实际上就是用一个唯一的单字节大小的整数值来表示每个字符。例如图 1-2 中给出了 `hello.c` 程序的 ASCII 码表示。

`hello.c` 程序以字节序列的方式存储在文件中。每个字节都有一个整数值，而该整数值对

应于某个字符。例如，第一个字节的整数值是35，它对应的就是字符‘#’；第二个字节整数值为105，它对应的字符是‘i’，依次类推。注意，每个文本行都是以一个不可见的换行符‘\n’来结束的，它所对应的整数值为10。像hello.c这样只由ASCII字符构成的文件称为文本文件，所有其他文件都称为二进制文件。

```
 #    i    n    c    l    u    d    e   <sp>  <    s    t    d    i    o    .
 35  105  110   99  108  117  100  101   32   60  115  116  100  105  111   46

 h    >    \n   \n   i    n    t   <sp>  m    a    i    n    (    )   \n   {
104   62   10   10  105  110  116   32  109   97  105  110   40   41   10  123

 \n  <sp> <sp> <sp> <sp>  p    r    i    n    t    f    (    "    h    e    l
 10   32   32   32   32  112  114  105  110  116  102   40   34  104  101  108

 l    o    ,   <sp>  w    o    r    l    d    \    n    "    )    ;    \n   }
108  111   44   32  119  111  114  108  100   92  110   34   41   59   10  125
```

图1-2　hello.c的ASCII文本表示

hello.c的表示方法说明了一个基本的思想：系统中所有的信息——包括磁盘文件、存储器中的程序、存储器中存放的用户数据以及网络上传送的数据，都是由一串位表示的。区分不同数据对象的唯一方法是我们读到这些数据对象时的上下文。比如，在不同的上下文中，一个同样的字节序列可能表示一个整数、浮点数、字符串或者机器指令。

作为程序员，我们需要了解数字的机器表示方式，因为它们与实际的整数和实数是不同的。它们是对真值的有限近似值，有时候会有意想不到的行为表现。这方面的基本原理将在第2章中详细描述。

C编程语言的起源

C语言是贝尔实验室的Dennis Ritchie于1969年～1973年间创建的。美国国家标准学会（American National Standards Institute，ANSI）在1989年颁布了ANSI C的标准，后来使C语言标准化成为了国际标准化组织（International Standards Organization，ISO）的责任。这些标准定义了C语言和一系列函数库，即所谓的“**C标准库**”。Kernighan和Ritchie在他们的经典著作中描述了ANSI C，这本著作被人们满怀感情地称为“K&R”[58]。用Ritchie的话来说[88]，C语言是“古怪的、有缺陷的，但也是一个巨大的成功”。为什么会成功呢？

- **C语言与Unix操作系统关系密切**。C语言从一开始就是作为一种用于Unix系统的程序设计语言而开发出来的。大部分Unix内核，以及所有支撑工具和函数库都是用C语言编写的。20世纪70年代后期到80年代初期，Unix在高等院校兴起，许多人开始接触C语言并喜欢上了它。因为Unix几乎全部是用C编写的，所以可以很方便地移植到新的机器上，这种特点为C和Unix赢得了更为广泛的支持。
- **C语言小而简单**。掌控C语言设计的是一个人而非一个协会，因此这是一个简洁明了、没有什么冗赘的一致的设计。K&R这本书用了大量的例子和练习描述了完整的C语言及其标准库，而全书不过261页。C语言的简单使它相对而言易于学习，也易于移植到不同的计算机上。
- **C语言是为实践目的设计的**。C语言是为实现Unix操作系统而设计的。后来，其他人发现能够用这门语言无障碍地编写他们想要的程序。

C语言是系统级编程的首选，同时它也非常适用于应用级程序的编写。然而，它也并非适

用于所有的程序员和所有的情况。C 语言的指针是造成困惑和程序错误的一个常见原因。同时，C 语言还缺乏对非常有用的抽象（例如类、对象和异常）的显式支持。像 C++ 和 Java 这样针对应用级程序的新程序设计语言解决了这些问题。

1.2　程序被其他程序翻译成不同的格式

hello 程序的生命周期是从一个高级 C 语言程序开始的，因为这种形式能够被人读懂。然而，为了在系统上运行 hello.c 程序，每条 C 语句都必须被其他程序转化为一系列的低级机器语言指令。然后这些指令按照一种称为可执行目标程序的格式打好包，并以二进制磁盘文件的形式存放起来。目标程序也称为可执行目标文件。

在 Unix 系统上，从源文件到目标文件的转化是由编译器驱动程序完成的：

```
unix> gcc -o hello hello.c
```

在这里，GCC 编译器驱动程序读取源程序文件 hello.c，并把它翻译成一个可执行目标文件 hello。这个翻译的过程可分为四个阶段完成，如图 1-3 所示。执行这四个阶段的程序（预处理器、编译器、汇编器和链接器）一起构成了编译系统（compilation system）。

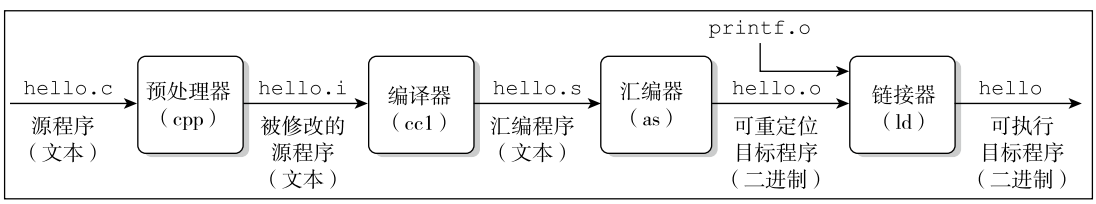

图 1-3　编译系统

- 预处理阶段。预处理器（cpp）根据以字符 # 开头的命令，修改原始的 C 程序。比如 hello.c 中第 1 行的 #include <stdio.h> 命令告诉预处理器读取系统头文件 stdio.h 的内容，并把它直接插入到程序文本中。结果就得到了另一个 C 程序，通常是以 .i 作为文件扩展名。
- 编译阶段。编译器（cc1）将文本文件 hello.i 翻译成文本文件 hello.s，它包含一个汇编语言程序。汇编语言程序中的每条语句都以一种标准的文本格式确切地描述了一条低级机器语言指令。汇编语言是非常有用的，因为它为不同高级语言的不同编译器提供了通用的输出语言。例如，C 编译器和 Fortran 编译器产生的输出文件用的都是一样的汇编语言。
- 汇编阶段。接下来，汇编器（as）将 hello.s 翻译成机器语言指令，把这些指令打包成一种叫做可重定位目标程序（relocatable object program）的格式，并将结果保存在目标文件 hello.o 中。hello.o 文件是一个二进制文件，它的字节编码是机器语言指令而不是字符。如果我们在文本编辑器中打开 hello.o 文件，看到的将是一堆乱码。
- 链接阶段。请注意，hello 程序调用了 printf 函数，它是每个 C 编译器都会提供的标准 C 库中的一个函数。printf 函数存在于一个名为 printf.o 的单独的预编译好了的目标文件中，而这个文件必须以某种方式合并到我们的 hello.o 程序中。链接器（ld）就负责处理这种合并。结果就得到 hello 文件，它是一个可执行目标文件（或者简称为可执行文件），可以被加载到内存中，由系统执行。

GNU 项目

GCC 是 GNU（GNU 是 GNU's Not Unix 的缩写）项目开发出来的众多有用工具之一。GNU 项目是 1984 年由 Richard Stallman 发起的一个免税的慈善项目。该项目的目标非常宏大，就是开

