第 5 章 操作系统

CHAPTER 5

计算机通常包含一个或多个处理器、内存、硬盘、键盘、鼠标、显示器、网络接口以及各种输入/输出(I/O)设备。操作系统是一种特殊软件,它的任务是对计算机上的硬件资源和运行的各种应用程序进行管理,并为应用程序提供一个简洁并且一致的硬件访问接口。

5.1 操作系统简介

5.1.1 操作系统的功能

大多数人都接触过一些常用的计算机操作系统,例如 Windows、Linux 或 macOS。很多人认为操作系统就是启动计算机后所面对的图形界面或命令行环境。图形用户界面称为 GUI,命令行环境则称为 Shell,它们实际上是计算机上安装的程序,严格来说并不是操作系统的一部分。

图 5-1 描绘了计算机总体的层次结构。从图中可以看出,底层是硬件,包括 CPU(Central Processing Unit,中央处理器)、内存、硬盘、显示器、键盘、鼠标等。硬件上面是软件。绝大部分计算机有两种运行模式:内核模式和用户模式。内核模式拥有最高权限,能访问所有硬件和执行任何指令。操作系统位于软件中最基础的层面,运行在内核模式。其余软件运行在用户模式,只能执行部分指令,并且不能直接执行 I/O 指令。启动计算机后,Shell 和 GUI 这些用户接口程序位于用户模式的最底层,用户可以通过 Shell 或 GUI 启动浏览器、字处理软件等其他应用程序。这些程序都要依赖操作系统提供的服务。操作系统一般直接运行在裸机上,为所有其他软件提供基础的运行环境。

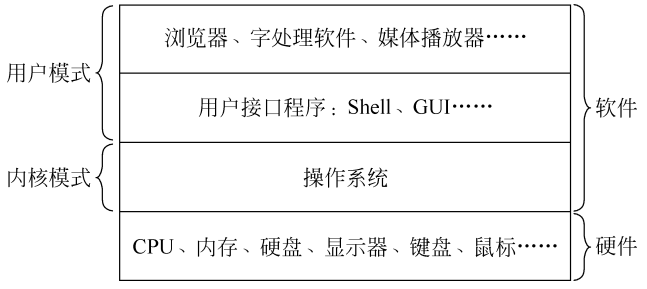


图 5-1 计算机总体的层次结构

IBM的创立者托马斯·沃森曾说："全世界只需要5台计算机就足够了"。如果没有操作系统，这句话可能是对的。但操作系统的出现使得程序员和用户对计算机的操控变得极为便利。Windows的易用性有目共睹，苹果计算机更是工业设计的永恒经典，并且这些系统还在不断演化，变得越来越美观易用。而近年的发展则让人们看到了另一个经典的工业设计典范UNIX的意外复兴。以UNIX为模板的开源操作系统Linux获得了网络服务器和嵌入式系统的很大市场份额，并且为智能手机操作系统的发展奠定了坚实的基础。

操作系统主要运行于内核模式，但也不能一概而论。有些操作系统，尤其是一些嵌入式操作系统，可能没有内核模式；有些操作系统则会让一些组件运行于用户模式。因此不能用是否运行于内核模式区分操作系统与应用软件。

从应用程序和程序员的角度来看，操作系统主要提供两大功能：一是管理各种硬件资源；二是对纷繁复杂的硬件进行抽象，为应用程序和程序员提供简洁一致的硬件访问接口。

计算机有多种类型和体系结构，CPU指令、内存组织、I/O和总线结构各不相同。不仅在机器指令层面编程是异常繁杂的工作，而且还要面对各种硬件形态。以硬盘为例，常见的就有IDE、SATA、SCSI等多种接口，其中任何一种接口在机器指令层面进行编程都是极为复杂的工作。如果编写应用程序要掌握各种接口类型硬盘的细节，将是不可能完成的任务。为了解决这个问题，操作系统提供了硬盘驱动程序，负责与硬盘进行交互，并为其他程序提供读/写硬盘数据块的接口。对于其他I/O设备，操作系统也有相应的驱动程序提供硬件控制和访问接口。

但就算是在这个层面，对于大多数应用来说还是太过繁杂。例如，存储数据的硬件可能是硬盘、闪存，甚至某个网络存储位置。如果每个有数据存取需求的应用程序都需要了解所有硬件，编程仍然是很困难的任务，而且很难应对新的硬件。因此，所有的操作系统都为数据存储介质提供了另一层抽象——文件。这样，编写应用程序时，只需要知道如何创建和读写文件即可，不用涉及具体的硬件细节，甚至不用关心使用的是哪种硬件。

从底层的角度来看，文件抽象也简化了所存储数据的模型。底层实现不用关心用户存储的是数码照片、电子邮件、歌曲、Web页面还是电影。这样文件抽象就成了上层应用和底层硬件之间的一个接口，从而大大简化了各方面的工作，使得各种复杂的系统架构的发展成为可能。

就操作系统来说，可以认为操作系统就是一个接口，隐藏复杂的硬件细节，给程序（以及程序员）提供良好、清晰、优雅、一致的抽象和接口。正因为有了这个接口作为坚实的基础，程序员才能腾出手来集中精力发展各种功能强大、界面美观的应用。例如，基于Linux的内核就发展出了Shell命令行环境、Gnome和KDE这些非常不一样的桌面环境以及安卓手机界面，各手机厂商又在此基础上进一步衍生出了各自的操作界面，所有这些都有赖于Linux内核提供的简洁而一致的接口作为支撑。

除了为应用程序提供硬件抽象，操作系统还需要对处理器、存储器、时钟、磁盘、鼠标、网络接口等各种设备进行管理。现代计算机系统允许多道程序同时运行。如果应用程序直接访问，同时运行的多道程序同时在打印机上输出信息，打印的结果很可能是一团糟。不仅打印机，应用程序在使用存储器、I/O设备以及其他资源（文件、数据库等）时，都有可能互相干扰。因此，操作系统需要对所有这些资源的使用进行管理，记录哪个程序在使用什么资源，对资源请求进行分配，评估使用代价，并且为不同的程序和用户调解互相冲突的资源请求。

有一些资源的管理方式是轮流使用，即在时间上复用资源。例如，若系统中只有一个CPU，而多个程序需要在该CPU上运行，操作系统首先把该CPU分配给某个程序，在它运行了一段时间之后，让另一个程序开始在CPU上运行，然后是下一个，如此进行下去。由于这种轮流可以进行得极快，以至于用户会感觉程序是在同时连续运行。又如打印机，当多个打印作业要在同一台打印机上打印时，就可以采取排队的方式，轮流依次打印。

有一些资源的管理方式则是空间复用。例如，让多个运行的程序同时进驻内存，甚至让运行的程序只有部分指令段和数据段进驻内存。显然，这会带来效率和内存保护等问题，这些都有赖于操作系统来解决。磁盘也是典型的空间复用的例子。磁盘可能同时为许多用户保存文件，分配磁盘空间并记录谁正在使用哪个磁盘块，也是操作系统的典型任务。

5.1.2　操作系统的发展历史

操作系统的发展与计算机体系结构的发展关联密切。不同的计算机的发展阶段对应着不同的操作系统。

1937年，阿兰·图灵在论文中提出了图灵机和可计算性的概念，为计算机的研究打下了理论基础。在第二次世界大战之后，战争的需要刺激了计算机研究的发展。英国的Colossus和美国的ENIAC相继问世。随后冯·诺依曼对计算机的研发进行了总结，提出了由存储器、运算器、控制器和输入/输出设备组成的计算机体系结构，并且建议将程序和数据都存放在存储器中。冯·诺依曼的设计成了现代计算机的蓝图。

在20世纪40年代，计算机还只能用继电器和真空管等器件制造，程序则是用机器语言编写的，甚至还需要通过将上千根电缆接到电路板的方式来设计程序。当时还没有程序设计语言，更没有操作系统。使用机器得预约，程序员在墙上的机时表上预约一段时间，然后到机房中将电路板连到计算机里，运行过程中真空管还经常被烧坏。当时计算机主要是做简单的数学运算，例如计算三角函数表和对数表，或者计算弹道等。到了20世纪50年代，开始有了穿孔卡片，这时就可以将程序写在卡片上，不需要再插拔电路板，但其他过程则依然如旧。

到了20世纪50年代，晶体管的发明使得计算机的发展突飞猛进。计算机的成本显著降低，稳定性则大为提高，终于可以离开实验室，实现产品化。操作员、程序员和维护人员之间也有了明确的分工。这些机器现在被称作大型机，需要有专门的机房，由专业操作员进行操作。只有少数大企业、要害部门或资深大学才有实力配备。程序员先将程序写在纸上(用FORTRAN语言或汇编语言)，然后制成穿孔卡片，再将卡片盒带到机房交给操作员，等待程序运行结束，输出结果。计算机运行完程序后，在打印机上输出计算结果，再由操作员将结果交还程序员。然后，操作员从送来的卡片盒中取出下一个任务输入计算机。这样做的效率很低，由于当时的计算机很昂贵，最好是能充分利用机时，因此批处理系统应运而生。

批处理的做法是首先批量地收集程序作业，然后用一台相对便宜的计算机将它们读到磁带上。在收集了一定数量的程序任务后，将磁带装到大型机的磁带机上。然后，操作员运行一个特殊的计算机操作程序，这个程序会从磁带上读入第一个程序任务并执行，并将结果输出到另一盘磁带上。每个任务结束后，操作程序会自动地从磁带上读入下一个任务并执行。当磁带上的一批程序任务全部执行结束后，操作员取下输入和输出磁带，将准备好的下一卷输入磁带装上去，再把输出磁带送去打印，早期的批处理系统如图5-2所示。

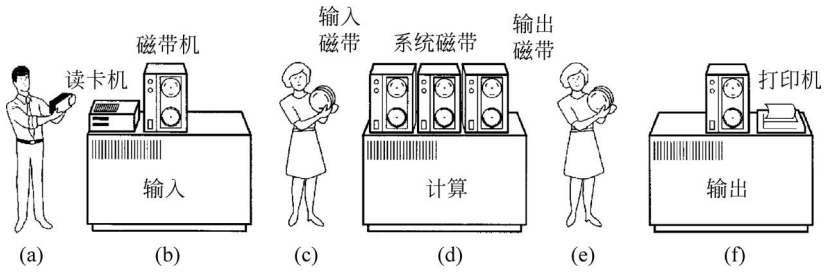


图 5-2　早期的批处理系统

典型的程序任务结构如图 5-3 所示。首先是 $JOB 卡片，它记录了程序的一些信息，例如最长运行时间、程序员的名字和账号；然后是 $FORTRAN 卡片，指令操作程序运行 FORTRAN 语言编译器；之后是待编译的源程序，然后是 $LOAD 卡片，指令操作程序加载编译好的目标程序；接着是 $RUN 卡片，指令操作程序执行该程序并使用后面的数据，$END 卡片标识任务结束。这些控制卡片起的作用类似现在的批处理命令和脚本程序的先驱，操作程序可以认为是操作系统的前身。

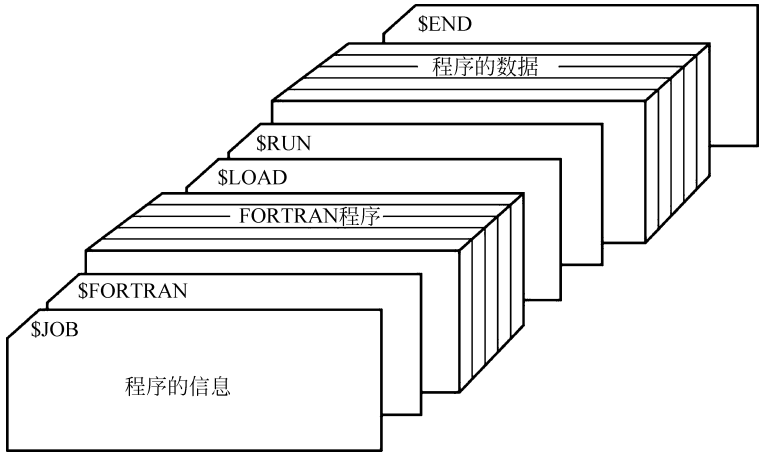


图 5-3　典型的程序任务结构

到了 20 世纪 60 年代，集成电路(Integrated Circuit，IC)技术开始成熟，采用了 IC 技术的计算机的性价比有了很大提升。大多数计算机厂商开始分化出两条不同的产品线：一条是大型科学用计算机，主要用于高计算强度的科学和工程计算；另一条是商用计算机，主要用于卡片磁带转录和打印服务。一开始这两种计算机无法兼容，但很快人们希望为小型计算机开发的程序也能放到大型机上运行。IBM 公司试图通过引入 System/360 解决这两个问题。360 是一个计算机系列，既有低档机也有高档机。这些计算机的价格和性能(速度、内存容量)各不相同，但是有相同的体系结构和指令集，因此，为一种型号编写的程序也可以在其他型号上运行。

360 系列既能适应简单的商用也能适应繁重的科学计算，同时还能挂载磁带机和打印机等各种外设。为了实现这个目的，IBM 开发了一个庞大的操作系统 OS/360，由数千名程序员编写的数百万行汇编语言代码构成，体量庞大。同时它还引入了后来很关键的一些技术，例如多道程序设计和外设联机并行操作(SPOOLing)技术。

