注意事项

●注意事项

本书内容仅供参考。请读者基于自身的判断使用本书。对于由使用本书而导致的所有后果，出版社和作译者概不负责。

本书内容基于截至 2018 年 1 月的信息，读者在使用本书时，可能遇到当前信息与本书有所不同的情况。

在没有事先说明的情况下，本书关于软件的描述均基于截至 2018 年 1 月的版本。有的软件可能会更新，所以实际情况可能与本书的描述有所不同。在购买本书前，请一定确认相应软件的版本号。

请在同意以上注意事项的前提下阅读本书。如果您在没有了解以上注意事项的情况下联系出版社或作译者，我们将无法进行相关处理，敬请知悉。

- 书中记载的所有产品名都是相关公司的商标或注册商标。书中省略了 ™、®、© 等标识。

第 1 章　计算机系统的概要

本章将简要说明什么是 OS，以及 OS 与硬件设备的关系。本章有很多比较抽象的描述，因此读者也可以暂时跳过本章，在需要时再回来查看相关内容。

世界上充满了各种计算机系统，比如大家身边的个人计算机、智能手机、平板电脑，以及平时不怎么接触的商用服务器等。虽然这些计算机系统上的硬件结构存在各种各样的差异，但大体上为如图 1-1 所示的结构。

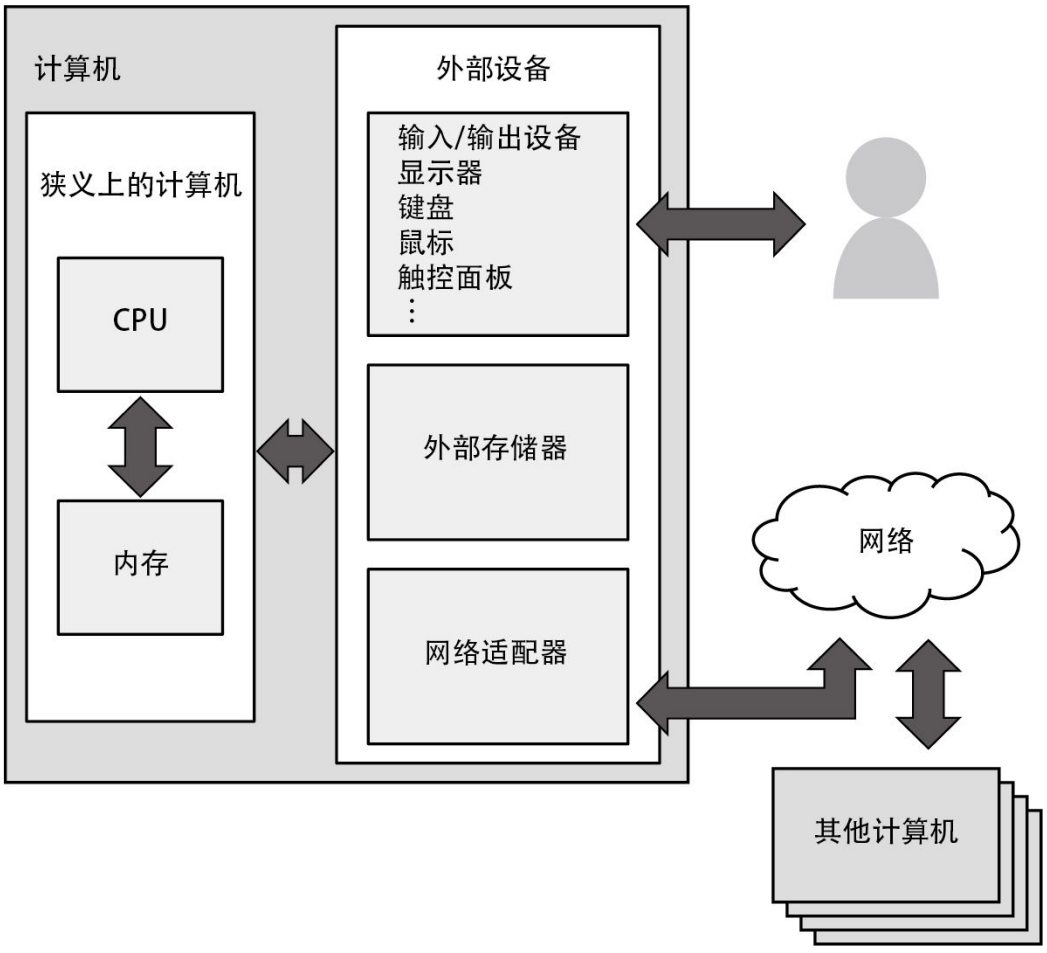

图 1-1　计算机系统的硬件结构

在计算机系统运行时，在硬件设备上会重复执行以下步骤。

① 通过输入设备或网络适配器，向计算机发起请求。

② 读取内存中的命令，并在 CPU 上执行，把结果写入负责保存数据的内存区域中。

③ 将内存中的数据写入 HDD（Hard Disk Drive，硬盘驱动器）、SDD

（Solid State Disk，固态硬盘）等存储器，或者通过网络发送给其他计算机，或者通过输出设备提供给用户。

④回到步骤①。

由这些重复执行的步骤整合而成的对用户有意义的处理，就称为程序。程序大体上分为以下几种。

- 应用程序：能让用户直接使用，为用户提供帮助的程序，例如计算机上的办公软件、智能手机和平板电脑上的应用
- 中间件：将对大部分应用程序通用的功能分离出来，以辅助应用程序运行的程序，例如 Web 服务器、数据库系统
- OS：直接控制硬件设备，同时为应用程序与中间件提供运行环境的程序，例如 Linux

