持久化：I/O设备

邰颖
南京大学
智能科学与技术学院

I/O 设备

- I/O 设备很重要，因为进程需要一些输入，然后产生一些输出。

- 例如：
  - 输入 => 键盘 / 磁盘
  - 输出 => 屏幕

- 这些 I/O 设备是如何由操作系统管理的？（这与 CPU或内存的管理类似吗？）

系统架构

- 为何采用分层结构？
  - •物理限制
  - •成本因素
- **将高性能需求的部件尽可能靠近 CPU！**
- 制造商通过开发专用芯片组以提升性能而竞争

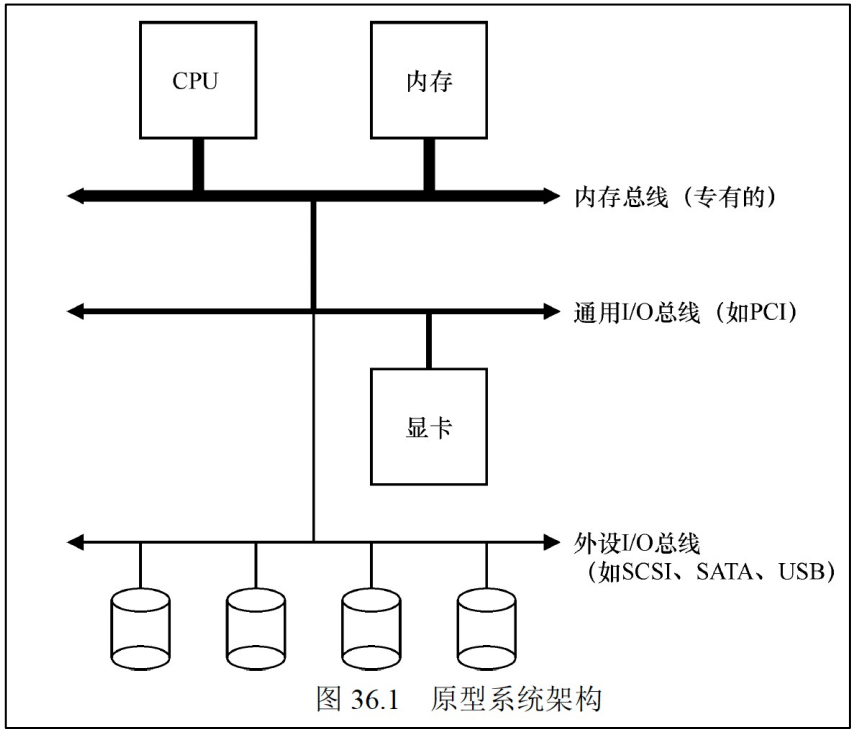
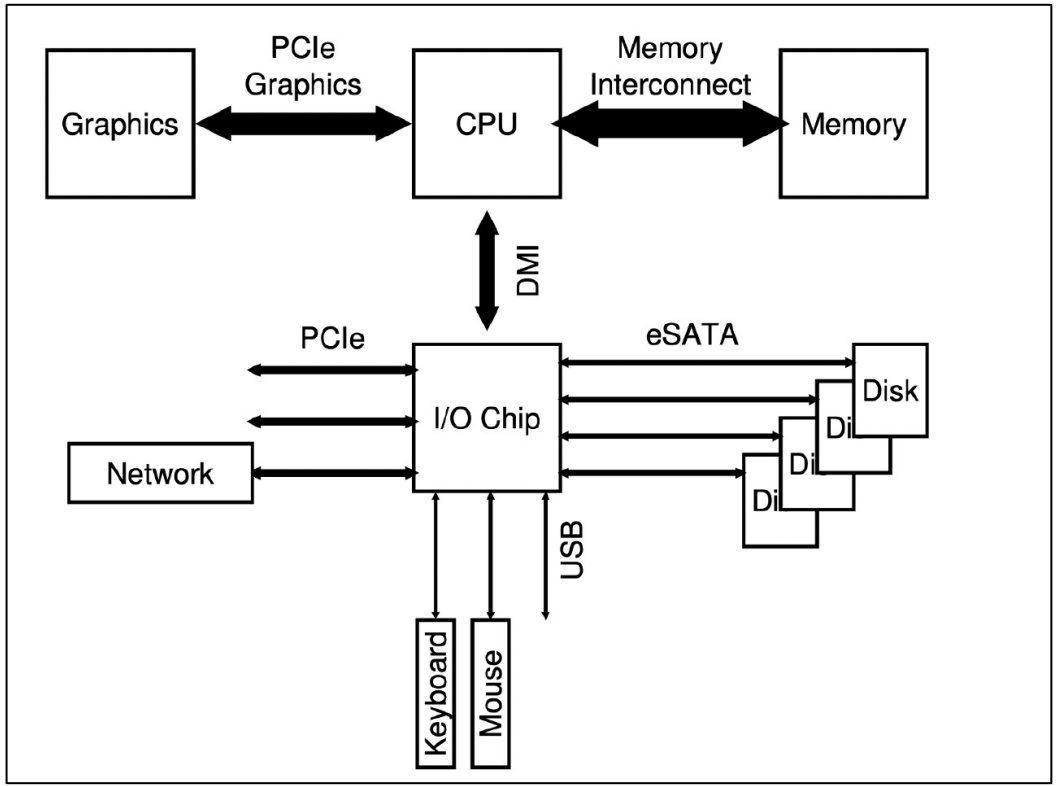


