第5章 I/O设备管理

从某种意义上来说，操作系统的目标是给用户提供一个高层的机器接口（虚拟机），把所有的硬件细节都封装在这个虚拟机中。我们知道，计算机最核心的硬件部件有两个：CPU和内存。其中，CPU负责执行指令，内存负责存储数据。但光有这两个部件是不行的，试想一下，如果摆在我们面前的是一块主板，上面只连着CPU和内存，那么怎么才能让它工作起来呢？对于普通的计算机用户来说，根本无法直接与CPU和内存打交道。

为了解决这个问题，需要引入输入/输出（Input/Output，I/O）设备，用户正是通过这些输入/输出设备来使用计算机。为了加深读者的印象，来看一个网络上流传的小笑话。

显示器说：我好惨，整天被人看。

键盘说：我更惨，每天被人打。

鼠标说：我才惨呢，每天被人摸。

主机说：你们有我惨吗？每天被人按肚脐眼！

在一个现代计算机系统中，存在着大量的输入/输出设备，它们种类繁多，差异很大，包括控制方式上的差异、数据传输速度上的差异等。而且随着科学技术的发展，新设备也层出不穷。在20世纪90年代初期，在一台计算机中，只有一些最基本的输入/输出设备，如键盘、磁盘和显示器。后来随着图形用户界面的引入，鼠标也逐渐成为标准配置。在20世纪90年代中期，随着多媒体计算机的出现，又增加了音箱、麦克风和CD-ROM等设备，有了这些设备以后，计算机的应用水平又提高了一个层次。CD-ROM的存储容量超过600MB，这就使得大规模的应用程序（如比较大的游戏软件）成为可能。而声卡的出现使得计算机从无声世界进入到有声世界，使用户能在计算机上听歌曲、看电影。之后，随着硬件价格的下降，很多计算机又配置了打印机、扫描仪和光盘刻录机等设备。最早的光盘刻录机只有单倍速，而且价格在万元人民币以上。而U盘和移动硬盘的出现，使得存储容量小、不易携带、可靠性低的软盘彻底成为历史。再后来，随着摄像头的微型化和普及，人们可以很方便地进行网络视频聊天或者召开在线会议。总之，在计算机系统中存在着各种不同类型的输入输出设备，因此，如何对这些设备进行管理，使得各种资源能够得到充分、合理的利用，这是操作系统的一个重要任务，也是本章将要讨论的内容。

5.1 I/O硬件

如前所述，I/O设备种类繁多。事实上，如果访问一个电商网站，然后单击它的计算机配件网页，那么在上面就会罗列各种各样的、当前主流的I/O设备，如SSD硬盘、显示器、显卡、鼠标、键盘、U盘、移动硬盘、摄像头等。另外，还有各种游戏装备，如专门为玩游戏而配

置的显示器、显卡、键盘、鼠标和游戏手柄等。

需要指出的是，不同专业的人眼中的硬件设备是不一样的。对于电子专业的人来说，他们关心的是硬件本身，他们眼中的输入/输出设备可能是由一组芯片、导线、电源和马达等物理元器件组成的一个硬件。而对于计算机专业的人来说，他们是从操作系统的角度来看待输入/输出设备的，他们所关心的并不是某个硬件自身的设计、制造和维护，而是如何来对它进行编程，如何让它运转起来。也就是说，这个硬件所接收的控制命令是什么，它所完成的功能是什么，以及它所返回的出错报告有哪些。因此，本书是从这个角度来理解输入/输出设备的。

5.1.1 I/O设备的类型

可以从不同的角度，把输入/输出设备划分为不同的类型。

首先，从设备的交互对象来看，可以把设备分为以下三类。

- 人机交互设备：包括视频显示设备、键盘、鼠标和打印机等。
- 与计算机或其他电子设备交互的设备：包括磁盘、磁带、传感器等。
- 计算机之间的通信设备：包括网卡、调制解调器等。

其次，根据设备的交互方向，可以把设备分为以下三类。

- 输入设备：如键盘、鼠标、麦克风和扫描仪等，数据通过这些设备输入到计算机。
- 输出设备：如显示器和打印机，数据从计算机输出到这些设备。
- 双向交互设备：如磁盘和网卡，它们既能输入，也能输出。

再次，还可以按照数据的组织方式，把设备分为以下两类。

- 块设备：以数据块来作为信息的存储和传输单位，每个数据块都有一个地址，可以直接定位和访问。数据块之间的读写操作是相互独立的，如硬盘。
- 字符设备：以字符来作为信息的存储和传输单位，数据即字节流，只能顺序访问，无定位无寻址，如鼠标、串口和键盘等。

另外，还可以按照数据的传输速率，把设备分为低速设备、中速设备和高速设备；或者从程序的使用角度，把设备分为逻辑设备和物理设备。

总之，对于各种类型的输入/输出设备，它们之间的差别主要表现在数据的传输速率、应用领域、控制的复杂程度、数据的传输单位、数据的表示方法，以及出错条件等方面。

5.1.2 设备控制器


扫码观看

刚才讨论的是输入/输出设备的类型，请读者思考一个问题：有了这样的一些设备，是不是就能够实现各种输入/输出功能呢？或者说，各种I/O设备千奇百怪，那么能否有一个统一的、标准的硬件访问接口呢？

以最常用的两种I/O设备键盘和鼠标为例。对于普通的计算机用户来说，他们在使用键盘时，是直接跟键盘上的按键打交道，想按哪个键就按哪个键。但是对于操作系统来说，如果想让键盘正常地工作，就必须了解其内部的工作原理。如果下次读者家里的键盘坏了，在扔到垃圾桶之前，可以先做一件事情：把它拆开，而且是彻底地拆开，拆成最小的零部件。这时就会发现，键盘的基本单位是按键，这些按键通过一些电路和电子元器件(电阻、电容等)连接起来。对于每一个按键，把它拆开以后，里面是一些机械装置，包括开关帽、底座、触

点金属片、弹簧、固定卡和跳线等。显然,发明和制造键盘的,应该是一些电子和机械专业的工程师。

而对于鼠标来说,最外面是一个塑料外壳和两个按键,把它拆开来以后,里面有无线模块、滚轮、微动开关、主控芯片、存储芯片、快捷键等。显然,鼠标的内部装置和工作原理与键盘是完全不同的。

这样问题就来了:在一个计算机系统当中,有着各种不同类型的输入/输出设备,而每一种设备的内部装置和工作原理都是各不相同的。在这种情形下,如果你是操作系统的设计者,你该怎么办?更重要的是,操作系统的设计者往往是计算机专业出身,而不是电子或机械专业出身,要想让他们去弄明白所有设备的所有电子和机械原理,这是完全不可能的。

为了解决这个问题,一种自然而然的想法就是把不同类型的硬件设备的内部实现细节封装在一个黑箱中,然后对外提供一个标准的、统一的接口。这样一来,对于程序员来说,就不用去关心它内部的实现原理,什么开关帽、触点金属片、弹簧和滚轮,这些都不用去关心,只要知道这个接口的访问方式即可,然后所有设备的接口都是类似的。这个接口就是设备控制器(Device Controller)。

图5.1是一个简单的计算机系统的体系结构图。从图5.1中可以看出,对于每一个输入/输出单元来说,它一般是由两个部分组成的,一个是机械部分,一个是电子部分。这两个部分相互合作,共同来完成系统当中的各种输入/输出功能。有的读者可能会问,为什么不把这两个部分合二为一呢,为什么要把它们作为相互独立的部分呢,这主要是为了在设计时,能够更加模块化、更加通用。

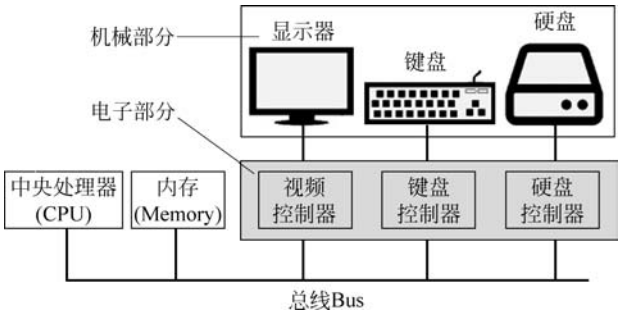

图5.1 简单计算机系统的体系结构

机械部分就是前面所讲的输入/输出设备本身,而电子部分称为设备控制器或适配器(Adapter)。控制器与适配器的区别就是:适配器一般是印制电路卡的形式,它可以很方便地插入到主板的扩充槽当中。而控制器一般是一组芯片,它主要是集成在主板上或者是I/O设备的内部。但不管是适配器还是控制器,它们的功能都是一样的,也就是完成设备与主机之间的连接和通信。例如,为了实现显示功能,把字符或图像显示出来,那么当然需要一个显示器。但光有显示器还是不够的,它只是一个外在的机械设备,光有这个设备还不能完成显示功能。为了实现显示功能,还需要一个视频控制器,通常的称呼是显卡。显示器是连在显卡上,而显卡是插在计算机的主板上。只有当显示器和显卡相互配合时,才能够在显示器上显示各种图像。类似地,对于键盘和磁盘驱动器这些输入/输出设备来说,它们都有相应的设备控制器。有的读者可能会说,我的键盘买回来就直接能用,并没有去买什么键盘

控制器。这是因为键盘控制器的功能很简单，价格也便宜，所以就直接集成在主板上，不用单独去买了。

在设备控制器上通常有一个插槽，可以用电缆把它和相应的输入/输出设备连接起来。另外，在控制器和输入/输出设备之间的接口可以定义为一个标准接口。例如，符合 ANSI、IEEE 或者是 ISO 这样的国际标准，或者是某种事实上的工业标准。这样一来，对于不同的制造商来说，它们有的是生产控制器的，有的是生产输入/输出设备的，但这些都没有关系，只要它们生产出来的产品能够符合这个标准，那么不同厂家之间的产品就能够随意地配对组合，并且正常使用。这样就提高了设备的通用性。例如，如果组装一台计算机，那么对于显示器，可以购买各种不同的品牌，如三星、冠捷或者飞利浦，而对于显卡来说，也可以选择不同的品牌，这都没有关系，可以把它们任意组合起来。

5.1.3 I/O 地址

如前所述，每一种类型的输入/输出设备都有一个相应的设备控制器，设备与设备控制器结合起来，才能完成相应的输入/输出功能。如图 5.2 所示，输入/输出设备本身并不直接跟 CPU 打交道，而是通过它的设备控制器来跟 CPU 打交道。具体来说，每个设备控制器里面都有一些寄存器，用于和 CPU 进行通信，包括控制寄存器、状态寄存器和数据寄存器等。例如，通过往控制寄存器中写入不同的值，操作系统就可以命令该设备去执行发送数据、接收数据、打开和关闭等操作。另外，操作系统也可以通过读取状态寄存器的值，来了解该设备的当前状态，如就绪、繁忙等。除了这些寄存器以外，很多设备还有一个数据缓冲区，可以供操作系统来读写。

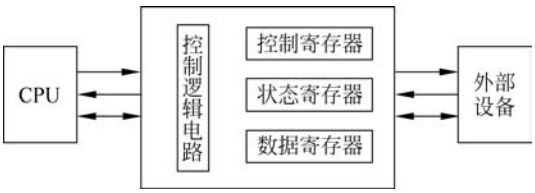

图 5.2 设备控制器的内部结构

现在的问题是：CPU 如何与这些控制寄存器以及数据缓冲区进行通信呢？事实上，在 CPU 中执行的是一条条的指令，在执行指令时，如果是对普通的内存单元进行访问，那么很简单，只要指明这个内存单元的起始地址即可。但现在要处理的是设备控制器里面的寄存器，这又如何来访问它们当中的内容呢？解决方法主要有三种：I/O 独立编址、内存映像编址以及混合编址。

1. I/O 独立编址

I/O 独立编址的基本思路是：由于系统中有很多 I/O 设备，每个设备都有一个设备控制器，而每一个控制器中都有若干个寄存器。因此可以给所有设备控制器中的每一个寄存器分配一个唯一的 I/O 端口编号，也称为 I/O 端口地址，然后用专门的输入/输出指令来对这些端口进行操作。这些端口地址所构成的地址空间是完全独立的，与内存地址空间没有任何关系。

图 5.3 是 I/O 独立编址的一个示意图。总共有两个地址空间，一个是内存地址空间，

其中的每一个地址,都对应于一个内存单元;另一个是I/O端口地址空间,其中的每一个地址,都对应于某一个设备控制器中的一个寄存器。

例如,对于汇编指令:

```
MOV R1,[2]
```

这是一条普通的内存访问指令,它表示把内存地址为2的那个内存单元的内容读进来,并且保存在CPU的寄存器R1当中。

而对于指令:

```
IN R1,[2]
```

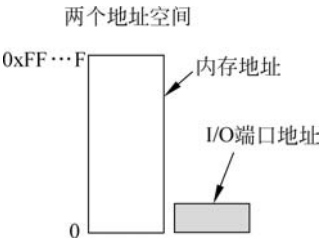

图5.3 I/O独立编址

这条指令也是去访问地址为2的单元,但它是一条专门的输入/输出指令,它表示把I/O端口地址为2的那个寄存器的内容读进来,并且保存在寄存器R1当中。类似地,如果要把一个寄存器的值写入到某个I/O端口地址,可以用OUT指令。

采用I/O独立编址的方法,其优点是:输入/输出设备不会占用内存地址空间。而且在编写程序时,对于每一条指令,很容易区分它是对内存进行访问,还是对输入/输出端口进行访问。因为对于不同的操作来说,其指令的形式是不一样的。

I/O独立编址的典型例子是早期的8086/8088芯片,它给I/O端口分配的地址空间为64KB,只能用IN和OUT指令来对这些端口地址进行读写操作。

```
mov al, #0x11
out #0x20, al ! 发送到中断控制器 8259A-1
mov al, #0x20 ! 硬件中断的起始 (0x20)
out #0x21, al
mov al, #0x28
out #0xA1, al
…
in al, #0x64 ! 键盘控制器的状态寄存器端口
test al, #2
jnz empty_8042
…
```

以上是Linux系统启动时执行的一些代码,其中使用了IN指令和OUT指令,分别对中断控制器和键盘控制器内部的不同寄存器进行了访问。

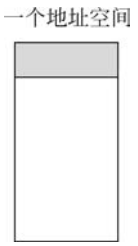

图5.4 内存映像编址

2. 内存映像编址

内存映像编址的基本思路是:把所有设备控制器当中的每一个寄存器都映射为一个内存地址,专门用于输入/输出操作。从操作的层面来看,对这些地址的访问与普通的内存访问是完全相同的。

如图5.4所示,在内存映像编址方式下,端口地址空间与内存地址空间是统一编址的。总共只有一个地址空间,端口地址空间是内存地址空间的一部分,它一般位于内存地址的高端,而地址低端为普通的内存地址。

在内存映像编址方式下，如图 5.5 所示，当 CPU 要访问内存或 I/O 设备时，就把一个地址打在地址总线上。如果该地址位于内存地址区间，则去访问内存；如果该地址位于 I/O 地址空间，则去访问 I/O 设备。那么系统如何判断该地址是位于哪一个区间内呢？有几种不同的做法，例如，可以把地址的高位作为片选信号，或者是在存储管理单元（MMU）中设置相应的地址范围。

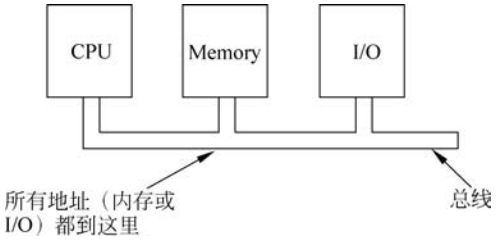

图 5.5　内存映像编址方式下的寻址

内存映像编址的优点是：

- 编程方便，无须专门的输入/输出指令，可以像普通的内存单元那样来访问输入/输出端口，甚至可以用 C/C++等高级语言来编程。
- 对普通的内存单元可以进行的所有操作指令，都可以同样作用于输入/输出端口。包括基本的读操作、写操作，也包括其他一些内存操作，如 test 指令，它用来测试一个内存单元的值是否为 0。
- 在内存映像编址方式下，无须专门的保护机制来防止用户进程去执行 I/O 操作。如前所述，I/O 操作是一种特权指令，不允许用户进程直接使用，而必须通过系统调用的方式，由系统态的内核程序来完成。为了做到这一点，只要借用存储管理当中的内存保护机制即可。也就是说，把这些端口地址排除在用户进程的逻辑地址空间之外，不允许它们去访问。

内存映像编址的缺点是：

- 不能对控制寄存器的内容进行 Cache，必须关闭。
- 每一次都要判断访问的是内存还是 I/O。

如前所述，CPU 在访问内存时，需要使用 Cache，这样能减少对内存的访问次数，提高访问速度。但是在访问 I/O 设备时，不能使用 Cache。因为对于内存单元来说，无论是读操作还是写操作，都是由 CPU 来控制的。如果 CPU 不去修改，则内存单元的值就不会改变。因此，Cache 中的值总是最新的。但设备控制器不一样，即使 CPU 不去访问它，它里面的值也可能会发生变化。例如，如果一次 I/O 操作完成，则状态寄存器的值就会发生变化，从繁忙变为就绪。在这种情形下，为了确保每次读入的都是最新的值，就不能使用 Cache，而必须直接去访问 I/O 端口。打个比方，有一个人做了一锅汤，想尝一尝汤的咸淡。他拿了一个小碗盛了点汤，一尝，发现淡了。于是往锅里加了一些盐，然后又拿起刚才的小碗尝了尝汤，发现还是淡了，因为小碗里的汤并没有发生变化。就这样，当他往锅里加了无数盐之后，小碗里的汤还是淡的。毕竟，碗里的汤是最早的旧的汤，而锅里的汤才是当前最新的汤。这就好比 Cache 当中的内容，仍然是过去的内容，而不是 I/O 设备的最新内容。

3. 混合编址

混合编址的基本思路是把以上两种编址方法混合在一起。具体来说,对于所有设备控制器当中的寄存器来说,它采用的是I/O独立编址的方法,每一个寄存器都有一个独立的I/O端口地址。而对于设备的数据缓冲区来说,它采用的是内存映像编址的方法,把它们的地址统一到内存地址空间当中。例如,Intel公司的Pentium处理器采用的就是这种方案。它把内存地址空间当中640KB~1MB这一段区域保留起来,专门用作设备的数据缓冲区。另外,它还有一个独立的I/O端口地址空间,大小为64KB。图5.6是混合编址的示意图,在内存空间中有一块I/O地址空间,另外还有一个独立的I/O地址空间。

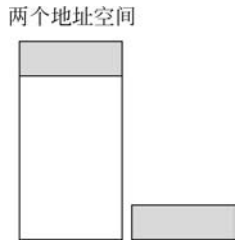

图5.6 混合编址

在图5.7中,列出了个人计算机上的部分I/O端口地址。例如,对于可编程中断控制器(Programmable Interrupt Controller),它的端口地址是从0020到0021,占用了两个端口地址。这说明在这个控制器当中,有两个寄存器。当然,对于不同的计算机,这些端口地址可能会有一些差别。读者可以看一看自己计算机上各种设备控制器的端口地址是什么。以Windows 10系统为例,在菜单栏的"Windows系统"目录下单击"控制面板",然后在随后出现的对话框中单击"硬件和声音",这时,在"设备和打印机"一行,就会出现一个"设备管理器"按钮,单击该按钮,就会出现一个单独的应用程序。然后在它的"查看"菜单栏中选择"按类型列出资源",这样就可以看到每一种不同类型的I/O设备所占用的内存、输入/输出端口和中断请求等资源。图5.7列出的是其中的输入/输出端口资源。

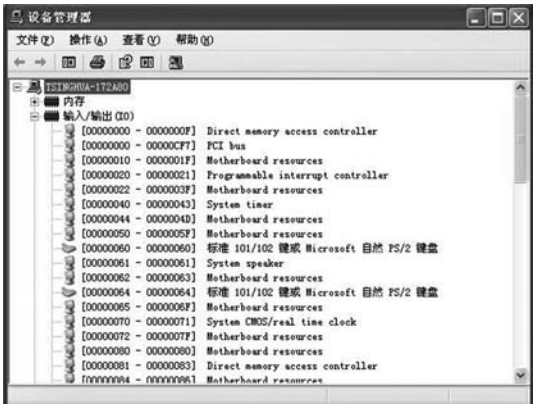

图5.7 PC上的部分I/O端口地址

到目前为止,已经介绍了I/O设备的类型、设备控制器以及I/O端口地址。那么根据这些知识,能不能开始编程使用这些设备,来完成相应的输入/输出功能呢?如果能,那应该如何去做呢?答案就是采用I/O控制方式。