以前的计算机没有多道程序的概念，如果当前任务要进行磁带或其他 I/O 操作，CPU 就只能空转，直至 I/O 操作完成。如果是计算强度高的科学和工程计算，I/O 操作的占比还相对较少，但如果是商业应用，打印等 I/O 操作的占比则相对较高，因此浪费的 CPU 时间也较多。多道程序的解决方案是将内存分几个部分，用于存放不同的程序任务。当一个任务在等待 I/O 操作完成时，另一个任务可以使用 CPU，从而提高 CPU 的利用率。如果在内存中同时驻留多个程序，则需要特殊的硬件来对程序进行保护，以避免程序的信息被其他程序破坏或窃取，如图 5-4 所示。

外设联机并行操作的实质是在输入/输出设备和主机之间增加缓存。例如计算机在执行当前任务时，可以让外设将下一任务的卡片读入磁盘，一旦当前任务结束，操作系统就能从磁盘读出新的任务，装进空出来的内存区域运行。在打印时也不是直接向打印机输出，而是将需要打印的文件存入特定的存储区或设备进行排队，让打印机外设依次打印队列中的输出文件，这样就提高了 I/O 速度，并且实现了多个程序任务共享外设。

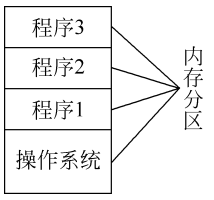


图 5-4　多道程序的概念

在多道程序设计的基础上，很快又衍生出了分时系统的技术。在分时系统中，有多台终端供用户同时登录。由于用户与计算机之间通常只有简短的命令交互，所以计算机能够为许多用户提供快速的交互式服务，同时还可以利用 CPU 的空闲时间在后台运行其他任务。当时的想法是计算机很昂贵，因此一个学校甚至整个城市都只需要一台大型计算机就够了，然后所有人都可以通过终端连接计算机。分时技术出现后，MIT、贝尔实验室和通用电气决定开发一种能够同时支持数百名分时用户的系统，该系统称作 MULTICS，设计目标是用一台计算机为波士顿地区所有用户提供计算服务。在当时看来，人手一台计算机的想法完全是不现实的。

到 20 世纪末，随着个人计算机的大行其道，计算服务的概念很快被抛弃。但是随着互联网的兴起，这个概念却以客户机/服务器架构和云计算的形式回归。在这种形式中，相对小型的计算机（包括智能手机、平板电脑等）连接到规模庞大的远程数据中心的服务器，本地计算机处理用户界面，而服务器进行计算。电子商务已经向这个方向演化了，简单的客户端连接着高性能服务器，这同 MULTICS 的设计精神非常类似。尽管 MULTICS 在商业上失败了，但 MULTICS 对随后的操作系统却产生了巨大的影响。

另一个主要的进展是小型机的崛起，其中包括 DEC 的 PDP 系列。PDP-1 计算机只有 4K 个 18 位的内存，每台售价 12 万美元，该机型非常热销。随着 PDP-1 的成功，很快有了一系列 PDP 机型，直至 PDP-11。曾参与 MULTICS 研发的贝尔实验室计算机科学家 Ken Thompson 找到了一台无人使用的 PDP-7 机器，并开始开发一个简化的 MULTICS 单用户版。他的工作导致了 UNIX 操作系统的诞生。UNIX 很快在学术界、政府部门和许多公司中流行起来。

由于源代码很容易得到，多个机构发展了自己的 UNIX 版本，并且互不兼容，从而导致了混乱。UNIX 有两个主要的版本：AT&T 的 System V 和加州大学伯克利分校的 BSD，另外还有一些小的变种。为了改变这种混乱的局面，IEEE 提出了一个 UNIX 标准，称作 POSIX，以便让程序能够在遵循标准的任何 UNIX 版本上编译运行，目前大多数 UNIX 版本都支持它。POSIX 定义了一套 UNIX 必须支持的系统调用接口。现在，很多非 UNIX 系

统也支持 POSIX 接口。

出于技术共享的目的，1991 年，芬兰的 Linus Torvalds 决定编写一个免费的类 UNIX 系统——Linux，并且在开放源码运动的相互推动下，发展成了现在世界上应用最广泛的操作系统之一。

进入小型机时代后，20 世纪 70 年代初，大规模集成电路（Large-Scale Integrated circuit，LSI）技术开始成熟，在每平方厘米的硅片芯片上可以集成数千个晶体管，个人计算机时代到来了。从体系结构上看，个人计算机（最早称为微型计算机）与 PDP-11 是一样的，但价格却相去甚远。以往，公司的一个部门或大学里的一个院系才能配备一台小型机，而微处理器却使每个人都能拥有自己的计算机。1974 年，Intel 公司发布了第一代通用 8 位 CPU 8080，并请 Gary Kildall 为它写了一个基于磁盘的操作系统 CP/M。由于 Intel 公司不认为基于磁盘的微型计算机有什么前景，所以当 Kildall 要求 CP/M 的版权时，Intel 公司同意了他的要求。Kildall 在此基础上组建了一家公司 Digital Research，进一步开发和销售 CP/M。1977 年，Digital Research 重写了 CP/M，使其可以在使用 8080、Z80 等 CPU 的多种微型计算机上运行，从此控制了微型计算机世界达 5 年之久。

20 世纪 80 年代早期，IBM 公司设计了 IBM PC，想寻找可在上面运行的软件。IBM 公司同比尔·盖茨联系有关他的 BASIC 解释器的许可证事宜，同时询问他是否知道可在 PC 上运行的操作系统。盖茨向 IBM 推荐了 CP/M。让人遗憾的是 Kildall 拒绝与 IBM 会见，从而错过了一次改变历史的机会。结果，IBM 回头询问盖茨是否可以提供一个操作系统。盖茨了解到一家本地计算机制造商有合适的操作系统 DOS。他联系对方将 DOS 买了下来，然后雇用了 DOS 的作者 Tim Paterson，根据 IBM 的要求进行了修改，修改后的版本称为 MS-DOS，并随 IBM PC 捆绑销售，从此开启了微软的商业传奇。

用于早期微型计算机的 CP/M、MS-DOS 和其他操作系统都是采用命令行形式交互、通过键盘输入命令的。事实上，早在 20 世纪 60 年代，Doug Engelbart 就发明了图形用户界面，包括窗口、图标、菜单和鼠标。这些思想被施乐帕克研究中心用在了他们所研制的机器中。1979 年，乔布斯在帕克中心参观时见到了这套系统，立即意识到它的潜在价值。乔布斯从施乐公司挖来十几位工程师进行研发。1983 年，苹果公司推出使用鼠标的苹果计算机——LISA，由于过于昂贵，在商业上失败了。1984 年，苹果公司推出了里程碑式的计算机——Macintosh，由于价格更加便宜，并且界面对用户非常友好，从而取得了巨大的成功，并成了个人计算机历史上的传奇。苹果公司后来在开发 MAC 操作系统时采用了卡内基梅隆大学开发的 UNIX 内核 Mach。因此，尽管有着截然不同的界面，MAC OS X 实际上也是基于 UNIX。

受到 Macintosh 的启发，微软也开发了基于 GUI 的系统 Windows。早期的 Windows 是以 MS-DOS 为基础，仅仅是运行在 MS-DOS 上层的一个图形环境。直到 1995 年，Windows 95 才开始独立于 MS-DOS，只是用 DOS 来运行老的 DOS 程序。

另一个微软操作系统是 Windows NT，它同 Windows 95 兼容，但是内部是完全重新编写的。Windows NT 5.0 在企业网络市场取得了成功。1999 年，Windows NT 5.0 改名为 Windows 2000，微软期望它成为 Windows 98 和 Windows NT 5.0 的接替者。后来微软又发布了 Windows 98 的升级版 Windows Me。2001 年，发布了 Windows 2000 的一个升级版 Windows XP。Windows XP 取得了成功，基本上替代了原来所有的 Windows 版本。

2007年，微软公司发布了Windows XP的后继版——Vista，没有得到市场认可。随后全新的且并不那么消耗资源的Windows 7取得了成功。后来微软又发布了它的后继者Windows 8和Windows 10。微软希望Windows 10会成为台式机、便携式电脑、笔记本电脑、平板电脑、手机、家庭影院电脑等各种设备上的主流操作系统。然而，很多计算机系统还是停留在Windows 7。

个人计算机操作系统的另一个主要竞争者是基于UNIX的Linux。Linux在网络和企业服务器以及平板电脑和智能手机等领域占有很大市场，在个人计算机上也很常见，成了替代Windows的流行选择。

FreeBSD源自Berkeley的BSD，也是一个流行的UNIX变体。所有现代Macintosh计算机都运行着FreeBSD的某个修改版。在使用高性能RISC芯片的工作站上，UNIX系统是标准配置，它的衍生系统在移动设备上被广泛使用，例如那些运行iOS和Android的设备。

几乎所有的UNIX系统都支持X Window系统（如众所周知的X11）。X11具有基本的视窗管理功能，允许用户通过鼠标创建、删除、移动和变化视窗。通常在X11之上还提供一个完整的GUI，如Gnome或KDE，使得UNIX在外观和感觉上类似于Macintosh或Windows。

操作系统的另一个发展脉络是网络操作系统和分布式操作系统。在网络操作系统中，用户能够登录到一台远程机器上并将文件从一台机器复制到另一台机器，每台计算机都运行自己本地的操作系统，并有自己的本地用户。网络操作系统需要有网络接口控制器以及相应底层软件，同时还需要一些程序来进行远程登录和远程文件访问，除此以外与单处理器的操作系统没有本质区别。

分布式操作系统则是以一种传统单处理器操作系统的形式呈现在用户面前的，尽管它实际上是由多处理器组成的。用户一般不知道自己的程序在何处运行，也不知道自己的文件存放于何处，这些应该由操作系统自动和有效地处理。真正的分布式系统与集中式系统有着本质的区别。例如，分布式系统通常允许一个应用在多台处理器上同时运行，因此，需要更复杂的处理器调度算法来获得最大的并行度优化。网络中的通信延迟往往导致分布式算法必须能适应信息不完备、过时甚至不正确的环境，这与单机系统完全不同。

20世纪70年代，第一台手持电话开始出现。移动电话最初是身份的象征，现在移动电话已经渗入普通人的生活。虽然在电话上将通话和计算合二为一的想法在20世纪70年代就已经出现了，但第一台真正的智能手机直到20世纪90年代中期才出现。最初诺基亚和爱立信是智能手机市场的引领者。随着智能手机逐渐普及，手机操作系统之间的竞争也变得更加激烈。在智能手机出现后的第一个十年中，手机操作系统市场基本被Symbian主导。三星、索尼、爱立信、摩托罗拉和诺基亚都是使用Symbian。然后，随着手机市场形势的变化，Symbian的市场份额逐渐被侵蚀，这其中包括RIM公司的Blackberry OS和苹果公司的iOS。2011年，诺基亚放弃Symbian并且宣布将Windows Phone作为自己的主流平台，从此开始进入低潮。在一段时间内，苹果公司和RIM公司是市场的宠儿。2008年谷歌发布了基于Linux的操作系统Android，由于具有开源的优势，各手机厂商很容易基于Android开发自己的衍生系统，并且是基于Java编程，很快就占据了最大的市场份额。

5.1.3　操作系统的分类

随着计算机系统和应用环境的复杂化和多样化，操作系统也演化出了各种类型，可以把

操作系统大致分为如下一些类型。

1. 大型机操作系统

虽然个人计算机的计算能力发展迅速，人类对高计算复杂度问题的追求却不会止步。在很多科研院所和大型公司都装备了规模庞大的大型机系统，并不断进行升级。这种大型机与个人计算机的主要差别在于其 CPU 数量和内存数量以及 I/O 能力。用于大型机的操作系统主要面向多个作业的同时处理，系统主要提供三类服务：批处理、事务处理和分时。批处理系统完成不需要交互式用户干预的周期性作业。事务处理系统负责大量小的请求，例如，电子商务网站交易处理。每笔交易的计算量都不大，但是系统必须每秒钟处理成千上万笔交易。分时系统允许多个远程用户同时登录计算机。目前，大型机操作系统基本被 Linux 占据，谷歌、阿里、腾讯等公司会根据自身需要开发特制的文件系统。

2. 服务器操作系统

服务器操作系统在服务器上运行，服务器可以是大型的个人计算机、工作站，甚至是大型机。它们通过网络同时为若干个用户服务，并且允许用户共享硬件和软件资源。服务器主要提供文件服务、数据库服务和 Web 服务。大型网站往往有多台服务器同时运行，为用户提供支持，处理海量 Web 请求。典型的服务器操作系统有 Solaris、FreeBSD、Linux 和 Windows Server。

3. 多处理器操作系统

获得大规模计算能力的另一个常用方式是用多个 CPU 组成单个的系统，依据连接和共享方式的不同，这些系统被称为并行计算机、多计算机或多处理器。它们采用的操作系统通常是配有通信、连接和一致性等专门功能的服务器操作系统的变体。

