第 36 章　I/O 设备

在深入讲解持久性部分的主要内容之前，我们先介绍输入/输入（I/O）设备的概念，并展示操作系统如何与它们交互。当然，I/O 对计算机系统非常重要。设想一个程序没有任何输入（每次运行总会产生相同的结果），或者一个程序没有任何输出（为什么要运行它？）。显而易见，为了让计算机系统更有趣，输入和输出都是需要的。因此，常见的问题如下。

关键问题：如何将 I/O 集成进计算机系统中

I/O 应该如何集成进系统中？其中的一般机制是什么？如何让它们变得高效？

36.1　系统架构

开始讨论之前，我们先看一个典型系统的架构（见图 36.1）。其中，CPU 通过某种内存总线（memory bus）或互连电缆连接到系统内存。图像或者其他高性能 I/O 设备通过常规的 I/O 总线（I/O bus）连接到系统，在许多现代系统中会是 PCI 或它的衍生形式。最后，更下面是外围总线（peripheral bus），比如 SCSI、SATA 或者 USB。它们将最慢的设备连接到系统，包括磁盘、鼠标及其他类似设备。

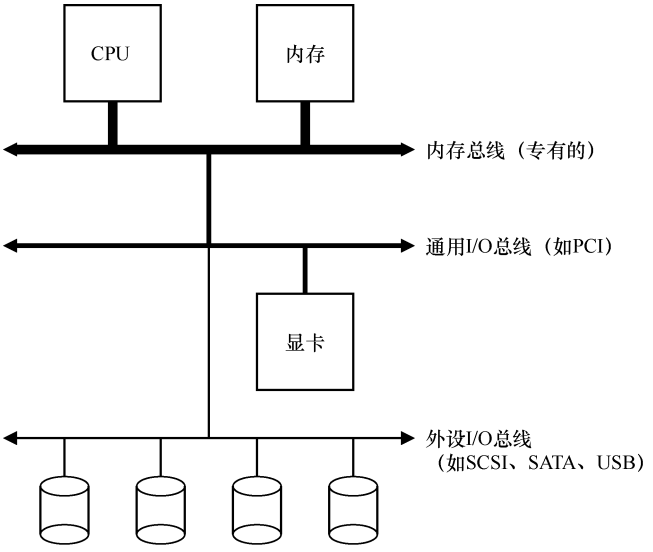


图 36.1　原型系统架构

你可能会问：为什么要用这样的分层架构？简单回答：因为物理布局及造价成本。越快

的总线越短，因此高性能的内存总线没有足够的空间连接太多设备。另外，在工程上高性能总线的造价非常高。所以，系统的设计采用了这种分层的方式，这样可以让要求高性能的设备（比如显卡）离 CPU 更近一些，低性能的设备离 CPU 远一些。将磁盘和其他低速设备连到外围总线的好处很多，其中较为突出的好处就是你可以在外围总线上连接大量的设备。

36.2 标准设备

现在来看一个标准设备（不是真实存在的），通过它来帮助我们更好地理解设备交互的机制。从图 36.2 中，可以看到一个包含两部分重要组件的设备。第一部分是向系统其他部分展现的硬件接口（interface）。同软件一样，硬件也需要一些接口，让系统软件来控制它的操作。因此，所有设备都有自己的特定接口以及典型交互的协议。

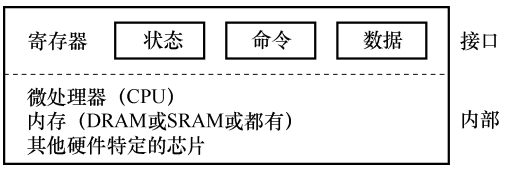


图 36.2 标准设备

第 2 部分是它的内部结构（internal structure）。这部分包含设备相关的特定实现，负责具体实现设备展示给系统的抽象接口。非常简单的设备通常用一个或几个芯片来实现它们的功能。更复杂的设备会包含简单的 CPU、一些通用内存、设备相关的特定芯片，来完成它们的工作。例如，现代 RAID 控制器通常包含成百上千行固件（firmware，即硬件设备中的软件），以实现其功能。