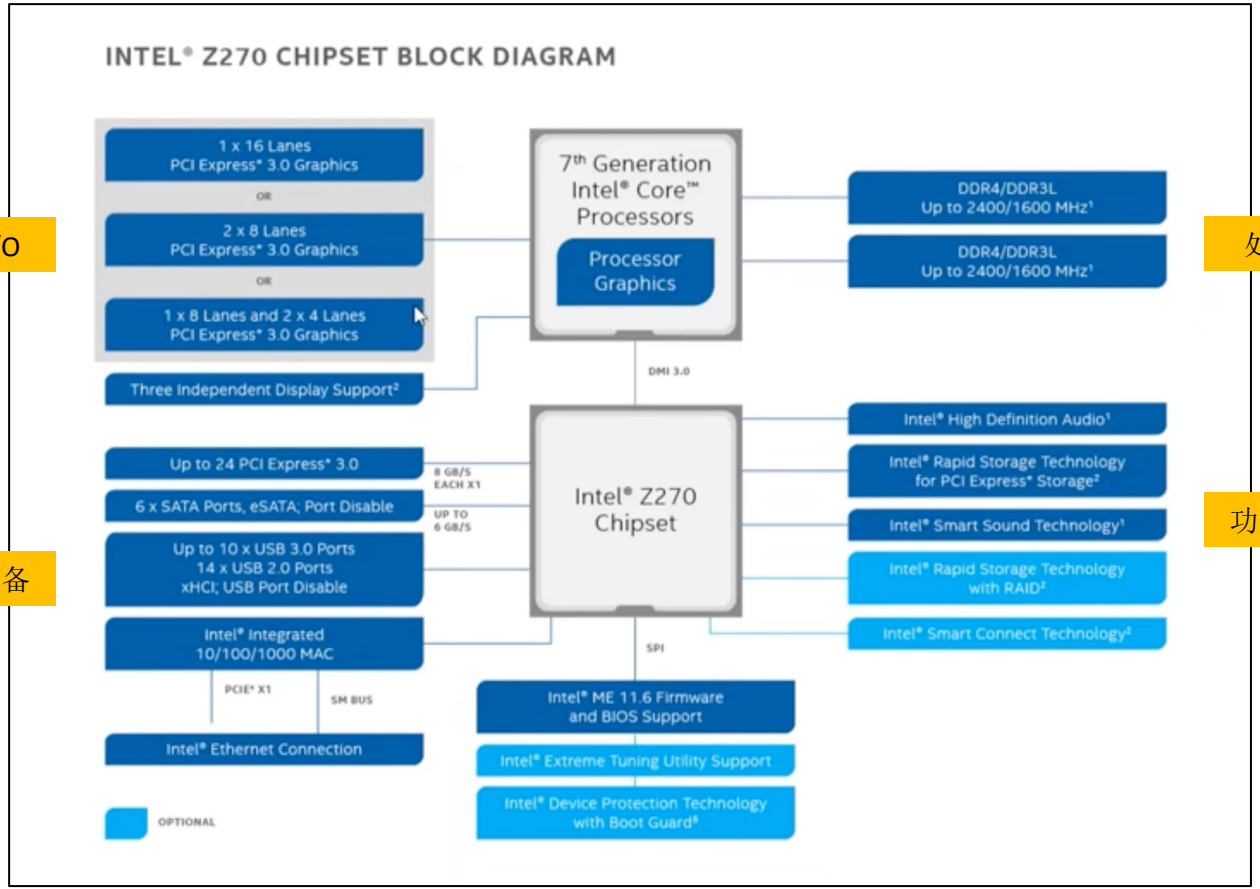
图 36.1　原型系统架构

系统架构

- 现代系统通常使用专用芯片组来提高性能



例子：Intel® Z270 芯片组框图



高速I/O

低速设备

处理器

功能模块

标准设备

- 设备提供的接口 (interface) 通常由若干寄存器组成
  - 状态寄存器 (status register): 设备当前的状态
  - 命令寄存器 (command register): 指示设备执行特定的任务