5.2 I/O控制方式

当我们拿到一个新的硬件设备时,光有硬件本身是不行的,要想让它正常地运转起来,还需要编写软件,用软件来指挥它运行,这就需要用到I/O控制方式。具体来说,I/O控制

就是在 CPU 上执行指令，与 I/O 设备的设备控制器中的各种寄存器进行通信。例如，往控制寄存器中写入各种操作命令，从状态寄存器中读出当前状态，以及从数据寄存器中读入或写入数据，所有这些过程都必须在 CPU 中通过指令的形式来完成。

I/O 控制既有非常简单的方式，也有比较复杂的方式。有的不需要任何额外的硬件支持，单凭软件即可完成。有的则需要中断机制或 DMA 控制器等硬件的支持。具体采用哪一种 I/O 控制方式，取决于具体的 I/O 设备本身，以及系统的软硬件配置、对性能的要求等因素。一般来说，当前的 I/O 控制方式主要有三种：程序循环检测方式(Programmed I/O)、中断驱动方式(Interrupt-driven I/O)和直接内存访问方式(Direct Memory Access)，每一种方式都有各自的优点和局限性。

5.2.1 程序循环检测方式


扫码观看

何谓程序循环检测方式？先来看一个生活中的例子：小朋友在家里吃饭。对于一个只有一两岁的小朋友，他或她一般不太愿意也不太会自己吃饭，一定要大人喂。那么大人喂饭的过程一般是这样：如果小朋友的嘴巴没空(如上一口饭菜尚未吃完)，那么此时不能再喂，而必须耐心等待。至于等待时间的长短，往往取决于对方的心情。等到嘴巴空出来以后，就再盛一勺饭菜，喂到小朋友的嘴里。然后重复上述步骤，直到一碗饭全部吃完。

对于基于程序循环检测的 I/O 控制方式，它的基本思路也是类似的。具体来说，在程序(一般是设备驱动程序)当中，通过不断地检测 I/O 设备的当前状态，来控制一个 I/O 操作的完成。在进行 I/O 操作之前，先要循环地去检测设备是否已经就绪。也就是说，要去查看该设备的控制器中的状态寄存器，看它是否空闲。如果不空闲，就只能等待；如果空闲，就向控制器发出一条命令，启动这次 I/O 操作。然后，在这个操作的进行过程中，也要循环地去检测设备的当前状态，看它是否已经完成。最后，在 I/O 操作完成以后，如果这是一次输入操作，那么还要把读进来的数据保存到内存中的某个位置。总之，从硬件上来说，在 I/O 操作的整个过程中，控制 I/O 设备的所有工作都是由 CPU 来完成的。这种方式也称为繁忙等待方式或轮询方式。它的缺点是在进行一个 I/O 操作时，要一直占用着 CPU，而 CPU 和 I/O 设备之间的运行速度是不匹配的，CPU 很快而 I/O 设备很慢，这样就会浪费大量的 CPU 时间。还是以喂饭为例，小朋友吃饭的速度是非常慢的，他们是边吃边玩，一顿饭得吃 1 小时甚至更长的时间。在这段时间当中，喂饭的大人什么也干不了，只能饿着肚子在那里陪着。

需要说明的是，I/O 控制与 I/O 操作是不一样的，两者不能混为一谈。I/O 控制是由 CPU 来进行，而 I/O 操作是由设备自己来完成。例如，假设我们去磁盘读入一个数据块，那么读磁盘的命令是由 CPU 发出的，具体方式就是往磁盘控制器内部的控制寄存器当中写入命令。但是真正去访问磁盘设备、把数据读进来，这个操作是由磁盘控制器和磁盘驱动器这两个硬件相互配合、自己完成的，与 CPU 没有关系。因此，对于程序员来说，关心的是如何编写程序，对 I/O 设备进行控制，对控制器内部的各个寄存器进行读写。至于 I/O 设备内部的工作原理，根本就不用去关心。举一个生活中的例子，当我们在使用洗衣机来洗衣服的时候，需要做的事情是：把脏衣服放进去，再倒入洗衣液，最后按一下启动按钮，这样就可以了。至于洗衣机的内部工作原理，它是如何洗干净衣服的，我们根本不用关心。

下面来看一个具体的例子。在一个嵌入式系统中，有一个字符显示设备，能够把一个个

字符显示在一块小屏幕上。已知在系统中,I/O 地址采用的是内存映像编址方式,现在需要在这个字符设备上显示一个字符串“ABCDEFGH”。对于操作系统来说,要完成这个任务其实很简单,只要把这 8 个字符一个接一个地送到该显示设备的相应的 I/O 端口即可。而且由于采用的是内存映像编址的方式,因此这些端口地址就是普通的内存地址。对这些地址所进行的操作,就是普通的内存访问操作,可以用 C 之类的高级语言来实现,而不需要去使用什么专门的输入/输出指令。

如图 5.8 所示,需要显示的这个字符串被保存在系统内核的一个缓冲区中,它的起始地址保存在指针 p 当中。在内存的地址空间中,有一个单元对应于字符显示设备控制器当中的状态寄存器,其地址保存在指针 display_status_reg 当中。另一个单元对应于设备控制器当中的数据寄存器,其地址保存在指针 display_data_reg 当中。现在要做的事情,就是把这 8 个字符一个接一个地放入到 display_data_reg 所指向的内存单元中。这个操作表面上是一次内存访问操作,但实际上是把字符送到设备的控制器当中,而控制器就会自动地把这些字符送到显示设备的屏幕上去。另外,在送数据的过程中,要不断地去查询状态寄存器的值。需要说明的是,p、display_status_reg 和 display_data_reg 这三个指针变量同样是位于内存空间当中的,图 5.8 只是为了直观起见,把它们单独画在旁边。

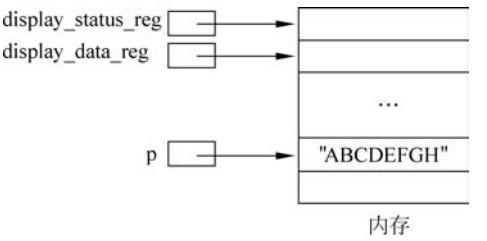

图 5.8　I/O 控制的一个例子

根据上述要求,可以编写如下 C 语言程序来完成这项功能。在这段程序中,count 表示需要显示的字符个数,i 是循环变量,表示当前正在打印的是第 i 个字符,其初始值为 0。在循环体中,第一条语句是一个循环体为空的循环检测语句,用来判断显示设备是否已经就绪。如果尚未就绪,就在该处循环等待;如果已经就绪,则退出循环检测语句,然后把第 i 个字符复制到设备的数据寄存器当中。从代码中可以看出,在程序员眼中,这个数据寄存器其实就是一个普通的内存单元,因此,这个操作就是一个简单的赋值操作。但它的功能和普通的赋值操作有所不同,它相当于是给设备发出了一个命令,命令它去显示一个字符。

```
for(i = 0; i < count; i++)
{
        while( * display_status_reg != READY);
        * display_data_reg = p[i];
}
```

如果读者对上述这段代码还有所疑虑,不妨把喂饭那个例子也写成相应的伪代码,仅供参考和比较。

```
while(饭未吃完)
{
      while(小朋友嘴巴没空) 等待;
      装一勺饭菜,喂到小朋友嘴里;
}
```

显然,这两段代码的程序结构几乎是完全一样的,其功能也是差不多的。

此外,有的读者可能还有另外一个疑问,就是为什么这个程序是一个两重的循环语句,即为什么每次在显示下一个字符之前,都要先用一个循环语句来检测一下,如果不这么做行不行。具体来说,能否把这个程序修改为以下形式?

```
while( * display_status_reg != READY);
for(i = 0; i < count; i++)
{
      * display_data_reg = p[i];
}
```

这段代码的意思是:首先查询显示设备的状态是否就绪,如果尚未就绪,就在该处循环等待;如果已经就绪,那么就一个接一个地把每一个字符送入到它的数据寄存器。

这种写法是不行的,由于 CPU 与 I/O 设备的运行速度不匹配,将会造成显示数据的丢失。具体来说,对于赋值语句

```
* display_data_reg = p[i];
```

它仅仅是一条赋值语句,因此,它的执行速度是非常快的。事实上,CPU 在执行指令的时候,速度是纳秒一级的。因此,这条语句将会很快执行完,然后会执行 i++操作,把循环变量加 1。再判断循环控制条件,发现条件成立,因此又开始执行新一轮的循环,去显示下一个字符。但是相对来说,字符显示设备是一个慢速设备,它在执行这个显示命令时,在把一个字符真正显示在屏幕上时,不可能像 CPU 那么快,而是需要一定的时间来完成,如毫秒一级。这样,当 CPU 再一次执行到循环体中的赋值语句,试图去显示第二个字符时,对于设备控制器来说,它虽然已经把第一个字符给取走了,但是还没有来得及处理,也就是说,它还是处于繁忙状态,还不能接收新的数据。如果此时硬要把第二个字符塞给它,就会造成数据的丢失。因此,正确的做法应该是:当 CPU 把上一个字符交给了数据寄存器之后,就应该循环等待,等到对方处理完这个字符以后,其状态就会变成就绪状态,这时才能把下一个字符放进去。

另外,在上面这个例子中,I/O 编址采用的是内存映像编址方式,如果改为 I/O 独立编址,那么这段程序该如何修改呢?

其实程序的基本框架是差不多的,也是两重循环。只不过在访问设备控制器中的寄存器时,所用到的指令不太一样。在读取状态寄存器的值时,需要使用 IN 指令。而在把数据写入到数据寄存器时,需要使用 OUT 指令。换句话说,如果采用的是 I/O 独立编址,那么就必须使用汇编语言来编程实现。或者说,程序的大部分内容仍然可以用 C 语言来实现,只不过在访问状态寄存器和数据寄存器时,要在 C 语言中嵌入一些汇编语言指令,这样才

能使用 IN 和 OUT 指令。

5.2.2 中断驱动方式

循环检测的控制方式会占用大量的 CPU 时间。事实上,在一个 I/O 操作的整个过程中,所有的控制 I/O 设备的工作都是由 CPU 来完成的,这样就会造成 CPU 时间的浪费。例如,在上面的例子中,假设字符显示设备的显示速度为 100 字符/秒,那么在循环检测的方式下,当一个字符被写入到显示设备的数据寄存器以后,CPU 需要等待 10ms 才能把下一个字符写进去。而在这 10ms 的时间内,CPU 一直在执行循环等待,这样 CPU 时间就被白白浪费掉了。10ms 看上去似乎不太多,但是不要忘了,指令的执行速度是纳秒一级,因此 10ms 能执行很多条指令。如何解决这个问题?一种办法就是让 CPU 在这 10ms 的时间内,不要在那里干等着,而是去做一些其他的、有用的事情,如去运行其他的进程。然后等到显示设备已经处理完上一个字符时,CPU 再接着去输出下一个字符。而要做到这一点,就必须依赖于硬件的支持,采用中断技术。因此,这种方法被称为中断驱动的控制方式。


扫描观看

在介绍中断驱动方式之前,同样先来看一个生活中的例子。假设小朋友现在已经长到三四岁了,已经上幼儿园了。那么在幼儿园当中,小朋友是如何吃饭的呢?肯定不能像在家里那样专门有人喂,因为如果采用这种方式(即循环检测方式),那么就必须单独占用一个大人,这样这个大人别的事情就都干不了,只能在那儿专门喂饭。但是在幼儿园,一个班往往有二三十个孩子,而老师通常只有三个,因此不可能采用这种方法。事实上,在幼儿园,小朋友的吃饭过程一般如下。

- 老师从食堂取来一大桶米饭和一大桶菜。
- 如果小朋友们尚未准备好吃饭,则循环等待,直到他们准备就绪。
- 老师将饭菜装入每个小朋友的小碗。
- 小朋友们开始吃饭,而老师则去做别的事情。
- 在吃饭过程中,小朋友们可以通过各种信号打断老师。

也就是说,在幼儿园,小朋友们是自己吃饭的,通常不需要老师喂。这样,当小朋友们在吃饭的时候,老师就可以在旁边忙活其他的事情,如收拾房间、制作教具等。然后小朋友们如果有需要,可以通过各种信号来打断老师。当老师被打断以后,她就会去查看发生了什么事情,具体来说:

- 如果小朋友举着小手,这说明他/她碗里的饭已经吃完了,还想再吃一碗,因此就给他/她添饭。
- 如果小朋友举着拳头,这说明他/她碗里的汤已经喝完了,还想再喝一碗,因此就给他/她添汤。
- 如果小朋友吃完饭了,就给他/她收拾碗和勺子。

当老师处理完一个小朋友的中断请求之后,又回到刚才的状态,去收拾房间或制作教具。需要说明的是,以上例子中的举手、举拳头等信号,都是清华洁华幼儿园的规定,而其他的幼儿园不一定是这样。至于为什么要举手、举拳头,而不是直接告诉老师你的需求,那是因为在吃饭时是不允许说话的,怕噎着。

在上面这个例子中,把教师比喻为 CPU,把小朋友比喻为 I/O 设备。对于教师而言,她的工作主要有两块,一块是与就餐有关的,另一块是与就餐无关的(收拾房间或制作教具)。

对于前者，又包括两个部分，一是在准备就绪后启动就餐；二是在就餐过程中及时响应小朋友的请求。对于基于中断驱动的I/O控制方式，它的基本原理也是类似的。

那么在一个计算机系统当中，如何实现基于中断驱动的I/O控制方式呢？为了回答这个问题，先要弄清楚中断的概念。

对于一个操作系统而言，中断是非常重要的。有人把中断对于操作系统的重要性比喻为机器中的驱动齿轮。没有齿轮的机器是没有办法工作的，同样，没有中断的操作系统，也是没有办法正常运行的。因此，有人把操作系统称为由“中断驱动”或者“中断事件驱动”的。

所谓中断，指的是由于某个事件的发生，改变了正在CPU上执行的指令的顺序。这种事件对应于CPU芯片内部或外部的硬件电路所生成的电信号。中断处理的过程一般是这样：当中断事件发生时，CPU会暂停当前正在执行的程序，并且在保留现场后自动转去执行相应事件的处理程序，当处理完以后再返回断点，继续执行被打断的程序。

中断可以分为两大类，即同步中断和异步中断。

所谓同步中断，是指当CPU正在执行指令时，由CPU的控制单元所发出的中断，也称为“异常”。异常又可以分为以下两类。

- 由CPU检测到的异常，包括错误(Fault)、陷阱(Trap)和中止(Abort)。例如，算术溢出、被零除、用户态下使用了特权指令等，都会引发相应的异常。
- 由程序主动来设定的异常，也就是说，程序员通过int、int3等指令来发出的中断请求，也称为软中断，它主要用来实现系统调用服务。

所谓异步中断，指的是由CPU以外的其他硬件设备在任意时刻所发出的中断，简称为“中断”。它也可以分为以下两类。

- 可屏蔽中断，即I/O中断。它是当外部设备操作正常结束或发生错误时所发生的中断。例如，打印机打印完成或缺纸，读磁盘时驱动器中没有磁盘等。
- 不可屏蔽中断，即由掉电、存储器校验错误等硬件故障引起的硬件中断。

这里讨论的主要是I/O中断，即外部输入/输出设备引发的中断。

如图5.9所示为一个典型的计算机系统中的中断机制。中断控制器负责管理系统中的所有I/O中断，只有它才能向CPU发出中断请求。而对于普通的I/O设备，当它需要发送中断时，不是直接发给CPU，而是先发给中断控制器，并由它来决定是否要转发给CPU。在硬件层面上，当一个I/O设备完成了CPU交给它的I/O任务以后，它的设备控制器就会向中断控制器发出一个信号，该信号会被中断控制器检测到。这时，中断控制器就会判断一下，看看当前是不是已经有一个中断正在处理，或者是否有一个更高优先级的中断同时出现。如果都没有，那么就开始处理这个信号。一方面，它会把一个编号放在地址总线上，这个编号标明了是哪一个设备所发出的中断请求；另一方面，它会向CPU发出一个中断信号。然后CPU就会中断当前的工作，并且用这个编号作为索引去访问一个中断向量表。在中断向量表中，存放的是每一个中断处理程序的起始地址。这样就能找到与该中断相对应的中断处理程序的起始地址，然后跳转到该程序去运行。当这个中断处理程序开始运行后不久，就会向中断控制器发出一个确认信号，表示这一个中断已经被处理。这样，中断控制器就可以发出新的中断请求了。

还是以前面的字符显示的例子来说明中断驱动方式的基本思路。在这种方式下，对于

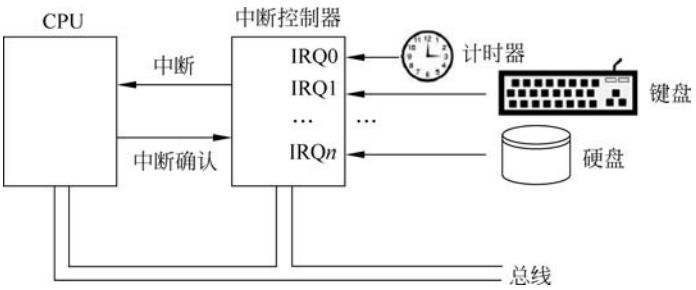

图 5.9　中断机制

用户进程来说,它需要做的事情就是把字符串"ABCDEFGH"放到一个缓冲区 buffer 中,然后调用一个系统调用函数 display()去把它打印出来。

用户进程代码如下。

```
strcpy(buffer, "ABCDEFGH");
display(buffer, strlen(buffer));
```

对于系统调用函数 display(),它的代码如下。

```
copy_from_user(buffer, p, count);      // p: 内核缓冲区
enable_interrupts( );
while( * display_status_reg != READY);
i = 0;
 * display_data_reg = p[i];
scheduler( );
```

这段代码的基本思路:首先把 buffer 缓冲区中的字符串复制到一个字符数组 p 中,p 位于系统内核中。为什么要做这件事情呢?为什么不直接去使用 buffer 缓冲区呢?原因在于:buffer 缓冲区是位于用户地址空间当中,而现在需要在内核空间中运行,这两个空间是相互独立的。在存储管理一章曾经讨论过,在现代操作系统中,对于每一个进程,都有一个 4GB 的虚拟地址空间。其中 2GB 是用户地址空间,用来存放用户进程自己的、独立的内容,包括代码和数据。而另外 2GB 是内核地址空间,用来存放操作系统的代码和数据,而且所有进程的内核地址空间是共享的,内容是一致的。在本例中,用户进程的代码和数据缓冲区 buffer 是位于用户区,而系统调用函数 display()和数据缓冲区 p 是位于内核区。由于 display()函数是运行在内核态,因此要把用户缓冲区 buffer 当中的内容复制到内核空间。

那么 copy_from_user()这个函数如何实现呢?其实很简单,直接去访问即可。当进程发起系统调用,从用户态进入到内核态以后,并没有发生进程切换,仍然是同一个进程。因此,页表是完全一样的。只不过在用户态下,使用的是页表的下半部分。而在内核态下,使用的是页表的上半部分。而且在内核态下,完全可以去访问用户空间。

接下来是打开中断,即把程序状态字寄存器中的中断位打开,这样,CPU 就能够处理中断。然后是一个循环检测语句,判断字符显示设备的当前状态是否空闲。如果不空闲,就一

直循环等待。当发现它空闲后，就把循环变量 i 初始化为 0，并把需要显示的第一个字符 p[0]放入显示设备的数据寄存器当中，让它显示出来。与循环检测方式不同的是，这里仅仅是把第一个字符放入到数据寄存器中，然后就不管了。至于这个字符何时显示结束，以及剩余的那些字符如何显示，这些都不是在 display()函数中完成的。事实上，在把第一个字符放入到数据寄存器之后，当前进程就会去调用系统调度程序，切换到另一个进程去执行，而自己则会被阻塞起来，并且挂到相应的阻塞队列中。