36.3 标准协议

在图 36.2 中，一个（简化的）设备接口包含 3 个寄存器：一个状态（status）寄存器，可以读取并查看设备的当前状态；一个命令（command）寄存器，用于通知设备执行某个具体任务；一个数据（data）寄存器，将数据传给设备或从设备接收数据。通过读写这些寄存器，操作系统可以控制设备的行为。

我们现在来描述操作系统与该设备的典型交互，以便让设备为它做某事。协议如下：

```
While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    (Doing so starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request
```

该协议包含 4 步。第 1 步，操作系统通过反复读取状态寄存器，等待设备进入可以接收命令的就绪状态。我们称之为轮询（polling）设备（基本上，就是问它正在做什么）。第 2 步，操作系统下发数据到数据寄存器。例如，你可以想象如果这是一个磁盘，需要多次写入操作，将一个磁盘块（比如 4KB）传递给设备。如果主 CPU 参与数据移动（就像这个示例协议一样），我们就称之为编程的 I/O（programmed I/O，PIO）。第 3 步，操作系统将命令

写入命令寄存器；这样设备就知道数据已经准备好了，它应该开始执行命令。最后一步，操作系统再次通过不断轮询设备，等待并判断设备是否执行完成命令（有可能得到一个指示成功或失败的错误码）。

这个简单的协议好处是足够简单并且有效。但是难免会有一些低效和不方便。我们注意到这个协议存在的第一个问题就是轮询过程比较低效，在等待设备执行完成命令时浪费大量 CPU 时间，如果此时操作系统可以切换执行下一个就绪进程，就可以大大提高 CPU 的利用率。

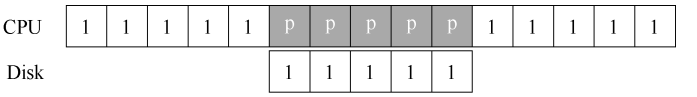
关键问题：如何减少轮询开销

操作系统检查设备状态时如何避免频繁轮询，从而降低管理设备的 CPU 开销？

36.4 利用中断减少 CPU 开销

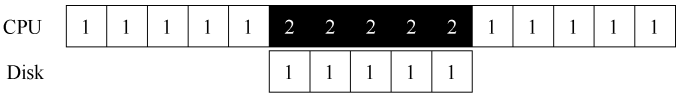
多年前，工程师们发明了我们目前已经很常见的中断（interrupt）来减少 CPU 开销。有了中断后，CPU 不再需要不断轮询设备，而是向设备发出一个请求，然后就可以让对应进程睡眠，切换执行其他任务。当设备完成了自身操作，会抛出一个硬件中断，引发 CPU 跳转执行操作系统预先定义好的中断服务例程（Interrupt Service Routine，ISR），或更为简单的中断处理程序（interrupt handler）。中断处理程序是一小段操作系统代码，它会结束之前的请求（比如从设备读取到了数据或者错误码）并且唤醒等待 I/O 的进程继续执行。

因此，中断允许计算与 I/O 重叠（overlap），这是提高 CPU 利用率的关键。下面的时间线展示了这一点：



其中，进程 1 在 CPU 上运行一段时间（对应 CPU 那一行上重复的 1），然后发出一个读取数据的 I/O 请求给磁盘。如果没有中断，那么操作系统就会简单自旋，不断轮询设备状态，直到设备完成 I/O 操作（对应其中的 p）。当设备完成请求的操作后，进程 1 又可以继续运行。

如果我们利用中断并允许重叠，操作系统就可以在等待磁盘操作时做其他事情：



在这个例子中，在磁盘处理进程 1 的请求时，操作系统在 CPU 上运行进程 2。磁盘处理完成后，触发一个中断，然后操作系统唤醒进程 1 继续运行。这样，在这段时间，无论 CPU 还是磁盘都可以有效地利用。