发出一个完整的类 Unix 的系统，其源代码能够不受限制地被修改和传播。GNU 项目已经开发出了一个包含 Unix 操作系统的所有主要部件的环境，但内核除外，内核是由 Linux 项目独立发展而来的。GNU 环境包括 EMACS 编辑器、GCC 编译器、GDB 调试器、汇编器、链接器、处理二进制文件的工具以及其他一些部件。GCC 编译器已经发展到支持许多不同的语言，能够针对许多不同的机器生成代码。支持的语言包括 C、C++、Fortran、Java、Pascal、面向对象 C 语言（Objective-C）和 Ada。

GNU 项目取得了非凡的成绩，但是却常常被忽略。现代开放源码运动（通常和 Linux 联系在一起）的思想起源是 GNU 项目中自由软件（free software）的概念。（此处的 free 为自由言论（free speech）中“自由”之意，而非免费啤酒（free beer）中“免费”之意。）而且，Linux 如此受欢迎在很大程度上还要归功于 GNU 工具，因为它们给 Linux 内核提供了环境。

1.3　了解编译系统如何工作是大有益处的

对于像 `hello.c` 这样简单的程序，我们可以依靠编译系统生成正确有效的机器代码。但是，有一些重要的原因促使程序员必须知道编译系统是如何工作的，其原因如下：

- *优化程序性能。*现代编译器都是成熟的工具，通常可以生成很好的代码。作为程序员，我们无需为了写出高效代码而去了解编译器的内部工作。但是，为了在 C 程序中做出好的编码选择，我们确实需要了解一些机器代码以及编译器将不同的 C 语句转化为机器代码的方式。例如，一个 `switch` 语句是否总是比一系列的 `if-then-else` 语句高效得多？一个函数调用的开销有多大？`while` 循环比 `for` 循环更有效吗？指针引用比数组索引更有效吗？为什么将循环求和的结果放到一个本地变量中，与将其放到一个通过引用传递过来的参数中相比，运行速度要快很多呢？为什么我们只是简单地重新排列一下一个算术表达式中的括号就能让一个函数运行得更快？

  在第 3 章中，我们将介绍两种相关的机器语言：IA32 和 x86-64。IA32 是 32 位的，目前普遍应用于运行 Linux、Windows 以及较新版本的 Macintosh 操作系统的机器上；x86-64 是 64 位的，可以用在比较新的微处理器上。我们会介绍编译器是如何把不同的 C 语言结构转换成它们的机器语言的。第 5 章，你将学习如何通过简单转换 C 语言代码，以帮助编译器更好地完成工作，从而调整 C 程序的性能。在第 6 章，你将学习到存储器系统的层次结构特性，C 语言编译器将数组存放在存储器中的方式，以及 C 程序又是如何能够利用这些知识从而更高效地运行。
- *理解链接时出现的错误。*根据我们的经验，一些最令人困扰的程序错误往往都与链接器操作有关，尤其是当你试图构建大型的软件系统时。例如，链接器报告它无法解析一个引用，这是什么意思？静态变量和全局变量的区别是什么？如果你在不同的 C 文件中定义了名字相同的两个全局变量会发生什么？静态库和动态库的区别是什么？我们在命令行上排列库的顺序有什么影响？最严重的是，为什么有些链接错误直到运行时才会出现？在第 7 章，你将得到这些问题的答案。
- *避免安全漏洞。*多年来，缓冲区溢出错误是造成大多数网络和 Internet 服务器上安全漏洞的主要原因。存在这些错误是因为很少有人能理解限制他们从不受信任的站点接收数据的数量和格式的重要性。学习安全编程的第一步就是理解数据和控制信息存储在程序栈上的方式会引起的后果。作为学习汇编语言的一部分，我们将在第 3 章中描述堆栈原理和缓冲区溢出错误。我们还将学习程序员、编译器和操作系统可以用来降低攻击威胁的方法。

1.4　处理器读并解释存储在存储器中的指令

此刻，hello.c 源程序已经被编译系统翻译成了可执行目标文件 hello，并存放在磁盘上。要想在 Unix 系统上运行该可执行文件，我们将它的文件名输入到称为外壳（shell）的应用程序中：

```
unix> ./hello
hello, world
unix>
```

外壳是一个命令行解释器，它输出一个提示符，等待你输入一个命令行，然后执行这个命令。如果该命令行的第一个单词不是一个内置的外壳命令，那么外壳就会假设这是一个可执行文件的名字，它将加载并运行这个文件。所以在此例中，外壳将加载并运行 hello 程序，然后等待程序终止。hello 程序在屏幕上输出它的信息，然后终止。外壳随后输出一个提示符，等待下一个输入的命令行。

1.4.1　系统的硬件组成

为了理解运行 hello 程序时发生了什么，我们需要了解一个典型系统的硬件组织，如图 1-4 所示。这张图是 Intel Pentium 系统产品系列的模型，但是所有其他系统也有相同的外观和特性。现在不要担心这张图的复杂性——我们将在本书分阶段介绍大量的细节。

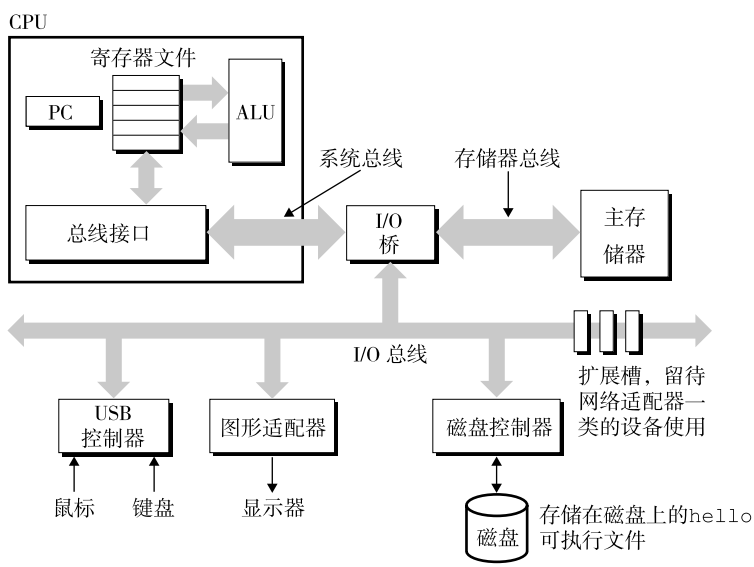

图 1-4　一个典型系统的硬件组成

CPU：中央处理单元；ALU：算术 / 逻辑单元；PC：程序计数器；USB：通用串行总线

1. 总线

贯穿整个系统的是一组电子管道，称做总线，它携带信息字节并负责在各个部件间传递。通常总线被设计成传送定长的字节块，也就是字（word）。字中的字节数（即字长）是一个基本的系统参数，在各个系统中的情况都不尽相同。现在的大多数机器字长有的是 4 个字节（32 位），有的是 8 个字节（64 位）。为了讨论的方便，假设字长为 4 个字节，并且总线每次只传送 1 个字。

2. I/O 设备

输入 / 输出（I/O）设备是系统与外部世界的联系通道。我们的示例系统包括 4 个 I/O 设备：作为用户输入的键盘和鼠标，作为用户输出的显示器，以及用于长期存储数据和程序的磁盘驱动

器（简单地说就是磁盘）。最初，可执行程序 hello 就存放在磁盘上。

每个 I/O 设备都通过一个*控制器*或*适配器*与 I/O 总线相连。控制器和适配器之间的区别主要在于它们的封装方式。控制器是置于 I/O 设备本身的或者系统的主印制电路板（通常称为主板）上的芯片组，而适配器则是一块插在主板插槽上的卡。无论如何，它们的功能都是在 I/O 总线和 I/O 设备之间传递信息。

第 6 章会更多地说明磁盘之类的 I/O 设备是如何工作的。在第 10 章，你将学习如何在应用程序中利用 Unix I/O 接口访问设备。我们将特别关注网络类设备，不过这些技术对于其他设备来说也是通用的。