以上这些程序如图 1-2 所示相互协作着运行。

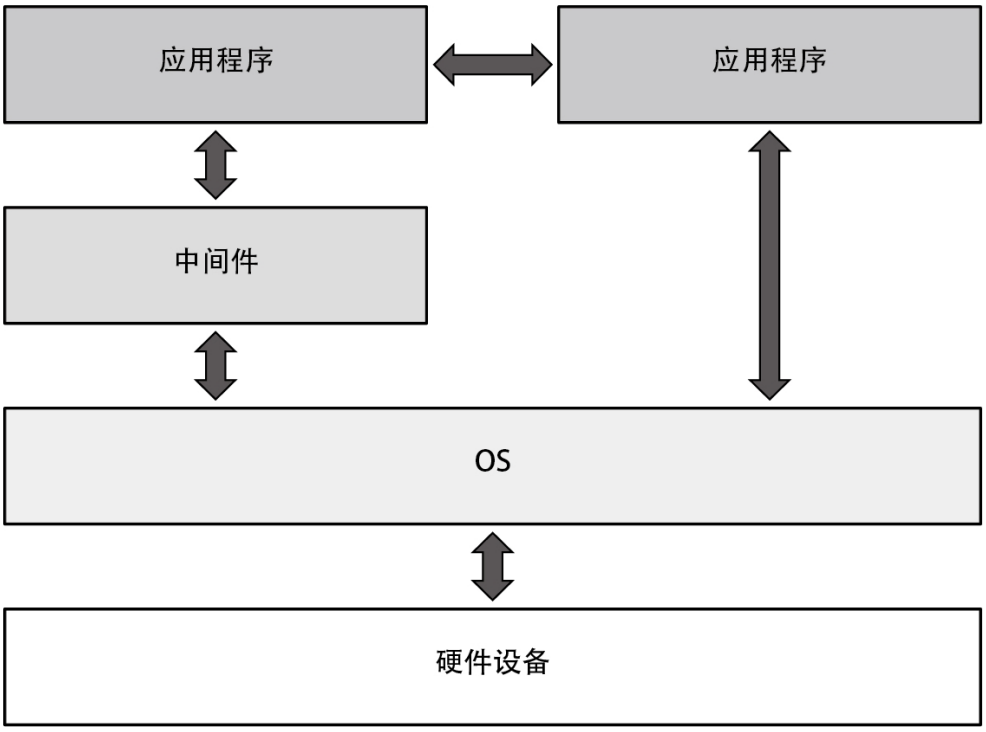

图 1-2　各种程序相互协作着运行

通常情况下，程序在 OS 上以进程为单位运行。每个程序由一个或者多个进程构成（图 1-3）。包括 Linux 在内的大部分 OS 能同时运行多个进程。

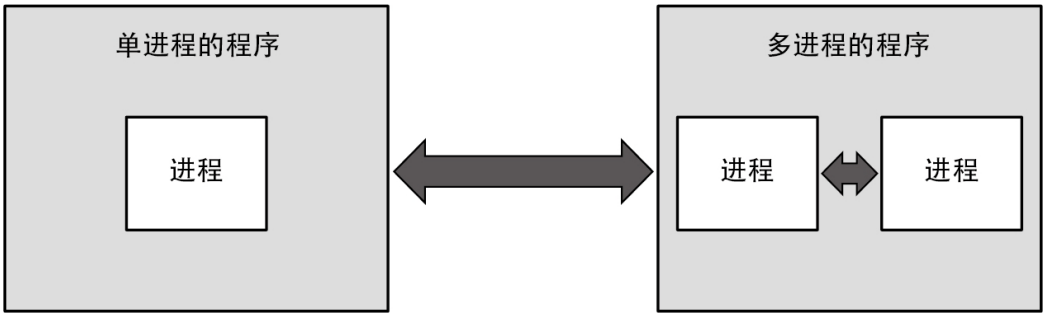

图 1-3　程序由一个或者多个进程构成

下面我们来介绍一下 Linux，以及 Linux 与硬件设备的关系。虽然下面的很多内容不仅适用于 Linux，也适用于其他 OS，但为了便于说明，在此不作具体区分。