注意，使用中断并非总是最佳方案。假如有一个非常高性能的设备，它处理请求很快：通常在 CPU 第一次轮询时就可以返回结果。此时如果使用中断，反而会使系统变慢：切换到

其他进程，处理中断，再切换回之前的进程代价不小。因此，如果设备非常快，那么最好的办法反而是轮询。如果设备比较慢，那么采用允许发生重叠的中断更好。如果设备的速度未知，或者时快时慢，可以考虑使用混合（hybrid）策略，先尝试轮询一小段时间，如果设备没有完成操作，此时再使用中断。这种两阶段（two-phased）的办法可以实现两种方法的好处。

提示：中断并非总是比 PIO 好

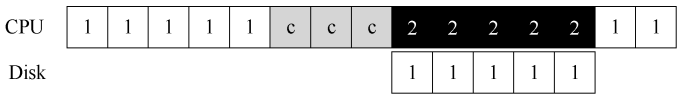
尽管中断可以做到计算与 I/O 的重叠，但这仅在慢速设备上有意义。否则，额外的中断处理和上下文切换的代价反而会超过其收益。另外，如果短时间内出现大量的中断，可能会使得系统过载并且引发活锁[MR96]。这种情况下，轮询的方式可以在操作系统自身的调度上提供更多的控制，反而更有效。

另一个最好不要使用中断的场景是网络。网络端收到大量数据包，如果每一个包都发生一次中断，那么有可能导致操作系统发生活锁（livelock），即不断处理中断而无法处理用户层的请求。例如，假设一个 Web 服务器因为“点杠效应”而突然承受很重的负载。这种情况下，偶尔使用轮询的方式可以更好地控制系统的行为，并允许 Web 服务器先服务一些用户请求，再回去检查网卡设备是否有更多数据包到达。

另一个基于中断的优化就是合并（coalescing）。设备在抛出中断之前往往会等待一小段时间，在此期间，其他请求可能很快完成，因此多次中断可以合并为一次中断抛出，从而降低处理中断的代价。当然，等待太长会增加请求的延迟，这是系统中常见的折中。参见 Ahmad 等人的文章[A+11]，有精彩的总结。

36.5 利用 DMA 进行更高效的数据传送

标准协议还有一点需要我们注意。具体来说，如果使用编程的 I/O 将一大块数据传给设备，CPU 又会因为琐碎的任务而变得负载很重，浪费了时间和算力，本来更好是用于运行其他进程。下面的时间线展示了这个问题：



进程 1 在运行过程中需要向磁盘写一些数据，所以它开始进行 I/O 操作，将数据从内存拷贝到磁盘（其中标示 c 的过程）。拷贝结束后，磁盘上的 I/O 操作开始执行，此时 CPU 才可以处理其他请求。

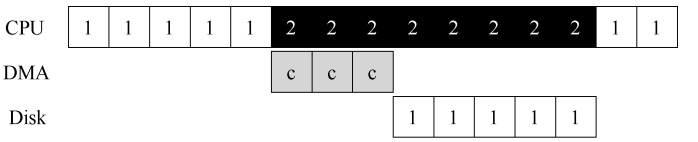
关键问题：如何减少 PIO 的开销

使用 PIO 的方式，CPU 的时间会浪费在向设备传输数据或从设备传出数据的过程中。如何才能分离这项工作，从而提高 CPU 的利用率？

解决方案就是使用 DMA（Direct Memory Access）。DMA 引擎是系统中的一个特殊设备，它可以协调完成内存和设备间的数据传递，不需要 CPU 介入。