3. 主存

*主存*是一个临时存储设备，在处理器执行程序时，用来存放程序和程序处理的数据。从物理上来说，主存是由一组*动态随机存取存储器*（DRAM）芯片组成的。从逻辑上来说，存储器是一个线性的字节数组，每个字节都有其唯一的地址（即数组索引），这些地址是从零开始的。一般来说，组成程序的每条机器指令都由不同数量的字节构成。与 C 程序变量相对应的数据项的大小是根据类型变化的。例如，在运行 Linux 的 IA32 机器上，short 类型的数据需要 2 个字节，int、float 和 long 类型需要 4 个字节，而 double 类型需要 8 个字节。

第 6 章将具体介绍存储技术，如 DRAM 芯片是如何工作的，以及它们又是如何组合起来构成主存的。

4. 处理器

中央处理单元（CPU），简称*处理器*，是解释（或*执行*）存储在主存中指令的引擎。处理器的核心是一个字长的存储设备（或*寄存器*），称为*程序计数器*（PC）。在任何时刻，PC 都指向主存中的某条机器语言指令（即含有该条指令的地址）。㊀

从系统通电开始，直到系统断电，处理器一直在不断地执行程序计数器指向的指令，再更新程序计数器，使其指向下一条指令。处理器看上去是按照一个非常简单的指令执行模型来操作的，这个模型是由指令集结构决定的。在这个模型中，指令按照严格的顺序执行，而执行一条指令包含执行一系列的步骤。处理器从程序计数器（PC）指向的存储器处读取指令，解释指令中的位，执行该指令指示的简单操作，然后更新 PC，使其指向下一条指令，而这条指令并不一定与存储器中刚刚执行的指令相邻。

这样的简单操作并不多，而且操作是围绕着主存、*寄存器文件*（register file）和*算术 / 逻辑单元*（ALU）进行的。寄存器文件是一个小的存储设备，由一些 1 字长的寄存器组成，每个寄存器都有唯一的名字。ALU 计算新的数据和地址值。下面列举一些简单操作的例子，CPU 在指令的要求下可能会执行以下操作：

- *加载*：把一个字节或者一个字从主存复制到寄存器，以覆盖寄存器原来的内容。
- *存储*：把一个字节或者一个字从寄存器复制到主存的某个位置，以覆盖这个位置上原来的内容。
- *操作*：把两个寄存器的内容复制到 ALU，ALU 对这两个字做算术操作，并将结果存放到一个寄存器中，以覆盖该寄存器中原来的内容。
- *跳转*：从指令本身中抽取一个字，并将这个字复制到程序计数器（PC）中，以覆盖 PC 中原来的值。

处理器看上去只是它的指令集结构的简单实现，但是实际上现代处理器使用了非常复杂的机制来加速程序的执行。因此，我们可以这样区分处理器的指令集结构和微体系结构：指令集结构

㊀ PC 也普遍地被用来作为“个人计算机”的缩写。然而，两者之间的区别应该可以很清楚地从上下文中看出来。

描述的是每条机器代码指令的效果；而微体系结构描述的是处理器实际上是如何实现的。第 3 章我们研究机器代码时考虑的是机器的指令集结构所提供的抽象性。第 4 章将更详细地介绍处理器实际上是如何实现的。

1.4.2　运行 hello 程序

前面简单描述了系统的硬件组成和操作，现在开始介绍当我们运行示例程序时到底发生了些什么。在这里我们必须省略很多细节稍后再做补充，但是从现在起我们将很满意这种整体上的描述。

初始时，外壳程序执行它的指令，等待我们输入一个命令。当我们在键盘上输入字符串“./hello”后，外壳程序将字符逐一读入寄存器，再把它存放到存储器中，如图 1-5 所示。

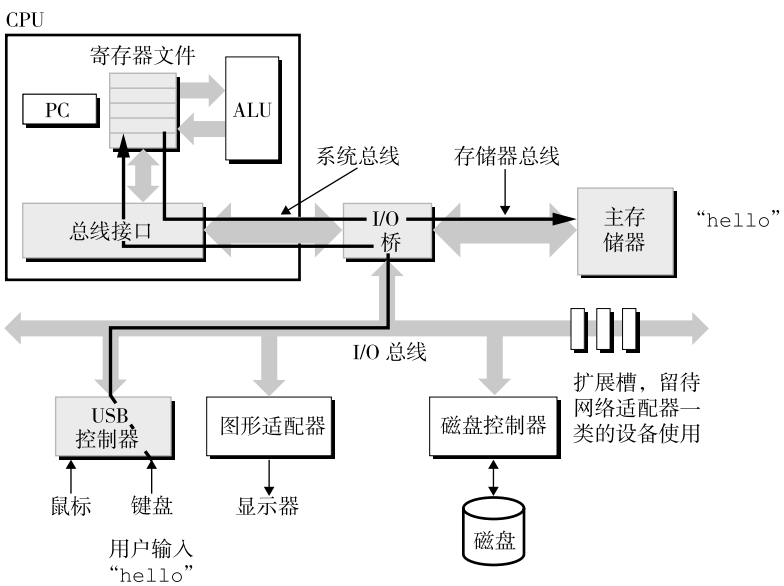

图 1-5　从键盘上读取 hello 命令

当我们在键盘上敲回车键时，外壳程序就知道我们已经结束了命令的输入。然后外壳执行一系列指令来加载可执行的 hello 文件，将 hello 目标文件中的代码和数据从磁盘复制到主存。数据包括最终会被输出的字符串“hello, world\n”。

利用直接存储器存取（DMA，将在第 6 章讨论）的技术，数据可以不通过处理器而直接从磁盘到达主存。这个步骤如图 1-6 所示。

一旦目标文件 hello 中的代码和数据被加载到主存，处理器就开始执行 hello 程序的 main 程序中的机器语言指令。这些指令将“hello, world\n”字符串中的字节从主存复制到寄存器文件，再从寄存器文件中复制到显示设备，最终显示在屏幕上。这个步骤如图 1-7 所示。

1.5　高速缓存至关重要

这个简单的示例揭示了一个重要的问题，即系统花费了大量的时间把信息从一个地方挪到另一个地方。hello 程序的机器指令最初是存放在磁盘上的，当程序加载时，它们被复制到主存；当处理器运行程序时，指令又从主存复制到处理器。相似地，数据串“hello, world\n”初始时在磁盘上，然后复制到主存，最后从主存上复制到显示设备。从程序员的角度来看，这些复制就是开销，减缓了程序“真正”的工作。因此，系统设计者的一个主要目标就是使这些复制操作尽可能快地完成。