调用外部设备（以下简称“设备”）是 Linux 的一个重要功能。如果没有 Linux 这样的 OS，就不得不为每个进程单独编写调用设备的代码（图 1-4）。

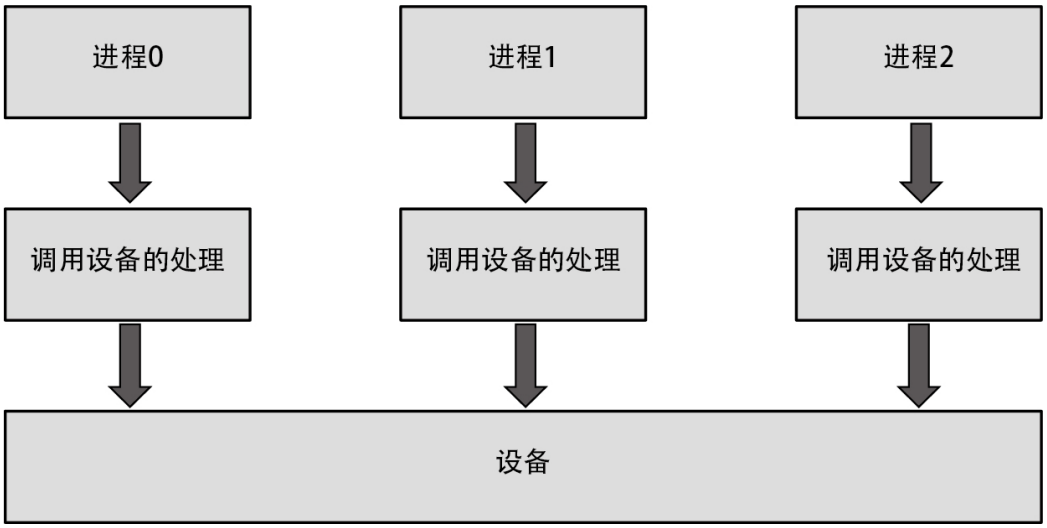

图 1-4　当不存在 OS 时的设备调用

在这种情况下，会存在以下缺点。

- 应用程序开发人员必须精通调用各种设备的方法
- 开发成本高
- 当多个进程同时调用设备时，会引起各种预料之外的问题

为了解决上述问题，Linux 把设备调用处理整合成了一个叫作**设备驱动程序**的程序，使进程通过设备驱动程序访问设备（图 1-5）。

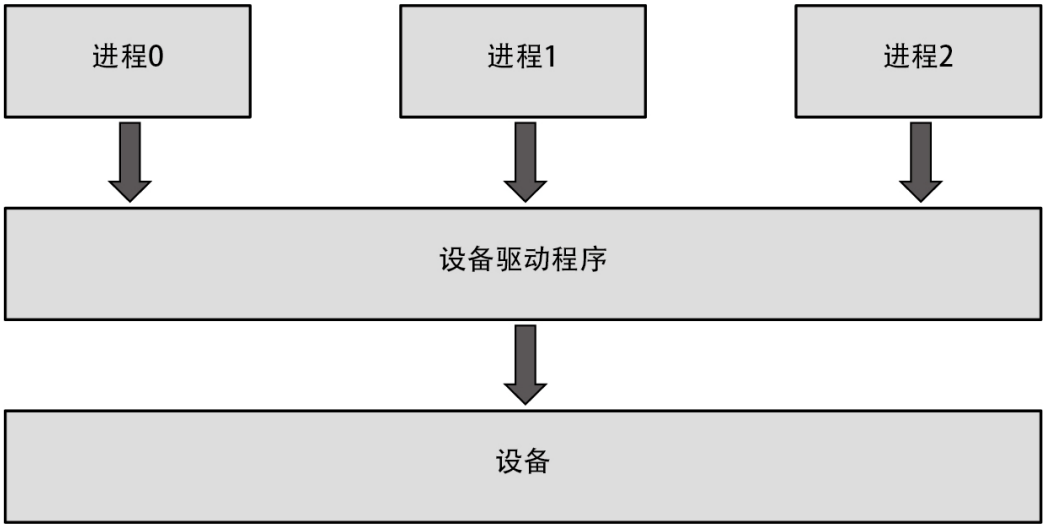

图 1-5　进程通过设备驱动程序访问设备

虽然世界上存在各种设备，但对于同一类型的设备，Linux 可以通过同一个接口进行调用（图 1-6）。

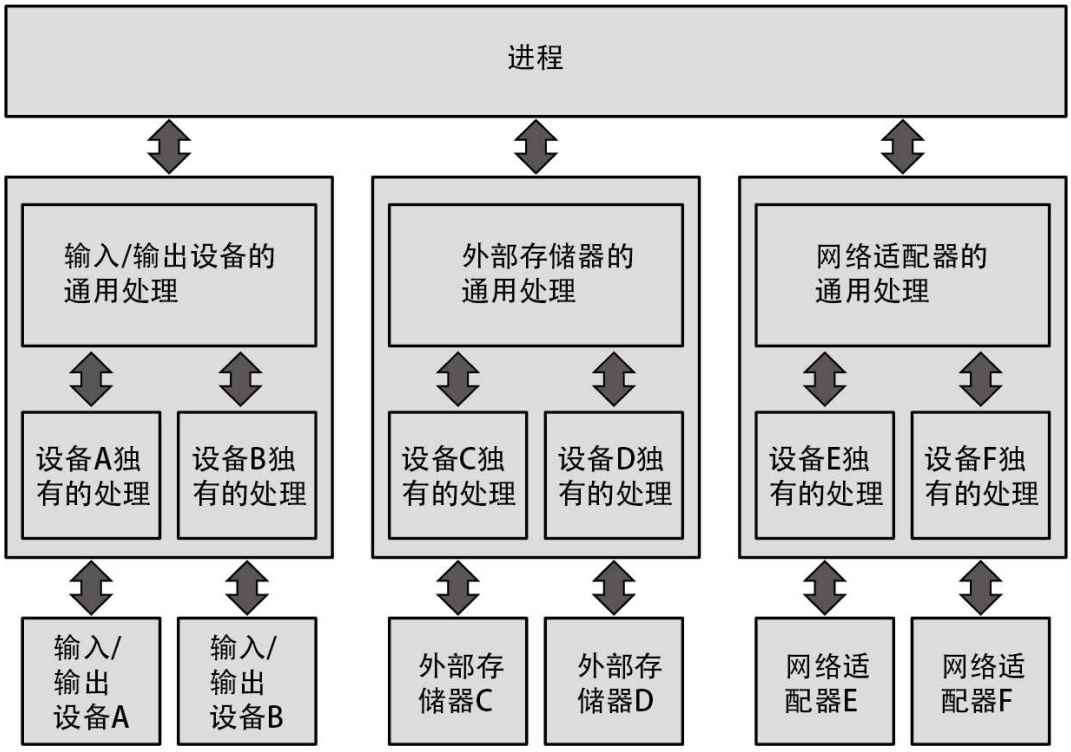

图 1-6　通过同一个接口调用同一类型的设备