DMA 工作过程如下。为了能够将数据传送给设备，操作系统会通过编程告诉 DMA 引

擎数据在内存的位置，要拷贝的大小以及要拷贝到哪个设备。在此之后，操作系统就可以处理其他请求了。当 DMA 的任务完成后，DMA 控制器会抛出一个中断来告诉操作系统自己已经完成数据传输。修改后的时间线如下：



从时间线中可以看到，数据的拷贝工作都是由 DMA 控制器来完成的。因为 CPU 在此时是空闲的，所以操作系统可以让它做一些其他事情，比如此处调度进程 2 到 CPU 来运行。因此进程 2 在进程 1 再次运行之前可以使用更多的 CPU。

36.6 设备交互的方法

现在，我们了解了执行 I/O 涉及的效率问题后，还有其他一些问题需要解决，以便将设备合并到系统中。你可能已经注意到了一个问题：我们还没有真正讨论过操作系统究竟如何与设备进行通信！所以问题如下。

关键问题：如何与设备通信

硬件如何如与设备通信？是否需要一些明确的指令？或者其他的方式？

随着技术的不断发展，主要有两种方式来实现与设备的交互。第一种办法相对老一些（在 IBM 主机中使用了多年），就是用明确的 I/O 指令。这些指令规定了操作系统将数据发送到特定设备寄存器的方法，从而允许构造上文提到的协议。

例如在 x86 上，in 和 out 指令可以用来与设备进行交互。当需要发送数据给设备时，调用者指定一个存入数据的特定寄存器及一个代表设备的特定端口。执行这个指令就可以实现期望的行为。

这些指令通常是特权指令（privileged）。操作系统是唯一可以直接与设备交互的实体。例如，设想如果任意程序都可以直接读写磁盘：完全混乱（总是会这样），因为任何用户程序都可以利用这个漏洞来取得计算机的全部控制权。

第二种方法是内存映射 I/O（memory- mapped I/O）。通过这种方式，硬件将设备寄存器作为内存地址提供。当需要访问设备寄存器时，操作系统装载（读取）或者存入（写入）到该内存地址；然后硬件会将装载/存入转移到设备上，而不是物理内存。

两种方法没有一种具备极大的优势。内存映射 I/O 的好处是不需要引入新指令来实现设备交互，但两种方法今天都在使用。

36.7 纳入操作系统：设备驱动程序

最后我们要讨论一个问题：每个设备都有非常具体的接口，如何将它们纳入操作系统，

而我们希望操作系统尽可能通用。例如文件系统，我们希望开发一个文件系统可以工作在 SCSI 硬盘、IDE 硬盘、USB 钥匙串设备等设备之上，并且希望这个文件系统不那么清楚对这些不同设备发出读写请求的全部细节。因此，我们的问题如下。

关键问题：如何实现一个设备无关的操作系统

如何保持操作系统的大部分与设备无关，从而对操作系统的主要子系统隐藏设备交互的细节？

这个问题可以通过古老的抽象（abstraction）技术来解决。在最底层，操作系统的一部分软件清楚地知道设备如何工作，我们将这部分软件称为设备驱动程序（device driver），所有设备交互的细节都封装在其中。

我们来看看 Linux 文件系统栈，理解抽象技术如何应用于操作系统的设计和实现。图 36.3 粗略地展示了 Linux 软件的组织方式。可以看出，文件系统（当然也包括在其之上的应用程序）完全不清楚它使用的是什么类型的磁盘。它只需要简单地向通用块设备层发送读写请求即可，块设备层会将这些请求路由给对应的设备驱动，然后设备驱动来完成真正的底层操作。尽管比较简单，但图 36.3 展示了这些细节如何对操作系统的大部分进行隐藏。