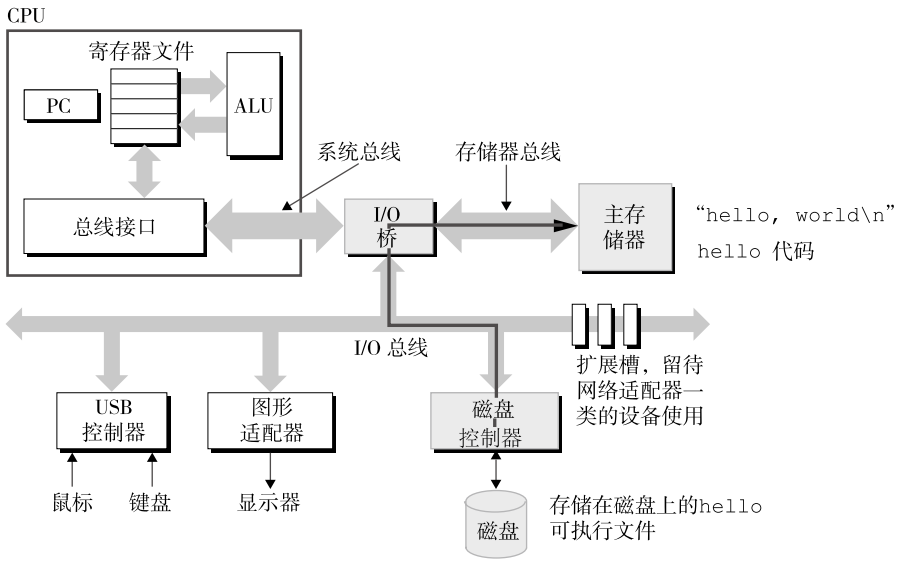

图 1-6　从磁盘加载可执行文件到主存

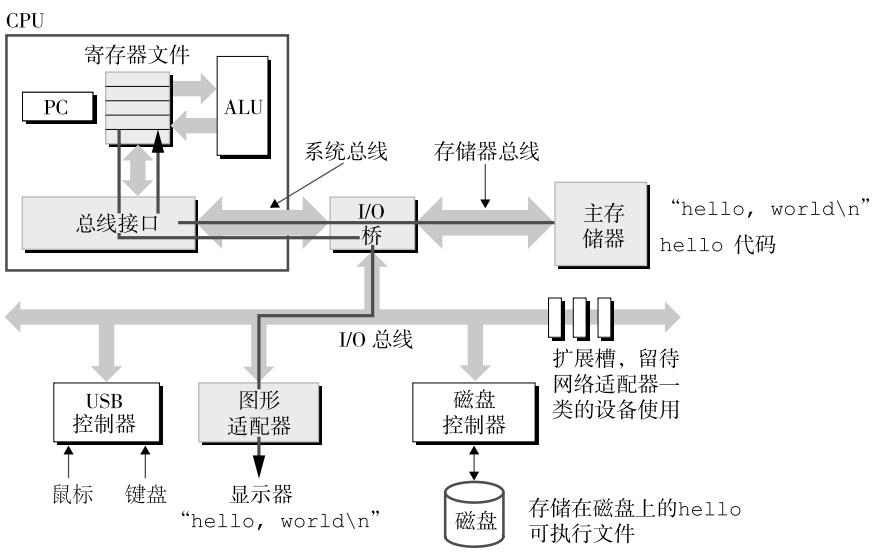

图 1-7　将输出字符串从内存写到显示器

根据机械原理，较大的存储设备要比较小的存储设备运行得慢，而快速设备的造价远高于同类的低速设备。例如，一个典型系统上的磁盘驱动器可能比主存大 1000 倍，但是对处理器而言，从磁盘驱动器上读取一个字的时间开销要比从主存中读取的开销大 1000 万倍。

类似地，一个典型的寄存器文件只存储几百字节的信息，而主存里可存放几十亿字节。然而，处理器从寄存器文件中读数据的速度比从主存中读取几乎要快 100 倍。更麻烦的是，随着这些年半导体技术的进步，这种处理器与主存之间的差距还在持续增大。加快处理器的运行速度比加快主存的运行速度要容易和便宜得多。

针对这种处理器与主存之间的差异，系统设计者采用了更小、更快的存储设备，即*高速缓存存储器*（简称高速缓存），作为暂时的集结区域，用来存放处理器近期可能会需要的信息。图 1-8 展示了一个典型系统中的高速缓存存储器。位于处理器芯片上的 L1 高速缓存的容量可以达

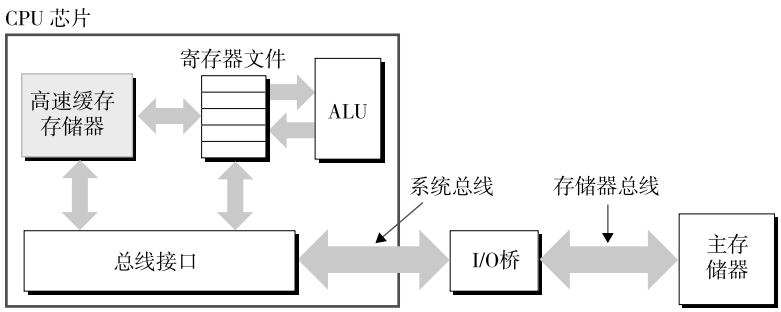

图 1-8　高速缓存存储器

到数万字节，访问速度几乎和访问寄存器文件一样快。一个容量为数十万到数百万字节的更大的 L2 高速缓存通过一条特殊的总线连接到处理器。进程访问 L2 高速缓存的时间要比访问 L1 高速缓存的时间长 5 倍，但是这仍然比访问主存的时间快 5 ～ 10 倍。L1 和 L2 高速缓存是用一种叫做*静态随机访问存储器*（SRAM）的硬件技术实现的。比较新的、处理能力更强大的系统甚至有三级高速缓存：L1、L2 和 L3。系统可以获得一个很大的存储器，同时访问速度也很快，原因是利用了高速缓存的局部性原理，即程序具有访问局部区域里的数据和代码的趋势。通过让高速缓存里存放可能经常访问的数据的方法，大部分的存储器操作都能在快速的高速缓存中完成。

本书得出的重要结论之一，就是意识到高速缓存存在的应用程序员可以利用高速缓存将他们程序的性能提高一个数量级。你将在第 6 章学习这些重要的设备以及如何利用它们。

1.6　存储设备形成层次结构

在处理器和一个又大又慢的设备（例如主存）之间插入一个更小更快的存储设备（例如高速缓存）的想法已经成为了一个普遍的观念。实际上，每个计算机系统中的存储设备都被组织成了一个*存储器层次结构*，如图 1-9 所示。在这个层次结构中，从上至下，设备变得访问速度越来越慢、容量越来越大，并且每字节的造价也越来越便宜。寄存器文件在层次结构中位于最顶部，也就是第 0 级或记为 L0。这里我们展示的是三层高速缓存 L1 到 L3，占据存储器层次结构的第 1

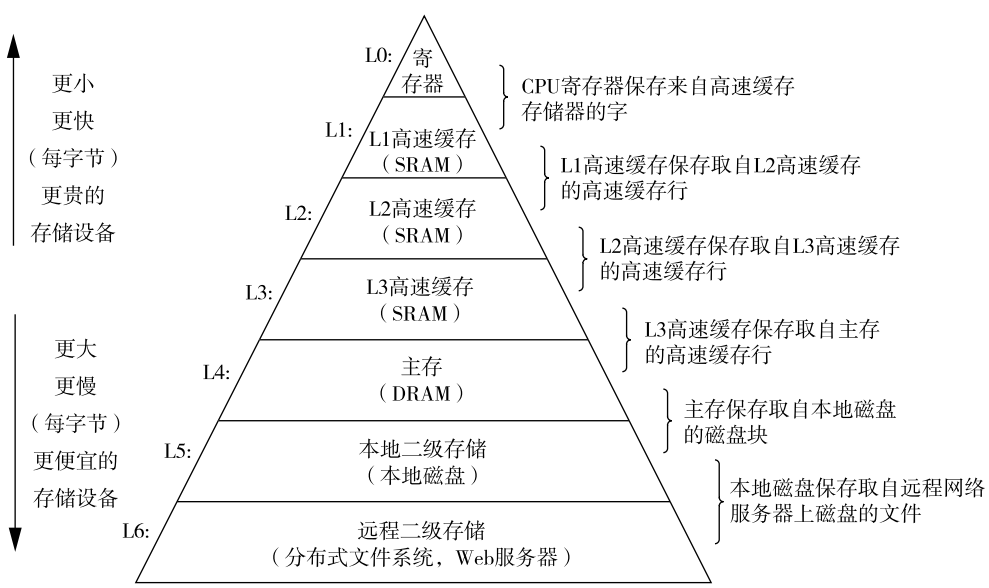

图 1-9　一个存储器层次结构的示例

层到第 3 层。主存在第 4 层，以此类推。

存储器层次结构的主要思想是一层上的存储器作为低一层存储器的高速缓存。因此，寄存器文件就是 L1 的高速缓存，L1 是 L2 的高速缓存，L2 是 L3 的高速缓存，L3 是主存的高速缓存，而主存又是磁盘的高速缓存。在某些具有分布式文件系统的网络系统中，本地磁盘就是存储在其他系统中磁盘上的数据的高速缓存。