对于剩余的那些字符，它们是在哪里处理的呢？是在中断处理程序当中。以下是中断处理程序的代码。

中断处理程序

```
if(i >= count - 1)          //所有字符显示完毕
{
      unblock_user( );
}
else
{
      i = i + 1;
      * display_data_reg = p[i];
}
acknowledge_interrupt( );
return_from_interrupt( );
```

需要指出的是：中断处理程序是什么时候被调用执行的呢？当然是在中断发生的时候。而中断又是在什么时候发生的呢？是在 I/O 设备已经完成了它的 I/O 操作的时候。在本例中，每当设备显示完一个字符以后，就会发生一次中断。而这就说明，每当这个中断处理程序被调用的时候，它就已经知道了一个事实，即上一个字符已经被顺利地显示出来了。因此，这时就没有必要再去循环地检测设备的状态是否空闲，因为它肯定是空闲的，因此可以直接对设备进行操作了。

具体来说，首先检查一下循环变量 i 的值，如果它等于 count－1，说明所有的字符都已经显示完毕，因此就去相应的阻塞队列中把刚才被阻塞的用户进程唤醒，并把它挂到就绪队列中去。否则，说明后面还有需要显示的字符，因此就把循环变量 i 加 1，然后把下一个字符直接复制到数据寄存器当中，此时不需要检测设备是否空闲。接下来是一些后处理操作，先是向中断控制器发出一个确认信号，然后结束中断处理程序，返回到被中断的那个进程。

从整个执行过程来看，首先是一个用户进程（假设是进程 A）在运行，它调用了 display 函数，进入到内核态下运行。然后把第一个字符‘A’送到字符显示设备的设备控制器。控制器于是命令设备来显示这个字符，而这需要 10ms 的时间。与此同时，在 CPU 上原来是进程 A 在运行，现在 A 被阻塞，然后系统调度进程 B 去运行，B 是一个与 A 无关的进程。在 10ms 以后，‘A’字符显示完成。设备控制器会发出一个中断给 CPU，从而打断了进程 B 的执行（注意当前是进程 B 在执行），并且跳转到相应的中断处理程序。此时并没有发生进程的切换，因此仍然是进程 B 在运行，但执行的指令（中断处理程序）实际上是在做进程 A 的工作，是在帮助进程 A 完成此次 I/O 操作。在中断处理程序当中，把第二个字符‘B’送到设备控制器，然后回到进程 B 继续运行。又过了 10ms，当字符‘B’也显示完成后，设备控制

器又会发中断给CPU。如此往复,直到所有的字符都显示完毕。总之,在8个字符显示的整个过程中,在CPU上运行的主要是B进程,然后每隔一段时间会执行一下中断处理程序,而A进程始终处于阻塞状态。与此同时,字符显示设备也在一直工作,它和CPU是两个不同的资源,因此可以同时工作。当所有字符显示完成后,就会把进程A唤醒。

需要指出的是:在中断处理程序当中,在访问显示字符串的时候,使用的是内核中的数据缓冲区p。这就是为什么之前要把进程A中缓冲区buffer的内容复制到内核缓冲区p的原因,此处如果写的是buffer而不是p,那么就无法正确地访问。因为当中断处理程序在运行的时候,是在进程B的资源平台上。进程B的页表与进程A的页表是不同的,而buffer是进程A中的虚拟地址,它不能用进程B的页表来进行地址映射。而p是内核中的地址,无论是进程A还是进程B,它们的内核地址空间中的内容是相同的,因此能够顺利地访问。

最后把中断驱动方式的基本思路总结一下。当一个用户进程需要进行I/O操作时,它会去调用相应的系统调用函数,由这个函数来发起I/O操作,并且在发起之后把该用户进程阻塞起来,然后调度其他的进程去使用CPU。当所需的I/O操作完成时,相应的设备就会向CPU发出一个中断,然后系统可以在中断处理程序当中做进一步的处理。如果还有剩余的数据需要处理,那么就再次启动I/O操作。从这个过程可以看出,在中断驱动的控制方式下,数据的每一次读写还是通过CPU来控制完成的,只不过当I/O设备在进行数据处理的时候(这段时间往往比较长),CPU不必等待,而是可以继续执行其他的进程。

5.2.3 直接内存访问方式

在中断驱动的控制方式下,每一次的数据读写还是通过CPU来控制完成,而且每一次处理的数据量很少,因此中断出现的次数就很多,而中断的处理需要额外的系统开销,因此也会浪费一些CPU时间。例如,对于前面的字符显示的例子,假设设备的显示速度为100字符/秒,那么在循环检测的方式下,当一个字符被写入到设备的数据寄存器以后,CPU需要等待10ms才能写入下一个字符,也就是说,这10ms全部被浪费掉了。而在中断驱动的方式下,这10ms中的大部分会被用来执行其他的进程,但也有少部分用于系统开销。如果中断的次数比较多,那么这些额外的系统开销也还是不少。

如前所述,I/O操作一般分为两个环节。第一个环节是CPU或内存与设备控制器之间的通信,第二个环节是设备控制器与I/O设备之间的通信。在I/O操作启动或完成时,CPU需要访问控制器,并与之交换数据。例如,如果是一次写操作,那么在I/O操作启动时,CPU需要把数据从内存写入到设备控制器内部的缓冲区,然后设备控制器自己去和I/O设备打交道,把这些数据写到设备上。反之,如果是读操作,CPU先给控制器发信号,让它去启动I/O操作,把数据读入到控制器内部的缓冲区中,然后CPU再把这些数据读入到内存。因此,第一个环节是CPU与控制器打交道,第二个环节是控制器与I/O设备打交道,从而真正让I/O设备去工作。

对于第二个环节,即控制器与I/O设备打交道,这主要涉及各种硬件实现细节,不在本书的讨论范围。因此,这里只考虑第一个环节,即CPU与控制器之间的数据传送。CPU可以一字节一字节地向设备控制器请求数据,这是最基本的方法。在具体编程实现时,如果是I/O独立编址,那么就使用专门的IN和OUT指令;如果是内存映像编址,那么就使用普通的赋值语句。但不管是哪一种方式,每一次只能传送一字节或一个字。因此,如果需要交换

的数据量比较大，那么就需要重复执行很多次的传送指令，就会浪费大量的 CPU 时间。为了解决这个问题，一种方法是直接内存访问（Direct Memory Access，DMA）的控制方式。它可以避免通过编程来实现大规模的 I/O 数据移动，也就是说，不是用软件来做这件事情，而是让硬件来帮着做。

以磁盘读取操作为例，假设要从磁盘上读取一个数据块。如前所述，磁盘是一种块设备，它是以数据块来作为信息的存储和传输单位，每一个数据块都有一个地址。下面来看一下，如果不使用 DMA 方式，而是使用刚才所说的中断驱动的控制方式，那么整个过程是怎么样的。

（1）CPU 向磁盘控制器发出命令，读取一个数据块。或者准确地说，是磁盘驱动程序在 CPU 上运行的时候，向磁盘控制器发出命令，读取一个数据块。

（2）磁盘控制器从磁盘驱动器中一位接一位地读取这个数据块，该数据块可能包含一个或多个扇区。从磁盘驱动器中读出来的是一连串的位流，这样直到整个数据块都保存在控制器内部的缓冲区当中。

（3）磁盘控制器通过校验位来验证这个数据块是否传送正确，如果正确，就向 CPU 发出一个中断。因此，真正的 I/O 操作是由硬件自己来完成的，是由设备控制器和驱动器自己来完成的。

（4）当操作系统开始运行后，会利用一个循环语句，从磁盘控制器的缓冲区当中读出这个数据块。具体来说，即在每一次的循环内，从控制器的数据寄存器当中，读取一字节或一个字，并把它保存在内存当中。

总之，在中断驱动的方式下，第（1）步是由 CPU 来启动的，而第（2）步和第（3）步都是由 I/O 设备自己来完成的。在此期间，CPU 就可以去运行其他的进程。等到所有的数据都已经正确地从 I/O 设备传送到控制器内部的缓冲区以后，第（4）步再由 CPU 通过一个循环语句把这些数据复制到内存当中。因此，在中断驱动的方式下，CPU 负责第（1）步和第（4）步，而 I/O 设备负责第（2）步和第（3）步。但是在 DMA 方式下，可以把第（4）步也省略掉，不是由软件来完成，而是由 DMA 控制器硬件来完成。这样就进一步解放了 CPU，使它有更多的时间去运行别的进程。

要想使用直接内存访问的控制方式，首先在硬件上要有一个 DMA 控制器。这个控制器可以集成在设备控制器当中，也可以集成在主板上。DMA 控制器的一个特点是能够独立于 CPU，直接去访问系统总线，因此它能代替 CPU 去指挥 I/O 设备与内存之间的数据传送，从而为 CPU 腾出更多的时间。

另外，既然是一个控制器，那么在 DMA 的内部也会有一些寄存器，这些寄存器可以被 CPU 来读写。也就是说，CPU 可以通过写操作来向它发出命令，也可以通过读操作来了解它的当前状态。这些寄存器包括一个内存地址寄存器、一个字节计数器以及一个或多个控制寄存器。这些控制寄存器指明了 I/O 设备的端口地址、数据传送方向、传送单位，以及每一次传送的字节数。

下面来具体看一下，在使用了 DMA 以后，从磁盘上读取一个数据块的整个过程是怎么样的（如图 5.10 所示）。

（1）上层的 I/O 软件调用了磁盘的驱动程序，命令它去读取磁盘上的某一个数据块，并保存在特定地址的内存区域。因此，这个驱动程序就开始在 CPU 上运行。

(2) CPU(即驱动程序)对 DMA 控制器进行编程,即对它的各个寄存器的值进行设置,告诉它应该把什么数据传送到内存的什么地方,以及总共需要传送多少字节。

(3) CPU 向磁盘控制器发出命令,让它去读取一个数据块。

(4) 磁盘控制器从磁盘驱动器中把所需的数据块读进来,保存在它内部的缓冲区当中,并且验证数据的正确性。这项工作需要较长的时间,在此期间,CPU 就可以去运行别的进程。显然,CPU 和 DMA 并不去管数据是如何从 I/O 设备传送到设备控制器当中,这个过程是由设备自己来完成的。

(5) 磁盘控制器完成数据块的读入工作后,向 DMA 控制器发出信号,从而启动这一次的数据传送,即需要把数据块从磁盘控制器内部的缓冲区传送到内存。

(6) DMA 控制器通过总线向磁盘控制器发出一个读操作的请求信号,并且把将要写入的内存地址打在总线上。

(7) 磁盘控制器从内部缓冲区当中取出一字节,并按照 DMA 控制器所给出的地址写入到内存当中。

(8) 当这个写操作完成以后,磁盘控制器会通过总线向 DMA 控制器发出一个确认信号。然后 DMA 控制器就会把内存地址加 1,并且把需要传送的字节数减 1。如果这个计数器的值仍然大于 0,那么就转到第(6)步,传送下一字节。

(9) 当所有数据都传送完毕后,DMA 控制器就会向 CPU 发出一个中断,告诉它数据传输已经完成。这样,当中断处理程序开始运行时,它就知道,从磁盘驱动器当中读出来的数据块,不仅是到了磁盘控制器内部的缓冲区当中,而且已经在 DMA 的控制下,被传送到了内存当中。也就是说,当这个中断发生的时候,整个 I/O 过程实际上已经完成了。

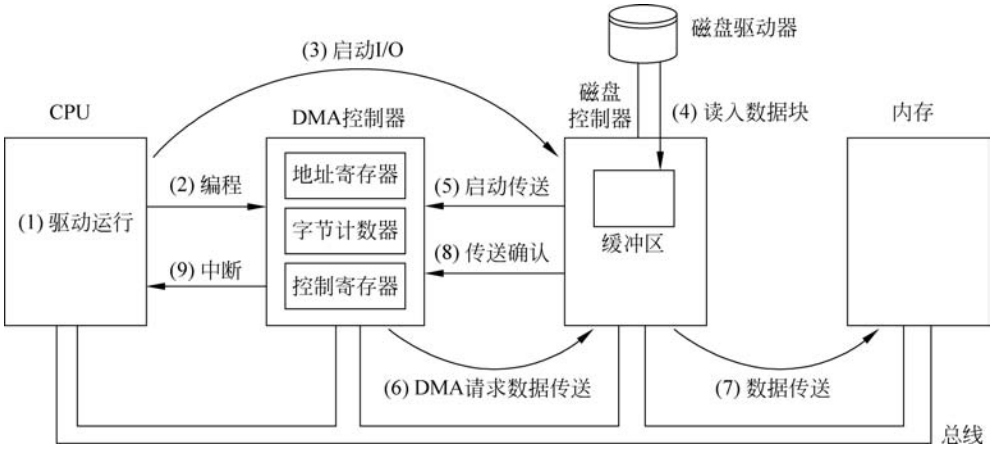

图 5.10　基于 DMA 的磁盘读取过程

再回到前面那个字符显示设备的例子,看看在 DMA 的控制方式下,是怎样的一个解决方案。需要指出的是,这个例子与图 5.10 的磁盘读取操作有两个区别。首先,它是输出而不是输入操作;其次,它是字符设备而不是块设备。

对于用户进程来说,没有任何变化,它需要做的事情仍然是把字符串保存在一个缓冲区 buffer 中,然后调用一个系统调用函数 display()去把它打印出来。

系统调用函数 display()的代码如下。

```
copy_from_user(buffer, p, count);
setup_DMA_controller(p, count);
scheduler( );
```

与中断驱动方式相比，这段代码非常简单。它的最主要的工作就是设置 DMA 控制器，也就是说，对 DMA 控制器进行编程，设置它的各个寄存器的值，包括把内存起始地址 p 放进去，把需要显示的字符个数 count 放进去。此外，还有其他一些初始化的工作。做完了这些事情以后，display()函数就完成了任务，因此就调用系统的调度程序，从就绪队列中选择另外一个进程去运行。而原来的用户进程则会被阻塞起来，直到这个 I/O 操作完成为止。