近来多核芯片逐渐普及，常规的个人电脑和手机也开始应用小规模的多核处理器，而且核的数量正与日俱增。不过多处理器操作系统的技术储备已足够成熟，很容易将其应用到多核处理器系统。难点在于应用程序如何充分运用多核的计算能力。许多主流操作系统，包括 Windows 和 Linux，都可以运行在多核处理器上。

4. 个人计算机操作系统

现代个人计算机操作系统都支持多道程序处理，通常都有几十个程序同时运行。随着多核处理器的普及化，它们的功能与服务器和多处理器操作系统已没有本质区别，主要区别在于个人系统需要为普通用户提供一流的交互体验。这类系统广泛用于字处理、电子表格、游戏和 Internet 访问。常见的个人计算机操作系统包括 Linux、Windows 7、Windows 10 和苹果公司的 OS X。

5. 掌上计算机操作系统

平板电脑和智能手机这类掌上计算机操作系统目前已得到普及。目前市场已经被谷歌的 Android（安卓）系统和苹果的 iOS 主导，虽然它们仍有很多竞争对手。现在大多数掌上计算机设备都是基于多核 CPU、GPS、摄像头、扬声器、麦克风、加速度传感器和各种用于身份识别的传感器，并且已经开发了多到数不清的第三方应用 App。

6. 嵌入式操作系统

嵌入式操作系统通常是指在某种设备中嵌入了计算芯片来控制设备和处理数据。典型的例子有微波炉、电视机、汽车等。主要的嵌入式操作系统有嵌入式 Linux、QNX 和 VxWorks 等。

7. 无线传感网络操作系统

随着计算和信息技术的普及化和泛在化，无线传感网络也在迅速发展，这类传感器网络可以用于建筑物周边保护、国土边界保卫、森林火灾探测、气象预测用的温度和降水测量、战场上敌方运动的信息收集等。传感器是一种内建有无线通信功能的电池驱动的小型计算机，它们能源有限，必须长时间工作在无人的户外环境中，通常是恶劣的条件下。其网络必须在个别结点失效的情况下仍然能稳定采集信息和传输数据。每个传感器结点都是一个计算机，因此结点上也需要运行一个小型的操作系统，通常这个操作系统是由事件驱动的，可以响应外部事件，或者基于内部时钟进行周期性的测量。该操作系统必须小且简单，因为这些结点的 RAM 很小，而且功耗要越低越好。TinyOS 就是一个用于传感器结点的操作系统。

8. 实时操作系统

实时操作系统的特征是将时间作为关键参数。例如，在工业过程控制系统中，计算机必须通过传感器收集生产过程的数据并根据数据及时控制执行器件做各种动作。这类系统通常必须满足严格的时间限制。例如，汽车在装配线上移动时，必须在限定的时间内进行规定的操作。如果焊接机器人焊接得太早或太迟都会毁坏汽车。如果某个动作必须绝对地在规定的时间内发生，这就是硬实时系统。工业过程控制、民用航空、军事以及类似应用中有很多这样的系统。另一类实时系统是软实时系统，在这种系统中，虽然有时间限制，但偶尔的超时是可以接受的，不会引起严重的后果。数字音频、多媒体系统和智能手机就是软实时系统。由于在实时系统中要满足时间限制，因此这类操作系统就是一个简单地与应用程序链接的库，各个部分必须紧密耦合并且彼此之间没有保护。这种实时系统的例子有 eCos。

掌上、嵌入式以及实时操作系统的分类有不少是彼此重叠的，所有这些系统都至少存在某种软实时情景。嵌入式和实时操作系统只运行系统设计师安装的软件，用户不能添加自己的软件，这样就使得保护工作很容易。掌上和嵌入式操作系统是给普通消费者使用的，而实时操作系统则更多用于工业领域。

9. 芯片卡操作系统

随着技术的发展和安全要求越来越高，现在的金融卡通常都包含有芯片，并且具备 CPU。这类智能卡通常具有加密、通信、数据校验等多项功能，因此也有专用的操作系统。

5.2 操作系统与计算机硬件

操作系统需要对计算机底层硬件进行有效管理，并向应用程序（和程序员）提供硬件访问接口。程序员编程时不需要了解所有的硬件细节，但至少需要了解操作系统提供的硬件访问接口。因此只有了解现代计算机中主要的计算机硬件，才能理解操作系统的具体细节。

从概念上来说，可以如图 5-5 所示来理解个人计算机的硬件结构。计算机包含 CPU、内存以及各种 I/O 设备，这些部件都通过总线连接起来并相互通信。实际上现代计算机包含多条总线，其结构与此相似。

5.2.1 处理器

处理器（CPU）是计算机的心脏，是最主要的资源，所有的程序都必须由处理器来解释

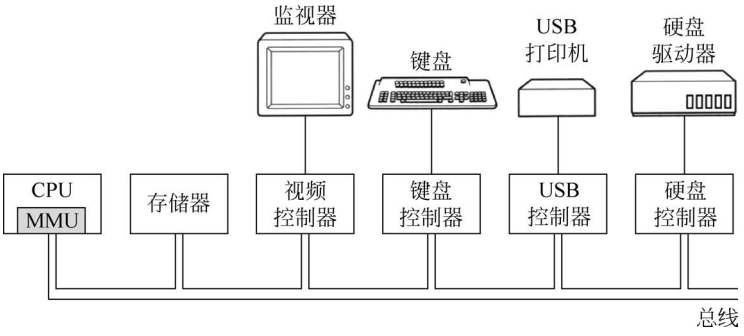


图 5-5　个人计算机中的部分硬件

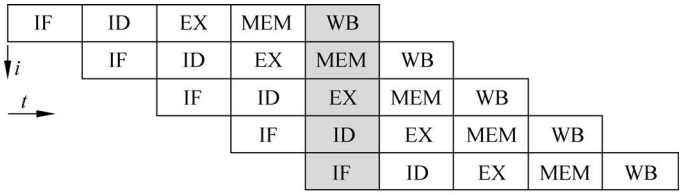
和执行。处理器管理的主要目的是对处理器的分配和调度实施最有效的管理，以最大限度地提高处理器的能力。CPU 按周期方式执行指令，每个周期包括取指、译码、取数、执行、结果写回等环节，然后从内存中取指并执行下一指令。每种 CPU 都有一套专门的指令体系，一般来说互不兼容。x86 处理器不能执行 ARM 程序，ARM 处理器也不能执行 x86 程序。另外 CPU 通常都是计算机上运行最快的器件，CPU 在内部执行指令的速度比 CPU 读写外部内存的速度要快得多，CPU 内部都有一些用来保存关键变量和临时数据的寄存器。通常，可以认为指令分成三类：一类指令将数据从内存取入寄存器，或是从寄存器存入内存；一类指令对已取入寄存器的数据进行算术或逻辑运算；还有一类指令控制程序流程，即进行循环或跳转等操作。

除了用来保存变量和临时结果的通用寄存器之外，多数计算机还有一些专用寄存器，包括程序计数器、栈指针寄存器和程序状态字。程序计数器存储了将要取出的下一条指令的内存地址。在指令取出之后，程序计数器就被更新指向下一条指令。栈指针寄存器存储当前栈的栈顶地址。当前栈中存储了正在执行的进程的栈帧。一个栈帧与进程中的一次函数调用相对应，栈帧中保存了该次函数调用的参数、局部变量和返回值。程序状态字（PSW）寄存器包含了条件码位（由比较指令设置）、CPU 优先级、模式（用户态或内核态）以及各种其他控制位。

操作系统经常会中止正在运行的某个进程并启动（或继续）另一个进程。每当停止一个运行的进程时，操作系统必须保存所有寄存器的值，这样当该进程再次运行时，可以恢复这些寄存器的值，因此操作系统必须知晓所有的寄存器。

为了提高 CPU 的运行效率，现代 CPU 大多采用流水线设计。CPU 内部被分成多个单元，每个单元负责指令处理周期的不同环节。取指单元负责将指令从内存取到 CPU，译码单元负责对指令进行译码，执行单元则负责执行指令等。多个单元可以并行工作，当执行单元在执行指令 n 时，译码单元可以同时对指令 $n+1$ 译码，取指单元则读取指令 $n+2$。图 5-6 描述了 5 阶段流水线示意图。

还有一些 CPU 使用超标量流水线设计，见图 5-7，在这种设计中有多个执行单元。例如，一个 CPU 用于整数算术运算，一个 CPU 用于浮点算术运算，一个 CPU 用于布尔运算。每次有两个或更多的指令被同时取出、译码并装入暂存缓冲区中，直至它们执行完毕。一旦执行单元有空闲，就检查保持缓冲区中是否还有可处理的指令，如果有，就把指令从缓冲区



(IF=取指；ID=译码；EX=执行；MEM=存储器访问；WB=写回)

图 5-6　5 阶段流水线示意图

中移出并执行。这种设计会带来一个问题，就是指令经常会不按顺序执行。多数情况下，硬件层面会保证这种运算的结果与顺序执行指令的结果相同，但还是有一些复杂情形需要操作系统处理。

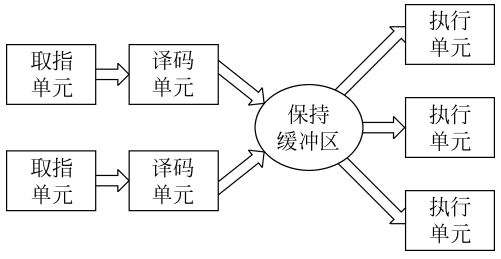


图 5-7　超标量流水线示意图

除了极少数简单的嵌入式系统 CPU 之外，多数 CPU 都有两种模式，即内核态和用户态。通常在 PSW 中有一个二进制位决定 CPU 处于哪种模式。当处于内核态时，CPU 可以执行指令集中的所有指令，并且使用硬件的所有功能。操作系统一般在内核态下运行，从而可以访问所有硬件。

用户程序则是在用户态下运行，仅允许执行整个指令集的一个子集和访问所有功能的一个子集。用户态一般不允许使用与 I/O 和内存保护有关的指令。当然，将 PSW 中的模式位设置成内核态也是不允许的。

为了使用操作系统所提供的服务，用户程序必须通过系统调用以进入内核。TRAP 指令把用户态切换成内核态，并启用操作系统。当有关工作完成之后，在系统调用后面的指令把控制权返回给用户程序。

随着 CPU 技术的迅速发展，一些更高级的特性也开始出现并迅速普及。例如多线程和多核技术。线程是操作系统能够进行运算调度的最小实际运作单位。多线程允许 CPU 同时持有两个不同的线程，并在纳秒级的时间尺度内来回切换。例如，如果某个线程要从内存中读一个字(需要等待多个时钟周期)，多线程 CPU 就可以切换至另一个线程。多线程并不是真正的并行处理。在一个时刻只有一个线程在运行，但是线程的切换时间减少到了纳秒级。

除了多线程，还出现了包含多个完整处理器或内核的 CPU，目前 8 核或 10 核 CPU 已经很常见。图形处理器(GPU)则更是包含成千上万个微核，GPU 擅长大规模并行处理简单运算，例如在图像应用中渲染多边形或模拟大规模神经网络。

操作系统需要对 CPU 是多线程还是多核进行区分，因为每个线程在操作系统看来就像是在单个 CPU 上运行。假设某个 CPU 有两个内核，每个内核支持两个线程。这样操作

系统就可以把它看成是4个CPU。如果在某个时间只需要运行两个线程，那么在一个内核上运行两个线程，而让另一个内核空转，就不如让两个线程在不同的内核上运行，那样效率要高得多。

5.2.2 内存

内存是计算机中的关键器件，也是计算机系统中的紧缺资源，所有的程序都必须调至内存中才能执行，内存管理在操作系统中占有极其重要的地位。内存与CPU之间的数据交互频繁。内存的速度应该尽可能快，以跟上CPU的速度；容量又要尽可能大一点，以驻留操作系统和多道客户程序以及数据；同时价格还要便宜。不过这三个要求很难同时满足，因此存储系统现一般采用层次结构，如图5-8所示，越上层的存储器速度越快，容量也更小。

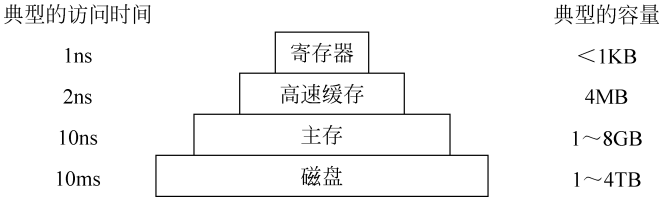


图5-8 典型的存储系统的层次结构

存储系统的顶层是CPU中的寄存器。它们在CPU中处于最核心的位置，对它们的访问是没有时延的。寄存器典型的存储容量在32位CPU中为32×32位，在64位CPU中为64×64位。程序必须通过指令自行管理这些寄存器。