正如可以运用不同的高速缓存的知识来提高程序性能一样，程序员同样可以利用对整个存储器层次结构的理解来提高程序性能。第 6 章将更详细地讨论这个问题。

1.7　操作系统管理硬件

我们继续讨论 hello 程序的例子。当外壳加载和运行 hello 程序，以及 hello 程序输出自己的消息时，外壳和 hello 程序都没有直接访问键盘、显示器、磁盘或者主存。取而代之的是，它们依靠操作系统提供的服务。我们可以把操作系统看成是应用程序和硬件之间插入的一层软件，如图 1-10 所示。所有应用程序对硬件的操作尝试都必须通过操作系统。

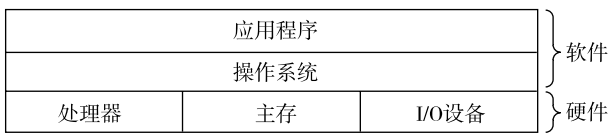

图 1-10　计算机系统的分层视图

操作系统有两个基本功能：1）防止硬件被失控的应用程序滥用。2）向应用程序提供简单一致的机制来控制复杂而又通常大相径庭的低级硬件设备。操作系统通过几个基本的抽象概念（进程、虚拟存储器和文件）来实现这两个功能。如图 1-11 所示，文件是对 I/O 设备的抽象表示，虚拟存储器是对主存和磁盘 I/O 设备的抽象表示，进程则是对处理器、主存和 I/O 设备的抽象表示。我们将依次讨论每种抽象表示。

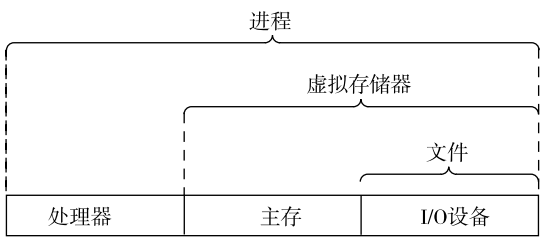

图 1-11　操作系统提供的抽象表示

Unix 和 Posix

20 世纪 60 年代是大型、复杂操作系统盛行的年代，如 IBM 的 OS/360 和 Honeywell 的 Multics 系统。OS/360 是历史上最成功的软件项目之一，而 Multics 虽然持续存在了多年，却从来没有被广泛应用。贝尔实验室曾经是 Multics 项目的最初参与者，但是考虑到该项目的复杂性和缺乏进展于 1969 年退出。鉴于 Multics 项目不愉快的经历，一组贝尔实验室的研究人员——Ken Thompson、Dennis Ritchie、Doug Mcllroy 和 Joe Ossanna，从 1969 年开始在 DEC PDP-7 计算机上完全用机器语言编写了一个简单得多的操作系统。这个新系统中的很多思想，如层次文件系统、作为用户级进程的外壳概念，都是来自于 Multics，只不过在一个更小、更简单的程序包里实现。1970 年，Brian Kernighan 给新系统命名为“Unix”，这也是一个双关语，暗指“Multics”的复杂性。1973 年用 C 语言重新编写其内核，1974 年，Unix 开始正式对外发布 [89]。

贝尔实验室以慷慨的条件向学校提供源代码，所以 Unix 在大专院校里获得了很多支持并继续发展。最有影响的工作是 20 世纪 70 年代晚期到 80 年代早期，在美国加州大学伯克利分校，伯克利的研究人员在一系列发布版本中增加了虚拟存储器和 Internet 协议，称为 Unix 4.xBSD（Berkeley Software Distribution）。与此同时，贝尔实验室也在发布自己的版本，即 System V Unix。其他厂商的版本，如 Sun Microsystems 的 Solaris 系统，则是从这些最初的 BSD 和 System

V 版本中衍生而来。

20 世纪 80 年代中期，Unix 厂商试图通过加入新的、往往不兼容的特性来使它们的程序与众不同，麻烦也就随之而来了。为了阻止这种趋势，IEEE（电气和电子工程师协会）开始努力标准化 Unix 的开发，后来由 Richard Stallman 命名为“Posix”。结果就得到了一系列的标准，称做 Posix 标准。这套标准涵盖了很多方面，比如 Unix 系统调用的 C 语言接口、外壳程序和工具、线程及网络编程。随着越来越多的系统日益完全地遵从 Posix 标准，Unix 版本之间的差异正在逐渐消失。

1.7.1　进程

像 hello 这样的程序在现代系统上运行时，操作系统会提供一种假象，就好像系统上只有这个程序在运行，看上去只有这个程序在使用处理器、主存和 I/O 设备。处理器看上去就像在不间断地一条接一条地执行程序中的指令，即该程序的代码和数据是系统存储器中唯一的对象。这些假象是通过进程的概念来实现的，进程是计算机科学中最重要和最成功的概念之一。

进程是操作系统对一个正在运行的程序的一种抽象。在一个系统上可以同时运行多个进程，而每个进程都好像在独占地使用硬件。而并发运行，则是说一个进程的指令和另一个进程的指令是交错执行的。在大多数系统中，需要运行的进程数是多于可以运行它们的 CPU 个数的。传统系统在一个时刻只能执行一个程序，而先进的多核处理器同时能够执行多个程序。无论是在单核还是多核系统中，一个 CPU 看上去都像是在并发地执行多个进程，这是通过处理器在进程间切换来实现的。操作系统实现这种交错执行的机制称为上下文切换。为了简化讨论，我们只考虑包含一个 CPU 的单处理器系统的情况。我们会在 1.9.1 节讨论多处理器系统。

操作系统保持跟踪进程运行所需的所有状态信息。这种状态，也就是上下文，它包括许多信息，例如 PC 和寄存器文件的当前值，以及主存的内容。在任何一个时刻，单处理器系统都只能执行一个进程的代码。当操作系统决定要把控制权从当前进程转移到某个新进程时，就会进行上下文切换，即保存当前进程的上下文、恢复新进程的上下文，然后将控制权传递到新进程。新进程就会从上次停止的地方开始。图 1-12 展示了示例 hello 程序运行场景的基本理念。

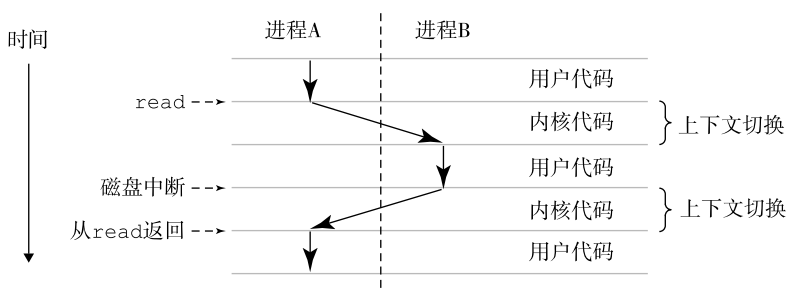

图 1-12　进程的上下文切换

示例场景中有两个并发的进程：外壳进程和 hello 进程。起初，只有外壳进程在运行，即等待命令行上的输入。当我们让它运行 hello 程序时，外壳通过调用一个专门的函数，即系统调用，来执行我们的请求，系统调用会将控制权传递给操作系统。操作系统保存外壳进程的上下文，创建一个新的 hello 进程及其上下文，然后将控制权传递给新的 hello 进程。hello 进程终止后，操作系统恢复外壳进程的上下文，并将控制权传回给它，外壳进程将继续等待下一个命令行输入。