以下是中断处理程序的代码。

```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

中断处理程序也非常简单，原因在于：当中断发生时，表明 I/O 操作已经全部完成，所有的字符都已经被送到了显示设备的设备控制器当中。此时，已经没有太多实质性的工作，仅仅是把先前被阻塞的进程唤醒，告诉它 I/O 操作已经完成，并把它挂到就绪队列中。也就是说，在这种情形下，在整个 I/O 操作过程中，中断只会出现一次。

从上述代码可以看出，在 DMA 控制方式下，无论是系统调用函数 display()，还是中断处理程序，都非常简单，都没有对字符显示设备进行直接的操作，而是通过 DMA 控制器来完成。CPU 需要做的事情，只是初始化和启动一下，然后 DMA 控制器就会代替 CPU，把内存中需要显示的字符一个接一个地送到设备控制器当中。这样一来，CPU 的工作量就大为减轻，可以腾出更多的时间去执行其他的进程。

5.3 I/O 软件

在前面的内容中，讨论的一直是硬件，以及如何对这些硬件进行控制。但是在学习操作系统时，我们更关心的是软件，与输入/输出有关的软件。具体来说，为了更好地管理系统当中各种各样的输入/输出设备，需要哪一些相关的软件？这些软件各自完成什么样的功能？它们之间的相互关系、组织结构又如何？一般来说，与 I/O 软件有关的角色主要有三个，即应用程序开发人员、操作系统设计者和 I/O 设备厂商。那么在 I/O 软件的设计与实现过程中，这三个角色分别承担什么任务呢？

5.3.1 I/O 软件的层次结构

在操作系统中，为了提高效率，实现模块化管理，I/O 软件的基本思想是采用分层结构，把各种设备管理软件组织成一系列的层次。如图 5.11 所示，一般来说可以分为四层，即中断处理程序、设备驱动程序、设备独立的系统软件以及用户空间的 I/O 软件。每一层都是用来实现特定的功能，相邻的层次之间有着良好的调用接口。其中，低层的软件(包括中断处理程序和设备驱动程序)负责与硬件打交道，与硬件的特性相关，它把硬件和较高层的软

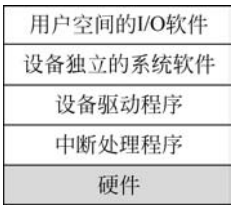

图 5.11 I/O 软件的层次结构

件隔离开来。而较高层的软件(包括设备独立的系统软件和用户空间的 I/O 软件)是独立于硬件的,与硬件的实现细节无关。

1. 中断处理程序

在 I/O 软件的最底层是中断处理程序。前面在讨论 I/O 控制方式时,已经见到过几个中断处理程序的例子。如前所述,当 I/O 设备完成一次 I/O 操作时,设备控制器会向中断控制器发信号,然后中断控制器再向 CPU 发信号,从而触发一次中断。在中断发生后,将跳转到相应的中断处理程序去执行。因此,在 I/O 操作中,中断处理程序是必不可少的。它与设备驱动程序一起合作,共同来完成相应的 I/O 功能。既然要合作,就需要同步。中断处理程序与设备驱动程序之间的同步方式可以采用各种进程间通信方式,如信号量和 P、V 原语。

中断是一种异步行为,难以处理,我们无法准确地知道中断会在什么时候发生。作为操作系统的设计者,应该把中断处理隐藏起来,使得对于用户程序和大多数的操作系统模块来说,它们不必去感知中断的存在。

最后,中断处理需要执行不少的 CPU 指令,如进程上下文的保存和恢复,这些都需要一定的系统开销。

2. 设备驱动程序

设备驱动程序就是与具体的设备类型密切相关的,用来控制设备运行的程序。它一般是由设备的生产厂商提供的。如果自己组装过计算机就会知道,当我们在购买声卡、网卡和显卡等设备时,除了设备本身以外,通常还需要安装相应的设备驱动程序,这些驱动程序可以存放在光盘、U 盘等移动存储介质中,也可以直接从公司的网站下载,它们是由设备生产商制作并提供的。而且对于不同的操作系统,它们往往会有不同的版本。

一般来说,在 I/O 软件当中,真正与 I/O 设备密切相关的,直接对它们进行控制的软件,就是设备驱动程序。只有它才会直接去对设备控制器当中的寄存器进行操作,去读状态命令,去写控制命令。前面讨论的 I/O 控制方式,即通过编写软件的方式来让设备能够正常地运行,实际上讲的就是驱动程序的编写。

设备驱动程序与 I/O 设备之间是密不可分、一一对应的。每一个 I/O 设备都需要相应的设备驱动程序,而每一个设备驱动程序一般也只能处理一种类型的设备。因为对于不同的设备来说,其设备控制器当中的寄存器的数目是各不相同的,而且控制命令的类型也各不相同。例如,对于一个鼠标驱动程序来说,它需要从鼠标这个设备中读取各种各样的信息,包括移动的位置、哪一个按键被按下等。而对于一个磁盘驱动程序来说,它为了进行磁盘的读写操作,就必须知道扇区、磁道、柱面和磁头等各种各样的参数,并使用这些参数来控制磁盘控制器。

如图 5.12 所示,设备驱动程序虽然是由硬件生产厂商提供的,但它一般也是操作系统的一部分,是位于系统的内核空间当中。它们直接对设备控制器进行控制,指挥它们去完成 I/O 操作。而设备控制器又是真正地去和硬件设备打交道。操作系统的其余部分则是与设备驱动程序打交道。

对于不同的 I/O 设备,它们的设备驱动程序的编写方式是不一样的,但也会有一些基本的套路。一般来说,很多设备驱动程序在具体实现时,会执行以下步骤。

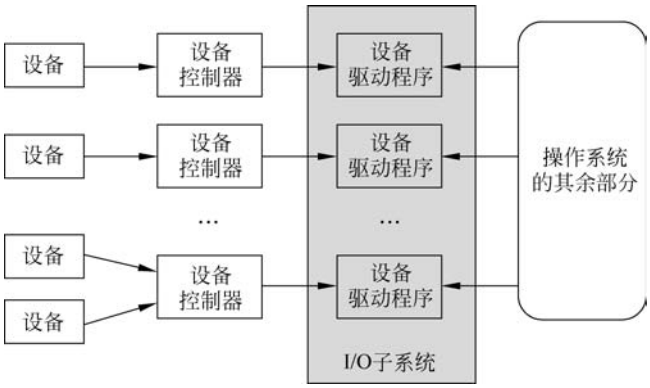

图 5.12　设备驱动程序是操作系统的一部分

(1) 初始化,如打开设备。

(2) 解释系统的命令,检查输入的参数是否有效。如果无效,则返回一个出错报告;如果有效,就要把输入的抽象参数转换为控制设备所需要的具体参数。例如,对于一个磁盘驱动程序来说,它所得到的输入参数可能是一个简单的数据块编号,即线性地址。例如,函数调用 read(fd,15,buf)表示读取磁盘的第 15 个数据块到 buf 中。这个函数是操作系统制定的一个标准接口函数,它适用于所有的块设备,而不仅仅是这个磁盘。但是对于磁盘驱动程序来说,它为了去控制磁盘控制器,当然不能使用这样一个参数,而必须对它进行转换,把它转换为相应的磁头号、磁道号、扇区号和柱面号等具体的参数,然后才能使用这些参数,向设备控制器发出命令。

(3) 检查设备当前是否空闲,如果设备正忙,则这一次的操作请求暂时无法完成,因此把它加入到等待队列,稍后再处理。如果设备空闲,那么再检查硬件的状态,看能否开始运行。

(4) 设备驱动程序向设备控制器发出一连串的命令,即把这些命令写入到控制器的各个寄存器当中,通过端口地址写进去。每发出一条命令以后,可能还需要去检查一下控制器是否已经收到了这条命令,并且已经准备好接收下一条命令。

(5) 当这个 I/O 操作完成以后,驱动程序会去检查出错的情况。如果一切正常,则程序运行结束,并返回一些状态信息给它的调用者。如果这是一个输入操作,那么还要把输入的数据上传给上一层的系统软件。

5.3.2　设备独立的系统软件

如前所述,真正的 I/O 控制的工作是由设备驱动程序(包括中断处理程序)来完成的,而设备驱动程序是由硬件厂商提供的,有了它以后,设备基本上就能正常地运转起来。那么在这种情形下,对于操作系统的设计者来说,需要做什么事情呢? 具体来说,对于操作系统的 I/O 管理模块,它的功能是什么? 事实上,在一些简单的嵌入式系统中,只要有驱动程序就可以正常工作了,连操作系统都不需要。

在计算机的操作系统当中设置 I/O 管理软件,主要有如下几个原因。

- I/O 设备的种类繁多、功能各异,需要标准化接口。
- I/O 设备不可靠,如存储介质失效或传输错误。

- I/O设备不可预测,且运行速度快慢不一。

换言之,操作系统的存在,不是用来解决设备能否使用的问题,而是用来解决设备如何用得更好的问题,即让设备使用起来更加方便。

为了加深读者的印象,先来看一个关于程序员的小段子。

两个程序员是好朋友,一个为iOS系统开发游戏,另一个为安卓系统开发游戏。两个人同时决定各自开发一款游戏给自己的阵营。

一个月过去了,iOS游戏开发者兴奋地跑去找安卓游戏开发者说:“我终于完成了!并且上架了,反响很好,我的第一桶金就要赚到了!你怎么样,搞定没有?”安卓程序员冷冷地说:“还早得很呢”。iOS程序员好奇地问:“为什么?”

安卓程序员依旧冷冷地回答:“因为我要开发的游戏需要支持3英寸、3.2英寸、3.5英寸、3.7英寸、3.8英寸、3.9英寸、4英寸、4.1英寸、4.2英寸、4.3英寸、4.5英寸、4.7英寸、4.8英寸、5英寸、5.5英寸、5.7英寸、6.8英寸、7英寸、7.2英寸、7.5英寸、7.8英寸、8英寸、8.7英寸、8.8英寸、8.9英寸、9英寸、9.2英寸、9.5英寸、9.7英寸、9.8英寸、9.9英寸、10.1英寸、11.1英寸、12英寸和13英寸等屏幕大小,以及240×320px、240×400px、240×480px、320×400px、320×480px、360×480px、360×640px、480×640px、480×720px、480×800px、480×854px、540×960px、600×800px、600×1024px、640×960px、720×1280px、752×1280px、768×1024px、800×1024px和800×1280px等分辨率。还有1核、2核、3核、4核、5核、6核、7核和8核的支持优化。还得要求多窗口运行而且不死机。”

这个段子告诉我们,在软件开发的时候,如果程序员需要直接面对各种各样、不同类型的I/O设备,而且这些设备来自不同的厂家,其设备驱动程序也各不相同,在这种情形下,程序员就会感到非常痛苦。

再举一个例子。假设要去访问一个数据文件,这个文件可能是存放在机械硬盘上,也可能是存放在固态硬盘、光盘、移动硬盘或U盘上。不同存储设备的生产厂商是不一样的,驱动程序也是不一样的,在这种情形下,对于程序员来说,应该如何来编写程序?难道要把所有的存储设备都枚举一遍?

总之,为了使程序员更好地去使用I/O设备,在操作系统当中就必须专门有一层软件,即设备独立的I/O软件,也称为内核I/O子系统。它的主要功能包括:给上层应用的统一接口、与设备驱动程序的统一接口、提供与设备无关的数据块大小以及缓冲技术等。

1. 应用程序与操作系统的接口

如图5.13所示,对于应用程序开发人员,他不会直接与底层的硬件设备打交道,而是与操作系统打交道,操作系统会提供一个应用程序编程接口(Application Programming Interface,API),让编程人员来调用。那么对于应用程序开发人员来说,他们希望操作系统提供什么样的接口呢?这些接口函数应该具备什么样的一些特点呢?

显然,最根本的目标只有一个,即越简单越好,使用越方便越好,就像“傻瓜相机”一样,一看就会用。具体来说,有如下三个方面。

- 设备独立性:使用户在编写程序、访问各种I/O设备时,无需事先指定特定的设备类型。例如,假设需要访问一个文件,该文件可能来自硬盘、U盘或光盘,显然,这些设备都是各不相同的,我们希望各种类型的设备之间的差异由操作系统来处理,对用户来说是透明的,不必关心。

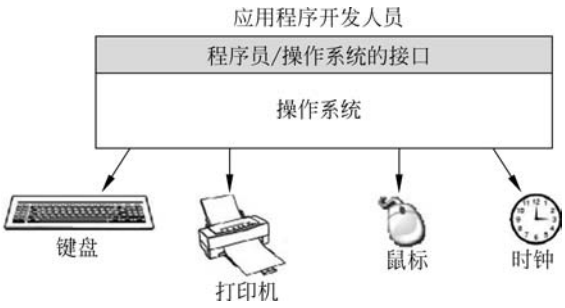

图 5.13　应用程序与操作系统的接口

- 统一命名：用简单的字符串或整数的方式来命名一个文件或设备。例如，在 UNIX 系统中，所有的文件和设备都采用相同的命名规则，即路径名。
- 阻塞与非阻塞 I/O：我们希望操作系统提供的 API 函数分为两类，一类是阻塞性的，即当进程启动一个系统调用后，它会被阻塞起来，直到这次 I/O 操作完成。另一类是非阻塞性的，即当进程启动一个系统调用后，不管这次 I/O 操作是否完成，都会立即返回，然后该进程继续往下运行。我们需要根据不同的应用背景，来选择不同类型的函数。

那么在一个实际的操作系统当中，API 函数是否具有上述特点呢？以下是 Windows 操作系统中的一个 API 函数。

```
HANDLE CreateFile(
  LPCTSTR lpFileName,                          //文件名
  DWORD dwDesiredAccess,                       //访问模式
  DWORD dwShareMode,                           //共享模式
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,  //安全属性
  DWORD dwCreationDisposition,                 //创建方式
  DWORD dwFlagsAndAttributes,                  //文件属性
  HANDLE hTemplateFile                         //模板文件的句柄
);
```

CreateFile()函数的功能是创建或打开以下的某种对象：控制台、通信资源(如串口)、目录、磁盘设备(分区)、文件(硬盘和光盘等)。

从 CreateFile()函数的功能和参数来看，这就是典型的设备独立性。也就是说，对于不同类型的设备，无论是硬盘、光盘、串口还是控制台，系统把对它们的访问都抽象为一种统一的形式，即文件，然后用相同的一组 API 函数来访问。CreateFile()函数是创建文件，另外还有读文件、写文件等相关的函数。因此，对于不同类型的设备，它们的访问方法是完全相同的，调用的是相同的一组函数。至于这些设备之间的差别，是由操作系统内部来处理的，程序员根本不需要知道。

另外，该函数的第一个参数 lpFileName，即文件名，这就是刚才所说的统一命名。也就是说，不管是什么类型的设备，都用统一的一种命名方式。例如，“A:\\1.txt”可能是软盘上的一个文件。“C:\\2.txt”可能是硬盘上的一个文件。“F:\\3.txt”可能是光盘上的一个文件。“COM1”是串口 1，“\\.\A:”是软盘设备，“\\.\C:”是硬盘分区，“CON”是控制台。总之，无论是访问什么类型的对象，使用的命名方式都是统一的路径名的方式。这样，程序

员使用起来就很方便。

下面再来看一下阻塞与非阻塞的I/O操作。所谓阻塞I/O,就是指一个进程在调用一个I/O操作的API函数后,该进程会被阻塞起来,直到这次I/O操作完成,然后被唤醒。这种方式易于使用,也易于理解,前面讨论的都是这种方式。例如,在一个典型的C语言程序中,经常使用scanf()函数,让用户从键盘输入一个数据。这个scanf()函数就是一个阻塞的I/O函数,当这个函数被执行、I/O操作被启动后,就会停在那里等待,而当前进程就会被阻塞起来,从而把CPU腾出来给别的进程。等到I/O操作完成后,该进程再继续往下运行。

但是在有些时候,可能需要非阻塞的I/O。也就是说,在调用一个I/O操作的API函数后,无论这个I/O操作是否完成,该函数都会立即返回。在这种方式下,程序的执行具有异步性。当I/O操作正在进行时,该进程可以继续执行,继续去做别的事情。然后当I/O操作完成时,I/O子系统会给进程发信号。这种方式的好处是调用者具有主动权,它能决定是否继续等下去。缺点是不太好理解,对程序员提出了更高的要求。因为这相当于是有两件事情在同时进行,所以一般都涉及多线程编程。在这种情形下,如何来协调各个并发线程之间的关系,有时候比较困难。

举个例子,假设在开始时,进程A在CPU上运行。后来A执行了一个I/O操作,如果是阻塞的I/O操作,那么进程A将会被阻塞,系统就会调度另一个进程B去运行。等这个I/O操作完成后,再把进程A唤醒。如果是非阻塞的I/O操作,那么进程A在启动I/O操作后,会立即返回,然后继续往下执行,此时不会发生进程的切换。

阻塞I/O示例代码如下。

```
HANDLE hCom;
hCom = CreateFile("COM1", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
//EV_RXCHAR: 一个字符已收到,并放在输入缓冲区中
SetCommMask(hCom, EV_RXCHAR);
WaitCommEvent(hCom, &dwEvtMask, NULL);
if(dwEvtMask & EV_RXCHAR)
{
    ReadFile(hCom, buf, NumBytesToRead, NumBytesRead, NULL);
}
```

上面的代码是一个阻塞I/O的例子,其功能是去访问计算机的串口设备,当串口接收到别的机器发来的数据时,就去把它读进来。这段程序是一个标准的C语言程序,只不过它用到了Windows系统的一些数据类型和API函数,因此在源文件的开头要把windows.h头文件包含进来。在这段程序中,首先定义了一个句柄,然后用CreateFile()函数打开了COM1串口。CreateFile()函数刚才已经讨论过,就是把串口这个外部设备当成一个普通的文件来打开。然后设置串口掩码为EV_RXCHAR,也就是说,当串口的输入缓冲区接收到一个字符以后,就会触发EV_RXCHAR这样一个事件。接下来调用WaitCommEvent()函数,等待事件的发生。然后判断一下,如果该事件是EV_RXCHAR,说明在输入缓冲区中有数据,然后就用ReadFile()函数把它读入到buf当中。显然,ReadFile()也是一个很好的设备独立性的函数,它更常用于读取一个硬盘文件的内容。另外,在这个例子当中,WaitCommEvent()和ReadFile()这两个函数的使用方法都是阻塞的I/O操作,即当它们被

调用时，会停在那里一直等待下去，等待事件的发生。当然，这种等待不是循环等待，浪费CPU时间，而是整个进程会被阻塞起来，然后把CPU让出来给其他进程使用。当等待的事件发生以后，该进程会被唤醒，从而可以继续往下执行。显然，这种阻塞性的I/O操作是易于理解的，也是我们通常访问I/O设备时所采用的方式。因为I/O设备的运行速度一般比较慢，比CPU要慢很多，所以为了提高CPU的使用效率，通常会采用这种方式。

非阻塞I/O示例代码如下。

```
HANDLE hCom;
OVERLAPPED o;
hCom = CreateFile("COM1", GENERIC_READ, 0, NULL, OPEN_EXISTING,
                    FILE_FLAG_OVERLAPPED, NULL);        //重叠 I/O
//EV_RXCHAR: 一个字符已收到,并放在输入缓冲区中
SetCommMask(hCom, EV_RXCHAR);
o.hEvent = CreateEvent(NULL, ...);
bR = WaitCommEvent(hCom, &dwEvtMask, &o);
if(!bR) ASSERT(GetLastError() == ERROR_IO_PENDING);
r = WaitForSingleObject(o.hEvent,INFINITE);
if(r == WAIT_OBJECT_0)                                  //有数据到达
{
    bR = ReadFile(hCom, buf, sizeof(buf), &nBytesRead, &o);
    if( !bR )
    {
        if(GetLastError() == ERROR_IO_PENDING)
        {
            r = WaitForSingleObject(o.hEvent, 2000);
            if(r == WAIT_OBJECT_0)
               GetOverlappedResult(hCom,&o,&nBytesRead,FALSE);
            else if(r == WAIT_TIMEOUT)...
    }
}
```

以上是非阻塞I/O的例子。基本功能与前面的例子相似，只是稍微修改了一下。在创建文件时，把文件的属性设置为FILE_FLAG_OVERLAPPED，即重叠的I/O。这样，在对这个文件进行访问时，就可以实现I/O操作与程序的执行并行进行。

在调用WaitCommEvent()函数时，用法与刚才不太一样。由于hCom是以重叠I/O的方式打开，因此最后一个参数传递的是一个OVERLAPPED结构体变量的起始地址。这种参数形式就表示这是一次非阻塞的I/O操作，即当该函数被执行时，无论是否有事件发生，都会立即返回。如果事件的确发生了，则返回值为真；如果事件没有发生，则返回值为假。此时可以调用GetLastError()函数，其返回值应该为ERROR_IO_PENDING，表示这个操作正在后台进行。将来一旦发生了该事件，即有字符到达串口，那么系统将会自动地把o.hEvent置位，表示有信号到来。因此，在代码中可以用WaitForSingleObject()来等待该事件，时间是无限期等待。当然，也可以设置一个特定的等待时间，即等待多长时间以后，就不再等待了，转而作为超时处理。

接下来是调用ReadFile()函数，从系统缓冲区中，把数据读入到用户自己的缓冲区中。

这个操作也是一个非阻塞的I/O操作,无论该操作是否成功,都会立即返回。然后在使用WaitForSingleObject()等待该操作完成时,设置了等待时间为2000ms。如果在这段时间内,没有读到数据,那么就进行超时处理。

有的读者可能不太理解为什么要使用这种非阻塞I/O的方式。事实上,对于一个应用程序来说,它往往需要同时去做很多事情,如屏幕显示、数据刷新、用户交互、串口通信等。因此,有时它就不能因为等待某个I/O操作而把自己阻塞起来,从而耽误了其他的工作。

再来看一个简单的阻塞和非阻塞I/O的例子,这里的背景是Linux操作系统。

(1) 阻塞地读取串口一个字符。

```
char buf;
fd = open("/dev/ttyS1", O_RDWR);
res = read(fd, &buf, 1);      //只有当串口有输入时才会返回
if(res == 1) printf("%c\n", buf);
```

(2) 非阻塞地读取串口一个字符。

```
char buf;
fd = open("/dev/ttyS1", O_RDWR|O_NONBLOCK);
while(read(fd, &buf, 1) != 1);      //串口无输入时也会返回
printf("%c\n", buf);
```

上面这段代码是阻塞地读取串口的一个字符,在调用read()函数时,只有当串口有输入时才会返回,否则当前进程会被阻塞。下面这段代码是非阻塞地读取串口一个字符,open()函数的参数是O_NONBLOCK,表示以非阻塞的方式来打开串口。然后在调用read()函数时,无论串口当前是否有输入都会立即返回,因此为了读到数据,需要用循环语句来不断尝试。

2. 操作系统与I/O设备的接口

我们知道,I/O设备种类繁多,类型各异,那么操作系统如何跟它们打交道呢?

如图5.14所示,为了提高I/O软件的可重用性和可扩展性,在操作系统与设备驱动之间,也有一个接口,每一种I/O设备的驱动程序都必须遵守该接口。由于设备驱动程序是由硬件厂商制作和提供的,那么从硬件厂商的角度来说,他们希望操作系统提供一个什么样的接口呢?

由于I/O设备种类繁多,因此,操作系统不太可能为每一种I/O设备都单独制定一个接口。一般来说,为了实现设备独立性,操作系统会把各种类型的设备划分为三大类:块设备、字符设备和网络设备,并为每一类设备定义了一个标准接口,而大多数设备驱动程序都支持其中之一。例如,键盘属于字符设备,硬盘属于块设备。

如图5.15所示,所有的设备被归结为三类,然后操作系统为每一类设备定义了一组接口函数。这些接口函数都是一些抽象的函数,它们并不会去做一些具体的操作,如命令设备控制器去执行I/O操作。这些函数都是抽象的,它们仅仅是作为接口函数来使用,被上层的操作系统软件所调用。这样的好处是能够把硬件设备的细节封装在设备驱动程序里面,而对于上层的系统内核中的I/O软件来说,它所面对的就是这个抽象的接口。换言之,它

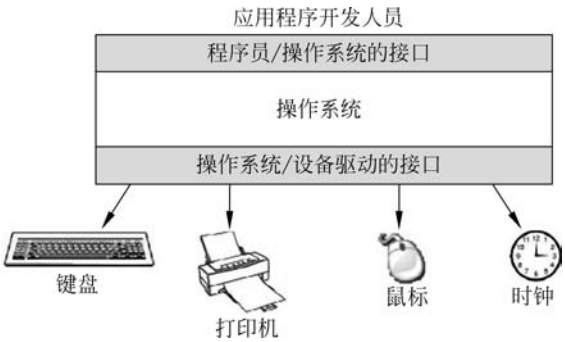

图 5.14　操作系统与设备驱动的接口

只会去调用这个接口中的函数，而不会直接跟底层的驱动程序打交道。由于这个接口是标准的、抽象的、固定不变的，因此，即使底层的硬件设备发生了变化，那么只需要去更新相应的设备驱动程序即可，而不会影响到上层软件对它的使用。事实上，上层软件不用做任何修改就可以继续使用，这样就有利于实现设备的独立性。

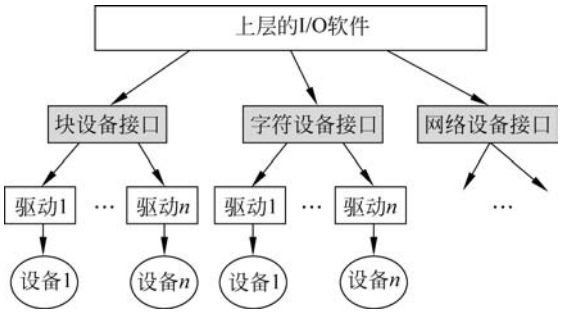

图 5.15　三种类型的接口函数

那么在这些接口中都包含哪一些函数呢？首先，无论是哪一类设备，无论是块设备还是字符设备，它们都需要一些共同的接口函数，例如：

- open(deviceNumber)：启动设备，初始化并分配资源，如缓冲区。
- close(deviceNumber)：关闭设备，释放资源。

当然，对于不同的设备来说，这两个函数的具体实现肯定是不一样的。但是从接口的角度来看，这些函数可以做成标准，包括函数名、函数的参数个数、参数的数据类型等，这些都可以是固定不变的。

对于字符设备，如键盘、鼠标和串口等，主要的接口函数包括：

- read(deviceNumber,buffer,size)：从一字节流设备中读入 size 字节写入到 buffer 缓冲区中。
- write(deviceNumber,buffer,size)：从 buffer 缓冲区中取出 size 字节，写入到一字节流设备中。

所有的字符设备都会提供这两个函数，但它们的具体实现是各不相同的。

对于块设备，如硬盘、U 盘和光盘等，主要的接口函数包括：

- read(deviceNumber,deviceAddr,buffer):从设备地址 deviceAddr 处读入一个数据块到 buffer 缓冲区。
- write(deviceNumber,deviceAddr,buffer):把 buffer 中的数据块写入到设备地址 deviceAddr。
- seek(deviceNumber,deviceAddress):把设备的访问指针定位到正确的位置。

3. 接口映射

刚才讨论了两个接口,一个是应用程序与操作系统的接口,另一个是操作系统与 I/O 设备的接口。显然,对于操作系统来说,它的一个基本任务就是实现这两个接口的映射,即把用户提交的接口函数调用转换为相应的设备驱动程序的接口函数调用。

例如,假设用户编写了如下 C 语言代码:

```
fp = fopen("C:\\1.txt","r");
fread(buffer,sizeof(char),100,fp);
```

这段代码的功能是去访问文件"C:\\1.txt",从该文件中读入 100B 到内存 buffer 数组中。那么对于操作系统来说,首先就要进行分析,这个文件到底是存放在什么地方。具体来说,用户在访问文件时,给出的参数往往是带有路径名的文件名,如"C:\\1.txt""F:\\2.txt"和"G:\\3.txt"等。在这种情形下,操作系统就要进行翻译,查明 1.txt 文件是位于硬盘,2.txt 文件是位于光盘驱动器,3.txt 文件是位于 U 盘,然后分别调用相应的设备驱动程序,完成文件的访问操作。这些都是操作系统内部要完成的工作,而对于程序员来说,他只要使用这种简单的文件名的形式来访问即可,而不必关心这个文件到底是存放在哪一种存储设备上,也不需要直接跟设备驱动程序打交道。

4. 设备无关的数据块大小

内核 I/O 子系统的另一个功能是提供与设备无关的数据块大小。我们知道,磁盘的访问是以扇区为单位,但是不同的磁盘可能会有不同的扇区大小,有的是 512B,有的是 1KB。因此,在系统内核的 I/O 软件模块,为了实现设备独立性,可以向它的上层掩盖这一事实,并提供统一的数据块大小。例如,它可以规定一个数据块的大小是 4KB,这样,在操作系统的内部,数据块的大小都是 4KB。然后在访问实际的 I/O 设备时,再根据它的扇区大小来确定每个数据块到底包含多少个扇区。例如,如果物理扇区的大小是 512B,相当于是将 8 个物理扇区合并为一个数据块。如果物理扇区的大小是 1KB,则相当于是将 4 个物理扇区合并为一个数据块。这样,对于上层的软件来说,它们所面对的就是一些抽象的设备,这些设备都使用相同大小的数据块,这样就把数据块的大小统一起来了。事实上,无论是访问接口的统一、命名规则的统一,还是数据块大小的统一,其目的都是为了实现设备独立性这个最终的目标。

5. 缓冲技术

内核 I/O 子系统的另一个功能是缓冲技术。我们知道,在 CPU 和内存之间存在缓冲,这个缓冲是位于 CPU 内部的高速缓存 Cache,即为了减少对内存的访问次数,提高内存的访问速度,可以把常用的一些数据保存在 Cache 中。而在 CPU 和磁盘设备之间也有缓冲,这个缓冲是位于内存当中,即为了减少对磁盘的访问次数,提高磁盘的访问速度,可以把常用的一些数据块保存在内存中,如图 5.16 所示。

缓冲技术之所以能够起作用，根本原因在于程序的局部性原理。它的基本思想是：在实现数据的输入/输出操作时，为了缓解 CPU 与外部设备之间速度不匹配的矛盾，提高资源的利用率，可以在内存当中开辟一个空间，作为缓冲区。这样，当我们从磁盘读数据时，先到缓冲区当中去查找。如果能找到，就不用再去访问磁盘，而是直接从缓冲区中读取。在写磁盘时，也是先写入到缓冲区当中，以后再写到磁盘上。例如，假设要把一个非常大的数据文件从 C 盘复制到 D 盘，那么在执行完这次复制操作之后，该文件的很多数据块就可能会临时存放在内存的缓冲区当中。如果过了一会儿，需要再次访问该文件，例如，把它再复制一份到 E 盘，这时就会发现，这次复制的速度会比刚才要快，因为有一些数据块已经在内存当中了。

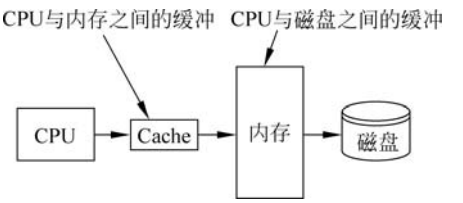

图 5.16　缓冲技术

在具体实现缓冲区技术时，可采用以下两种方案。

- 单缓冲：只有一个缓冲区，由 CPU 和外设轮流使用，在一方处理完之后就等待对方去处理。
- 双缓冲：有两个缓冲区，CPU 和外设都可以连续地处理而不需要等待对方。这种方式要求 CPU 和外设的速度比较接近，否则还是会有等待的现象。

缓冲技术是一种非常实用的技术，因为对于 I/O 设备的访问，也经常会满足程序的局部性原理。例如，在访问磁盘时，在一段时间内，可能会集中地访问其中的若干个数据块，因此设置缓冲区可以有效地减少对 I/O 设备的访问次数，从而提高系统的性能。换言之，缓冲技术的实质是以空间换时间。

5.3.3　用户空间的 I/O 软件

在 I/O 软件的最顶层是用户空间的 I/O 软件。前面介绍的各种 I/O 软件，都位于系统内核当中，是操作系统的一部分。但也有另外一小部分的 I/O 软件，它们并不在系统内核当中，主要可以分为以下两种。

- 库函数：与用户程序进行链接的库函数。例如，在 C 语言中与 I/O 有关的各种库函数，如 open()、write()、read()等。不过，这些库函数实质上只是一个空的外壳，在具体实现时，它们会把传给它们的参数再往下传递给相应的系统调用函数，并由后者来完成实际的 I/O 操作。
- SPOOLing 技术：这是一种完全运行在用户空间中的程序，它是在多道系统当中，一种处理独占设备的方法。

SPOOLing(Simultaneous Peripheral Operation On Line)一般称为假脱机技术，或者虚拟设备技术。它可以把一个独占设备转变为具有共享特征的虚拟设备，从而提高设备的利用率。它的基本思想是：在多道系统当中，对于一个独占的设备，专门利用一道程序，即 SPOOLing 程序，来完成对这个设备的输入/输出操作。

具体来说，如图 5.17 所示，一方面，SPOOLing 程序负责与这个独占的 I/O 设备进行数据交换，这可以称为“实际的 I/O”。如果这是一个输入设备，那么 SPOOLing 程序预先从该设备输入数据并加以缓冲，然后在需要时再交给应用程序；如果这是一个输出设备，那么

SPOOLing 程序会接受应用程序的输出数据并加以缓冲,然后在适当的时候再输出到该设备。另一方面,应用程序在进行 I/O 操作时,只是与 SPOOLing 程序交换数据,这可以称为"虚拟的 I/O"。这时,它实际上是从 SPOOLing 程序的缓冲区当中读出数据或者是把数据送入到这个缓冲区,而不是直接地与实际的设备进行 I/O 操作。

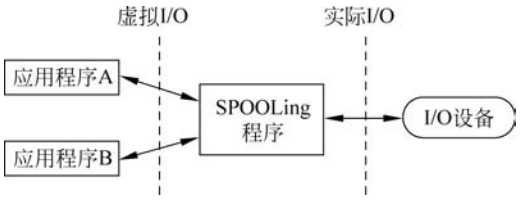

图 5.17　SPOOLing 技术

SPOOLing 技术的优点有以下两个。

- 高速的虚拟 I/O 操作:应用程序的虚拟 I/O 比实际的 I/O 速度要快,因为它只是在两个进程之前的一种通信,把数据从一个进程交给另一个进程。这种交换是在内存中进行的,而不是真正地让机械的物理设备去运作。这样,就能缩短应用程序的执行时间。
- 实现对独占设备的共享:由 SPOOLing 程序提供虚拟设备,然后各个用户进程就可以对这个独占设备依次地共享使用。

举个例子,打印机就是一种独占设备,在任何时候只能允许一个用户进程使用。在现代操作系统当中,对于打印机设备,普遍采用了 SPOOLing 技术。具体来说,首先创建一个 SPOOLing 进程,或称后台打印程序,以及一个 SPOOLing 目录。当一个进程需要打印一个文件时,首先会生成将要打印的文件,并把它放入 SPOOLing 目录当中,然后由这个后台打印进程来负责真正的打印操作。例如,在计算机上有各种各样的应用软件,如文字编辑软件、网页浏览器、图像编辑器等。在使用这些软件时,如果要打印一个文件,那么直接单击"打印"按钮即可。没过一会儿,软件就会提示"打印"已完成。此时,就可以把这个软件关闭。如果文件是存放在 U 盘上,甚至还可以把 U 盘拔掉,这些操作都没有问题。但实际上我们一看打印机,根本还没有开始打印,这是怎么回事呢?原来,对于应用软件来说,它所谓的打印,只是生成相应的打印文档,然后交给后台打印进程。然后对于它来说,整个任务就算完成了。而且由于这是两个进程之间的通信,数据是在内存中传送,因此速度会很快。剩下的就是后台打印进程与打印机之间的事情了,而且打印机是一个外部设备,它的运行速度比较慢,所以要过好一会儿,才会听到打印机开始工作。

5.3.4　I/O 实现举例

前面介绍了 I/O 软件的各个层次,即中断处理程序、设备驱动程序、设备独立的操作系统软件以及用户空间的 I/O 软件。每一层都是用来实现特定的功能,相邻的层次之间有着良好的调用接口。下面通过两个例子来把这些内容综合在一起,也就是说,为了完成一次完整的 I/O 功能,不同的角色应该如何分工,程序员需要做什么,操作系统需要做什么,设备驱动程序和中断处理程序需要做什么,这样就把整个过程给串起来了。

1. I/O 实现案例之一

假设在一个实验室当中,为了项目的需要,新开发了一个简单的字符输入设备。该设备

类似于键盘，只不过比较简陋，没有标准的键盘那么大。由于该设备是自己制作的个性化的I/O设备，没有现成的驱动程序可以使用，因此，需要自己为这个设备编写相应的驱动程序，使之能正常使用起来。换言之，如果是从市场上购买的标准的I/O设备，如键盘、鼠标等，这些设备一般由专门的硬件厂商制作，并配备有相应的设备驱动程序，所以直接就可以使用。但如果是自己设计并实现的设备，那么不仅要做硬件，而且要做软件，要自己编写相应的设备驱动程序。

那么如何来完成这个任务呢？首先要弄明白的是：为了让这个设备正常运转起来，到底需要做哪些事情？具体来说，要编写哪些代码，这些代码存放在什么地方，如何把它们提交给操作系统，使得操作系统能够管理这个定制的设备，从而使用户可以像普通的其他设备一样来使用它。

假设我们已经编程实现了该设备的驱动程序，并且已经把它提交给了操作系统，在这种情形下，它是如何来使用的呢？由于这是一个类似于键盘的输入设备，因此，用户在编程使用这个设备的时候，肯定是通过相应的函数调用来实现的。例如，在C语言中，如果要从键盘输入一个数据，那么一般会调用scanf()库函数。如图5.18所示，这就是这个设备的用户使用方式。对于应用程序开发人员来说，他编写了一个C语言应用程序，在main()函数当中，通过调用scanf()函数，从键盘读入一个整数，保存在变量x当中。所以这就是用户在使用该设备时所需要做的事情。

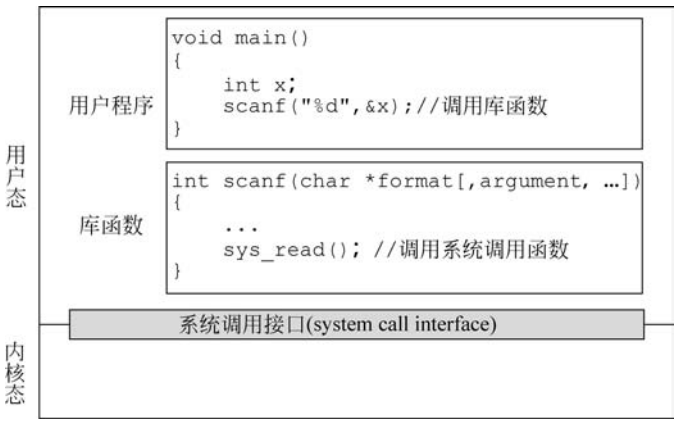

图5.18　用户态下的工作

我们知道，scanf()函数是C语言的库函数，即前面所说的用户空间的I/O软件，它在具体实现时，通常的做法是先进行一些参数验证和转换工作，然后就启动一次系统调用，让操作系统来完成此次I/O工作。我们不妨把这个系统调用称为sys_read()。需要指出的是，当进程在执行main()和scanf()函数时，都是运行在用户态，而一旦启动了系统调用，就会发生状态的切换，CPU就会从用户态切换为内核态，然后去执行操作系统内核的代码，如图5.19所示。

在内核态下，在sys_read()系统调用中，它会去做其他一些事情，这里不再详述，然后就去调用字符类的设备接口函数read()。如前所述，对于字符设备、块设备和网络设备，每一类设备都会提供一组标准的、统一的对外接口函数。然后在内核I/O子系统中，只会去调用这组接口函数，而不会直接与设备驱动程序打交道。接下来，在read()函数的具体实现

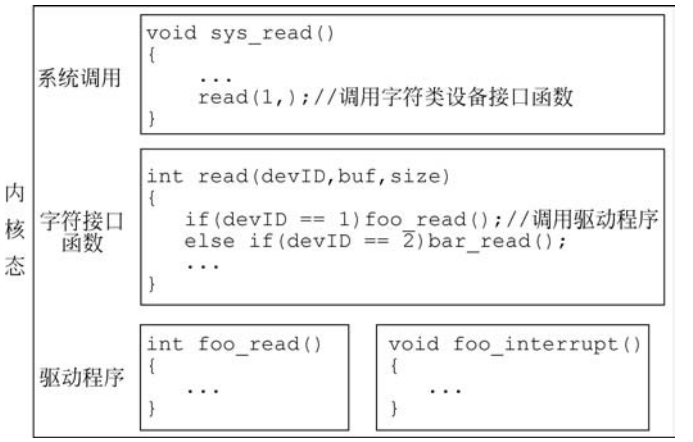

图 5.19　内核态下的工作

中,由于它是面向所有字符类设备的,包括键盘、鼠标和串口等,而每一种字符设备的驱动程序是不一样的,因此需要进行一个跳转。例如,如果该设备的设备号是 1,那么就跳转到 1 号字符设备的驱动程序 foo_read(); 如果该设备的设备号是 2,那么就跳转到 2 号字符设备的驱动程序 bar_read(),以此类推。假设这个新设备的设备号为 1,因此,如果选定了该设备,那么就会跳转到它的驱动程序 foo_read()去执行。另外,一般来说,设备驱动程序主要由两个函数组成:一个是 foo_read(),即该设备对 read()接口函数的具体实现;另一个是 foo_interrupt(),即中断处理函数。总之,到此为止,我们就能够明白,当在系统中新增加一个设备时,真正需要做的事情就是编程实现它的驱动程序,即 foo_read()和 foo_interrupt()这两个函数。而对于其他的那些内核函数,包括库函数,都是别人已经做好的接口,可以直接使用。

那么对于 foo_read()和 foo_interrupt()函数,它们又是如何实现的呢? 以下给出了一个参考的实现样例,该样例来源于一个真实的设备驱动程序,并进行了适当的裁剪。它的运行环境是一个类 Linux 的嵌入式操作系统。

(1) foo_read()函数。

```
size_t foo_read(struct file * filp, char * buf, size_t count, loff_t * ppos)
{
    foo_dev_t * foo_dev = filp->private_data;

    if(down_interruptible(&foo_dev->sem)              //互斥
        return -ERESTARTSYS;
    foo_dev->intr = 0;                                //同步
    outb(DEV_FOO_READ, DEV_FOO_CONTROL_PORT);
    wait_event_interruptible(foo_dev->wait,
                             (foo_dev->intr == 1)); //被阻塞
    if (put_user(foo_dev->data, buf)) return -EFAULT;
    up(&foo_dev->sem);
    return 1;
}
```

(2) foo_interrupt()函数。