下一层是高速缓存，它通常由硬件控制。主存与高速缓存之间的吞吐以块的方式进行。当CPU需要存取某个内存地址的数据时，高速缓存硬件会检查相应的内存块是否已经在高速缓存中。如果在，称为高速缓存命中，就不需要通过总线访问主存。高速缓存命中一般只需要两个时钟周期。如果高速缓存未命中就必须访问内存，所需的时钟周期要多得多。数据从缓存中清除时，缓存块数据如果没有被改写，从缓存中清除时就不用写回内存。高速缓存也位于CPU内部，成本昂贵，容量有限。有些CPU具有两级甚至三级高速缓存，每级高速缓存都比前一级慢并且容量更大。缓存面临的主要问题是：如果未命中，应该把哪一块从缓存中移走，以放入需要的数据块。当存在多级缓存时，这个问题会更加复杂。

在存储层次结构中，再往下一层是主存。主存通常称为随机存取存储器（Random Access Memory，RAM）。存储器的容量为几百兆字节至若干吉字节，并且其容量正在迅速增长。所有不能在高速缓存中得到满足的访问请求都会转往主存。

除了主存之外，计算机还有少量只读存储器（Read Only Memory，ROM）。它们与RAM不同，在电源切断之后，ROM中的数据不会丢失。只读存储器在工厂中就被编程完毕，然后再也不能被修改。ROM速度快且便宜，常被用于存储计算机启动时的引导加载模块以及底层I/O设备控制。

E^2PROM（Electrically Erasable Programmable ROM，电可擦可编程ROM）和闪存也是非易失性的，但是存储的数据可以擦除和重写，不过重写的速度比RAM要慢得多。闪存常用作便携式电子设备的存储媒介。

5.2.3　磁盘

存储层次结构再往下一层是磁盘(硬盘)。就单位容量成本来说,磁盘比 RAM 要低得多,容量也大得多,但磁盘的访问速度比 RAM 大约慢了三个数量级,因为磁盘需要做机械动作,如图 5-9 所示。

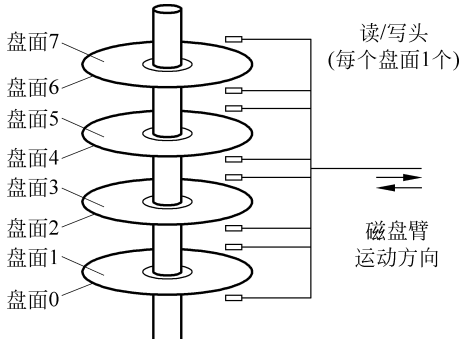


图 5-9　磁盘的大致构造

在一个磁盘中有一个或多个金属盘片,它们以 5400 转/分、7200 转/分、10 800 转/分或更高的速度旋转。安装有读写头的机械臂悬横在盘面上读写数据。数据记录在磁盘的同心圆磁道上,每个磁道划分为若干扇区。低端硬盘的速率是 50MB/s,而高速磁盘的速率是 160MB/s。

还有一类硬盘称为固态硬盘(Solid State Drive,SSD),它没有做机械动作的部分,外形也不是圆盘。固态硬盘的数据存储在掉电不失的闪存中。

现代操作系统大多支持虚拟内存技术,这种技术将磁盘当作内存使用,使得期望运行大于物理内存的程序成为可能。方法是将程序放在磁盘上,而将主存作为一种缓存,用来保存最频繁使用的部分程序,这种机制需要 CPU 中的存储器管理单元(Memory Management Unit,MMU)的支持。缓存和 MMU 的出现对系统的性能有着重要的影响。

5.2.4　I/O 设备

计算机如果有了 CPU 和存储器就可以运行程序,但应用价值不大,因为无法与外界进行交互。I/O 设备是计算机与外界交互的渠道。I/O 设备一般包括两个部分:设备控制器和设备本身。设备控制器从操作系统接收命令,然后驱使设备执行各种动作来完成命令。

在许多情形下,对这些设备的控制是非常复杂和具体的,所以,控制器的任务是为操作系统提供一个相对简单的接口。例如,磁盘控制器从操作系统接收命令,读取磁盘 2 的 11206 号扇区,然后,控制器把这个扇区号转化为具体的柱面、扇区和磁头。转换时要考虑到外柱面比内柱面的扇区多,而且还要记录哪些扇区已经坏了等。磁盘控制器计算出数据的具体机械位置后,驱使磁头臂做动作,使其移动到相应的柱面,等待对应的扇区转动到磁头下面,数据读出后,去掉引导块并校验数据,最后再把读到的二进制位组成字并存放到硬盘自身的缓存中。

每类设备都有各自的特点,虽然设备控制器大大地简化了对设备的访问,但对于应用程序员来说还是太过繁杂。因此操作系统需要对各种设备进行抽象,以提供简洁而统一的访

问接口。这就需要各设备厂商为操作系统提供设备驱动程序，将繁杂的设备控制转化为符合操作系统要求的统一的访问接口。而各操作系统的抽象接口不尽相同，所以设备厂商可能还需要为不同的操作系统提供不同的设备驱动程序。例如，扫描仪可能会配有适用于macOS、Windows 10、Windows 11 以及 Linux 的设备驱动程序。

设备驱动程序必须内嵌到操作系统中，这样才能在内核态运行。设备驱动程序也可以在内核外运行，现代的 Linux 和 Windows 操作系统都支持这种方式。不过绝大多数驱动程序仍然需要在内核态运行。只有很少一些系统在用户态运行全部驱动程序。在用户态运行的驱动程序必须能够以某种受控的方式访问设备，然而这并不容易。

将设备驱动程序装入操作系统有 3 个途径。第一个途径是将内核与设备驱动程序重新链接，然后重启动系统，UNIX 系统以这种方式工作。第二个途径是在一个操作系统文件中设置一个条目，描述清楚需要一个设备驱动程序，然后重启动系统。在系统启动时，操作系统找寻所需的设备驱动程序并装载，Windows 以这种方式工作。第三种途径是操作系统能够在运行时接收新的设备驱动程序并且立即将其安装好，无须重启系统。这种方式正在变得普及起来。USB 和 IEEE 1394 等热插拔设备都可以动态装载设备驱动程序。

操作系统对设备的操作一般都是通过读写设备控制器的寄存器进行。例如，磁盘控制器有用于指定磁盘地址、内存地址、扇区计数和读/写的寄存器。操作设备时，操作系统将参数传递给设备驱动程序的相应函数，然后设备驱动程序将命令翻译成对应的值，并写进设备寄存器中。设备寄存器一般都映射到了 I/O 端口地址空间。

在有些计算机中，设备寄存器不是映射到 I/O 空间，而是被映射到操作系统的地址空间（操作系统可使用的地址），这样，对它们的读写就不是用 I/O 指令，而是像常规地址一样读写。在这种计算机中，有专门硬件保护设备寄存器，防止应用程序直接读写这些存储器地址。在另外一些计算机中，设备寄存器被映射到一个专门的 I/O 端口地址空间中，每个寄存器都有一个端口地址。在这些机器中，提供了只有内核态才能使用的专门 IN 和 OUT 指令，供设备驱动程序读写这些寄存器用。这两种方式的应用都很广泛。

实现输入和输出的方式有 3 种。在最简单的方式中，用户程序发起一个系统调用，操作系统将其翻译成一个对应设备驱动程序的函数调用，然后设备驱动程序启动 I/O 并不断循环检查该设备，看该设备是否完成了工作。当 I/O 结束后，设备驱动程序把数据送到指定的地方并返回，然后操作系统将控制返回给调用者，这种方式称为轮询。

第二种方式是设备驱动程序启动设备并且让该设备在操作完成时发出一个中断，设备驱动程序直接返回，操作系统会阻塞调用者并安排执行其他任务。当设备驱动程序检测到该设备的操作完毕时，发出一个中断通知操作完成。

中断在操作系统中的作用非常重要，基于中断的 I/O 过程大致分为 4 步。

第 1 步：设备驱动程序通过写设备寄存器向设备控制器发命令，设备控制器操作设备执行命令。

第 2 步：当设备完成数据的读写后，控制器通过特定的总线发信号给中断控制器芯片。

第 3 步：如果中断控制器可以接收中断，它会向 CPU 的中断引脚发信号。

第 4 步：中断控制器把该设备的编号放到总线上，这样 CPU 读总线就可以知道哪个设备刚刚完成了操作。

图 5-10 给出了启动 I/O 设备并发出中断的过程。

一旦CPU决定响应中断，就会把程序计数器和PSW压入堆栈中，并且CPU被切换到用户态。通过设备编号可以在中断向量表中找到该设备中断服务程序的地址。中断服务程序结束后，CPU会恢复程序计数器和PSW的值，继续运行先前运行的用户程序。

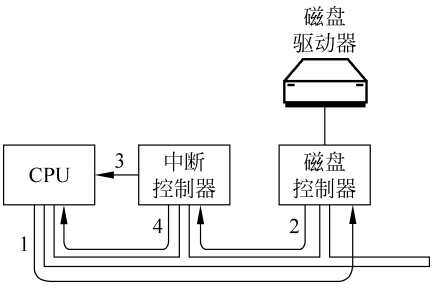


图5-10　启动I/O设备并发出中断的过程

第三种方式是使用直接存储器访问(Direct Memory Access，DMA)通道。DMA可以控制在内存和某些控制器之间的数据流，而无须CPU不断执行指令。CPU对DMA控制器进行设置，说明需要传送的字节数、数据传送的源地址和目标地址，然后启动DMA。DMA传输完数据后，会触发一个中断。

5.2.5　总线

总线(bus)是计算机各种功能部件之间传送信息的公共通信干线。计算机的总线可以划分为数据总线、地址总线和控制总线，分别用来传输数据、数据地址和控制信号。CPU与存储器、I/O设备的通信主要通过总线进行，不止一条总线，而且考虑到设备的多样性，总线还有多种类型，各种总线的传输速度和功能都不相同，其中最主要的总线是Intel发明的PCIE总线。操作系统必须了解所有总线的配置和管理。

总线通常由多个设备共享，因此，当多个设备同时需要发送数据时，需要由仲裁器决定哪个设备可以使用总线。PCIE不是这样，它使用独立的端到端通信。传统PCI总线使用并行方式，使用多条导线传送数据，例如32位的数据就要用32根导线。PCIE则是使用串行方式，用一根导线串行传递所有数据，这样做的好处是不用确保所有32位的数据在同一时刻精确地到达目的地。通过同时使用多个数据通路，仍可以达到并行的效果。例如，可以使用32个数据通路并行传输32条消息。随着网卡和显卡的速度迅速增长，PCIE标准也在持续更新。

在目前典型的计算机架构中，CPU通过DDR3总线与内存通信，通过PCIE总线与显卡通信，通过DMI总线经PCH控制器与所有其他设备对话。PCH控制器又通过通用串行总线与USB设备通信，通过SATA总线与硬盘和DVD驱动器通信，通过PCIE与网卡通信，通过PCI总线与旧的PCI设备通信。

USB将键盘、鼠标、打印机等所有慢速I/O设备与计算机连接。USB采用一种小型的4～11针连接器，其中一些针为USB设备提供电源或者接地。USB是一种集中式总线，其根设备每1ms轮询一次I/O设备，看是否有信息收发。USB 1.0的速度可以达到12Mb/s，USB 2.0可以达到480Mb/s，USB 3.0则能达到不小于5Gb/s的速率。

所有USB设备都可以实现即插即用，而不像以前的设备那样要求重启。在即插即用技术出现之前，每个设备都有一个固定的中断请求级别和用于其I/O寄存器的固定地址。例如，键盘的中断级别是1，并使用0x60～0x64的I/O地址，打印机是中断7级并使用0x378～0x37A的I/O地址等。由于I/O地址有限，而设备越来越多，这种固定配置就有可能产生冲突。即插即用的做法是由系统自动地收集有关I/O设备的信息，集中赋予中断级别和I/O地址，然后通知各设备所使用的数值。这个过程与计算机的启动密切相关。

5.2.6 计算机的启动过程

每台计算机上有一块主板，在主板上有一个称为基本输入输出系统(Basic Input Output System，BIOS)的程序。BIOS 内包含底层 I/O 程序，包括键盘输入、屏幕输出、磁盘 I/O 等。BIOS 存放在闪存中，是非易失性的，但是可以通过操作系统进行更新。

在计算机启动时会运行 BIOS 程序。它首先检查可用的内存数量、键盘和其他基本设备是否已安装并正常响应，然后扫描 PCIE 和 PCI 总线并找出连在上面的所有设备。即插即用设备也被记录下来。如果现有的设备和系统上一次启动时的设备不同，则对新的设备进行配置，然后，BIOS 尝试启动设备。用户可以在系统刚启动之后进入 BIOS 配置程序，对设备配置进行修改。系统会根据 BIOS 配置的顺序依次尝试从 CD-ROM、USB 或硬盘启动。启动设备上的第一个扇区被读入内存并执行，这个扇区中包含一个对保存在启动扇区末尾的分区表进行检查的程序，以确定哪个分区是可以启动的分区，然后，从该分区读入第二个启动引导模块，对操作系统进行加载并启动。

对于每种设备，操作系统通过 BIOS 获得配置信息。系统检查对应的设备驱动程序是否存在。如果没有，系统会要求用户在机器的存储设备上查找该设备的驱动程序或者从网络上下载。一旦有了设备驱动程序，操作系统就将它们加入内核，初始化相关的数据表，创建所需的所有上下文进程，并在终端上启动 Shell 或 GUI。