  - 数据寄存器 (data register): 将数据传递给设备或从设备获取数据

图 36.2　标准设备

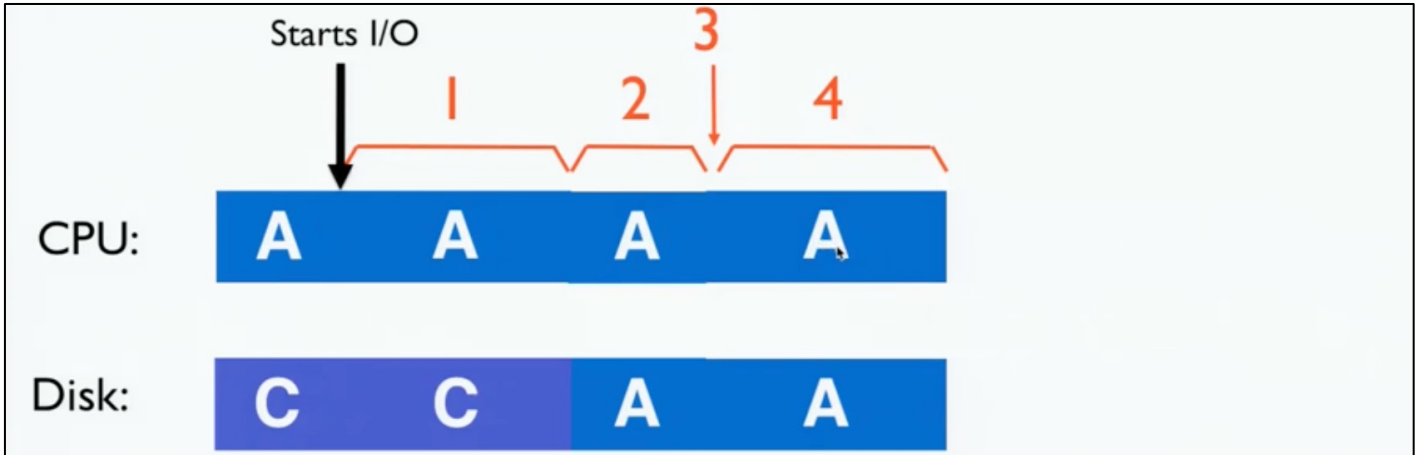
标准协议

- 现在来看“标准/典型协议”

```
While (STATUS == BUSY)
    ; // 等待设备空闲
写入数据到 DATA 寄存器
写入命令到 COMMAND 寄存器
    // 启动设备并执行命令
While (STATUS == BUSY)
    ; // 等待设备完成请求
```