在某个进程因为 Bug 或者程序员的恶意操作而违背了“通过设备驱动程序访问设备”这一规则的情况下，依然会出现多个进程同时调用设备的情况。为了避免这种情况，Linux 借助硬件，使进程无法直接访问设备。具体来说，CPU 存在**内核模式**和**用户模式**两种模式，只有处于内核模式时才允许访问设备。另外，使设备驱动程序在内核模式下运行，使进程在用户模式下运行（图 1-7）。

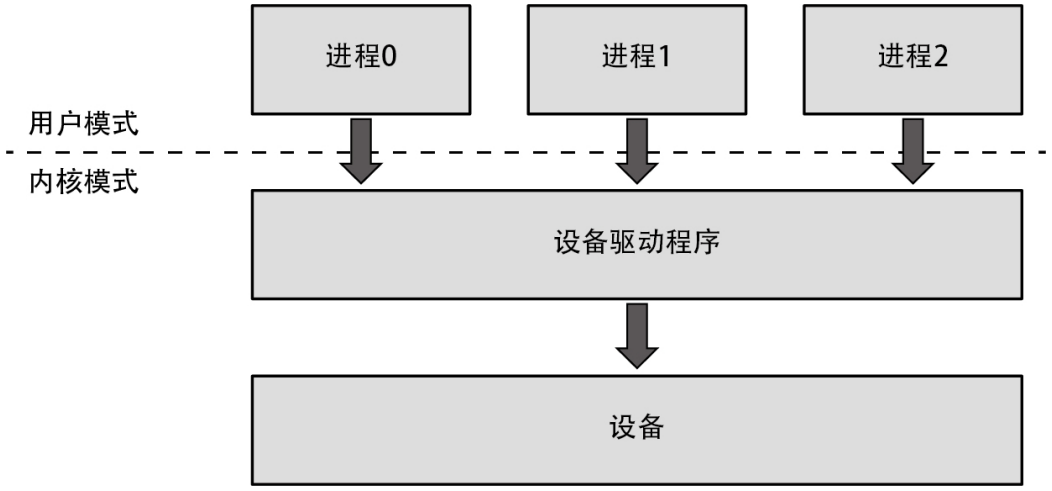

图 1-7　进程在用户模式下运行，设备驱动程序在内核模式下运行

除此之外，还有许多不应被普通进程调用的处理程序，如下所示。

- **进程管理系统**
- **进程调度器**
- **内存管理系统**

这些程序也全都在内核模式下运行。把这些在内核模式下运行的 OS 的核心处理整合在一起的程序就叫作**内核**。如果进程想要使用设备驱动程序等由内核提供的功能，就需要通过被称为**系统调用**的特殊处理来向内核发出请求。

需要指出的是，OS 并不单指内核，它是由内核与许多在用户模式下运行的程序构成的。关于 Linux 中的在用户模式下运行的功能，以及作为进程与内核的通信接口的系统调用，我们将在第 2 章具体说明。