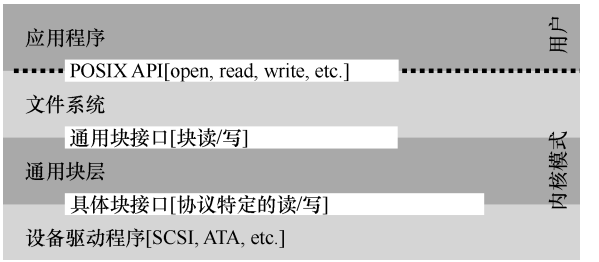


图 36.3 文件系统栈

注意，这种封装也有不足的地方。例如，如果有一个设备可以提供很多特殊的功能，但为了兼容大多数操作系统它不得不提供一个通用的接口，这样就使得自身的特殊功能无法使用。这种情况在使用 SCSI 设备的 Linux 中就发生了。SCSI 设备提供非常丰富的报告错误信息，但其他的块设备（比如 ATA/IDE）只提供非常简单的报错处理，这样上层的所有软件只能在出错时收到一个通用的 EIO 错误码（一般 IO 错误），SCSI 可能提供的所有附加信息都不能报告给文件系统[G08]。

有趣的是，因为所有需要插入系统的设备都需要安装对应的驱动程序，所以久而久之，驱动程序的代码在整个内核代码中的占比越来越大。查看 Linux 内核代码会发现，超过 70%的代码都是各种驱动程序。在 Windows 系统中，这样的比例同样很高。因此，如果有人跟你说操作系统包含上百万行代码，实际的意思是包含上百万行驱动程序代码。当然，任何安装进操作系统的驱动程序，大部分默认都不是激活状态（只有一小部分设备是在系统刚开启时就需要连接）。更加令人沮丧的是，因为驱动程序的开发者大部分是“业余的”（不是全职内核开发者），所以他们更容易写出缺陷，因此是内核崩溃的主要贡献者[S03]。

36.8 案例研究：简单的 IDE 磁盘驱动程序

为了更深入地了解设备驱动，我们快速看看一个真实的设备——IDE 磁盘驱动程序[L94]。我们总结了协议，如参考文献[W10]所述。我们也会看看 xv6 源码中一个简单的、能工作的 IDE 驱动程序实现。

IDE 硬盘暴露给操作系统的接口比较简单，包含 4 种类型的寄存器，即控制、命令块、状态和错误。在 x86 上，利用 I/O 指令 in 和 out 向特定的 I/O 地址（如下面的 0x3F6）读取或写入时，可以访问这些寄存器，如图 36.4 所示。

```
Control Register:
  Address 0x3F6 = 0x80 (0000 1RE0): R=reset, E=0 means "enable interrupt"


Command Block Registers:
  Address 0x1F0 = Data Port
  Address 0x1F1 = Error
  Address 0x1F2 = Sector Count
  Address 0x1F3 = LBA low byte
  Address 0x1F4 = LBA mid byte
  Address 0x1F5 = LBA hi  byte
  Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive
  Address 0x1F7 = Command/status


Status Register (Address 0x1F7):
    7      6    5     4    3    2    1    0
   BUSY  READY FAULT SEEK  DRQ  CORR IDDEX ERROR


Error Register (Address 0x1F1): (check when Status ERROR==1)
    7      6    5      4    3    2    1    0
   BBK    UNC    MC    IDNF  MCR  ABRT T0NF AMNF

   BBK  = Bad Block
   UNC  = Uncorrectable data error
   MC    = Media Changed
   IDNF = ID mark Not Found
   MCR  = Media Change Requested
   ABRT = Command aborted
   T0NF = Track 0 Not Found
   AMNF = Address Mark Not Found
```

图 36.4 IDE 接口

下面是与设备交互的简单协议，假设它已经初始化了，如图 36.5 所示。

- **等待驱动就绪**。读取状态寄存器（0x1F7）直到驱动 READY 而非忙碌。
- **向命令寄存器写入参数**。写入扇区数，待访问扇区对应的逻辑块地址（LBA），并将驱动编号（master=0x00，slave=0x10，因为 IDE 允许接入两个硬盘）写入命