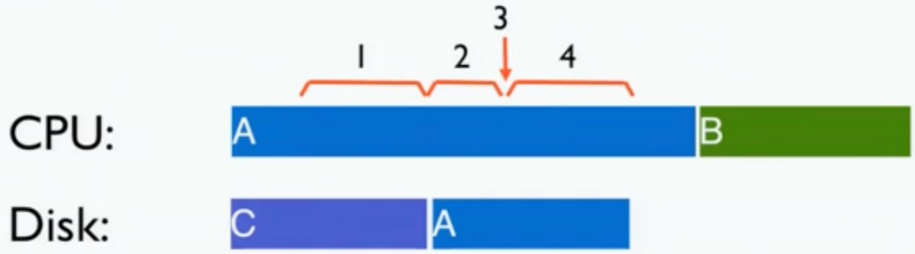
  - 这是一种轮询（polling）方式
  - 该过程称为程序控制I/O（Programmed I/O，PIO）
  - 特点：**简单、有效**

示例：标准协议




```
while (STATUS == BUSY)                  // 1
   ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
   ;
```

示例：标准协议（续）




```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```

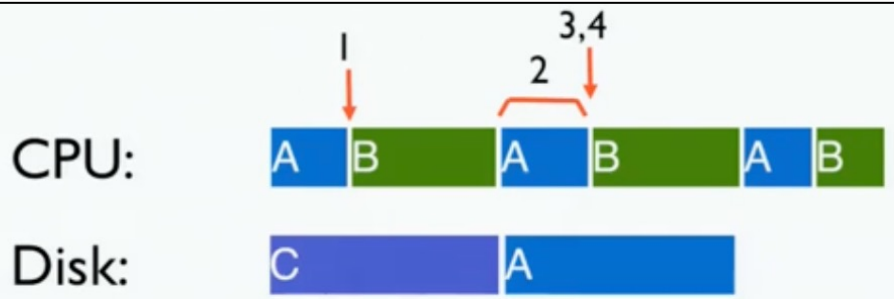
how to avoid spinning?

利用中断减少CPU开销

- **这个协议好吗？**
  - 不好，原因是忙等待（busy waiting）

- **中断机制（Interrupts）**
  - 已经在系统调用（syscalls）和时钟中断（timer）中介绍过中断！
  - 一旦发出 I/O 请求，当前线程进入睡眠状态（其他线程可以被调度运行）
  - I/O 完成时，硬件会触发中断，执行中断处理程序
  - 唤醒发起请求的线程

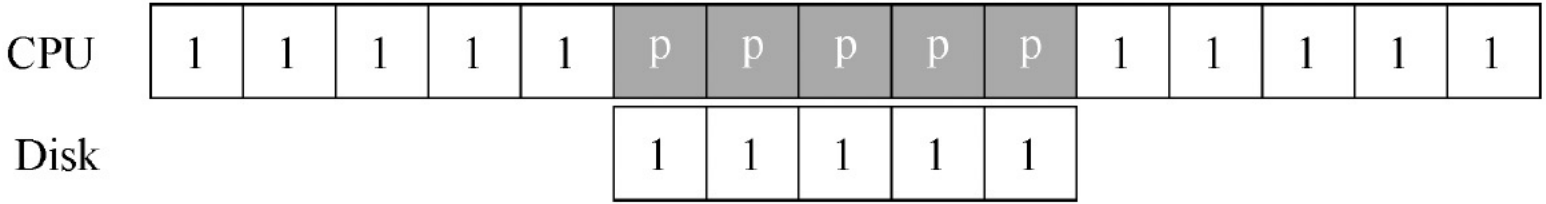
利用中断减少CPU开销（续）



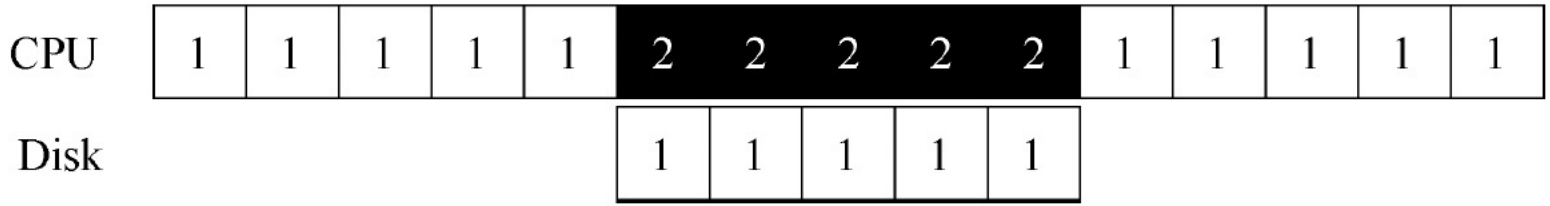