实现进程这个抽象概念需要低级硬件和操作系统软件之间的紧密合作。我们将在第 8 章揭示这项工作的原理，以及应用程序是如何创建和控制它们的进程的。

1.7.2　线程

尽管通常我们认为一个进程只有单一的控制流，但是在现代系统中，一个进程实际上可以由多个称为*线程*的执行单元组成，每个线程都运行在进程的上下文中，并共享同样的代码和全局数据。由于网络服务器对并行处理的需求，线程成为越来越重要的编程模型，因为多线程之间比多进程之间更容易共享数据，也因为线程一般来说都比进程更高效。当有多处理器可用的时候，多线程也是一种使程序可以更快运行的方法，我们将在 1.9.1 节讨论这个问题。你将在第 12 章学习到并发的基本概念，以及如何写线程化的程序。

1.7.3　虚拟存储器

*虚拟存储器*是一个抽象概念，它为每个进程提供了一个假象，即每个进程都在独占地使用主存。每个进程看到的是一致的存储器，称为*虚拟地址空间*。图 1-13 所示的是 Linux 进程的虚拟地址空间（其他 Unix 系统的设计也与此类似）。在 Linux 中，地址空间最上面的区域是为操作系统中的代码和数据保留的，这对所有进程来说都是一样的。地址空间的底部区域存放用户进程定义的代码和数据。请注意，图中的地址是从下往上增大的。

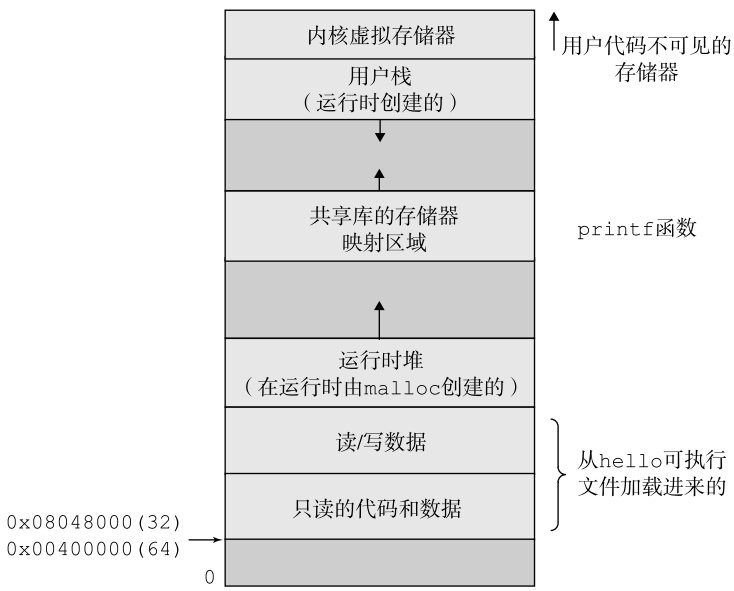

图 1-13　进程的虚拟地址空间

每个进程看到的虚拟地址空间由大量准确定义的区构成，每个区都有专门的功能。本书的后续章节将介绍更多的有关这些区的知识，但是先简单了解每一个区将是非常有益的。我们是从最低的地址开始，逐步向上介绍。

- *程序代码和数据*。对于所有的进程来说，代码是从同一固定地址开始，紧接着的是和 C 全局变量相对应的数据位置。代码和数据区是直接按照可执行目标文件的内容初始化的，在示例中就是可执行文件 `hello`。第 7 章我们研究链接和加载，你将会学习到更多有关地址空间的内容。
- *堆*。代码和数据区后紧随着的是运行时堆。代码和数据区是在进程一开始运行时就被规定了大小，与此不同，当调用如 `malloc` 和 `free` 这样的 C 标准库函数时，堆可以在运行时动态地扩展和收缩。第 9 章学习管理虚拟存储器时，我们将更详细地研究堆。
- *共享库*。大约在地址空间的中间部分是一块用来存放像 C 标准库和数学库这样共享库的代码和数据的区域。共享库的概念非常强大，也相当难懂。第 7 章介绍动态链接时，我们将

学习共享库是如何工作的。

- 栈。位于用户虚拟地址空间顶部的是用户栈，编译器用它来实现函数调用。和堆一样，用户栈在程序执行期间可以动态地扩展和收缩。特别是每次我们调用一个函数时，栈就会增长；从一个函数返回时，栈就会收缩。在第 3 章，你将学习编译器是如何使用栈的。
- 内核虚拟存储器。内核总是驻留在内存中，是操作系统的一部分。地址空间顶部的区域是为内核保留的，不允许应用程序读写这个区域的内容或者直接调用内核代码定义的函数。

虚拟存储器的运作需要硬件和操作系统软件之间精密复杂的交互，包括对处理器生成的每个地址的硬件翻译。其基本思想是把一个进程虚拟存储器的内容存储在磁盘上，然后用主存作为磁盘的高速缓存。第 9 章将解释虚拟存储器如何工作，以及它为什么对现代系统的运行如此重要。

1.7.4　文件

文件就是字节序列，仅此而已。每个 I/O 设备，包括磁盘、键盘、显示器，甚至网络，都可以视为文件。系统中的所有输入输出都是通过使用一小组称为 Unix I/O 的系统函数调用读写文件来实现的。

文件这个简单而精致的概念其内涵是非常丰富的，因为它向应用程序提供了一个统一的视角，来看待系统中可能含有的所有各式各样的 I/O 设备。例如，处理磁盘文件内容的应用程序员非常幸福，因为他们无需了解具体的磁盘技术。进一步说，同一个程序可以在使用不同磁盘技术的不同系统上运行。第 10 章将介绍 Unix I/O。

Linux 项目

1991 年 8 月，芬兰研究生 Linus Torvalds 谨慎地发布了一个新的类 Unix 的操作系统内核，内容如下：

来自：torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)

新闻组：comp.os.minix

主题：在 minix 中你最想看到什么？

摘要：关于我的新操作系统的小调查

时间：1991 年 8 月 25 日 20:57:08 格林尼治时间

每个使用 minix 的朋友，你们好。

我正在做一个（免费的）用在 386（486）AT 上的操作系统（只是业余爱好，它不会像 GNU 那样庞大和专业）。这个想法从 4 月份起就开始酝酿，现在快要完成了。我希望得到各位对 minix 的任何反馈意见，因为我的操作系统在某些方面是与它相类似的（其中包括相同的文件系统的物理设计（因为某些实际的原因））。

我现在已经移植了 bash（1.08）和 gcc（1.40），并且看上去能运行。这意味着我需要用几个月的时间使它变得更实用一些，并且我想知道大多数人想要的特性。欢迎提出任何建议，但是我无法保证都能实现。: -)

Linus (torvalds@kruuna.helsinki.fi)

接下来，如他们所说，这就成为了历史。Linux 逐渐发展成为一个技术和文化现象。通过结合 GNU 项目的力量，Linux 项目发展成为一个完整的、符合 Posix 标准的 Unix 操作系统的版本，包括内核和所有支撑的基础设施。从手持设备到大型计算机，Linux 在范围如此广泛的计算机上得到了应用。IBM 的一个工作组甚至把 Linux 移植到了一块腕表中！

1.8　系统之间利用网络通信

系统漫游至此，我们一直是把系统视为一个孤立的硬件和软件的集合体。实际上，现代系

统经常通过网络和其他系统连接到一起。从一个单独的系统来看，网络可视为一个 I/O 设备，如图 1-14 所示。当系统从主存将一串字节复制到网络适配器时，数据流经过网络到达另一台机器，而不是其他地方，例如本地磁盘驱动器。相似地，系统可以读取从其他机器发送来的数据，并把数据复制到自己的主存。