5.3 操作系统的相关概念

操作系统的主要任务是为用户程序服务。为用户程序提供服务又可以归结为四方面的问题：一是为运行的程序提供 CPU 资源，即 CPU 管理或进程管理；二是为运行的程序提供内存资源，即内存管理；三是为运行的程序提供输入输出抽象，也就是文件系统；四是对计算机上的设备进行管理。

5.3.1 进程

1. 进程的概念

进程(process)是操作系统中最重要的概念之一。进程是系统进行资源分配和调度的一个可并发执行的独立单位。

进程本质上是一个正在执行的程序。每个进程都有一个地址空间，在这个地址空间中，进程可以进行读写。地址空间中存放有程序的可执行指令、数据以及堆栈。

程序与进程的区别如下。

(1) 相同的程序可以在两个以上的进程中运行(如可以创建多个进程运行网页浏览器程序)。

(2) 程序是作为文件存放在磁盘中，运行时读到内存；而进程是在系统运行期间动态创建的，生命周期不会跨越系统运行周期。

(3) 程序只有程序语句及有初值数据变量和无初值变量；而进程需要有要处理的输入数据。

通过分析多道程序系统，可以了解创建进程的工作方式。用户先打开短消息聊天客户

端，然后启动一个下载工具，下载一个很大的视频文件，然后打开文本编辑工具写课程作业，顺便还打开浏览器准备随时上网查资料。同时，后台还运行了查收电子邮件的进程。这样就有了至少五个活动进程：聊天软件客户端、下载工具软件、字处理软件、Web 浏览器以及电子邮件接收程序，此外操作系统为了提供各种服务还要在后台运行许多进程。对用户来说，它们似乎是同时在运行，事实上，操作系统是不断在周期性地挂起一个进程，然后运行另一个进程。

2. 进程的状态

在任何时刻，一个进程要么正在执行，要么没有执行，因此可以认为总是处于以下两种状态之一：运行态或未运行态。但是进程处于未运行态时可能有几种原因：一是现在可以运行，但是需要等待分配 CPU 资源；二是进程在等待某些事件发生（例如 I/O 操作完成）。因此一般是将未运行态分成两个状态：就绪态和阻塞态。另外进程的创建需要一个过程，比如先创建进程控制块，再把程序加载到内存，创建过程中的进程一般称为处于新建态。进程结束后处于释放过程中的进程则称为处于退出态。如图 5-11 所示为进程状态模型。

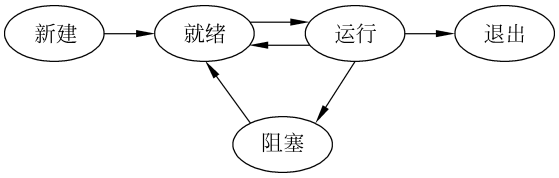


图 5-11　进程状态模型

进程可能有以下这些状态转换过程。

（1）新建→就绪。操作系统新建好一个进程后，将其从新建态转换到就绪态。

（2）就绪→运行。切换运行进程时，操作系统从就绪态的进程中选择一个。

（3）运行→退出。当前正在运行的进程已经完成或取消，将被操作系统终止。

（4）运行→就绪。正在运行的进程用完了分配的时间段，或是有优先级更高的进程需要运行，则运行进程转换为就绪态。

（5）运行→阻塞。如果进程请求了必须等待的某些事件，则进入阻塞态。

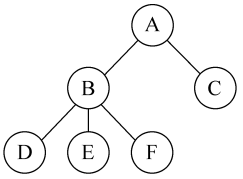
（6）阻塞→就绪。一旦所等待的事件发生了，处于阻塞态的进程转换到就绪态。

操作系统提供了各种对进程进行管理的系统调用，包括创建进程和终止进程的系统调用。例如，在用命令行形式与操作系统交互时，用户实际上是通过终端（键盘和显示器）与命令解释器或 Shell 的进程进行交互。当用户输入一条命令要求编译一个程序时，Shell 就会创建一个新进程执行编译程序。当执行编译的进程结束时，它会执行一个系统调用终止自己。

若一个进程能够创建一个或多个进程（称为子进程），而且这些进程又可以创建子进程，根据这些进程之间的创建关系就可以构造一棵树，这棵树称为进程树，见图 5-12。另外在完成某些任务时可能需要多个进程相互合作，这时相关进程之间经常需要彼此通信，这种通信称为进程间通信。

3. 进程控制块

一个进程被暂时挂起后，当该进程再次运行时，进程的状态必须与被挂起之前完全相同，因此在挂起时该进程的所有信息都要保存下来。例如，为了读写信息，进程打开了若干文件，对每个被打开文件有一个指向当前读写位置的指针。在进程暂时被挂起时，所有这些指针都必须保存起来，这样在该进程恢复运行时，执行的读写操作才能读写到正确的位置。



(进程A创建两个子进程B和C，进程B创建3个子进程D、E和F)

图 5-12　一棵进程树

所以，为了便于操作系统对进程进行管理，一个进程除了代码和数据之外，还需要一系列状态信息。与一个进程有关的所有信息，除了该进程自身地址空间的内容以外，均存放在操作系统的一张表中，这张表称为进程表，操作系统用数组或链表结构组织管理系统中的进程表，当前存在的每个进程都在其中占一项，这一表项称为进程控制块，如表 5-1 所示。进程控制块包括以下内容。

（1）标识符。每个进程都有一个唯一的标识符。

（2）状态。进程可能的状态包括就绪态、运行态和阻塞态等。

（3）优先级。相对于其他进程的优先等级。

（4）程序计数器。进程即将执行的下一条指令的地址。

（5）内存指针。包括代码段和数据段的指针以及和其他进程共享内存块的指针。

（6）上下文数据。进程执行时 CPU 寄存器中的数据。

（7）I/O 状态信息。包括 I/O 请求、分配给进程的 I/O 设备和进程使用的文件列表等。

（8）记账信息。可能包括处理器时间总和、使用的时钟数总和、时间限制等。

表 5-1　进程控制块

信　息　名	信 息 说 明	内　　容
标识信息	用于标识一个进程	进程名
说明信息	用于说明进程的各种情况	进程状态
		等待原因
		进程程序存放位置
		进程数据存放位置
现场信息	用于保留进程存放在处理器中的各种信息	通用寄存器内容
		控制寄存器内容
		程序状态字寄存器内容
控制信息	用于进程的调度	进程优先级
		队列指针

进程控制块是操作系统用来支持多道进程的重要数据结构。当进程被中断时，操作系统会把程序计数器和 CPU 寄存器（上下文数据）保存到进程控制块中，进程状态也被改变为其他的值，例如阻塞态或就绪态。然后操作系统可以另选一个处于就绪态的进程进入运行态，把这个进程的程序计数器和上下文数据加载到寄存器中，这个进程就可以开始执行。

为便于对系统中的各进程进行管理和控制，必须把所有进程的 PCB 按一定方式组织起来。可以用链表方式组织 PCB 表，对同种状态的进程，其 PCB 在一个链上，这个链称为一个队列，如图 5-13 所示。队列通常按状态分为执行队列、就绪队列、阻塞队列。

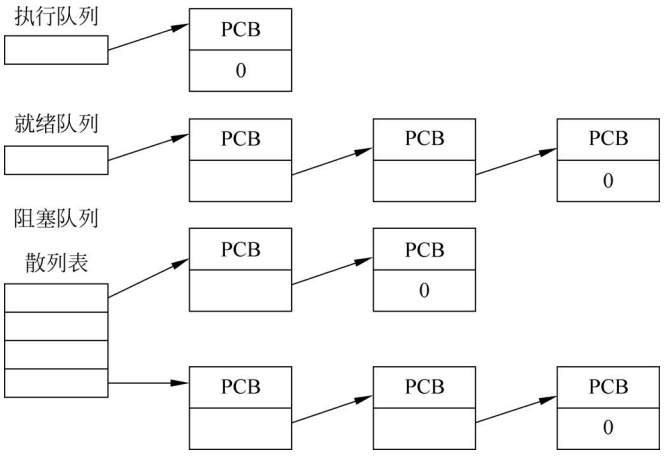


图 5-13　进程队列

4. 进程的调度

进程的调度是指由于操作系统管理了系统的有限资源，当有多个进程（或多个进程发出请求）要使用这些资源时，因为资源的有限性，必须按照一定的原则选择进程（请求）来占用资源。

进程调度目的是指控制资源使用者的数量同时选取资源使用者许可占用资源。

一般来说，调度分 3 个层次，如图 5-14 所示。

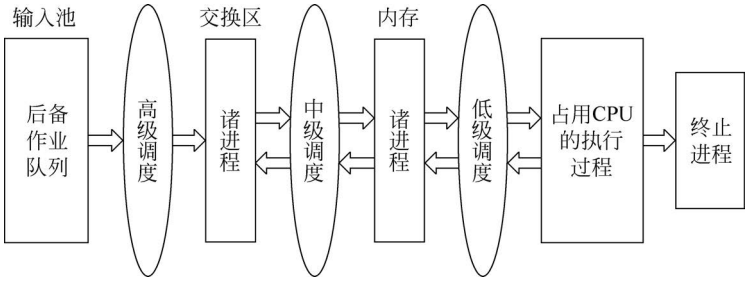


图 5-14　3 种调度的工作情况

（1）高级调度。高级调度又称作业调度，它决定处于输入池中的哪个后备作业可以调入系统，成为一个或一组就绪进程。

（2）中级调度。中级调度又称对换调度，它决定处于交换区中的就绪进程中哪一个可以调入内存，以便直接参与对 CPU 的竞争。在内存资源紧张时，将内存中处于阻塞状态的进程调至交换区。

（3）低级调度。低级调度又称进程调度或处理机调度，它决定驻在内存中的哪一个就绪进程可以占用 CPU，使其获得实在的执行。

进程调度方式分为不可剥夺（或不可抢占）方式和可剥夺方式两种。

（1）不可剥夺方式。一个进程在获得处理机后，除非运行结束或进入阻塞状态等原因主动放弃 CPU，否则一直运行下去。

（2）可剥夺方式。在某些条件下系统可以强制剥夺正在运行的进程使用处理机的权利，将其分配给另一个合适的就绪进程。

5. 进程调度的策略

进程调度的策略是在什么情况下用什么方式，在就绪进程之间进行切换和分配 CPU。

在设计进程调度的策略时，需要综合考虑很多因素，在统筹兼顾的基础上，应尽可能针对应用场景采取最合适的调度策略。

进程调度需要考虑的因素如下。

（1）资源利用的高效性。充分使用系统中各类资源，尽可能使多个设备并行工作。

（2）调度的低开销性。调度算法不能太复杂，不能带来大的开销。

（3）公平性。在考虑不同类型进程具有不同优先权的基础上，尽量公平地对待各个进程，使它们能均衡地使用处理机。

（4）针对性。考虑不同的设计目标，设计不同的策略。例如，对于批处理系统，应提高运行效率，取得最大的作业吞吐量和减少作业平均周转时间；对于交互式分时系统，应能及时响应用户的请求；对于实时系统，要求能对紧急事件作出及时处理和安全可靠。

6. 进程调度算法

进程调度算法要求满足高资源利用率、高吞吐量、用户满意等原则。

常用调度算法有先来先服务（First Come First Served，FCFS）调度算法、时间片轮转算法、优先级调度算法、多级反馈队列调度算法等。

（1）先来先服务（FCFS）调度算法。

思想：按照进程进入就绪队列的时间次序分配 CPU。

特点：具有不可抢占性的特点，一旦进程占用了 CPU，一直运行到结束，或者因阻塞而自动放弃 CPU，处在就绪队列头部的进程首先获得 CPU，一旦获得 CPU 的进程主动释放 CPU，要么进入阻塞状态，要么挂在就绪队列的尾部。

问题：当一个大作业运行时会使后到的小作业等待很长时间，这就增加了作业平均等待时间。对于 I/O 繁忙的进程，每进行一次 I/O 作业都要等待其他进程的一个运行周期结束后才能再次获得处理机，故大大延长了该类作业运行的总时间，使作业不能有效利用各种外部设备资源。该进程调度算法不能为紧急进程优先分配 CPU。

（2）时间片轮转算法。

思想：各就绪进程轮流运行一小段时间，这一小段时间称为时间片。在时间片内，如进程运行任务完成或因 I/O 等原因进入阻塞状态，该进程就提前让出 CPU。当一个进程耗费完一个时间片而尚未执行完毕，调度程序就强迫它放弃处理机，使其重新排到就绪队列末尾。

特点：时间片轮转为剥夺式调度算法，即当时间片用完后，即使当前进程没有执行结束，也会被剥夺 CPU。时间片轮转算法比较适合交互式分时系统。

系统的效率与时间片大小的设置有关。如时间片过大，系统与用户间的交互性就差，用户响应长；如时间片太小，进程间切换过于频繁，系统开销就增大。包括进程切换相关开销（保存、恢复现场等）大，频繁执行调度算法开销大。