```
while (STATUS == BUSY)                  // 1
    wait for interrupt;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
    wait for interrupt;
```

利用中断减少CPU开销（续）

- **轮询机制（ Programmed I/O，PIO ）：** CPU发出请求后无法执行其他任务（忙等待）



- **中断机制：** CPU 发出请求后切换执行其他任务（线程2）



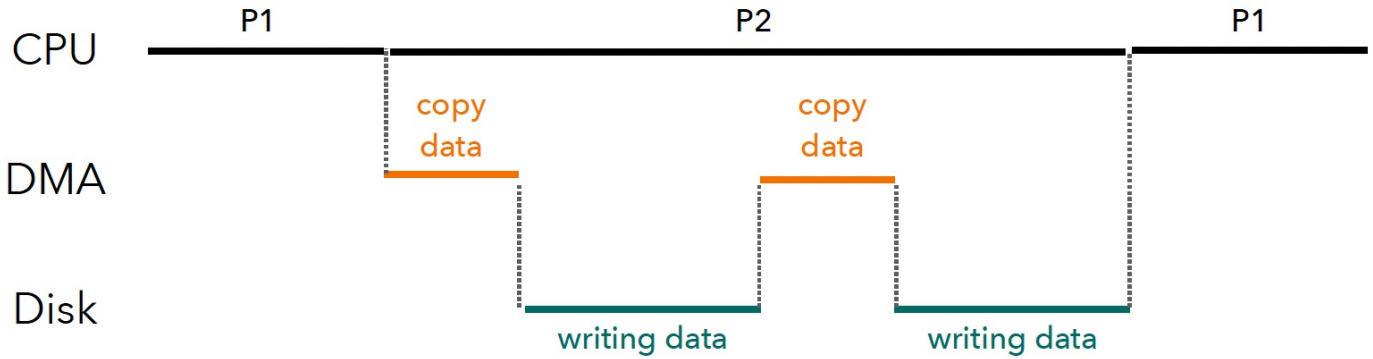
利用中断减少CPU 开销（续）

- **中断机制是最优方案吗？**
  - 并非如此。对于快速I/O，轮询（Polling）有时效果更好
  - 在高频网络场景中，中断可能导致活锁（Livelock）问题
- **其他优化方法：**
  - 混合机制（Hybrid）：先轮询一小段时间，再考虑使用中断
  - 中断合并（Coalescing）：合并多个中断事件，减少中断次数
- **理想方案：**
  - 不让 CPU 参与数据传输！
  - 使用**直接内存访问（ Direct Memory Access, DMA）**

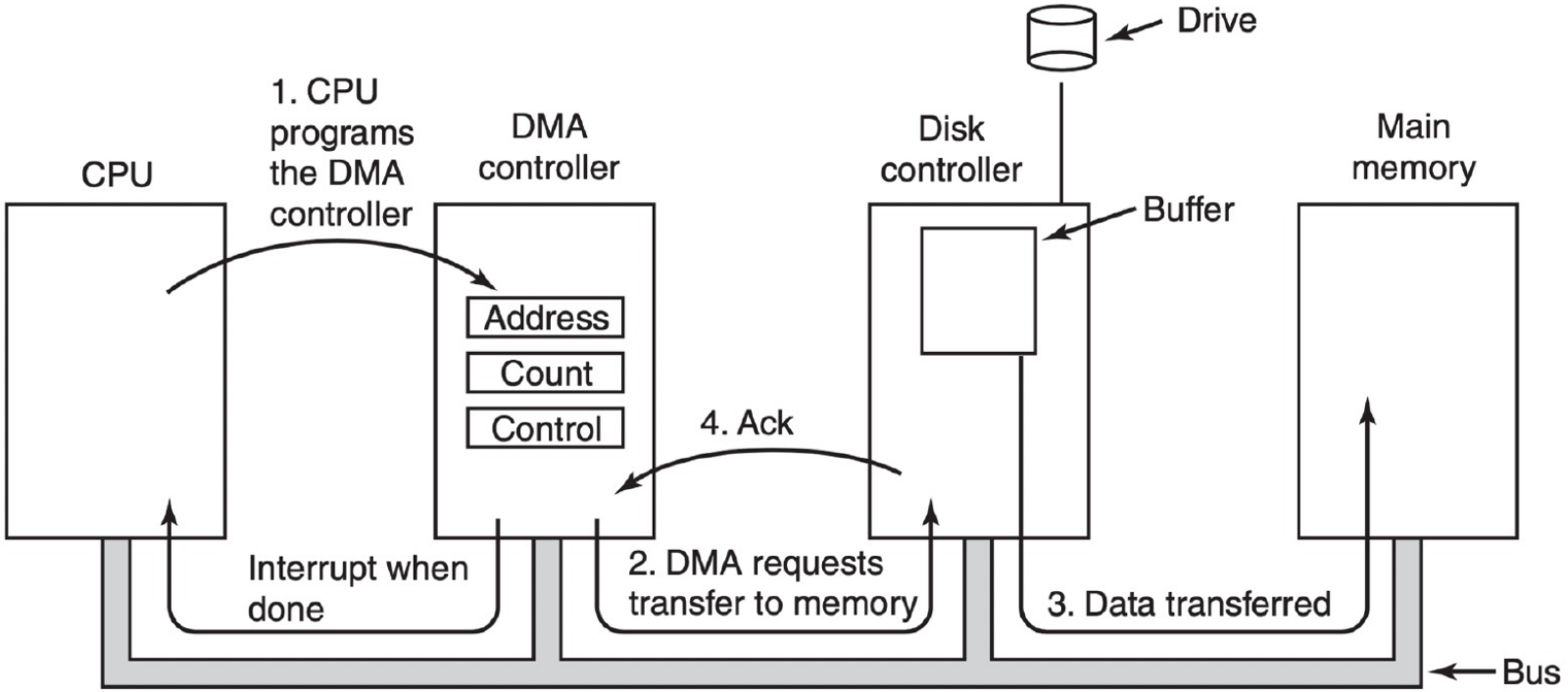
Direct Memory Access (DMA)

直接内存访问 (Direct Memory Access, DMA): 依靠一个特殊的硬件控制器来协调内存和设备寄存器之间的数据传输

- 操作系统首先向 DMA 发送指令 (where the data lives in, how much data to copy, and which device to send it to …)