```
void foo_interrupt(int irq,void * dev_id, struct pt_regs * regs)
{
    foo->data = inb(DEV_FOO_DATA_PORT);
    foo->intr = 1;
    wake_up_interruptible(&foo->wait);
}
```

在上述代码中,foo_dev-> sem 是一个信号量,用于实现进程间的互斥,保证在任何时候只有一个进程去使用这个硬件设备。down_interruptible()函数类似于前面提到的 P 原语,而 up()函数则类似于 V 原语。foo_dev-> intr 是一个标志位,用来与中断处理程序实现进程间同步,outb 语句用来向 I/O 设备控制器中写入数据,而 inb 语句用来从设备控制器中读出数据。另外,wait_event_interruptible()函数用于把当前进程阻塞起来,而 wake_up_interruptible()函数则负责唤醒进程。

总的来说,上述代码的执行过程是:首先是用户进程 A 执行,它执行了 scanf()函数,然后进入系统调用,然后到标准接口 read()函数,然后再到驱动程序 foo_read()函数,这些都是普通的函数调用,都是在进程 A 的资源平台上运行。接下来,在 foo_read()函数中,在使用 outb 指令启动了这次 I/O 操作后,进程 A 就会被阻塞起来。然后进行进程切换,调度另一个进程 B 去运行。当 B 在运行时,如果 I/O 操作完成,就会发生一次中断,把进程 B 打断,并跳转到中断处理程序 foo_interrupt()去执行,然后在这里再去唤醒进程 A。

2. I/O 实现案例之二

上述案例只适用于需要互斥访问的设备,如键盘、鼠标等字符设备,在任何时候只能有一个进程去读取它的数据。但是对于块设备,数据是可以共享的。例如,假设进程 A 要访问磁盘的第 i 个数据块,而进程 B 也要访问磁盘的第 i 个数据块。如果采用上述方案,那么就必须进行两次独立的 I/O 操作,从而造成浪费。因此,有必要对这种方案再进行进一步的优化。

案例二主要用在块设备当中。它的基本思路是:在数据结构上,为每一个设备设置一个请求队列。然后把驱动程序中的函数分为两个层次:上层函数和底层函数。其中,上层函数负责管理请求队列。而底层函数则负责与硬件打交道,完成真正的 I/O。

在这种方式下,I/O 操作的过程如图 5.20 所示。假设有一个用户进程 A,需要执行一次 I/O 操作,如读取文件中的一个数据块,那么就去执行相应的库函数,而库函数会调用系统调用函数,从而进入内核态。在系统调用函数中,又会执行一些操作,然后就进入设备驱动程序,调用它内部的某个上层函数,如 make_request(),从而把这一次的 I/O 请求提交进去。而对于 make_request()函数,它会在请求队列中,新增加一个请求。在以上整个过程中,一直都是函数调用,没有发生进程切换。做完这些工作以后,对于用户进程 A 来说,它的任务就已经完成了,因此可以执行一个等待操作,把自己阻塞起来。然后系统就会调度另一个进程 B 去运行。由此可见,I/O 请求的提交与真正实现是分离的。对于每一个用户进程,当它需要 I/O 操作时,都可以通过刚才的函数调用路径来提交 I/O 请求,然后把自己阻塞起来。在做这件事情时,进程之间是可以并行执行的,不存在互斥的问题,因为此时并没

有真正去访问I/O设备。

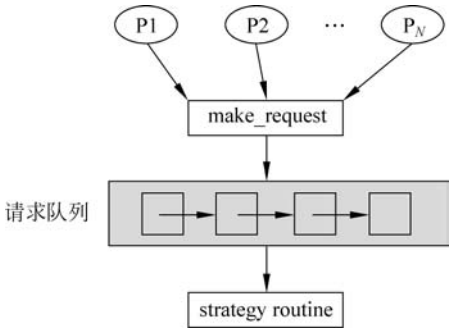

图5.20　块设备驱动程序中的I/O操作

那什么时候开始真正启动I/O操作呢？在进程被阻塞起来之前，可能要通过某种方式通知系统内核。例如，通知内核的某个后台进程，该进程会定期执行。然后它会去调用设备驱动程序中的下层函数，如strategy routine。该函数就会从请求队列中取出第一个请求，然后去完成它。具体做法就是去设置设备控制器，去读写它的端口地址，并有可能对DMA进行编程，让它来负责数据传输。做完这些事情后，该函数就结束了。

当这次I/O操作结束后，DMA控制器会向CPU发出一个中断，打断当前进程的运行。注意当前是进程B在运行，而进程A已经被阻塞。然后在中断处理程序中，会去唤醒进程A，告诉它I/O操作已完成。然后再检查一下请求队列，如果请求队列不为空，则再次执行strategy routine函数，去处理下一个请求。

采用这种方式，其优点是能对各个I/O请求进行优化。例如，假设进程A需要访问第3个扇区，进程B需要访问第4个扇区。在这种情形下，可以先执行A的请求然后再执行B的请求，这样磁头就不用移动，从而提高了访问的速度。

以下是UNIX系统SCSI磁盘驱动程序的一个例子，该设备驱动程序由如下若干个函数组成。

- sdstrategy：进行错误检查，如果设备不忙，发出一个请求。
- sdustart：将该请求放入到请求队列中，并发出启动信号。
- sdstart：为该请求申请所需的资源，如scsi总线或DMA资源。
- sdgo：往设备控制器中写入命令，设置中断向量，向设备控制器发出启动的信号。
- sdintr：中断处理程序，结束本次请求，唤醒被阻塞的进程。如果请求队列不为空，则执行下一个请求。

在这些函数中，前两个函数就是刚才所说的上层函数，负责对请求队列进行管理。而接下来的两个函数就是底层函数，负责去直接操作设备控制器，启动I/O操作。最后一个函数是中断处理程序，是在I/O操作完成、中断发生的时候被调用的。

扫码观看

5.4　磁　　盘

在本章的最后，来看两种具体的输入/输出设备，即磁盘和固态硬盘。本节主要讨论磁盘，它是一种非常普遍的外部存储设备，本节将学习磁盘的硬件、磁盘格式化、磁盘调度算法以及出错处理等方面的内容。

5.4.1　磁盘的硬件

磁盘包括软盘和硬盘。各位读者在使用计算机的时候，不知是否注意到，我们的外部存储设备，盘符一般是从C开始编号，如C盘、D盘和E盘等，但是却没有A盘和B盘，这是为什么呢？这其实是兼容性的原因，在历史上，A盘和B盘是给软盘驱动器预留的，其中A

盘是容量为360KB的5英寸盘，B盘是容量为1.44MB的3英寸盘。后来随着容量更大、性能更可靠的U盘的出现，软盘已经消失在历史的长河中了，因此这里不再赘述。如果读者在某部电影当中看到其中的角色仍然在使用软盘来作为存储介质（如碟中谍1），那就说明这部电影已经有一些年头了。事实上，“碟中谍”这部电影名中的“碟”字，就是指盘碟。

我们讨论的磁盘主要是硬盘，或者说机械硬盘。如图5.21所示，硬盘一般由一个或多个金属盘片组成。这些盘片组合被固定在一根旋转轴上，由同一个马达来驱动。当旋转轴开始旋转时，所有的盘片都会跟着旋转。每个盘片都有上、下两个盘面，在盘面上涂有磁性材料，信息就记录在这些盘面上。另外，在每一个盘面的上方，都有一个磁头，它被固定在一个磁头臂上，而这个磁头臂又固定在一个传动装置上。这个传动装置是可以移动的，它的移动方向就是沿着盘片半径的方向，左移或右移。当这个传动装置在移动时，所有的磁头臂都会跟着移动，从而带动它上面的磁头也跟着移动。

当这个传动装置固定在某个位置时，与之相连的磁头臂的位置就是固定的，因而磁头的位置也是固定的。在这种情形下，如果旋转轴匀速旋转一圈，那么对于每一个磁头来说，它所能访问的盘片表面的区域，是一个圆环的形状。我们把这样的一个圆环区域称为一个磁道（Track）。显然，不同的磁道半径是不同的，靠近圆心的磁道半径小，远离圆心的磁道半径大。另外，由于磁盘有多个盘面，因此，在所有盘面上，半径相同的所有磁道就组成了一个柱面（Cylinder）。

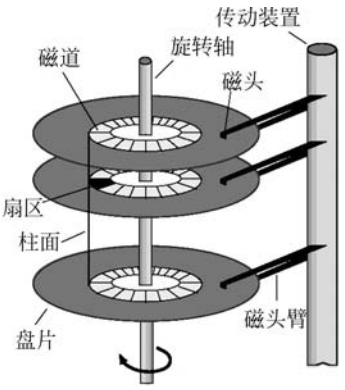

图5.21　硬盘

另外，由于磁头的大小有限，它每次只能访问一小块区域。因此，对于每一个磁道来说，又把它平均地划分为一个个小格子，每个小格子就是一个扇区（Sector）。一个扇区的大小一般是512B，也有的是4KB。在访问磁盘时，一般都以扇区为单位，具体过程如下。

当需要访问某个扇区时，首先要告诉磁盘驱动器，该扇区的地址是什么。具体来说，它是位于哪一个柱面，在这个柱面上又是位于哪一个磁道，在这个磁道上又是位于第几个扇区。只有在知道了这些地址信息以后，磁盘驱动器才能精确地定位这个扇区。也就是说，首先要移动传动装置，通过它来移动磁头，向左或向右移动，从而找到正确的柱面。然后，根据磁道号，可以选中相应的磁头。接下来，由于想要访问的扇区不一定正好就在该磁头的正下方，因此要让旋转轴转动起来，等到目标扇区正好路过磁头的正下方时，就可以对它进行读写操作了。

需要指出的是，磁盘的访问是以扇区为单位。即使只想读写一字节，也必须把它所在的整个扇区读入和写入，如图5.22所示。

磁盘的存储容量取决于它的扇区的个数以及扇区的大小。早期的5英寸软盘，只有一张盘片，该盘片有上、下两个面。然后每个盘面被划分为40个同心圆环，即40个磁道。每个磁道又被划分为9个扇区。这样一来，该软盘总共有40×2×9＝720个扇区，每个扇区的大小为512B，因此，该软盘的容量为360KB。而对于西部数据的WD3000HLFS硬盘，它大概有36 481个柱面，每个柱面有255个磁道，每个磁道平均约63个的扇区，因此，它总共有

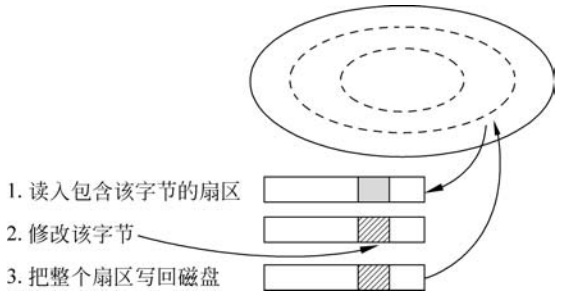

图 5.22　修改磁盘上的一字节

36 481×255×63=586 067 265 个扇区,每个扇区的大小为 512B,因此,该硬盘的容量大约为 300GB。当然,现在的机械硬盘容量就更大了,如 4TB。

在讨论硬盘的性能时,一个重要的方面就是随机访问时间,即当给定一个随机的数据块地址后,硬盘驱动器需要多长时间来完成相应的读写操作。如前所述,这个时间主要由两部分组成,一是把磁头臂移动到目标柱面所需要的时间,这个时间也称为柱面定位时间;二是目标扇区旋转到磁头正下方所需要的时间,这个时间也称为旋转延迟时间。无论是柱面定位还是旋转延迟,它们都属于机械运动,速度比较慢,一般需要几毫秒的时间。以旋转延迟时间为例,这个时间显然取决于旋转轴的旋转速度,如果旋转速度越快,那么相应的延迟时间就越短。大部分的硬盘驱动器每秒能够旋转 60～250 圈,或者以 RPM(Rotations Per Minute,每分旋转多少圈)为单位,即 3600～15 000RPM。例如,假设某个硬盘驱动器的转速为 10 000RPM,那么它旋转一圈所需要的时间就是 6ms。这意味着任何一个扇区的旋转延迟时间都不会超过 6ms。显然,在机械硬盘当中,由于柱面定位和旋转延迟时间较长,这就限制了硬盘的随机访问速度。读者如果感兴趣,可以大致估算一下这个访问速度的数量级别。因此,为了提高这个访问速度,在机械硬盘中往往会设置高速缓存,其原理与内存的高速缓存是类似的。

硬盘性能的另外一个方面是数据流在硬盘驱动器与内存之间的传输速度,这取决于硬盘驱动器是如何连接在计算机当中。它可以直接连接在系统总线上,也可以通过专门的 I/O 总线连上去。典型的总线包括 ATA(Advanced Technology Attachment,高级技术附件)、SATA(Serial ATA,串行 ATA)、USB(Universal Serial Bus,通用串行总线)等,其中最常用的连接方式是 SATA。

作为一个例子,图 5.23 给出了西部数据公司的一款硬盘 WD40EZRZ 的参数列表,其容量为 4TB,对外接口的类型为 SATA3.0,接口速率为 6Gb/s。高速缓存的容量为 64MB。转速为 5400RPM,这意味着旋转一圈所需要的时间为 11ms 左右。

产品型号	WD40EZRZ
产品容量	4TB
外形规格	3.5英寸
产品接口	SATA 6Gb/s
高速缓存	64MB
转速等级	5400RPM
尺寸/mm	147×101.6×26.1
重量/kg	0.68

图 5.23　WD40EZRZ 硬盘参数

虽然硬盘的最小访问单位是扇区,但实际上,我们在访问硬盘时,通常是以数据块为单位,每一个数据块往往是由连续的若干个扇区组成。这样一来,对于上层用户来说,整个硬盘可以看成是一个巨大的一维数组,其中每一个数组元素都是一个数据块。当然,也可以把每一个数组元素看成是一个扇区,然后相邻的几个扇区组成一个数据块。因此,从

用户的视角，这样来看待一块硬盘就可以了。每次访问硬盘时，只要给出一个块号即可，相当于是相应数组元素的下标。但是对于硬盘驱动器来说，它在真正工作时，是需要详细的内部地址的，即该数据块是由哪几个扇区组成，每个扇区的物理地址是什么。所谓扇区的物理地址，由三个部分组成：柱面号、该柱面上的磁道号和该磁道上的扇区号。有了这些地址信息，硬盘驱动器就能准确地定位到目标扇区，并且对它进行读写操作。换言之，这里需要进行一个地址映射，把一维地址即数据块号(或扇区编号)映射为三维地址，即柱面号、磁道号和扇区号。

例如，整个硬盘的扇区 0 可能是最外侧柱面上的第 1 个磁道上的第 1 个扇区，扇区 1 是相同柱面和相同磁道上的第 2 个扇区，以此类推。同一个磁道上的所有扇区都处理完以后，接下来的顺序是同一个柱面上的下一个磁道上的第 1 个扇区，然后又顺着该磁道上的扇区顺序依次编号。当同一个柱面上的所有磁道都处理完以后，再换到下一个柱面的第 1 个磁道的第 1 个扇区。或者可以这么来理解，把这个三维地址看成是一个三位数，然后把柱面号看成是该三位数中的百位数，磁道号看成是该三位数中的十位数，然后扇区号看成是该三位数中的个位数。对于个位数和十位数，每次满了十以后就进一位。

当然，以上的地址映射方法只是一种理论上的方法，在具体实现上还需要做一些修改和调整。这有两个方面的原因，首先，在硬盘上，有一些扇区可能是坏的，无法访问，所以在地址映射时需要用其他处的空闲扇区来替代它们。其次，每一个磁道上的扇区的个数可能是不一样的。这是因为对于不同半径的磁道来说，它们的圆环区域的面积是不一样的。越靠近圆心、半径越小的磁道，其圆环区域的面积越小；而越远离圆心、半径越大的磁道，其圆环区域的面积越大。因此，如果不去考虑这个问题，不去注意各个磁道在面积上的大小差异，而是简单地把所有磁道都划分为相同的扇区个数，就会带来一些不方便的地方。不过令人高兴的是，这种地址映射是由硬盘生产厂商自己来负责的，是在硬盘驱动器内部完成的，对于硬盘的使用者来说，不必关心其内部的实现细节。

5.4.2 磁盘格式化

下面简单介绍一下磁盘的格式化问题，主要讨论硬盘的格式化。假设我们手里拿到了一个全新的硬盘，并且已经把它安装到了计算机上，那么如何才能让它正常地工作起来，为我们存储数据呢？首先要做的事情就是对它进行格式化，没有格式化的磁盘是不能访问的。硬盘的格式化可以分为三个步骤：低级格式化、分区和高级格式化。

如前所述，磁盘的访问是以扇区为单位的，而扇区又在一个个磁道上。一个硬盘在刚刚出厂时，它是一个真正意义上的空盘，里面没有任何信息，就好像一张白纸。在这种情形下，我们根本就不知道哪里是磁道、哪里是扇区。因此，必须对它进行一种低级格式化，画出一个个扇区和磁道，并且在相邻的扇区之间用狭窄的间隙隔开。

每一个扇区一般由三部分的内容组成，即相位编码、数据区和纠错码。

- 相位编码：以某个特定的位组合模式开始，用来向硬件表明这是一个新扇区的开始。另外，它还包括柱面号、扇区号和扇区大小等类似的信息。
- 数据区：数据区的大小由低级格式化程序来确定，一般都设定为 512B。
- 纠错码：主要包含一些冗余信息，用来纠正在读取扇区时可能出现的错误。

在经过低级格式化以后，这个硬盘就不再是空白的了，而是画有各种各样的格式化信息。但是从另一个角度来说，它又是空白的，因为它里面还没有保存任何有用的用户数据。

换句话说,在进行低级格式化之前,磁盘可以说是一张白纸。在格式化以后,上面就画了很多个格子,即磁道和扇区,但每个格子里面都是空白的,还没有装数据。

在低级格式化以后,第二个步骤是分区,即用一个分区软件把整个硬盘划分为若干个逻辑分区,每一个逻辑分区都可以看成是一个独立的硬盘。具体来说,在物理上,硬盘只有一块,上面有很多个扇区。然后把相邻的一部分扇区作为分区1,另一部分扇区作为分区2,诸如此类。这样,在物理上这些分区都是位于同一个硬盘上,但是从逻辑上来说,可以把它们看成是各不相关的多个硬盘,可以分别使用。例如,假设一块硬盘有4TB,那么可以把它分为4个分区,即C盘、D盘、E盘和F盘。C盘一般用来安装操作系统和应用程序,而其余的分区则用来存放用户数据。

在大多数计算机上,一般都是用硬盘的第0个扇区来存放一些系统启动代码和一个分区表,在这个分区表当中,记录了每一个分区的起始扇区和大小。这有点像第4章中介绍的固定分区的存储管理方法,只不过这里管的不是内存,而是硬盘。

磁盘格式化的第三个步骤是高级格式化,即对每一个逻辑分区,分别进行一种高级的格式化操作。这其实就是我们平常在使用计算机时所说的格式化操作Format。当这个格式化操作完成以后,相应的逻辑分区上将会生成一个引导块、空闲存储管理的数据结构、根目录和一个空白的文件系统。而且对于不同的逻辑分区,可以使用不同的文件系统,如FAT、NTFS和Ext等。

需要指出的是,一旦对磁盘进行了格式化操作,那么磁盘上的原有数据就全部丢失了。当然,有网友说在格式化硬盘以后,计算机会变得轻了许多,这应该是谣言。

5.4.3 磁盘调度算法

如前所述,磁盘的访问是以扇区作为最小的寻址和存取单位的,在访问一个磁盘扇区时,所需的时间主要有以下三方面。

- 柱面定位时间:在寻找目标扇区时,首先要移动磁头,找到正确的柱面。从具体的实现来看,其实是磁盘的传动装置在移动,然后带动固定在它上面的磁头臂移动,而磁头臂的移动,又使得固定在它上面的磁头移动,最后就移动到了指定的柱面上。而这种移动是一种机械运动,它需要一定的时间,这段时间就称为柱面定位时间。
- 旋转延迟时间:当磁头移动到正确的柱面后,目标扇区可能并不在磁头的正下方,因此必须再等一会儿,等待该扇区旋转到磁头的下方。而这种盘片的旋转也是一种机械运动,它也需要一定的时间,这段时间就称为旋转延迟时间。显然,这段时间的长短与磁盘的转速有关,磁盘转得越快,则旋转延迟时间就越短。
- 数据传送时间:现在目标扇区已经在磁头的正下方,剩下的事情就是往这个扇区中写入数据,或者从这个扇区中读出数据,而这个操作也需要一定的时间。