第 3 章将对负责创建与终止进程的**进程管理系统**进行说明。

内核负责管理计算机系统上的 CPU 和内存等各种资源，然后把这些资源按需分配给在系统上运行的各个进程（图 1-8）。

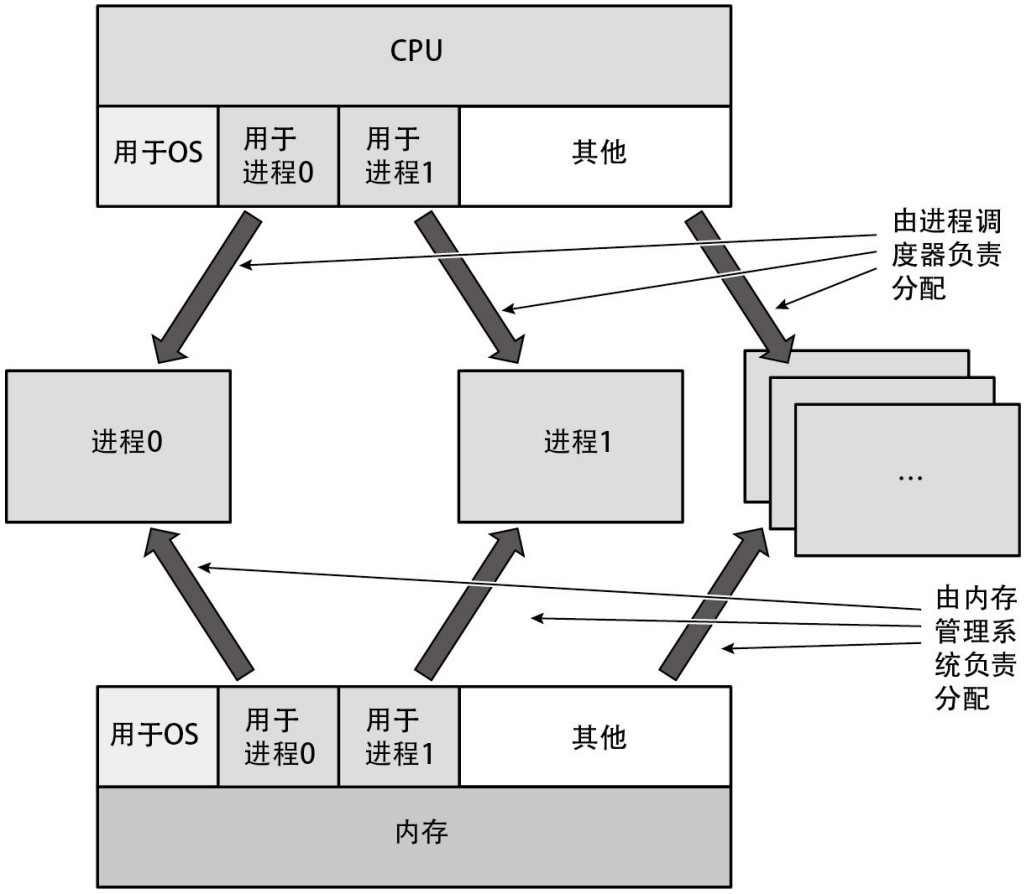

图 1-8　内核管理着 CPU 和内存等资源

图 1-8 中负责管理 CPU 资源的**进程调度器**的相关内容将在第 4 章详细说明，负责管理内存的**内存管理系统**的相关内容将在第 5 章详细说明。

在进程运行的过程中，各种数据会以内存为中心，在 CPU 上的寄存器或外部存储器等各种存储器之间进行交换（图 1-9）。这些存储器在容量、价格和访问速度等方面都有各自的优缺点，从而构成被称为**存储层次**的存储系统层次结构。从提高程序运行速度和稳定性方面来说，灵活有效地运用各种存储器是必不可少的一环。存储层次的相关内容将在第 6 章详细说明。