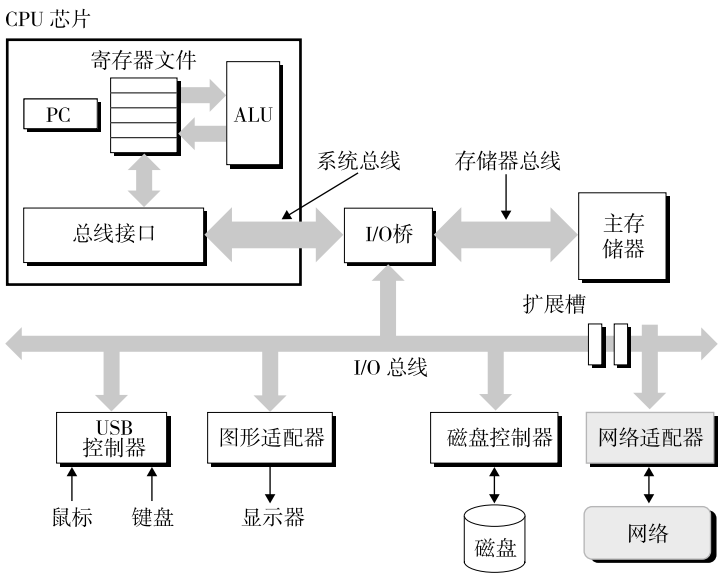

图 1-14　网络也是一种 I/O 设备

随着 Internet 这样的全球网络的出现，将一台主机的信息复制到另外一台主机已经成为计算机系统最重要的用途之一。例如，电子邮件、即时通信、万维网、FTP 和 telnet 这样的应用都是基于网络复制信息的功能。

继续讨论我们的 hello 示例，我们可以使用熟悉的 telnet 应用在一个远程主机上运行 hello 程序。假设用本地主机上的 telnet 客户端连接远程主机上的 telnet 服务器。在我们登录到远程主机并运行外壳后，远端的外壳就在等待接收输入命令。从这点开始，远程运行 hello 程序包括（如图 1-15 所示）的五个基本步骤。

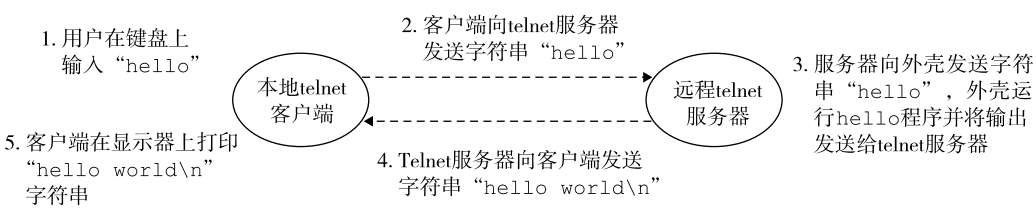

图 1-15　利用 telnet 通过网络远程运行 hello 程序

当我们在 telnet 客户端键入“hello”字符串并敲下回车键后，客户端软件就会将这个字符串发送到 telnet 的服务器。telnet 服务器从网络上接收到这个字符串后，会把它传递给远端外壳程序。接下来，远端外壳运行 hello 程序，并将输出行返回给 telnet 服务器。最后，telnet 服务器通过网络把输出串转发给 telnet 客户端，客户端就将输出串输出到我们的本地终端上。

这种客户端和服务器之间交互的类型在所有的网络应用中是非常典型的。在第 11 章，你将学到如何构造网络应用程序，并利用这些知识创建一个简单的 Web 服务器。

1.9　重要主题

在此，总结一下我们旋风式的系统漫游。这次讨论得出一个很重要的观点，那就是系统不仅仅只是硬件。系统是硬件和系统软件互相交织的集合体，它们必须共同协作以达到运行应用程序的最终目的。本书的余下部分会讲述硬件和软件的详细内容，通过了解这些详细内容，你可以写出更快速、更可靠和更安全的程序。

我们在此强调几个贯穿计算机系统所有方面的重要概念作为本章的结束。我们还会在本书中的多处讨论这些概念的重要性。

1.9.1　并发和并行

数字计算机的整个历史中，有两个需求是驱动进步的持续动力：一个是我们想要计算机做得更多，另一个是我们想要计算机运行得更快。当处理器同时能够做更多事情时，这两个因素都会改进。我们用的术语*并发*（concurrency）是一个通用的概念，指一个同时具有多个活动的系统；而术语*并行*（parallelism）指的是用并发使一个系统运行得更快。并行可以在计算机系统的多个抽象层次上运用。在此，我们按照系统层次结构中由高到低的顺序重点强调三个层次。

1. 线程级并发

构建进程这个抽象，我们能够设计出同时执行多个程序的系统，这就导致了并发。使用线程，我们甚至能够在一个进程中执行多个控制流。从 20 世纪 60 年代初期出现时间共享以来，计算机系统中就开始有了对并发执行的支持。传统意义上，这种并发执行只是模拟出来的，是通过使一台计算机在它正在执行的进程间快速切换的方式实现的，就好像一个杂技演员保持多个球在空中飞舞。这种并发形式允许多个用户同时与系统交互，例如，当许多人想要从一个 Web 服务器获取页面时。它还允许一个用户同时从事多个任务，例如，在一个窗口中开启 Web 浏览器，在另一窗口中运行字处理器，同时又播放音乐。在以前，即使处理器必须在多个任务间切换，大多数实际的计算也都是由一个处理器来完成的。这种配置称为*单处理器系统*。

当构建一个由单操作系统内核控制的多处理器组成的系统时，我们就得到了一个*多处理器系统*。其实从 20 世纪 80 年代开始，在大规模的计算中就采用了这种系统，但是直到最近，随着*多核*处理器和*超线程*（hyperthreading）的出现，这种系统才变得常见。图 1-16 列出了这些不同处理器类型的分类。

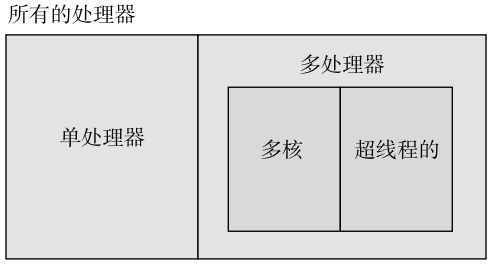

图 1-16　不同的处理器配置分类。随着多核处理器和超线程的出现，多处理器变得普遍了

多核处理器是将多个 CPU（称为“核”）集成到一个集成电路芯片上。图 1-17 描述的是 Intel Core i7 处理器的组织结构，其中微处理器芯片有 4 个 CPU 核，每个核都有自己的 L1 和 L2 高速缓存，但是它们共享更高层次的高速缓存，以及到主存的接口。工业界的专家预言他们能够将几十个、最终会是上百个核做到一个芯片上。

超线程，有时称为*同时多线程*（simultaneous multi-threading），是一项允许一个 CPU 执行多个控制流的技术。它涉及 CPU 某些硬件有多个备份，比如程序计数器和寄存器文件；而其他的硬件部分只有一份，比如执行浮点算术运算的单元。常规的处理器需要大约 20 000 个时钟周期做不同线程间的转换，而超线程的处理器可以在单个周期的基础上决定要执行哪一个线程。这使得 CPU 能够更好地利用它的处理资源。例如，假设一个线程必须等到某些数据被装载到高速缓存中，那 CPU 就可以继续去执行另一个线程。举例来说，Intel Core i7 处理器可以让一个核执行两个线程，所以一个 4 核的系统实际上可以并行地执行 8 个线程。