令寄存器（0x1F2-0x1F6）。

- **开启 I/O**。发送读写命令到命令寄存器。向命令寄存器（0x1F7）中写入 READ-WRITE 命令。
- **数据传送（针对写请求）**：等待直到驱动状态为 READY 和 DRQ（驱动请求数据），向数据端口写入数据。
- **中断处理**。在最简单的情况下，每个扇区的数据传送结束后都会触发一次中断处理程序。较复杂的方式支持批处理，全部数据传送结束后才会触发一次中断处理。
- **错误处理**。在每次操作之后读取状态寄存器。如果 ERROR 位被置位，可以读取错误寄存器来获取详细信息。

```
static int ide_wait_ready() {
  while (((int r = inb(0x1f7)) & IDE_BSY) || !(r & IDE_DRDY))
    ;                               // loop until drive isn't busy
}

static void ide_start_request(struct buf *b) {
  ide_wait_ready();
  outb(0x3f6, 0);                   // generate interrupt
  outb(0x1f2, 1);                   // how many sectors?
  outb(0x1f3, b->sector & 0xff);    // LBA goes here ...
  outb(0x1f4, (b->sector >> 8) & 0xff);    // ... and here
  outb(0x1f5, (b->sector >> 16) & 0xff);    // ... and here!
  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
  if(b->flags & B_DIRTY){
    outb(0x1f7, IDE_CMD_WRITE);     // this is a WRITE
    outsl(0x1f0, b->data, 512/4);   // transfer data too!
  } else {
    outb(0x1f7, IDE_CMD_READ);      // this is a READ (no data)
  }
}

void ide_rw(struct buf *b) {
  acquire(&ide_lock);
  for (struct buf **pp = &ide_queue; *pp; pp=&(*pp)->qnext)
    ;                               // walk queue
  *pp = b;                          // add request to end
  if (ide_queue == b)               // if q is empty
    ide_start_request(b);           // send req to disk
  while ((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
    sleep(b, &ide_lock);            // wait for completion
  release(&ide_lock);
}

void ide_intr() {
  struct buf *b;
  acquire(&ide_lock);
  if (!(b->flags & B_DIRTY) && ide_wait_ready() >= 0)
    insl(0x1f0, b->data, 512/4);   // if READ: get data
```

```
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);                          // wake waiting process
  if ((ide_queue = b->qnext) != 0) // start next request
    ide_start_request(ide_queue);  // (if one exists)
  release(&ide_lock);
}
```

图 36.5 xv6 的 IDE 硬盘驱动程序（简化的）

该协议的大部分可以在 xv6 的 IDE 驱动程序中看到，它（在初始化后）通过 4 个主要函数来实现。第一个是 ide_rw()，它会将一个请求加入队列（如果前面还有请求未处理完成），或者直接将请求发送到磁盘（通过 ide_start_request()）。不论哪种情况，调用进程进入睡眠状态，等待请求处理完成。第二个是 ide_start_request()，它会将请求发送到磁盘（在写请求时，可能是发送数据）。此时 x86 的 in 或 out 指令会被调用，以读取或写入设备寄存器。在发起请求之前，开始请求函数会使用第三个函数 ide_wait_ready()，来确保驱动处于就绪状态。最后，当发生中断时，ide_intr()会被调用。它会从设备中读取数据（如果是读请求），并且在结束后唤醒等待的进程，如果此时在队列中还有别的未处理的请求，则调用 ide_start_request()接着处理下一个 I/O 请求。

36.9 历史记录

在结束之前，我们简述一下这些基本思想的由来。如果你想了解更多内容，可以阅读 Smotherman 的出色总结[S08]。

中断的思想很古老，存在于最早的机器之中。例如，20 世纪 50 年代的 UNIVAC 上就有某种形式的中断向量，虽然无法确定具体是哪一年出现的[S08]。遗憾的是，即使现在还是计算机诞生的初期，我们就开始丢失了起缘的历史记录。