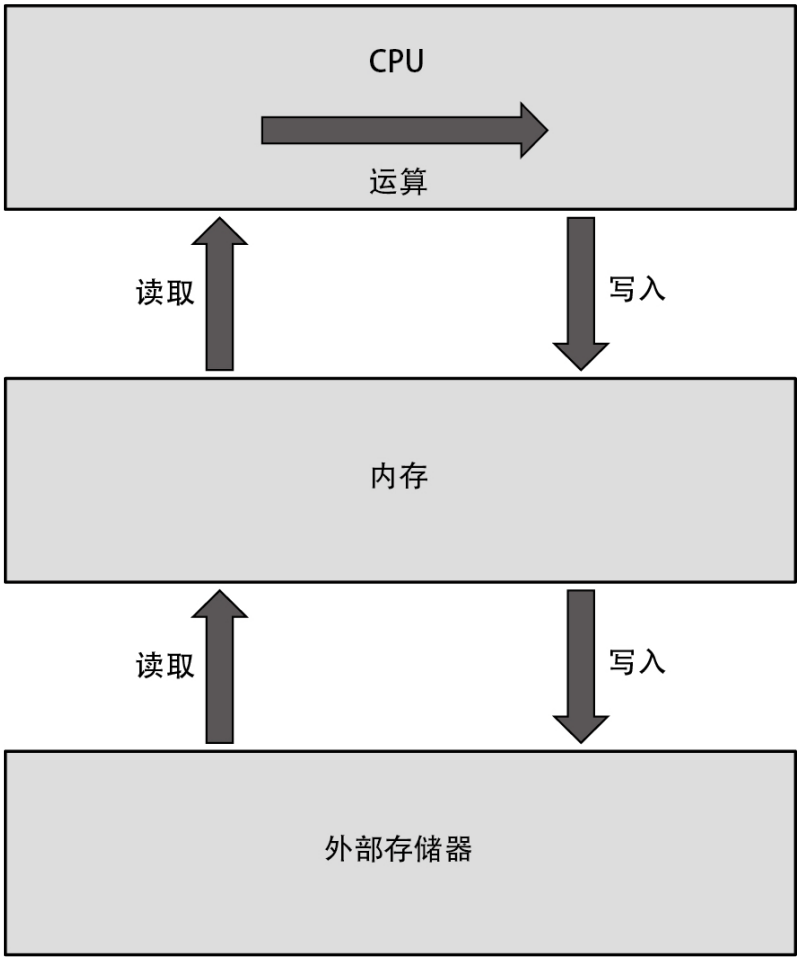

图 1-9　数据在各种存储器之间交换

虽然可以通过设备驱动程序访问外部存储器中的数据，但为了简化这一过程，通常会利用被称为**文件系统**的程序进行访问（图 1-10）。文件系统的相关内容将在第 7 章详细说明。

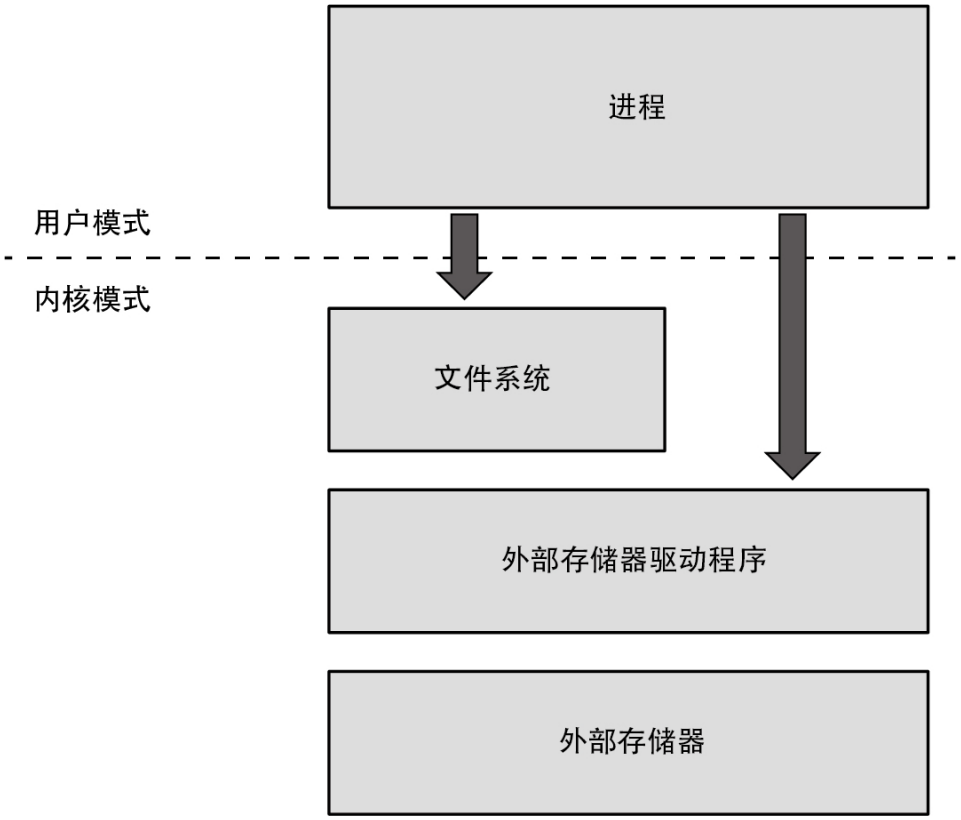

图 1-10　通常利用文件系统访问外部存储器

对于计算机系统来说，外部存储器是不可或缺的。在启动系统时，首先需要做的就是从**外部存储器**中读取 OS[1]。此外，为了防止在关闭电源时丢失系统运行期间在内存上创建的数据，必须在关闭电源前把这些数据写入外部存储器。第 8 章将详细介绍这些外部存储器的性能与特性，以及用于提高其性能的内核的功能。

[1]准确地说，在读取 OS 之前，还存在以下操作：①通过 BIOS（Basic Input Output System，基本输入 / 输出系统）或 UEFI（Unified Extensible Firmware Interface，统一可扩展固件接口）这种固化在硬件上的软件来初始化硬件设备；②运行引导程序来选择需要启动的 OS。

第 2 章　用户模式实现的功能

如第 1 章所述，OS 并非仅由内核构成，还包含许多在用户模式下运行的程序。这些程序有的以库的形式存在，有的作为单独的进程运行。这里我们先看一下计算机系统中的各种进程与 OS 的关系（图 2-1）。