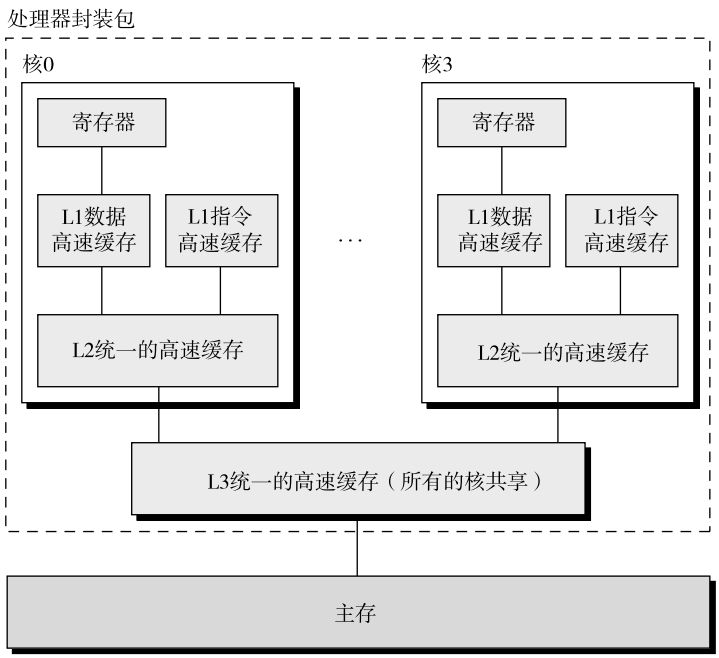

图 1-17 Intel Core i7 的组织结构。4 个处理器核集成到一个芯片上

多处理器的使用可以从两个方面提高系统性能。首先，它减少了在执行多个任务时模拟并发的需要。正如前面提到的，即使是只有一个用户使用的个人计算机也需要并发地执行多个活动。其次，它可以使应用程序运行得更快。当然，这必须要求程序是以多线程方式来书写的，这些线程可以并行地高效执行。因此，虽然并发原理的形成和研究已经超过 50 年的时间了，但是直到多核和超线程系统的出现才极大地激发了人们的一种愿望，即找到书写应用程序的方法利用硬件开发线程级并行性。第 12 章将更深入地探讨并发，以及使用并发来提供处理器资源的共享，使得程序的执行允许有更多的并行。

2. 指令级并行

在较低的抽象层次上，现代处理器可以同时执行多条指令的属性称为*指令级并行*。早期的微处理器，如 1978 年的 Intel 8086，需要多个（通常是 3 ～ 10 个）时钟周期来执行一条指令。比较先进的处理器可以保持每个时钟周期 2 ～ 4 条指令的执行速率。其实每条指令从开始到结束需要长得多的时间，大约 20 个或者更多的周期，但是处理器使用了非常多的聪明技巧来同时处理多达 100 条的指令。在第 4 章，我们将研究*流水线*（pipelining）的使用。在流水线中，将执行一条指令所需要的活动划分成不同的步骤，将处理器的硬件组织成一系列的阶段，每个阶段执行一个步骤。这些阶段可以并行地操作，用来处理不同指令的不同部分。我们会看到一个相当简单的硬件设计，它能够达到接近于一个时钟周期一条指令的执行速率。

如果处理器可以达到比一个周期一条指令更快的执行速率，就称之为*超标量*（superscalar）处理器。大多数现代处理器都支持超标量操作。第 5 章，将介绍超标量处理器的高级模型。应用程序员可以用这个模型来理解他们程序的性能。然后，他们就能写出拥有更高程度的指令级并行性的程序代码，因而也运行得更快。

3. 单指令、多数据并行

在最低层次上，许多现代处理器拥有特殊的硬件，允许一条指令产生多个可以并行执行的操作，这种方式称为*单指令、多数据*，即 SIMD 并行。例如，较新的 Intel 和 AMD 处理器都具有

并行地对 4 对单精度浮点数（C 数据类型 float）做加法的指令。

提供这些 SIMD 指令多是为了提高处理影像、声音和视频数据应用的执行速度。虽然有些编译器试图从 C 程序中自动抽取 SIMD 并行性，但是更可靠的方法是使用编译器支持的特殊向量数据类型来写程序，例如 GCC 就支持向量数据类型。作为对比较通用的程序优化讲述的补充，在网络旁注 OPT:SIMD 中描述了这种编程方式。

1.9.2　计算机系统中抽象的重要性

抽象的使用是计算机科学中最为重要的概念之一。例如，为一组函数规定一个简单的应用程序接口（API）就是一个很好的编程习惯，程序员无需了解它内部的工作便可以使用这些代码。不同的编程语言提供不同形式和等级的抽象支持，例如 Java 类的声明和 C 语言的函数原型。

我们已经介绍了计算机系统中使用的几个抽象，如图 1-18 所示。在处理器里，指令集结构提供了对实际处理器硬件的抽象。使用这个抽象，机器代码程序表现得就好像它是运行在一个一次只执行一条指令的处理器上。底层的硬件比抽象描述的要复杂精细得多，它并行地执行多条指令，但又总是与那个简单有序的模型保持一致。只要执行模型一样，不同的处理器实现也能执行同样的机器代码，而又提供不同的开销和性能。

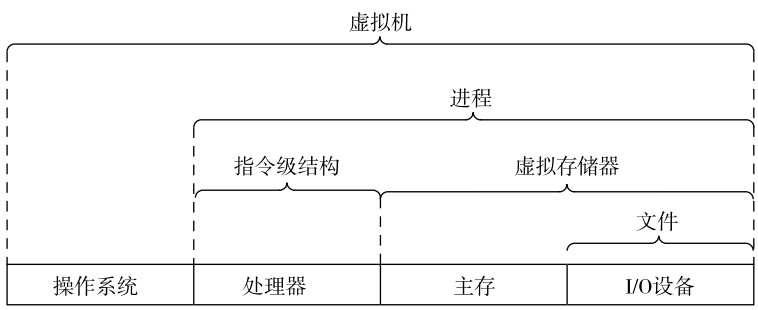

图 1-18　计算机系统提供的一些抽象

注：计算机系统中的一个重大主题就是提供不同层次的抽象表示，来隐藏实际实现的复杂性。

在学习操作系统时，我们介绍了三个抽象：文件是对 I/O 的抽象，虚拟存储器是对程序存储器的抽象，而进程是对一个正在运行的程序的抽象。我们再增加一个新的抽象：虚拟机，它提供对整个计算机（包括操作系统、处理器和程序）的抽象。虚拟机的思想是 IBM 在 20 世纪 60 年代提出来的，但是最近才显示出其管理计算机方式上的优势，因为一些计算机必须能够运行为不同操作系统（例如，Microsoft Windows、MacOS 和 Linux）或同一操作系统的不同版本而设计的程序。

在本书后续的章节中，我们会具体介绍这些抽象。

1.10　小结

计算机系统是由硬件和系统软件组成的，它们共同协作以运行应用程序。计算机内部的信息被表示为一组组的位，它们依据上下文有不同的解释方式。程序被其他程序翻译成不同的形式，开始时是 ASCII 文本，然后被编译器和链接器翻译成二进制可执行文件。

处理器读取并解释存放在主存里的二进制指令。因为计算机把大量的时间用于存储器、I/O 设备和 CPU 寄存器之间复制数据，所以将系统中的存储设备划分成层次结构——CPU 寄存器在顶部，接着是多层的硬件高速缓存存储器、DRAM 主存和磁盘存储器。在层次模型中，位于更

高层的存储设备比低层的存储设备要更快，单位比特开销也更高。层次结构中较高层次存储设备可以作为较低层次设备的高速缓存。通过理解和运用这种存储层次结构的知识，程序员可以优化 C 程序的性能。

操作系统内核是应用程序和硬件之间的媒介。它提供三个基本的抽象：1）文件是对 I/O 设备的抽象；2）虚拟存储器是对主存和磁盘的抽象；3）进程是对处理器、主存和 I/O 设备的抽象。

最后，网络提供了计算机系统之间通信的手段。从特殊系统的角度来看，网络就是一种 I/O 设备。

参考文献说明

Ritchie 写了关于早期 C 和 Unix 的有趣的第一手资料 [87，88]。Ritchie 和 Thompson 提供了最早出版的 Unix 资料 [89]。Silberschatz、Gavin 和 Gagne[98] 提供了关于 Unix 不同版本的详尽历史。GNU（www.gnu.org）和 Linux（www.linux.org）的网站上有大量的当前信息和历史资料。Posix 标准可以在线获得（www.unix.org）。