- 随后由 DMA 和 I/O 设备进行交互
- 当 I/O 操作结束时，DMA 控制器向 CPU 发送中断信号

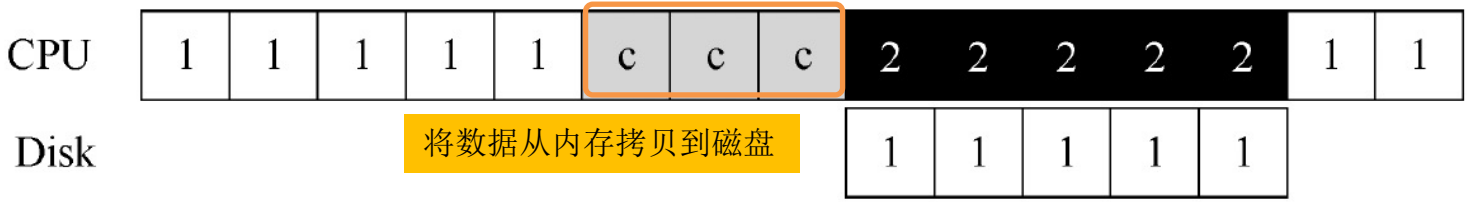


Direct Memory Access (DMA)

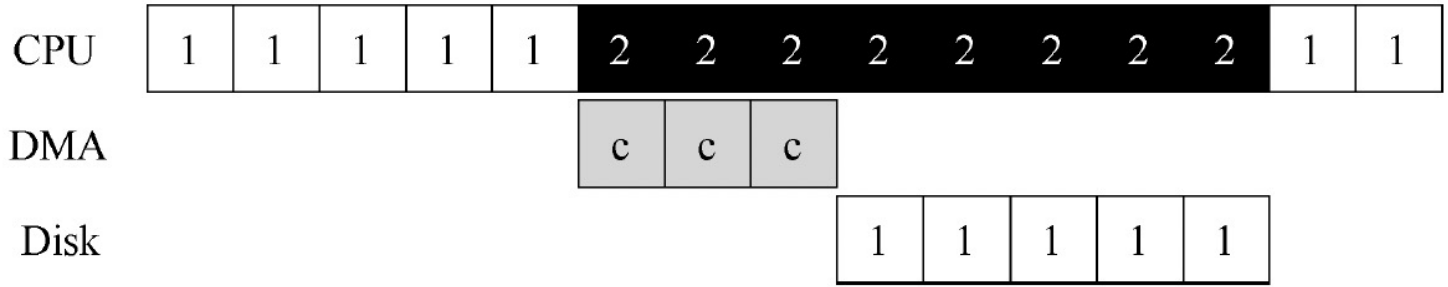


示例：利用DMA进行更高效的数据传送

- **PIO机制问题：这里存在什么问题？-> CPU需要花费时间处理数据传输**

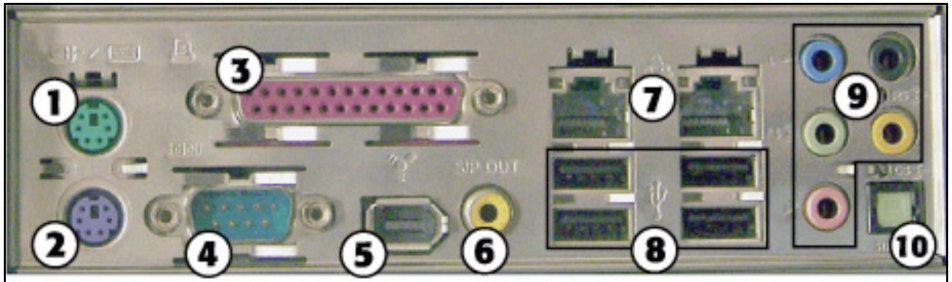


- **解决方案：DMA（直接内存访问）负责数据拷贝，CPU可以立刻切换进程**



CPU如何与设备通信：端口映射（明确指令）

- 每个控制寄存器被分配一个I/O端口号
- 使用明确的I/O指令（在x86上为in和out）
- 这些指令通常是特权指令



1. PS/2 mouse port
2. PS/2 keyboard port
3. Parallel port
4. Serial port
5. IEEE 1394a port
6. SPDIF coaxial digital audio port
7. Ethernet ports
8. USB ports
9. 1/8-inch mini-jack audio ports
10. SPDIF optical digital audio port

Port range	Summary
0x0000-0x001F	The first legacy DMA controller, often used for transfers to floppies.