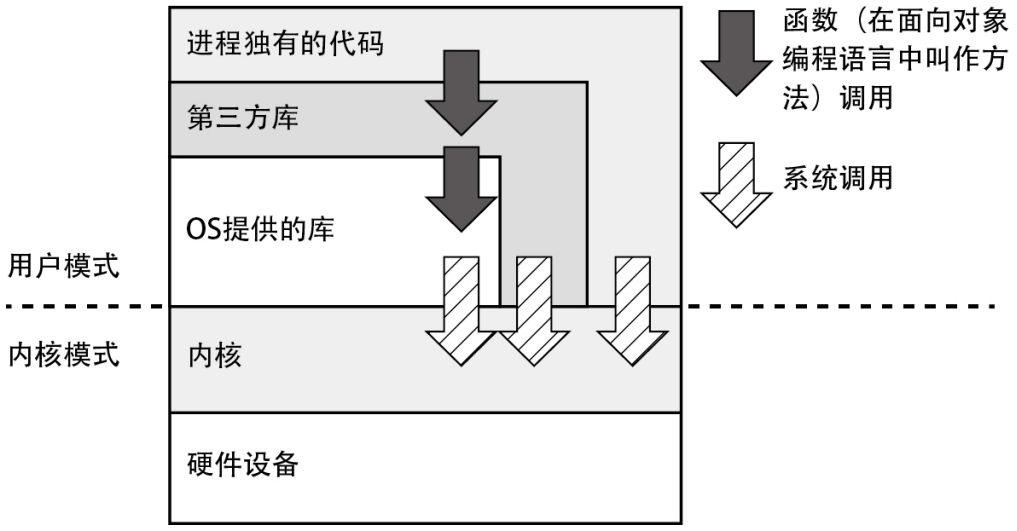

图 2-1　进程与 OS 的关系

一般来说，由在用户模式下运行的进程通过系统调用向内核发送相应的请求，其中存在进程独有的代码直接向内核发起请求的情况，也存在进程所依赖的库向内核发起请求的情况。另外，库分为 **OS 提供的库**与**第三方库**两种类型。

从整个系统来说，除了应用程序与中间件之外，OS 自身也提供了各种各样的程序。

本章后面将详细讲解系统调用、OS 提供的库和 OS 提供的程序的相关内容，以及 OS 提供这些库或程序的原因。

2.1　系统调用

如前所述，进程在执行创建进程、操控硬件等依赖于内核的处理时，必须通过系统调用向内核发起请求。系统调用的种类如下。

- 进程控制（创建和删除）
- 内存管理（分配和释放）
- 进程间通信
- 网络管理
- 文件系统操作
- 文件操作（访问设备）

关于这些系统调用，我们接下来会根据需要进行说明。

● CPU 的模式切换

系统调用需要通过执行特殊的 CPU 命令来发起。通常进程运行在用户模式下，当通过系统调用向内核发送请求时，CPU 会发生名为**中断**的事件。这时，CPU 将从用户模式切换到内核模式，然后根据请求内容进行相应的处理。当内核处理完所有系统调用后，将重新回到用户模式，继续运行进程（图 2-2）。

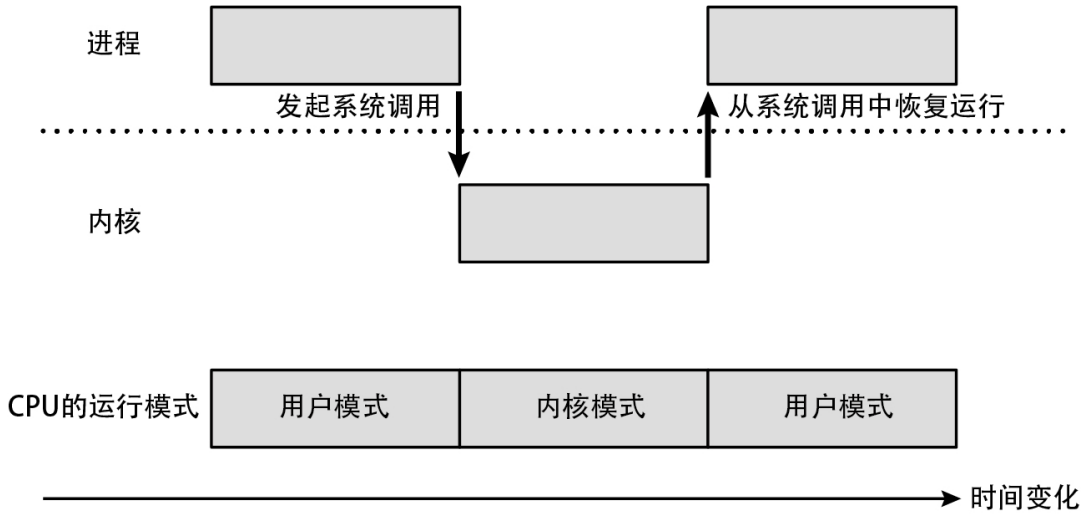

图 2-2　CPU 的模式切换

内核在开始进行处理时验证来自进程的请求是否合理（例如，请求的内存量是否大于系统所拥有的内存量等）。如果请求不合理，系统调用将执行失败。

需要注意的是，并不存在用户进程绕过系统调用而直接切换 CPU 运行模式的方法（假如有，内核就失去存在的意义了）。

● 发起系统调用时的情形

如果想了解进程究竟发起了哪些系统调用，可以通过 strace 命令对进程进行追踪。例如，通过 strace 命令运行一个输出消息的程序，也就是大家常说的 hello world 程序（代码清单 2-1）。