优化方案：可将时间片分成多个规格，如 10ms、20ms 或 50ms 等。按时间片大小将就绪进程排成多个队列。排在小时间片的进程被调度的频率比较高，将交互性强的进程排在小时间片队列，而将计算性较强的进程排在长时间片队列。这样可以提高系统的响应速度和减少周转时间。

(3) 优先级调度算法。

思想：为反映出各进程的重要和紧迫程度，系统赋予每个进程一个优先数，用于表示该进程的优先级。调度程序总是从就绪队列中挑选一个优先级最高的进程，使之占有处理机。

具体实现方式如下。

① 静态优先级调度。优先级在进程创建时已经确定。在进程运行期间该优先级保持不变。

② 动态优先级调度：优先级在进程运行中，可以动态调整。

分配优先级需要考虑的因素如下。

① 系统进程应当赋予比用户进程高的优先级。

② 短作业的进程可以赋予较高的优先级。

③ I/O 繁忙的进程应当优先获得 CPU。

④ 根据用户作业的申请，设置进程的优先级。

静态优先级调度比较适合实时系统，其优先级可根据事件的紧迫程度事先设定。动态优先级调度可根据实际情况调整优先级，处理更灵活，如表 5-2 所示。

表 5-2　动态优先级调度

实际情况	提高优先级	降低优先级
进程状态	CPU 忙的进程，在其 I/O 阶段就应提高其优先级	大规模运算阶段，可以降低该进程的优先级
实时交互	运行到某一阶段的进程，需要和用户交互才能继续，也应当在该阶段提高优先级，以减少用户等待的时间	
运行时间	进程在就绪队列中等待的时间越长，就可提高其优先级	进程占用 CPU 时间越长，就可降低其优先级
占用资源	根据进程在运行阶段占用的系统资源，如内存、外部设备的数量和变化来改变优先级	

(4) 多级反馈队列调度算法。

设置多个优先级队列；队列中进程分配的时间片大小不同；新进程进入系统，被置于最高优先级队尾，如图 5-15 所示。

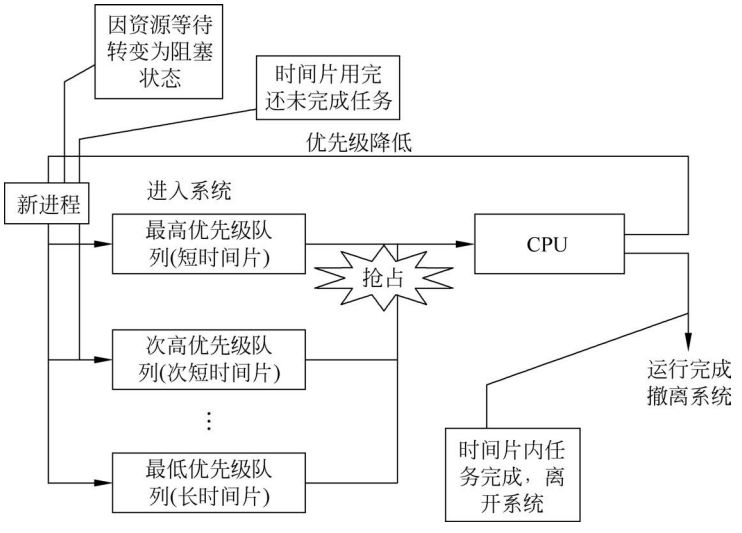


图 5-15　高优先级进程抢占低优先级进程

7. 死锁

(1) 死锁的定义。

在一个进程集合中，若每个进程都在等待某些事件(指释放资源)的发生，而这些事件又必须由这个进程集合中的某些进程来产生，就称该进程集合处于死锁状态。

例如，以下为一个竞争外设导致死锁的例子。

死锁示例如下所示：

进程 A	进程 B
① 申请输入设备	① 申请输出设备
② 申请输出设备	② 申请输入设备
③ 释放输入设备	③ 释放输出设备
④ 释放输出设备	④ 释放输入设备

如果执行次序为：进程 A①→进程 B①……，则发生死锁，如图 5-16 所示。

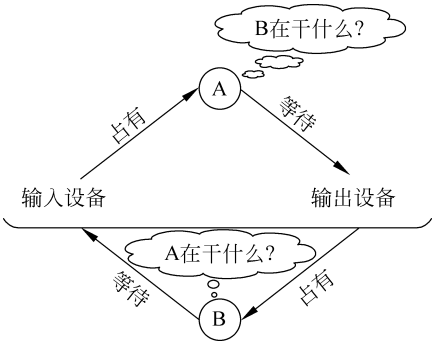


图 5-16 死锁示例

(2) 死锁的条件及死锁的防止。

出现死锁的系统必须同时满足下列四个必要条件。

① 互斥。必须存在需要互斥使用的资源。

② 占有等待。一定有占有资源而又等待其他资源的进程。

③ 非剥夺。系统中进程占有的资源未主动释放时不可以剥夺。

④ 循环等待。进程集合$\{P_0, P_1, \cdots, P_n\}$，$P_i$ 等待 P_i+1，P_n 等待 P_0。

死锁研究的对象主要包括死锁避免、死锁检测、死锁恢复和死锁防止。其中死锁防止最为重要，通常破坏死锁发生的 4 个必要条件中的任何一个即可防止死锁。

① 破坏互斥占用条件。让资源共享使用(但有些资源必须互斥)。

② 破坏占有等待条件。将进程所要资源一次性分给进程，要么没分到一个资源，要么全部满足(适合廉价资源的分配)，进程在下一轮申请资源时，释放所占的所有资源(用完一个再用下一个)。

③ 破坏非剥夺条件(用于内存管理、CPU 管理等)。

④ 破坏循环等待条件。采用资源顺序分配方法，给每类资源编号，进程只能按序号由小到大的顺序申请资源，若不满足则拒绝分配。若出现循环等待，则必会有小序号资源序号大于大序号资源序号。

8. 进程的系统调用

与进程管理有关的系统调用包括等待一个子进程结束、将当前进程所运行的程序替换为另一个程序等。

有时，需要向一个正在运行的进程传送信息，而该进程并没有等待接收信息。例如，一个进程通过网络向另一台机器上的进程发送消息进行通信。为了保证一条消息或消息的应答不会丢失，发送者要求它所在的操作系统在指定的若干秒后给一个通知，这样如果对方尚未收到确认消息就可以进行重发。在设定该定时器后，程序可以继续做其他工作。

在限定的秒数流逝之后，操作系统向该进程发送一个警告信号。此信号引起该进程暂时挂起，无论该进程正在做什么，系统将其寄存器的值保存到堆栈，并开始运行该进程预先设定好的一个信号处理过程，比如重新发送可能丢失的消息。这些信号是软件模拟的硬件中断，除了定时器之外，该信号可以由各种原因产生。许多由硬件检测出来的陷阱，如执行了非法指令或使用了无效地址等，也被转换成该信号并交给这个进程。

系统管理器给每个进程赋予一个用户号(UID)，这个 UID 就是启动该进程的用户 UID。子进程拥有与父进程一样的 UID。用户可以是某个组的成员，每个组也有一个组号(GID)。

在 Linux 系统中，有一个超级用户，在 Windows 中，也有一个管理员，它们具有特殊的权力，可以无视一些保护规则。

5.3.2　地址空间

每台计算机都有内存，用来存储正在执行的程序。较复杂的操作系统允许在内存中同时运行多道程序。为了避免它们互相干扰(包括操作系统)，需要能够对内存进行保护和重定位。现代操作系统大多数都通过将各种存储器抽象为地址空间解决两个问题。地址空间是一个进程可用于内存寻址的一套地址集合。每个进程都有自己的地址空间，并且这个地址空间独立于其他进程的地址空间。

逻辑地址又称虚地址，是相对地址，一般从 0 开始，每个程序的地址空间是独立的。

绝对地址又称实地址，是内存中真正的物理地址地址。程序在编译过程中会进行地址的转换，如图 5-17 所示。

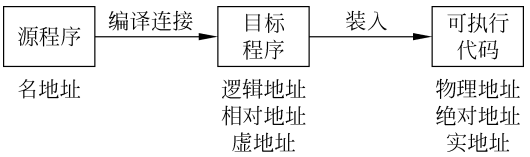


图 5-17　地址转换过程

为了能让每个程序都有独立的地址空间，一个解决办法是使用地址动态重定位技术。动态重定位技术将每个进程的地址空间映射到物理内存的不同位置，这通常是在 CPU 中配置两个特殊的寄存器：基址寄存器和界限寄存器。当进程运行时，程序的起始物理地址被装载到基址寄存器中，程序的长度则装载到界限寄存器中。每当进程访问内存取指或读写数据时，在地址被发送到地址总线之前，CPU 硬件会把基址值加到进程发出的地址值中。同时 CPU 还会检查程序提供的地址值是否大于界限寄存器的值，如果越界，就会产生错误并终止访问。

为了允许多道程序并存，一般来说内存越大越好。但实际上，在典型的现代操作系统中，计算机完成引导后就会启动几十甚至上百个进程，而且用户也会启动一些进程。所有进程所需的 RAM 数量总和通常要远远超出存储器能够支持的范围。因此操作系统需要应对内存超载的问题。常用于处理内存超载的方法有两种：一种是交换技术，即把空闲进程存回磁盘，在运行时再完整地放到内存上，如图 5-18 所示，加载 D 进程时，A 进程被交换出去，然后在 B 进程被交换出去后，A 进程再被交换进来；另一种方法是虚拟内存，该技术允许程序在运行时只有一部分被调入内存。虚拟内存的基本原理是将程序的地址空间分割成多个块，这些块称为页面。页面被映射到物理内存，但并不是所有页面都必须在内存中才能运行

程序。如果程序用到的地址属于已经映射到物理内存的页面，则由内存管理单元直接进行寻址；如果程序用到的地址属于没有映射到物理内存的页面，则由操作系统进行调度，将被引用的页面载入物理内存并重新执行失败的指令。

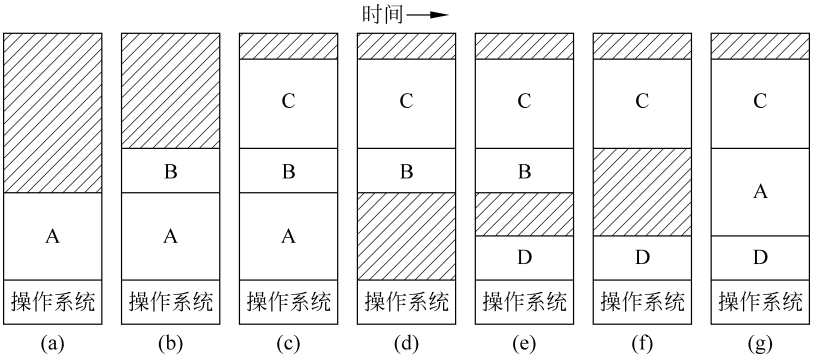


图 5-18　内存交换的情形

5.3.3　文件

文件系统是操作系统的另一个关键概念。操作系统的一项主要功能是隐藏磁盘等存储设备的数据块管理细节，为用户程序提供一个良好、清晰、一致的独立于设备的抽象文件模型，并将各种 I/O 设备的读写也抽象为文件读写。与文件系统有关的系统调用包括创建文件、删除文件、读文件和写文件等。相关的系统调用需要在读写文件前先在磁盘上定位和打开文件，在读写文件结束之后关闭该文件。

大多数操作系统用目录（文件夹）的概念对文件进行组织。操作系统也提供了相应的系统调用，如创建和删除目录，将文件放入目录中，从目录中删除文件等。目录项可以是文件或者目录，这样就用层次结构将文件组织了起来。目录层次结构中的每一个文件都可以通过从目录的顶部即根目录开始的路径名来确定。绝对路径名包含了从根目录到该文件的所有目录清单，目录之间用斜线隔开。

进程和文件都可以组织成树状结构，但这两种树状结构有不少不同之处。一般进程的树状结构层次不深，很少超过三层，而文件树状结构的层次常常多达四层、五层或更多层。进程和文件在所有权及保护方面也有所区别，子进程只有父进程才能控制和访问，而文件和目录通常文件所有者之外的其他用户也可以访问。

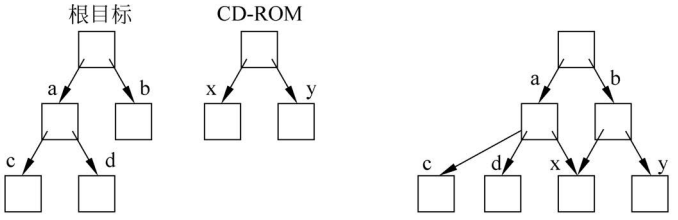
Linux 系统给每个文件赋予一个 9 位的二进制保护码，9 位保护码分 3 段，第 1 段用于文件所有者，第 2 段用于与所有者同组的其他成员，第 3 段用于其他用户。每个段包含 3 位，分别为 rwx 位，r 位用于读访问控制，w 位用于写访问控制，x 位用于执行访问控制。例如，保护码 rwxr-x--x 的含义是所有者可以读、写或执行该文件，其他的组成员可以读或执行，但不能写，而其他人可以执行，但不能读和写。