0x0020-0x0021	The first Programmable Interrupt Controller
0x0022-0x0023	Access to the Model-Specific Registers of Cyrix processors.
0x0040-0x0047	The PIT (Programmable Interval Timer)
0x0060-0x0064	The "8042" PS/2 Controller or its predecessors, dealing with keyboards and mice.
0x0070-0x0071	The CMOS and RTC registers
0x0080-0x008F	The DMA (Page registers)
0x0092	The location of the fast A20 gate register
0x00A0-0x00A1	The second PIC
0x00C0-0x00DF	The second DMA controller, often used for soundblasters
0x00E9	Home of the Port E9 Hack. Used on some emulators to directly send text to the hosts' console.
0x0170-0x0177	The secondary ATA harddisk controller.
0x01F0-0x01F7	The primary ATA harddisk controller.
0x0278-0x027A	Parallel port
0x02F8-0x02FF	Second serial port
0x03B0-0x03DF	The range used for the IBM VGA, its direct predecessors, as well as any modern video card in legacy mode.
0x03F0-0x03F7	Floppy disk controller
0x03F8-0x03FF	First serial port

CPU如何与设备通信：内存映射

- 内存映射I/O：
  - 将所有控制寄存器映射到内存地址空间中
  - 每个控制寄存器被分配一个唯一的内存地址
  - 为了访问特定的寄存器，操作系统发出一个load指令（读取）或store指令（写入）该地址
  - 然后硬件将load/store指令指向到设备而不是主存储器

纳入OS：设备驱动程序

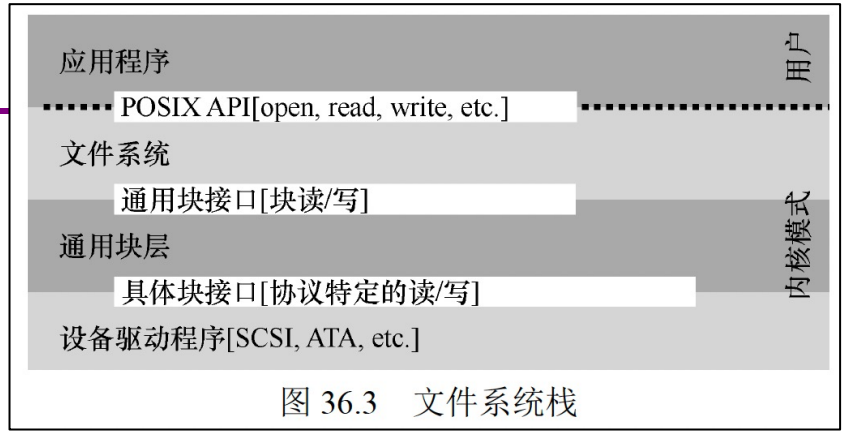


图 36.3　文件系统栈

- 操作系统创建了一个**分层的结构视图**
  - 即使在内核内部抽象也是不断使用的技术！
- 分层方法允许更低层部分容易更改
  - 例如，文件系统的实现独立于磁盘类型
- 管理 I/O 设备的代码在内核设备**驱动程序(device drivers)**
  - 70%的 Linux 代码是设备驱动程序，因为有太多不同的设备，每个设备都不同！
  - 设备驱动程序是内核错误的最常见来源
    - 这是由于给定的驱动程序可能只被少数系统使用，所以它不会被高度使用或仔细检查。

小结

- 介绍了I/O设备、系统架构
- 介绍了轮询、中断两种设备交互机制
- 介绍了PIO机制的问题，以及利用DMA提高设备效率
- 介绍了访问设备寄存器的两种方式，端口映射和内存映射
- 介绍了设备驱动程序的概念