代码清单 2-1　hello world 程序（hello.c）

```
#include <stdio.h>
int main(void)
{
    puts("hello world");
    return 0;
}
```

首先，不使用 strace 命令，尝试编译并运行一遍。

```
$ cc -o hello hello.c
$ ./hello
hello world
$
```

能在命令行中输出 hello world 就可以。接下来，通过 strace 命令来看看这个程序会发起哪些系统调用。此外，为了防止 strace 命令输出的数据与程序本身的输出混杂在一起，在使用 strace 命令时，我们加上 -o 选项，令其输出保存到指定的文件内。

```
$ strace -o hello.log ./hello
hello world
$
```

程序和上一次运行时一样，输出消息后就结束运行了。接下来，打开 hello.log 文件，看看 strace 命令的运行结果[1]。

[1]运行结果中的路径等因实际运行环境而不同。

```
$ cat hello.log
execve("./hello", ["./hello"], [/* 28 vars */]) = 0
brk(NULL)                                       = 0x917000
access("etcld.so.nohwcap". F_OK)        = -1 ENOENT  ↲
(No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|   ↲
MAP_ANONYMOUS, -1, 0)                           = 0x7f3ff46c2000
access("etcld.so.preload", R_OK)        = -1 ENOENT  ↲
(No such file or directory)
...
brk(NULL)                                       = 0x917000
brk(0x938000)                                   = 0x938000
write(1, "hello world\n", 12)                   = 12      ←①
exit_group(0)                                   = ?
+++ exited with 0 +++
$
```

strace 的运行结果中的每一行对应一个系统调用。虽然输出的内容很多，但现在只需关注①指向的这一行。通过这一行的内容可以了解到，进程通过负责向画面或文件等输出数据的 write() 系统调用，在画面上输出了 hello world\n 这一字符串。

在笔者的计算机中，该进程总共发起了 31 个系统调用。这些系统调用大多是由在 main() 函数之前或之后执行的程序的开始处理和终止处理（OS 提供的功能的一部分）发起的，无须特别关注。

虽然测试用的 hello world 程序是用 C 语言编写的，但无论使用什么编程语言，都必须通过系统调用向内核发起请求。接下来让我们确认一下。代码清单 2-2 所示为用 Python 编写的 hello world 程序。

代码清单 2-2　用 Python 编写的 hello world 程序（hello.py）

```
print("hello world")
```

我们通过 strace 命令来运行 hello.py 程序。

```
$ strace -o hello.py.log python3 ./hello.py
hello world
$
```

然后查看追踪到的信息。

```
$ cat hello.py.log
execve("usr/binpython3", ["python3", "./hello.py"],    ↲
[/* 28 vars */])                                  = 0
brk(NULL)                                         = 0x2120000
access("etcld.so.nohwcap". F_OK)          = -1 ENOENT   ↲
(No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ ↲
ANONYMOUS, -1, 0)                                 = 0x7f6a9a36d000
access("etcld.so.preload", R_OK)          = -1 ENOENT   ↲
(No such file or directory)
...
close(3)                                          = 0
write(1, "hello world\n", 12)                     = 12     ←②
rt_sigaction(SIGINT, {SIG_DFL, [], SA_RESTORER,          ↲
0x7f6a99f3e390}, {0x63f1d0, [], SA_RESTORER,             ↲
0x7f6a99f3e390}, 8)                               = 0
exit_group(0)                                     = ?
+++ exited with 0 +++
```

这次同样输出了大量内容，但现在只需关注②指向的这一行。可以发现，与 C 语言编写的 hello world 程序一样，本程序同样发起了 write() 这个系统调用。这种情况不仅存在于 hello world 这样简单的程序中，也存在于其他复杂的程序中。

需要指出的是，在 hello.py.log 中，除了 write()，其他都是由 Python 解析器的初始化处理和终止处理所发起的系统调用。最终共发起了 705 个系统调用[2]，这比起用 C 语言进行实验时要多得多，不过我们无须关注这部分内容。

> [2]关于各个系统调用的作用，可以利用 man 命令，通过 man 2 write 这样的指令来查询。

请各位读者务必通过 strace 追踪一下各自的程序，看看都发起了哪些系统调用，这是一件很有趣的事情（请注意，如果对运行时间较长的软件使用该命令，将出现大量的输出结果）。

● 实验

sar 命令用于获取进程分别在用户模式与内核模式下运行的时间比例。我们通过每秒[3]采集一次数据，来看看每个 CPU 核心到底在运行