那么如何尽可能减少磁盘的访问时间,提高磁盘的访问速度呢?一种思路是从硬件方面着手,通过工艺水平的提高来改进各种硬件设施。例如,让磁头的移动速度更快、让盘片的旋转速度更高、让数据的传送时间更短等。这种硬件的提升不属于本书的讨论范畴,我们主要考虑的是另外一种思路,即从软件方面着手,在硬件条件不变的情形下,通过软件的办法来尽可能提高磁盘的访问速度,这主要有两种方法。

先来看第一种方法,即通过合理地组织磁盘数据的存储位置来提高磁盘的访问速度。

例如，假设一个磁盘的转速为10 000rpm，每个磁道有300个扇区，每个扇区有512B，现在要读一个大小为150KB的文件，请问这需要多长时间？假设柱面定位的平均时间为6.9ms，旋转延迟的平均时间为旋转时间的一半，即3ms。每个扇区的传送时间为17μs。

情形1：假设该文件存放在同一个磁道的300个连续的扇区当中。在这种情形下，磁盘访问的总时间为：

$$6.9\text{ms}+3\text{ms}+6\text{ms}=15.9\text{ms}$$

具体来说，柱面定位时间需要6.9ms，旋转延迟时间3ms，这是因为需要从文件的第一个扇区开始读起，而该扇区不一定正好就在磁头的下方。接下来旋转一周，就把这个磁道上的每个扇区的内容都读出来了，这需要6ms，因此总的时间是15.9ms。

有的读者可能会有疑问，为什么不考虑扇区的传输时间？实际上，在盘片旋转一周的过程中，数据就已经被读走了，也就是说，盘片的旋转与数据的读取是同步进行的。考虑一个生活中的例子，当我们乘飞机到达目的地以后，要在机场取行李。取行李的地方一般是一个大圆盘，圆盘上有传送带，所有乘客的行李被依次放在传送带上。乘客可以站在圆盘的任意一个位置，当看到自己的行李经过时，就伸手去拿下来。显然，在理想状态下，这个大圆盘只要旋转一周，那么所有的行李都会被取走，而且所需要的总时间就是该圆盘旋转一周的时间，不必考虑乘客伸手去取行李的时间，因为这两者是重叠在一起的。

情形2：假设该文件的300个扇区随机分布在整个磁盘上。在这种情形下，磁盘访问的总时间为：

$$(6.9\text{ms}+3\text{ms}+0.017\text{ms})\times 300=2975.1\text{ms}$$

也就是说，每个扇区的访问，都要经历完整的三个步骤，即柱面定位、旋转延迟和数据传送，因此总的时间是2975.1ms。

由此可见，对于相同的一个文件，如果在磁盘上的存放位置不同，那么所需要的访问时间也是完全不同的。在上述例子中，情形2所需要的访问时间是情形1的187倍。当然，这两种情形其实都是最极端的情形，一种最好，一种最差。而对于实际的磁盘访问来说，一般介于这两者之间。总之，合理地组织磁盘数据的存储位置，是非常重要的，它直接影响到磁盘的访问速度。一般来说，当计算机使用了一段时间以后，随着不断地增加文件、删除文件，就会使各个文件在磁盘上的存储位置变得非常凌乱，而且不连续，因此需要定期使用磁盘整理工具去整理一下，使得各个文件能够尽可能地连续存放，从而提高磁盘的访问速度。例如，在Windows系统中，有一个工具软件defrag，就是用于磁盘的碎片整理。

提高磁盘访问速度的第二种方法是磁盘调度。通过上面的例子可以看出，对于大多数磁盘来说，柱面定位时间（磁头移动时间）在总的访问时间当中占有主要部分。因此，对于整个系统来说，如果能减少平均柱面定位时间，那么将有效地改善系统的输入/输出性能。

磁盘调度的问题描述是：在操作系统的I/O子系统当中，来自不同进程的磁盘访问请求，会构成一个随机分布的请求队列，如图5.24所示。

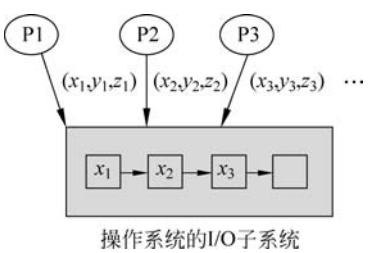

图5.24　磁盘调度

对于每一个进程，在访问磁盘时都会给出一个三维的地址(x,y,z)，即柱面号、磁道号和扇区号。这里只考虑柱面号，因此把所有地址中的柱面号单独抽取出来，构成一个序列$x_1,x_2,x_3,\cdots$。磁盘调度的基本思路就是通

过调整这些I/O请求的执行顺序,来减少整个请求序列所需要的平均柱面定位时间。而磁盘调度程序所采用的算法就是磁盘调度算法。

下面介绍几种常用的磁盘调度算法,包括先来先服务算法、最短定位时间优先算法和电梯算法。

1. 先来先服务算法

先来先服务(First-Come First-Served,FCFS)算法是最简单的一种算法,它的基本思路就是按照访问请求到达的先后顺序来依次执行。其优点是简单、公平。缺点是效率不高。因为对于相邻的两次访问请求,它们可能没有任何关系,可能访问的是不同的文件,因而在磁盘上的存储位置相距甚远。这样,就有可能使得磁头反复移动,而且每次移动的距离都比较远,从而增加了柱面定位的时间。

例如,假设一个磁盘总共有200个柱面,其编号为0～199。现有一批进程在同时访问该磁盘,这些访问请求的到达顺序为38,184,99,123,15,126,66,70,这些编号都是各个访问请求中的柱面号。已知磁头的起始位置在第60个柱面上,现在要计算,当这些访问请求被执行完后,磁头移动的总距离是多少。

如图5.25所示,在使用了先来先服务算法以后,访问请求的执行顺序就是它们的到达顺序。因此,在这8次磁盘访问中,磁头总共移动的距离为560,平均移动距离为70。

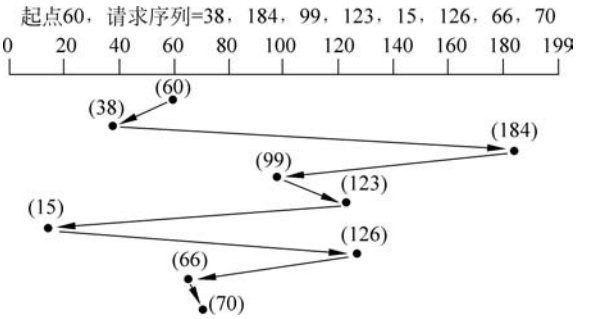

图5.25　先来先服务算法

从这个例子可以看出,在FCFS算法下,磁头每一次移动的距离都比较大。因此,这种算法的效率不是很高。

2. 最短定位时间优先算法

最短定位时间优先(Shortest Seek Time First,SSTF)算法的基本思路是:从磁盘访问的请求队列当中,选择从当前的磁头位置出发,移动距离最短的那个访问请求去执行。这个算法的目标是使每一次的磁头移动距离最短,因此是一种局部最优算法。当然,它并不一定能够得到整体最短的平均柱面定位时间,即不一定能找到全局最优方案。但是一般来说,它比先来先服务算法具有更好的性能。

此外,SSTF算法还有一个问题:如果需要访问的扇区是位于磁盘中间的柱面上,那么就会比较有利,因为它被执行的机会更多,只要在它左边或右边相邻的柱面上,有一个扇区被访问了,那么它就很可能被访问;反之,如果要访问的扇区是位于磁盘两侧的柱面上,那么就不太有利,因为它的邻居比较少,所以被访问的机会也就比较小。有时甚至可能会处于"饥饿"状态,即始终没有机会去执行。

对于刚才的例子,如图5.26所示,在使用了最短定位时间优先算法以后,访问请求的执

行顺序为：(60),66,70,99,123,126,184,38,15。在这 8 次磁盘访问中，磁头总共移动的距离为 293，平均移动距离为 36.625。这就比使用先来先服务算法好很多了。

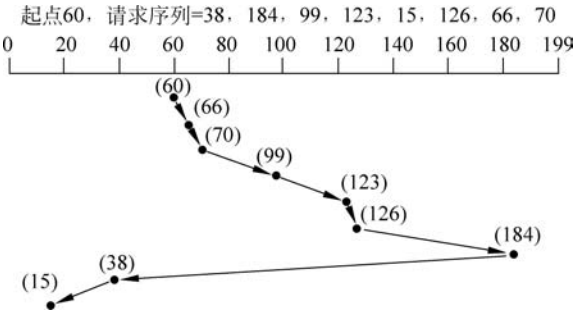

图 5.26　最短定位时间优先算法

3. 电梯算法

电梯算法(Elevator Algorithm)，也称为扫描(Scan)算法。它的基本思路是：磁头从当前位置开始，先沿着一个方向移动，并且依次执行这条路径上所有的访问请求。直到前面已经没有任何访问请求，然后再换一个方向，回过头来继续进行。

电梯算法的优点是：它克服了最短定位时间优先算法的缺点，既考虑了距离的因素，又考虑了方向的因素。在 SSTF 算法中，如果需要访问的扇区位于磁盘两侧的柱面上，那么这个访问请求被执行的机会就很少，甚至有可能处于“饥饿”状态，始终得不到执行的机会。例如，当一个柱面号较偏的访问请求正在等待时，如果不断地有新的访问请求到来，而这些访问请求都是位于中间位置，那么它们总是会被优先执行。而对于那个正在等待的访问请求，却始终得不到执行的机会。但是在电梯算法中，就没有这个问题。因为只要电梯的当前移动方向是正确的，那么就会越来越近。它不会走到一半的时候又掉头往回走。

电梯算法有一个比较好的性质，即对于任何一组访问请求，磁头移动的总距离有一个固定的上界，即柱面总数的两倍。这其实很好理解，因为无论有多少个访问请求在等待，无论这些访问请求在什么位置，只要把磁头从最左边移动到最右边，然后再从最右边移动到最左边，只要走完这两趟，那么所有的访问请求都能够执行完毕。也就是说，磁头移动的总距离不会超过柱面总数的两倍。

对于刚才的例子，如图 5.27 所示，在使用了电梯算法以后，假设初始方向为往左，那么访问请求的执行顺序为：(60),38,15,66,70,99,123,126,184。在这 8 次磁盘访问中，磁头总共移动的距离为 214，平均移动距离为 26.75。

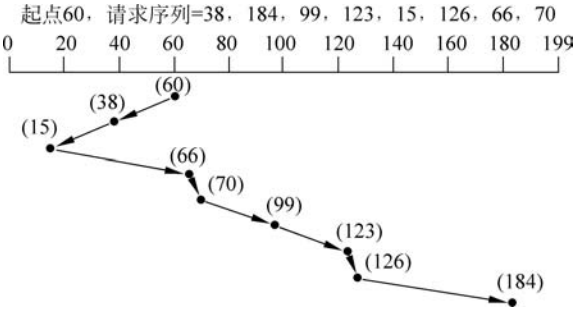

图 5.27　电梯算法

5.4.4 出错处理

从磁盘的发展历史来看,它的一个必然的趋势就是存储容量越来越大。但是对于磁盘的盘面来说,其大小是固定的,甚至越来越小。因为磁盘越小,携带就越方便。在这种情形下,为了增加磁盘的容量,生产厂商只能不断地去提高盘面数据的密度。所谓数据密度,是指在单位长度的磁介质上能够存放的数据的位数。这样一来,就对磁盘的工艺水平提出了更高的要求,而且不可避免地会带来瑕疵,使得磁盘上的某些地方、某些数据位不能正确地访问。这样,这些数据位所在的扇区就成为一个坏扇区,即写进去的数据不能完全正确地读出来。由于磁盘的访问是以扇区为单位,因此在一个扇区当中,只要有一个数据位不能正常访问,那么整个扇区的数据都不能使用。当然,如果出错的位数不是很多,只有少量的几个,都可以通过扇区的纠错码来校正。但如果错误位再多一些,那就没有办法了,整个扇区都将无法使用。

对于磁盘中的坏扇区,有以下两种处理策略。

- 由设备控制器来处理:在磁盘出厂前,对整个磁盘进行测试,然后用一个列表来记录所有的坏扇区,并把它写入磁盘。然后对其中的每一个坏扇区,用一个备用的扇区来替代它。一般来说,在磁盘的性能参数中,会有磁盘的容量,即该磁盘包含多少个扇区。事实上,这其实是一个对外的数字,而对内一般会更多一些。这些多出来的扇区就是备用扇区。如果磁盘中有一些扇区坏了,就可以使用这些备用扇区来代替。
- 由操作系统来处理:操作系统对整个磁盘进行测试,以获得一个坏扇区的列表。在此基础上,可以构造一个重映射表,对扇区编号进行调整。另外,为了避免这些坏扇区被再次使用,可以构造一个特殊的“文件”,该文件不是用来存放数据,而是用来“占用”所有的坏扇区。这样,这些扇区就不会再分配出去。

5.5 固态硬盘

在本章的最后,我们来看另外一种类型的硬盘,即固态硬盘,在当代计算机中,逐渐采用固态硬盘来替代原来的机械硬盘。

5.5.1 闪存

固态硬盘来源于闪存,所谓闪存,即闪速存储器(Flash Memory)。1984年,日本东芝公司的工程师Fujio Masuoka首先提出了闪存的概念。1988年,美国Intel公司推出了一款256KB的闪存芯片,从而成为世界上第一个将闪存商业化并投放市场的公司。闪存是一种存储器,那为什么叫Flash这个名字呢?我们知道,在英文当中,Flash的意思是闪光灯,即在使用照相机照相时,如果光线不太好,可以使用闪光灯来增加环境的亮度。那么当闪存这个技术在刚刚发明时,如果要擦除其内容,必须一次性把所有内容都擦除,给人的感觉就好像是闪光灯一闪,然后所有内容都没了,变成一片空白,因此就给它起了这样一个名字。事实上,以前有一部科幻电影“黑衣人”,里面就有类似的镜头。黑衣人是星际移民局的警察,他们在调查案件时,会询问一些目击证人。在询问结束后,他们希望目击证人把所看到的事

情全部忘记，因此就会拿出一个特殊的设备：记忆消除器。然后一按开关，只见一道白光闪过（就好像闪光灯一样），这时，目击证人的所有记忆就被消除了。当然，为了避免被误伤，黑衣人在使用该设备之前，要提前戴上一副墨镜。

闪存是一种存储器，它可以读也可以写。它的基本单元电路（即存储细胞）是双层浮空栅 MOS 管，然后带电表示存入 0，不带电表示存入 1。闪存的访问速度比机械硬盘要快，因为硬盘内部有机械装置，例如，需要把磁头移来移去，而且盘面还要不停地转动，这些都是机械运动，需要较长的时间。而闪存内部没有机械装置，它是通过电气的方法来进行擦写，因此速度比较快。另外，与磁盘一样，闪存也是非易失型的，即在断电以后也不会丢失信息。它还有一个优点就是经久耐用，能够忍受很大的压力和极端的温度，并且不怕水。这实际上是一个非常好的优点，尤其是对于一些马大哈来说。有一年，笔者的一个 U 盘忽然不见了，找了很久都没有找到，当时以为丢了，就没再管它。到了第 2 年，天气变凉，又到了穿外套的时间，有一天在穿一件衣服时，里面突然掉出来一个 U 盘，一看就是去年丢的那一个，原来它在外套里躲了一年，然后一试，还能用。这件衣服在去年收起来的时候，肯定是洗过的，换言之，这个 U 盘经历了洗衣机的考验。这是非常不简单的事情，因为衣服是泡在水里面，而且水里还有洗衣粉。然后洗衣机在工作时，衣服在里面翻来覆去，来回转动。尤其是在最后的甩干环节，旋转的速度和力度是非常大的。此外，在洗衣机洗完之后，衣服还要挂在太阳底下晒干，也就是说，这个 U 盘还要经历潮湿和高温的双重考验。在这种情形下，最后这个 U 盘还能继续使用，的确是很不容易。总之，由于闪存的上述这些特点，使它在嵌入式系统中得到了广泛的应用，然后现在又扩展到 PC 领域。

根据结构的不同，Flash 闪存又分为两种类型：NOR Flash 和 NAND Flash。对于 NOR Flash，它的设计目标是替代原来的只读存储器 ROM，但是又比 ROM 要好，可以方便地进行重写，所以它一般用来存储那些不经常更新的程序代码。NOR Flash 的读操作能力非常强，不仅速度快，而且提供完全的地址和数据总线，可以随机地访问任何一个存储单元。这一点类似于普通的随机访问存储器 RAM，即只要给出一个地址，就能访问相应的内存单元。在这种情形下，CPU 可以直接去执行存放在 NOR Flash 上面的程序，而不用事先把它装入到内存。对于写操作，它也能随机写，但是速度比较慢，而且在写入时有限制，只能把数据位从 1 变成 0，而不能反过来。如果要想把 0 变成 1，只能先进行擦除操作，但擦除必须以块为单位，即把整个块中的每一个数据位都变成 1。

NAND Flash 的设计目标是尽量缩小芯片的面积，实现大容量的存储，以匹敌磁介质的存储设备，如硬盘。换言之，它的设计目标是去替代传统的机械硬盘。根据这个目标，NAND Flash 有它自己的一些特点。首先，普通的硬盘是可读可写的，而 NOR Flash 的读操作能力较强，但写操作能力较弱。因此 NAND Flash 对此进行了改进，与 NOR Flash 相比，它的擦除和写入的速度都比较快。其次，硬盘的特点是容量大，因此 NAND Flash 在设计时，尽可能地进行压缩，每个存储单元都比较小，这样总的体积就比较小，存储密度也大，而且使用的元器件比较少，所以成本也就更低。当然，NAND Flash 也有缺点，它的 I/O 接口不提供随机访问的外部地址总线，换言之，不能对它进行随机访问，所有的操作（包括读、写和擦除）都必须以块为单位来进行。基于这个原因，NAND Flash 不适合取代原来的 ROM，因为微处理器不能直接执行存储在它上面的代码，它主要还是用来替代硬盘这样的块设备。

请读者思考一个问题,在一个嵌入式系统(如数码相机或摄像机)中,可能会用到哪些不同类型的存储器?

首先,需要一个存储器来存放系统的引导程序,所谓引导程序,即在开机后执行硬件检测、初始化和系统装入等功能的代码。也就是说,在这个存储器中存放的主要是代码,其内容一般不会修改,但必须是可以随机访问的,即可以直接执行它里面的代码,而不需要额外的其他存储器。另外,该存储器必须是非易失型的,即在断电以后其内容还在。那么什么样的存储器适合上述要求呢?根据这些特点,可以用ROM或NOR Flash来实现。

其次,在系统正常运行时,需要一个存储器来存放程序的代码和数据。该存储器必须是可以随机访问的,而且访问速度要快。其内容可读可写,但不要求是非易失型的。显然,这个存储器就是通常所说的内存,内存一般用DRAM存储器来实现。

再次,系统的固件也需要存放在一个存储器当中。所谓系统的固件,包括设备驱动程序、操作系统、图形用户界面和各种应用程序等。要求这个存储器必须是可读可写的,而且有一定的容量,是非易失型,但不要求可以随机访问。根据上述特点,可以用NAND Flash来实现,一般不会用机械硬盘。

最后,需要一个存储器来存放用户产生的数据文件,如用户拍摄的照片和录像等。这个存储器的特点是可读可写,可以永久保存,而且容量越大越好,但不要求是随机访问的。根据这些特点,可以用NAND Flash或机械硬盘来实现。硬盘的好处是容量大,价格便宜。缺点是比较重、不太方便,而且有噪声。

5.5.2 NAND Flash

如前所述,NAND Flash的设计目标是替代传统的机械硬盘,因此我们再对它进行进一步的阐述。机械硬盘的最小访问单位是扇区,一个扇区一般是512B或4KB,然后若干个扇区组成一个数据块。那么在NAND Flash当中,它的做法也是类似的。最小的访问单位是页(Page),这个页就类似于磁盘中的扇区,每一页由若干个KB组成,如4KB。然后连续的若干个页就组成了一个块(Block),这个块就类似于磁盘中的数据块。如图5.28所示,在这个NAND Flash中有12个页,下标从0到11。然后每4个相邻的页组成一个块,如0～3页组成了块0,4～7页组成了块1,8～11页组成了块2,以此类推。另外,对于每一页,它又是由两个部分组成,一个部分是有效容量,用来真正存储数据;另一个部分用来存放附加的校验信息。

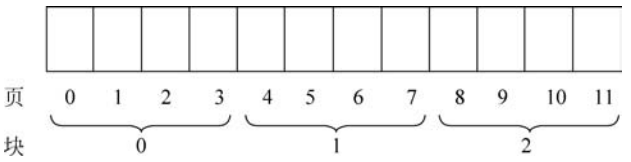

图5.28 页与块

在NAND Flash当中,可以进行读取一页的操作,即读取某一页的所有内容。这跟磁盘是一样的,每次读磁盘至少要读一个扇区。读一页所需要的时间开销是10～20μs。显然,这个时间是非常短的,速度很快。与之相比较,在读取一个硬盘扇区时,需要3个部分的时间,柱面定位大概4ms,旋转延迟大概2ms,数据传送大概1.4μs。而NAND Flash没有

机械部件，因此没有柱面定位和旋转延迟时间，所以速度就快很多。另外，NAND Flash 读取一页所需要的时间与页号和之前的访问请求无关。也就是说，连续的两次页访问之间是没有关系的，是相互独立的。而对于磁盘，连续的两次扇区访问之间是有关系的，如果是相邻的两个扇区，那么访问速度就更快，因为不再需要柱面定位和旋转延迟；如果是随机的两个扇区，那么访问速度就比较慢。总之，NAND Flash 的读操作能力是非常强的。