关于什么机器第一个使用 DMA 技术也有争论。Knuth 和一些人认为是 DYSEAC（一种“移动”计算机，当时意味着可以用拖车运输它），而另外一些人则认为是 IBM SAGE[S08]。无论如何，在 20 世纪 50 年代中期，就有系统的 I/O 设备可以直接和内存交互，并在完成后中断 CPU。

这段历史比较难追溯，因为相关发明都与真实的、有时不太出名的机器联系在一起。例如，有些人认为 Lincoln Labs TX-2 是第一个拥有向量中断的机器[S08]，但这无法确定。

因为这些技术思想相对明显（在等待缓慢的 I/O 操作时让 CPU 去做其他事情，这种想法不需要爱因斯坦式的飞跃），也许我们关注“谁第一”是误入歧途。肯定明确的是：在人们构建早期的机器系统时，I/O 支持是必需的。中断、DMA 及相关思想都是在快速 CPU 和慢速设备之间权衡的结果。如果你处于那个时代，可能也会有同样的想法。

36.10 小结

至此你应该对操作系统如何与设备交互有了非常基本的理解。本章介绍了两种技术，

中断和 DMA，用于提高设备效率。我们还介绍了访问设备寄存器的两种方式，I/O 指令和内存映射 I/O。最后，我们介绍了设备驱动程序的概念，展示了操作系统本身如何封装底层细节，从而更容易以设备无关的方式构建操作系统的其余部分。

参考资料

[A+11]“vIC: Interrupt Coalescing for Virtual Machine Storage Device IO”Irfan Ahmad, Ajay Gulati, Ali Mashtizadeh
USENIX ’11
对传统和虚拟化环境中的中断合并进行了极好的调查。

[C01]“An Empirical Study of Operating System Errors”
Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, Dawson Engler SOSP ’01
首批系统地研究现代操作系统中有多少错误的文章之一。除了其他漂亮的研究结果之外，作者展示了设备驱动程序的 bug 数是主线内核代码中的 bug 数的 7 倍。

[CK+08]“The xv6 Operating System”
Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich.
请参阅 ide.c 中的 IDE 设备驱动程序，其中有更多细节。

[D07]“What Every Programmer Should Know About Memory”Ulrich Drepper
November, 2007
关于现代内存系统的极好的阅读材料，从 DRAM 开始，一直到虚拟化和缓存优化算法。

[G08]“EIO: Error-handling is Occasionally Correct”
Haryadi Gunawi, Cindy Rubio-Gonzalez, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Ben Liblit
FAST ’08, San Jose, CA, February 2008
我们自己的工作，构建一个工具来查找 Linux 文件系统中没有正确处理错误返回的代码。 我们发现了成百上千个错误，其中很多错误现在已经修复。

[L94]“AT Attachment Interface for Disk Drives”Lawrence J. Lamers, X3T10 Technical Editor
Reference number: ANSI X3.221 - 1994
关于设备接口的相当枯燥的文档。你可以尝试读一下。

[MR96]“Eliminating Receive Livelock in an Interrupt-driven Kernel”Jeffrey Mogul and K. K. Ramakrishnan
USENIX ’96, San Diego, CA, January 1996
Mogul 和同事在 Web 服务器网络性能方面做了大量的开创性工作。这篇论文只是其中之一。

[S08]“Interrupts”
Mark Smotherman, as of July ’08

关于中断历史、DMA 以及早期相关计算思想的宝库。

[S03]“Improving the Reliability of Commodity Operating Systems”Michael M. Swift, Brian N. Bershad, and Henry M. Levy
SOSP ’03
Swift 的工作重新燃起了对操作系统更像微内核方法的兴趣。至少，它终于给出了一些很好的理由，说明基于地址空间的保护在现代操作系统中可能有用。

[W10]“Hard Disk Driver”Washington State Course Homepage
很好地总结了一个简单的 IDE 磁盘驱动器接口，介绍了如何为它建立一个设备驱动程序。