每个进程都有一个工作目录，对于没有以斜线开头给出绝对地址的路径，将默认在这个工作目录下寻找。进程可以通过系统调用指定新的工作目录。

在读写文件之前，首先要打开文件，检查其访问权限。若权限许可，系统将返回一个小整数，称作文件描述符，供后续操作使用。若禁止访问，系统则返回一个错误码。

在 Linux 系统中文件系统还有安装的概念。很多计算机都有光驱和 USB 接口，可以插

入光盘或 USB 存储盘。Linux 系统对这些可移动介质的处理办法是将光盘或 USB 盘上的文件系统安装到主文件树上。以图 5-19 为例，在 mount 调用之前，根文件系统与光盘上的文件系统是分离的。此时光盘上的文件系统还无法使用，因为无法说明文件路径。Linux 系统不允许在路径前面加上驱动器名称，因为这会引入操作系统应当屏蔽的硬件细节。代替的方法是用 mount 系统调用将光盘上的文件系统连接到根文件系统的某个目录上。在图 5-19(b)中，光盘上的文件系统安装到了目录 b 上，这样就可以访问文件/b/x 以及/b/y。



(a) 安装前，CD-ROM上的文件不可访问 (b) 安装后，它们成了文件层次的一部分

图 5-19　安装前后的文件

在 Linux 系统中另一个重要的概念是特殊文件。提供特殊文件是为了使对 I/O 设备的访问像对文件的访问一样。这样，就可以用读写文件的系统调用来对 I/O 设备进行读写。特殊文件有两种类型：块特殊文件和字符特殊文件，分别对应两类设备。块特殊文件对应那些由可随机存取的块组成的设备，如磁盘等；字符特殊文件用于打印机、键盘和其他接收或输出字符流的设备。在 Linux 系统中，特殊文件保存在/dev 目录中。

还有一种特殊的文件是管道。管道是一种虚文件，它可连接两个进程，如图 5-20 所示。如果进程 A 和 B 之间建立了管道，当进程 A 对进程 B 发送数据时，可以把数据写入管道，就好像写文件一样；进程 B 可以通过读管道得到数据，就好像读文件一样。通过管道，在 Linux 中两个进程之间的通信就与读写普通文件一样了。

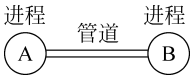


图 5-20　由管道连接的两个进程

5.3.4　输入/输出

所有的计算机都有用于输入/输出的设备，包括键盘、显示器、打印机等。操作系统需要对这些设备进行管理。每个操作系统都有管理其 I/O 设备的 I/O 子系统。有一些 I/O 软件是独立于设备的，可以应用于许多 I/O 设备。还有一些 I/O 软件，例如设备驱动程序，是专门为特定的 I/O 设备设计的。

I/O 设备大致分为两类：块设备和字符设备。块设备将信息存储在固定大小的块中，每个块有自己的地址，所有传输以一个或多个完整的连续块为单位。块设备的基本特征是每个块都能独立于其他块进行读写。硬盘、光盘、USB 盘是最常见的块设备。字符设备以字符为单位发送或接收一个字符流，而不考虑任何块结构。字符设备是不可寻址的。打印机、鼠标、键盘等都是字符设备。

对 I/O 设备的控制和访问需要一系列 I/O 软件的支持。在设计 I/O 软件时对应着几

个关键问题。

（1）如何让应用程序具有设备独立性。也就是说设计出的应用程序应当可以访问任意I/O设备而无须事先指定设备。例如视频播放软件能够从硬盘、光盘或USB盘上读取文件，而这些对于视频播放软件以及程序员来说应当都是无差别的。同样的，对于需要打印输出结果的软件来说，无论输出的目的地是打印机还是屏幕，应当也是无差别的。这在现代操作系统中一般是通过将设备抽象为文件，并通过文件系统的方式进行组织。

（2）错误处理。一般来说，错误应当尽可能在接近硬件的层面得到处理。如果在控制器层面能够处理，控制器就应当自己设法去处理。如果控制器处理不了，设备驱动程序就应当处理。在大部分情况下，错误的恢复可以在底层进行，而高层软件甚至不用知道发生了错误。

（3）设备共享。有些设备可以供多个应用程序同时使用，例如多个用户可以同时打开同一个磁盘上的文件，有些设备则需要由单个用户独占使用，独占设备还会带来死锁的问题。

5.3.5 Shell

操作系统的主要功能是对资源进行管理并为应用程序提供服务，如果是无须与人交互的嵌入式系统，仅此就够了，但如果是个人计算机或智能手机这样需要与人交互的系统，则还需要提供命令行或图形交互环境，它们其实是操作系统附带的程序，不是操作系统的组成部分。

在Linux系统中，Shell因为高效而受到程序员的欢迎。用户登录时，系统根据配置文件会启动一个Shell或GUI。Shell以终端作为标准输入和标准输出。首先显示提示符，提示用户Shell正在等待接收命令。假如用户输入：

```
date
```

Shell会创建一个子进程，并运行date程序。在该子进程运行期间，Shell会等待它结束。在子进程结束后，Shell再次显示提示符，并等待下一行输入。

用户可以将标准输出重定向到一个文件，如：

```
date > file
```

也可以将标准输入重定向，如：

```
sort < file1 > file2
```

该命令启动sort程序，将file1作为输入，将file2作为输出。

一个程序的输出可以通过管道作为另一程序的输入，例如：

```
cat file1 file2 file3 | sort >/dev/lp
```

这个命令会启动cat程序，将3个文件合并，结果输出到sort程序进行排序。sort的输出又被重定向到文件/dev/lp中，也就是打印机。

如果用户在命令后加上一个“&”符号，则Shell将不等待其结束，而直接显示出提示符，启动的程序会在后台运行，用户则可以继续与命令行交互。

5.4 系统调用

操作系统的功能主要是为用户程序提供抽象和管理计算机资源。就用户程序来说，重要的是前者，例如，创建、写入、读出和删除文件。资源管理部分对用户程序来说可以视为黑

箱，由操作系统自动完成。这样，用户程序和操作系统之间的交互主要是通过抽象，也就是系统调用接口。作为用户程序的程序员，必须了解和熟悉这个接口。不同的操作系统提供的调用接口也各不相同。

以读文件系统调用 read 为例，见图 5-21。read 有 3 个参数：第一个参数说明要从哪里读，也就是读哪个文件，第二个参数说明数据要读到哪里去，也就是目标缓冲区，第三个参数说明要读多少数据，也就是读取的字节数。用 C 程序调用的形式如下：

```
count = read(fd, buffer, nbytes);
```

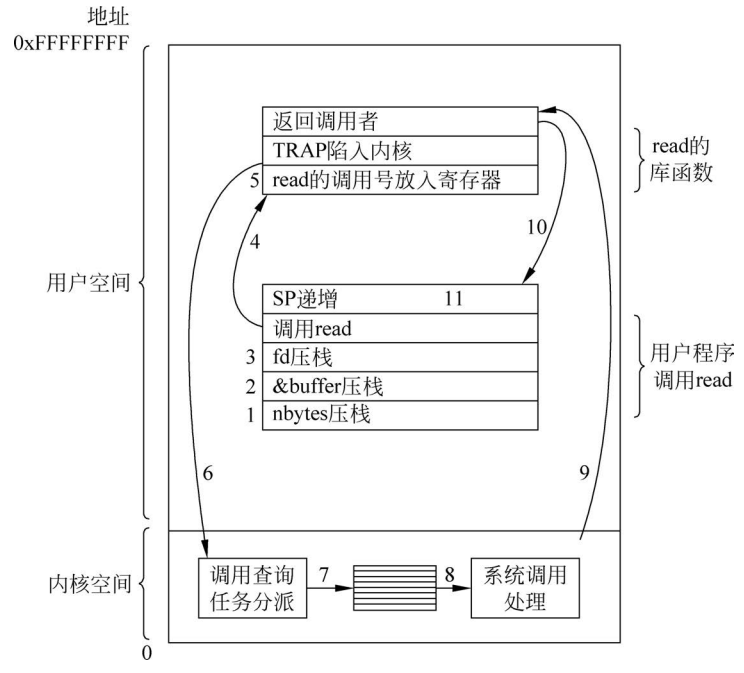


图 5-21　系统调用 read(fd,buffer,nbytes)的完成步骤

系统调用会在返回值中返回实际读出的字节数。一般这个值和 nbytes 一样，但也可能更小，例如如果遇到了文件末尾就读不到那么多数据。

如果系统调用不能执行，返回值会被置为－1，并且在全局变量 errno 中会放入错误号。系统调用因为涉及资源占用等诸多因素，有可能发生各种异常。因此应用程序在进行系统调用时都应当检查调用返回值，以便在发生异常时进行处理。

在进行系统调用时，系统调用编号会被放在指定的寄存器中，然后执行 TRAP 指令，将用户态切换到内核态，并从内核的一个固定地址开始执行。TRAP 指令与调用指令非常类似，它们都跳转到某个远处位置的指令，并将返回地址压栈。但是，TRAP 指令会切换到内核态，而普通的函数调用指令不会改变运行模式。另外，TRAP 指令并不能跳转到任意地址上，而是给出内存中一张表格的索引号，这张表格中含有跳转地址。

然后内核代码根据系统调用号查找正确的系统调用指针，再进行具体的系统调用处理。一旦系统调用处理完成，控制权就会返回给位于用户空间的系统调用库函数，并进一步返回到用户程序。最后用户程序会进行弹栈操作，继续执行后面的指令。

Linux 系统遵循的可移植操作系统接口（Portable Operating System Interface of

UNIX，POSIX）标准给出了 100 个过程调用，主要分成进程管理、文件管理、目录和文件系统管理以及杂项 4 类，Linux 系统调用见表 5-3～表 5-6。

表 5-3 进程管理

调　　用	说　　明
pid＝fork()	创建与父进程相同的子进程
pid＝waitpid(pid,&statloc,options)	等待一个子进程终止
s＝execve(name,argv,environp)	替换一个进程的核心映像
exit(status)	终止进程执行并返回状态

表 5-4 文件管理

调　　用	说　　明
fd＝open(file,how,…)	打开一个文件进行读写
s＝close(fd)	关闭一个打开的文件
n＝read(fd,buffer,nbytes)	把数据从文件读到缓冲区中
n＝write(fd,buffer,nbytes)	把数据从缓冲区写到文件中
position＝lseek(fd,offset,whence)	移动文件指针
s＝stat(name,&buf)	获取文件的状态信息

表 5-5 目录和文件系统管理

调　　用	说　　明
s＝mkdir(name,mode)	创建一个新目录
s＝rmdir(name)	删去一个空目录
s＝link(d1,d2)	创建一个新目录项 d2，并指向 d1
s＝unlink(name)	删去一个目录项
s＝mount(special,name,flag)	安装一个文件系统
s＝umount(special)	卸载一个文件系统

表 5-6 杂项

调　　用	说　　明
s＝chdir(dirname)	改变工作目录
s＝chmod(name,mode)	修改一个文件的保护位
s＝kill(pid,signal)	发送信号给一个进程
seconds＝time(&seconds)	自 1970 年 1 月 1 日起的流逝时间

在 Linux 系统中，只有 fork 可以创建新的进程。fork 会创建一个当前进程的副本，包括所有的文件描述符、寄存器等内容。在 fork 之后，原来的进程与其子进程就开始分别运行，然后各自从 fork 函数返回。在子进程中，fork 会返回零，在父进程中，fork 返回的则是子进程的标识符（PID）。这样根据 fork 的返回值，进程就能知道自己是父进程还是子进程。

多数情形下，在 fork 之后，子进程需要执行与父进程不同的代码。以 Shell 进程为例，Shell 会从终端读取命令，创建一个子进程，然后子进程执行命令，原来的 Shell 进程则等待子进程结束，再读入下一条命令。为了等待子进程结束，父进程会执行 waitpid 系统调用，它只是将父进程挂起，直至某个子进程结束。waitpid 可以等待一个特定的子进程，也可以将第一个参数设为－1，等待任何一个子进程。waitpid 返回时，会将子进程的退出状态（是

否正常退出等信息)写到第二个参数 statloc 所指向的内存位置。

Shell 调用 fork 创建一个新的子进程之后,这个子进程应当执行用户的命令,通常是执行用户指定的某个程序,这时子进程需要被替换成目标程序。通过 execve 系统调用可以实现这一点,这个系统调用会导致主调进程的整个核心映像被替换为指定文件中的程序。以下给出了简化的 Shell 程序。假设用户输入命令:

```
cp file1 file2
```

该命令将 file1 复制到 file2。在 Shell 创建子进程之后,子进程就会执行 cp,并将源文件名和目标文件名传递给它。以下为简化的 Shell 程序。

```
while (1)
{
   type_prompt( );                          //在屏幕上显示提示符
   read_command(command, parameters);       //从终端读取输入
   if (fork( ) != 0)                        //派生子进程
   {
      waitpid( -1, &status, 0);             //父进程代码
   }
   else
   {
      execve(command, parameters, 0);       //子进程代码,执行命令
   }
}
```