NAND Flash 的写操作有点奇特。写操作也可以以页为单位，每次写一页的内容，但是只能将数据位从 1 写成 0，而不能从 0 写成 1。例如，如果当前的值为 1111，那么可以对它进行修改，把它修改为 1110，即把最后一位从 1 改成 0。但是反过来就不行，也就是说，如果当前值为 1110，那么就不能把它修改为 1111。如果确实需要把某一个数据位从 0 改为 1，那么就必须先执行擦除操作，把所有的数据位都初始化为 1，然后再把相应的 1 修改为 0。但问题是，擦除操作又不是以页为单位，而是以块为最小单位，需要把这个页所在的整个块全部都初始化为 1。在时间开销上，擦除一个块需要 1～2ms，这就比读操作要慢很多。然后写操作需要 20～200μs，这虽然比擦除操作要快，但也比读操作要慢。而且在进行写操作之前，经常要先进行擦除，所以一次写操作所需要的总时间往往等于一次擦除时间再加上一次写入时间。另外，擦除操作是有次数限制的，超过了这个次数可能就无法再使用了。

NAND Flash 的写操作有点类似于生活中的一个例子。例如，国内在举办一些运动会的时候，经常会组织很多人去翻牌子，就是每个人手里拿着几块不同颜色的牌子，然后听从导演的指挥，在不同的时候举着不同颜色的牌子，这样就能拼出各种文字和图案。那么这就有点像 NAND Flash 的写操作。例如，背景颜色是红色，就是把所有的数据位都初始化为 1，然后再让其中的一些人举着白色的牌子，拼出来一幅图案，这就好比是把某些数据位从 1 修改为 0。

NAND Flash 的这种写操作显然是不太方便的。具体来说，在一次写入操作中，如果所有的修改仅限于把某些 1 修改为 0，那么没有问题，这次写入操作可以成功。反之，如果这次写入操作需要把某些 0 修改为 1，那么就不能直接写入，而要先增加一次擦除操作。

例如，假设某个存储单元当前的值为字符'c'，即二进制的 01100011，在这种情形下，如果想把这个存储单元的值修改为字符'b'，那是没有问题的，可以直接把这个字符写入。因为字符'b'的二进制是 01100010，所以如果要把字符'c'修改为字符'b'，那么唯一要做的事情就是把字符'c'末尾的那个 1 修改为 0 即可。但是如果想把这个存储单元的值修改为字符'G'(其二进制为 01000111)，那么就不行了，因为如果要把 01100011 修改为 01000111，那么这涉及两个变动，一是把第 3 位的 1 修改为 0，这是允许的；二是把第 6 位的 0 修改为 1，而这是不允许的。所以不能直接把字符'G'写入，而是要先进行擦除，即把该存储单元的值修改为 11111111，然后再把这个字符写入。

有了 NAND Flash 以后，就有了一种新的存储器，可以用来存放数据。但闪存本身并不是一个完整的、独立的存储设备，并不能直接把它连在计算机上。如前所述，对于一个 I/O 设备，除了设备本身以外，还需要有设备控制器，存储设备也不例外。因此，只有给闪存配上了设备控制器以后，才能真正去使用它们。

5.5.3 U 盘

闪存在移动存储领域的一个重大成功案例是 U 盘。U 盘全称为 USB 闪存驱动器，它

是一种使用USB接口的无需物理驱动器的微型高容量移动存储产品,通过USB接口与计算机连接实现即插即用。2002年7月,我国朗科公司“用于数据处理系统的快闪电子式外存储方法及其装置”的专利获得国家知识产权局正式授权,从而揭开了移动式存储设备领域新的篇章,彻底地将软盘扫入了历史的长河。

图5.29是最初的U盘的系统结构图,它主要由两个部分组成:存储控制电路和快闪存储器(即闪存)。存储控制电路需要解决两个问题,一是对外的接口,二是对内的控制。对外的接口采用的是通用串行总线(Universal Serial Bus,USB),这是由英特尔、微软和康柏等公司于1995年联合制定的一种数据通信方式,并逐渐成为行业标准。USB总线作为一种高速串行总线,具有传输速度快、供电简单、安装配置便捷(支持即插即用和热插拔)、易于扩展、传输方式多样化以及兼容性良好等优点,自推出以来,已成功替代串口和并口,成为现代计算机和智能设备的标准扩展接口和必备接口之一,目前已发展到USB 4.0版本。

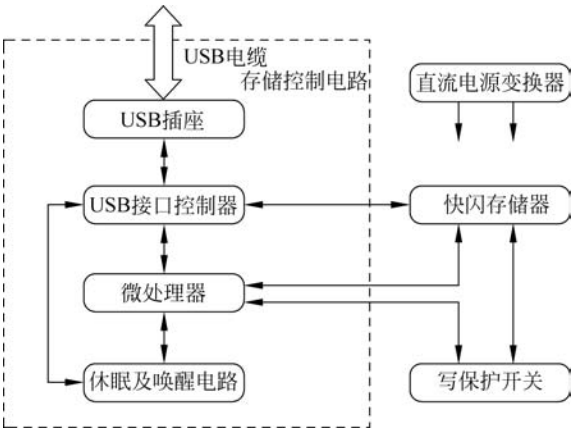

图5.29 U盘的系统结构图

存储控制电路实现控制功能的核心部件是微处理器,其内部有一个“快闪电子式外存储装置固件”(Firmware),用于直接控制闪存的存取并实现接口的标准功能。

当然,除了硬件以外,为了使U盘能正常工作,还需要有相应的软件,即设备驱动程序,设备驱动程序是安装在操作系统内核中的。

对U盘的读操作主要包括如下步骤。

- 操作系统接受用户的读命令,并将该命令发送给设备驱动程序。
- 设备驱动程序将读命令转换为内部固件能够理解并执行的特殊读操作指令,并通过USB接口控制电路传送给微处理器中的固件。
- 固件执行读操作,并将结果及状态返回给驱动程序。

对U盘的写操作主要包括如下步骤。

- 操作系统接受用户的写命令,并将该命令发送给设备驱动程序。
- 设备驱动程序判断U盘是否打开了写保护开关,如果有,则本次操作结束;如果没有,则继续往下进行。
- 设备驱动程序将写命令转换为内部固件能够理解并执行的多个特殊操作指令,并通过USB接口控制电路逐一传送给微处理器中的固件。

- 固件先按读操作指令对欲写入的存储区域进行读操作，并将读出的数据传回给驱动程序。
- 固件再按擦除操作指令对该存储区域进行擦除操作，并将擦除结果传回给驱动程序。
- 驱动程序将读出的数据同欲写入的数据进行整合，并将整合后的数据及写操作指令发送给固件，由固件将整合后的数据重新写入目标存储区域。
- 固件将写入后的结果与状态传回给驱动程序。

显然，写操作比读操作要复杂得多，原因主要有两个，一是对于闪存来说，在写入之前先要进行擦除操作，而擦除会破坏相邻的其他数据，所以先要用读操作把它们保存起来；二是擦除的最小单位是块。

5.5.4 SSD

闪存的另外一个应用案例是固态硬盘(Solid State Drive，SSD)。SSD是一种固态存储设备，使用集成电路部件来永久地存储数据，它的外部接口与传统的机械硬盘相同，可以像普通的硬盘那样来使用。SSD的存储介质主要是NAND Flash，早期也有用DRAM的，但DRAM是易失型存储器，因此为了不丢失数据，还得配一个电池，这样就不太方便。既然固态硬盘的存储介质是NAND Flash，因此前面讨论过的NAND Flash的优点和缺点也都存在。它的优点是：可靠、无噪声(没有机械装置)、读取速度快。另外，对外部环境不是太敏感，不像机械硬盘那么娇弱，害怕碰撞和震动。事实上，它的存在形式主要就是一块电路板，上面焊着一些芯片，没有什么精密的零部件，因此更加皮实一些。从缺点来看，SSD的价格比较贵，比机械硬盘贵很多，因此，虽然现在主流的计算机基本上都采用SSD而不是机械硬盘来作为外部存储器，但是存储容量却变小了。不过，从半导体行业的发展历史来看，这个问题应该很快就会得到解决。另外，SSD的写入速度较慢，而且擦除的次数有限。

图5.30是固态硬盘的系统结构图，它与U盘组成结构的基本逻辑是差不多的，只是更加复杂一些。

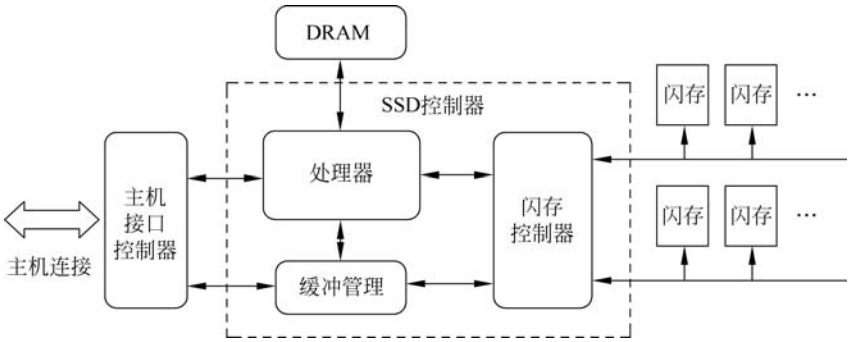

图5.30 固态硬盘的系统结构图

首先，和机械硬盘一样，SSD也需要采用某种方式连接到计算机上，因此，它会有一个主机接口单元(包括控制器和连接线缆等)来完成这个功能。常用的接口包括SATA、M.2

和 PCI-E 等。

其次,固态硬盘的核心或者说大脑是 SSD 控制器,也叫主控芯片,它的存在形式通常是一块独立的芯片,由专门的芯片设计产商提供。SSD 控制器的主要功能就是承上启下,实现数据的中转,将硬盘内部的闪存与外部的主机接口连接起来,并合理地调配数据在各个闪存芯片上的负载。这些工作是通过 CPU 处理器中的固件来完成的。

最后,SSD 控制器中的闪存控制器与闪存芯片相连,它通过各种专门的控制指令,管理着数据的读取和写入。

在访问方式上,固态硬盘与机械硬盘没有任何区别,都是把它看成一个块设备,然后通过数据块的地址来访问。但是从另外一个方面来说,由于它们内部的实现机理是不一样的,因此在使用时也要注意有所区别。

首先,对于机械硬盘,考虑到柱面定位和旋转延迟问题,因此在使用了一段时间以后,要进行碎片整理工作,即把文件内部的数据块尽量连续存放,以提高文件的访问速度。但是对于固态硬盘来说,由于它根本就没有柱面定位和旋转延迟环节,因此就不需要进行碎片整理。而且由于它的擦写次数是有限的,因此如果硬要进行碎片整理,那么不仅没有好处,反而会磨损硬盘。举一个生活中的例子,对于机械硬盘,好比是统一用一辆校车来接送全校的孩子,因此,如果能让孩子们搬家住在一块儿(即碎片整理),那么校车的接送就会很方便,每次只要跑一个地方即可。而对于固态硬盘,好比是每个家庭都是家长自己开车接送孩子,这样孩子们是否住在一起就没有什么影响。

其次,有的读者认为,既然固态硬盘的访问速度比机械硬盘要快,因此,有了固态硬盘以后,就可以疯狂地下载电影了,这也是不对的。固态硬盘和机械硬盘各有优点和缺点,因此,最好的做法是扬长避短,把两者的优点结合起来。固态硬盘的优点是速度快,缺点是价格高。因此,为了提高它的使用效率,可以把那些经常要使用的、以读为主的以及运行速度较慢的内容保存在固态硬盘,这样能提高访问的速度;而对于那些很少使用的或者运行速度已经足够快的内容,则可以保存在另外一块机械硬盘中。如电影文件,偶尔才看一次,然后文件又特别大,因此不适合保存在固态硬盘。另外,对于那些需要频繁更新的文件,最好也保存在机械硬盘,因为固态硬盘的擦写次数是有限的。

最后,如前所述,为了提高硬盘的访问速度,操作系统会在内存提供缓存功能,即把最近访问过的一些磁盘数据块放在内存中,这样能减少对磁盘的访问次数。那么在引入了固态硬盘以后,考虑到它的访问速度比较快,这时是否还需要缓存功能呢?当然还是需要的,不管是什么类型的硬盘,有了内存缓存后,都能减少对硬盘的访问次数,提高访问速度。尤其是对于写操作,使用缓存后可以减少对硬盘的擦除和写入次数,从而延长硬盘的工作时间。事实上,在固态硬盘的内部通常也会有一块 DRAM 芯片,作为缓存来使用。

习　　题

一、单项选择题

1.(　　)是直接存取(Direct Access)的 I/O 设备。

A. 磁盘　　B. 磁带　　C. 打印机　　D. 键盘

2. 下列哪一个是软件？（　　）

A. Device Controller　　B. DMA

C. Hard Disk Drive　　D. Device Driver

3. 在单处理机系统中，可并行的是（　　）。

Ⅰ 进程与进程　　Ⅱ 处理机与设备　　Ⅲ 处理机与 DMA　　Ⅳ 设备与设备

A. Ⅰ、Ⅱ和Ⅲ　　B. Ⅰ、Ⅱ和Ⅳ　　C. Ⅰ、Ⅲ和Ⅳ　　D. Ⅱ、Ⅲ和Ⅳ

4. 下列选项中，能引起外部中断的事件是（　　）。

A. 键盘输入　　B. 除数为 0　　C. 浮点运算下溢　　D. 访存缺页

5. 在使用 I/O 设备时，以下哪一种情形不会产生 I/O 中断？（　　）

A. 打印机脱纸　　B. 数据传输结束　　C. 数据开始传输　　D. 键被按下

6. 使用 DMA 可以节省（　　）。

A. 内存访问时间　　B. 磁盘访问时间

C. 总线访问时间　　D. CPU 时间

7. 下列关于 I/O 的工作，哪一个不是在设备驱动程序中运行？（　　）

A. 在读磁盘时，将抽象的参数转换为柱面、磁道、扇区等具体的参数

B. 向设备控制器发出各种命令

C. 对于磁盘来说，磁盘的调度程序

D. 为了维护最近所访问的数据块而设置的缓冲区

8. 引入缓冲区的主要目的是（　　）。

A. 节省内存

B. 改善 CPU 和 I/O 设备之间速度不匹配的情况

C. 提高 CPU 的利用率

D. 提高 I/O 设备的运行效率

9. 为了缓解 CPU 与 I/O 设备之间速度不匹配的矛盾，系统通常会采用缓冲技术。那么这里所说的缓冲区是位于（　　）中。

A. 外存　　B. 内存　　C. ROM　　D. 寄存器

10. SPOOLing 技术是一种实现虚拟（　　）的技术。

A. 处理器　　B. 设备　　C. 存储器　　D. 链路

11. SPOOLing 技术提高了（　　）的利用率。

A. 独占设备　　B. 共享设备　　C. 文件　　D. 内存

12. 磁盘上的文件是以（　　）为单位来进行读写的。

A. 块　　B. 记录　　C. 柱面　　D. 磁道

13. 关于辅助存储器，（　　）的提法是正确的。

A. “不是一种永久性的存储设备”　　B. “是 CPU 与内存之间的缓冲存储器”

C. “是文件的主要存储介质”　　D. “可以像内存一样被 CPU 直接访问”

14. 磁盘调度的目的是缩短（　　）。

A. 柱面定位时间　　B. 旋转延迟时间

C. 数据传送时间　　D. 启动时间

二、填空题

1. 在计算机系统中,可以按照数据组织的形式,把I/O设备分为两类,一类是块设备,一类是字符设备。请各举一个例子。块设备:_____,字符设备:_____。
2. 每个I/O单元由两部分组成,一个是机械部分,即I/O设备本身;另一个是电子部分,即_____。
3. 在设计I/O软件时,一个非常关键的概念或设计目标是:_____。
4. I/O地址的编址方式有三种,即_____、_____和混合编址。
5. I/O设备的控制方式有三种,即_____、_____和_____。
6. 是否所有的I/O设备都需要用到DMA?(回答是或不是)_____。
7. 在I/O软件中,直接对设备控制器进行操作的软件是:_____。
8. 操作系统通过_____技术,可以把独占设备转换为具有共享特征的虚拟设备。
9. 当我们使用Word应用程序来打印一篇文档的时候,必须等到打印机已经完成此次打印任务以后,才能够把Word关闭,否则可能会丢失打印数据。以上这段话是否正确?_____。
10. 在访问一个磁盘扇区时,所需的时间主要包括三部分,即_____、_____和数据传送时间。
11. 假设磁盘的转速为10 000rpm,每个磁道有300个扇区,每个扇区有512B,现要读一个50KB的文件。假设柱面定位(平均)时间为6.9ms,旋转延迟(平均)时间为3ms,扇区数据传送时间为17μs。①如果文件由同一个磁道上的100个连续扇区构成,那么总共需要的时间为:_____;②如果文件由100个随机分布的扇区构成,那么总共需要的时间为:_____。

三、简答题

1. 在一个I/O设备的设备控制器当中,主要有哪些寄存器?CPU又是如何去访问这些寄存器的(即I/O编址方式有哪几种)?
2. 是否每一个I/O设备都有相应的设备控制器?在一个设备控制器当中,主要有哪些寄存器?在I/O软件中,谁负责去访问这些寄存器?如何访问这些寄存器?
3. 以磁盘读取操作为例,说明DMA的工作原理。
4. I/O设备管理软件分为哪几个层次?其中哪几个层次是与硬件设备有关?哪几个层次是与硬件设备无关?
5. 在I/O软件的层次结构中,设备驱动程序是由谁提供的?当它在运行的时候,CPU处于什么状态?设备驱动程序和中断处理程序之间如何同步?设备独立的I/O软件是由谁编写的?操作系统与设备驱动程序之间的接口是由谁定义的?
6. 磁盘的最小访问单位是什么?假设系统要去修改磁盘上的某一字节,应当如何实现这个过程?

四、应用题

1. 某硬盘的参数为：盘片数 5；柱面数 100；扇区/磁道：16。
   假设分配以扇区为单位，若使用位示图管理磁盘空间，请问位示图需要占用多大的空间？
2. 某软盘有 40 个磁道，磁头从一个磁道移至另一磁道需要 6ms。文件在磁盘上非连续存放，逻辑上相邻的数据块的平均距离为 13 磁道，每块的旋转延迟时间及传输时间分别为 100ms 和 25ms，请问读取一个 100 块的文件需要多长时间？
3. 假设一个磁盘总共有 100 个柱面，它们的编号为 0～99，访问请求的到达顺序为（柱面号）20，44，40，4，80，12，76，磁头的起始位置在 40，假设每移动一个柱面需要 3ms，请分别采用先来先服务算法、最短定位时间优先算法和电梯算法（起始方向为指向第 0 个柱面的方向），来确定这些访问请求的实际执行顺序，并计算总共花费的柱面定位时间。
4. 假设一个磁盘有 100 个柱面（编号为 0～99），每个柱面上有 12 个磁道（编号 0～11），每个磁道上有 200 个扇区（编号 0～199）。现在有 7 个磁盘访问的请求，每个访问请求用一个三维地址来表示，即柱面号，磁道号，扇区号。假设这些访问请求的到达顺序为（10，0，10）、（22，1，20）、（20，5，100）、（8，5，50）、（40，10，50）、（6，11，120）、（36，8，100），并且已知上一次磁盘访问的扇区地址为（20，1，70）。请分别采用先来先服务算法、最短定位时间优先算法和电梯算法（起始方向为指向第 0 个柱面的方向）来确定这些访问请求的实际执行顺序，并计算磁头移动的总距离（不需要画出磁头移动的轨迹图）。
5. 假设磁盘的转速为 10 000rpm，每个磁道有 300 个扇区，每个扇区 512B，现要读一个 150KB 的文件。若柱面定位（平均）时间为 6.9ms，旋转延迟（平均）时间为旋转时间的一半，扇区数据传送时间为 17μs。
   （1）如果该文件由同一个磁道上的 300 个连续扇区构成，那么访问该文件总共需要多长时间？
   （2）如果该文件由 300 个随机分布的扇区构成，则访问该文件需要多长时间？
   （3）假设该磁盘有 12 个盘片、24 000 个柱面，那么该磁盘的容量是多大？
   （4）对于柱面定位、旋转延迟和数据传送，磁盘调度算法试图减少的是其中哪一部分时间？这部分代码位于什么地方？
6. 假设计算机系统采用 CSCAN（循环扫描）磁盘调度策略，使用 2KB 的内存空间记录 16 384 个磁盘块的空闲状态。
   （1）请说明在上述条件下如何进行磁盘块空闲状态的管理。
   （2）设某单面磁盘的旋转速度为每分钟 6000 转，每个磁道有 100 个扇区，相邻磁道间的平均移动的时间为 1ms。若在某时刻，磁头位于 100 号磁道处，并沿着磁道号增大的方向移动（如图 5.31 所示），磁道号的请求队列为 50，90，30，120。对请求队列中的每个磁道需读取 1 个随机分布的扇区，计算读完这些扇区总共需要多少时间，并给出计算过程。

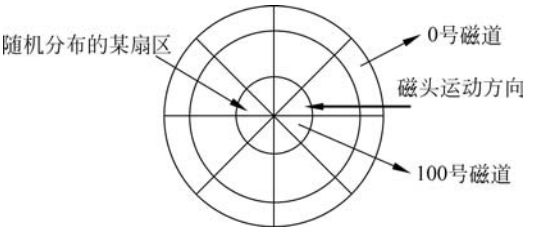

图 5.31　磁盘状态