如果进程需要读写文件,先要用系统调用 open 打开该文件。调用 open 时,要在第一个参数中用绝对路径名或指向工作目录的相对路径名指定要打开文件的名称,在第二个参数中用 O_RDONLY、O_WRONLY 或 O_RDWR 指定打开模式是只读、只写或是两者都可以,如果要创建一个新文件,可以使用 O_CREAT 作为参数。open 会返回相应的文件描述符,然后进程就可以用文件描述符进行读写操作。读写结束后,可以用系统调用 close 关闭文件。

在所有的系统调用中,最常用的调用可能是 read 和 write。每个打开的文件都有一个与之相关联的指向文件当前位置的指针。在顺序读写时,该指针通常指向下一个将要读写的字节。系统调用 lseek 可以改变该位置指针的值,这样就可以调整后续的 read 或 write 调用所读写的位置。调用 lseek 需要提供 3 个参数:第一个是文件描述符;第二个是位置偏移量;第三个说明该偏移量是相对于文件起始位置、当前位置还是文件的结尾。在修改了指针之后,lseek 会返回指针指向的绝对位置。

Linux 系统为每个文件保存了该文件的类型(普通文件、特殊文件、目录等)、大小、最后修改时间等其他信息。程序可以通过系统调用 stat 查看这些信息。调用 stat 需要提供两个参数:第一个参数指定了要查看的文件;第二个参数是一个地址值,stat 会将查到的信息写入该地址。如果文件已经打开,可以用系统调用 fstat 完成同样的工作,fstat 需要的参数是已打开文件的描述符。

还有一些系统调用与目录或文件系统有关,mkdir 和 rmdir 分别用于创建和删除空目录,link 则允许同一个文件以多个名称出现在文件系统树上的多个位置。例如某个开发团队中允许若干个成员共享一个共同的文件,每个人都在自己的目录中有该文件的链接并指向同一个文件。这种共享文件方式与每个团队成员都有一个副本不是同一回事,因为每个

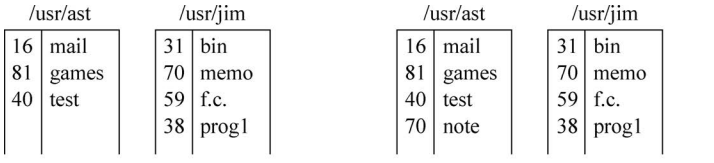
人所拥有的链接指向的实际上是同一个文件，任何成员所做的修改都立即为其他成员所见，而对副本的修改不会影响到其他的副本。

在 Linux 系统中，目录其实也是文件，只不过是一种特殊的文件，这个文件其实是一张表格，目录中的每个文件在表格中占一个条目，条目中记录了该文件的名称和一个与该文件对应的编号(inode)，通过 inode 号可以在一个特定的表格中查找到文件的信息，例如文件的拥有者和磁盘块的位置等。

假设/usr 目录中有两个子目录 ast 和 jim，在程序中执行语句：

```
link("/usr/jim/memo","usr/ast/note");
```

就会在 ast 目录中生成一个新的条目 note，/usr/jim/memo 和/usr/ast/note 这两个条目指向的是同一个文件，因此有相同的 inode 号。如果使用 unlink 系统调用将其中一个条目删除，实际文件不会被删除，因为还有其他目录中的条目指向这个文件。如果指向某个文件的所有条目都被删除了，操作系统就会把该文件从磁盘中删除。图 5-22 给出了系统调用 link 的过程。



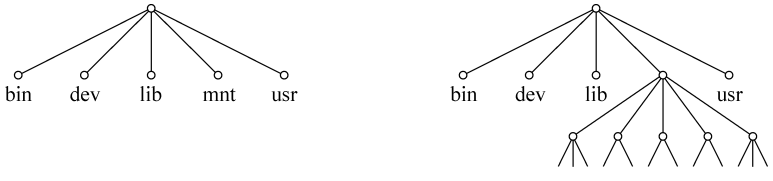
(a) link调用之前的两个目录　　(b) link调用之后，ast目录中生成了新的条目note，与jim目录中的条目memo有相同的inode号，指向的是同一个文件

图 5-22　系统调用 link 示例

系统调用 mount 允许将一个文件系统安插到另一个文件系统的树上。例如用户插入了一个 USB 软盘，通过在程序中调用 mount，就可以将这个 USB 软盘上的文件系统添加到根文件系统中：

```
mount("/dev/sdb0","/mnt",0);
```

这里，第一个参数是 USB 驱动器 0 的块特殊文件名称；第二个参数是要被安插到树中的位置；第三个参数说明将要安装的文件系统是可读写的还是只读的，见图 5-23。



(a) mount前的文件系统　　(b) mount后的文件系统

图 5-23　mount 前后的文件系统

当不再需要一个文件系统时，可以用系统调用 umount 进行卸载。通过 mount 调用，Linux 系统可以将多个存储介质整合成一个统一的文件层次树，不用考虑文件在哪个驱动器上。

系统调用 chdir 可以改变进程的当前工作目录。在程序中调用：

```
chdir("/usr/ast/test");
```

之后，相对路径 xyz 对应的文件就是/usr/ast/test/xyz。工作目录的概念避免了总是需要使用冗长的绝对路径名。

在 Linux 系统中，每个文件都设置有一个保护模式。系统调用 chmod 可以改变文件的保护模式。例如，执行语句：

```
chmod("file",0644);
```

会将 file 文件的保护模式位改成 110100100，表示文件拥有者可以读写，同组其他用户和普通用户只能读。

系统调用 kill 会向目标进程发送一个信号。如果目标进程设置好了这个信号的处理程序，在进程收到信号时，相应的信号处理程序就会被执行，执行过程类似中断处理。如果该进程没有设置相应的处理程序，那么收到信号的进程就会被杀掉。

Windows 系统也有相应的系统调用。微软定义了一套函数接口，称为 Win32 应用编程接口。Win32 API 接口定义的系统调用有数千个，程序员可以通过这套函数获得操作系统的服务。表 5-7 给出了 Linux 系统调用对应的部分 Win32 API 调用。从 Windows 95 开始的所有 Windows 版本都支持或部分支持这个接口。由于 Windows 发行新版时微软会在接口中添加新的系统调用，所以很难定义 Win32 编程接口是由哪些系统调用构成。另外 Win32 API 中有一大批调用完全是在用户空间进行的，而且一些系统调用在某个 Windows 版本中在内核执行，在另一个版本中又在用户空间执行。

表 5-7　Linux 系统调用对应的部分 Win32 API 调用

UNIX	Win32	说　明
fork	CreateProcess	创建一个新进程
waitpid	WaitForSingleObject	等待一个进程退出
execve	（无）	CreateProcess＝fork＋execve
exit	ExitProcess	终止执行
open	CreateFile	创建一个文件或打开一个已有的文件
close	CloseHandle	关闭一个文件
read	ReadFile	从一个文件读数据
write	WriteFile	把数据写入一个文件
lseek	SetFilePointer	移动文件指针
stat	GetFileAttributesEx	取得文件的属性
mkdir	CreateDirectory	创建一个新目录
rmdir	RemoveDirectory	删除一个空目录
link	（无）	Win32 不支持 link
unlink	DeleteFile	销毁一个已有的文件
mount	（无）	Win32 不支持 mount
umount	（无）	Win32 不支持 umount
chdir	SetCurrentDirectory	改变当前工作目录
chmod	（无）	Win32 不支持安全性(但 NT 支持)
kill	（无）	Win32 不支持信号
time	GetLocalTime	获得当前时间

Win32 API 中有许多系统调用与 Linux 系统中的系统调用有直接对应关系。CreateProcess 用于创建一个新进程，它相当于 Linux 系统中 fork 和 execve 的结合。Windows 中没有类似 Linux 的进程层次，所以不存在父进程和子进程的概念。在进程创建之后，创建者和被创建者是平等的。WaitForSingleObject 用于等待一个事件，等待的事件可以是多种可能的事件。如果有参数指定了某个进程，调用者将等待所指定进程的退出事件。进程退出使用的系统调用是 ExitProcess。Win32 API 中也有许多进行文件操作的系统调用，文件可被打开、关闭和写入。SetFilePointer 以及 GetFileAttributesEx 调用可以设置文件指针的位置和获取文件的属性。Windows 中有目录，目录分别用系统调用 CreateDirectory 以及 RemoveDirectory 创建和删除。系统调用 SetCurrentDirectory 可以设置当前目录。GetLocalTime 可获得当前时间。

Win32 接口中没有文件的链接和文件系统安装的概念，所以也不存在相应的系统调用。当然，Win32 中也有大量 Linux 中不存在的其他调用，特别是管理 GUI 的各种调用。

5.5 华为鸿蒙操作系统

华为鸿蒙操作系统（HUAWEI Harmony OS）是华为公司在 2019 年 8 月 9 日于东莞举行的华为开发者大会上正式发布的操作系统。它的问世，在全球引起反响。近年来，美国打压华为对鸿蒙操作系统的问世起了催生作用，代表中国高科技开展的一次战略突围，是中国解决诸多“卡脖子”问题的一个带动点。它的诞生改变了操作系统的全球格局。

5.2.1 鸿蒙操作系统简介

鸿蒙操作系统是华为公司耗时 10 年，由 4000 多名研发人员投入开发的一款基于微内核，面向 5G 物联网，面向全场景的分布式操作系统。鸿蒙的英文名是 Harmony，意为和谐。

鸿蒙操作系统是一款全新的面向全场景的分布式操作系统，创造了一个超级虚拟终端互联的世界，将人、设备、场景有机地联系在一起，将消费者在全场景生活中接触的多种智能终端实现极速发现、极速连接、硬件互助、资源共享，用合适的设备提供场景体验。

鸿蒙操作系统不是安卓系统的分支或简单修改，是与安卓、iOS 不一样的操作系统；它在性能上不弱于安卓系统，而且华为还为基于安卓生态开发的应用平稳迁移到鸿蒙操作系统上做好了衔接，能将相关系统及应用迁移到鸿蒙操作系统上，完成迁移及部署。新的操作系统将打通手机、计算机、平板电脑、电视、工业自动化控制、无人驾驶、车机设备、智能穿戴，统一成一个操作系统，并且该系统是面向下一代技术而设计的，能兼容全部安卓应用的所有 Web 应用。若安卓应用重新编译，在鸿蒙操作系统上，运行性能提升超过 60%。鸿蒙操作系统架构中的内核会把之前的 Linux 内核、鸿蒙操作系统微内核与 LiteOS 合并为一个鸿蒙操作系统微内核。创造一个超级虚拟终端互联的世界。同时由于鸿蒙系统微内核的代码量只有 Linux 宏内核的千分之一，其受攻击概率也大幅降低。

5.2.2 技术特性

1. 无缝连接

鸿蒙操作系统采用分布式架构和分布式虚拟总线技术，提供共享通信平台、分布式数据

管理、分布式任务调度和虚拟外设。使用鸿蒙操作系统，应用程序开发人员将不必处理分布式应用程序的底层技术，从而使他们能够专注于自己的个人服务逻辑。

开发分布式应用程序将比以往任何时候都容易。基于鸿蒙操作系统构建的应用程序可以在不同的设备上运行，同时提供跨所有场景的无缝协作体验。

2. 流畅的性能

保证了延时引擎和高性能进程间通信（IPC）技术实现系统的流畅。系统通过确定性延迟引擎和高性能 IPC 技术解决性能不佳的挑战。

3. 安全性更高

系统采用全新的微内核设计，拥有更强的安全特性和低时延等特点。微内核设计的基本思想是简化内核功能，在内核之外的用户态尽可能多地实现系统服务，同时加入相互之间的安全保护。微内核只提供最基础的服务，比如多进程调度和多进程通信等。通过统一 IDE 支撑一次开发，多端部署，实现跨终端生态共享。

4. 一致性

鸿蒙操作系统凭借多终端开发 IDE，多语言统一编译，分布式架构 KIT 提供屏幕布局控件以及交互的自动适配，支持控件拖拽，面向预览的可视化编程，从而使开发者可以基于同一工程高效构建多端自动运行 App，实现真正的一次开发，多端部署，在跨设备之间实现共享生态。华为方舟编译器是首个取代安卓虚拟机模式的静态编译器，可供开发者在开发环境中一次性将高级语言编译为机器码。此外，方舟编译器未来将支持多语言统一编译，可大幅提高开发效率。

5.6　小结

可以从两个角度认识操作系统：资源管理的角度和计算机抽象的角度。从资源管理的角度来看，操作系统的任务是高效管理计算机系统的各个部分。从计算机抽象的角度来看，操作系统的任务是为应用程序以及应用程序员提供比实际机器更便于理解和运用的抽象。这些抽象包括进程、地址空间以及文件等概念。

操作系统的发展与现代计算机技术的发展伴随始终，操作系统最早是替代计算机操作员，后来发展到早期批处理系统，然后又发展到多道程序系统以及个人计算机系统。

操作系统同硬件交互密切，计算机由处理器、存储器以及 I/O 设备组成，这些部件通过总线连接。

所有操作系统构建所依赖的基本概念是进程、存储管理、I/O 管理、文件管理和安全。

操作系统的核心是它所提供的系统调用，这些系统调用体现了操作系统提供给用户程序的接口和服务。