第五章：中断和设备驱动

驱动是操作系统中管理特定设备的代码，它有如下功能：1、配置设备相关的硬件，2、告诉设备需要怎样执行，3、处理设备产生的中断，4、与等待设备I/O的进程进行交互。驱动程序的代码写起来可能很棘手，因为驱动程序与它所管理的设备会并发。此外，驱动必须了解设备的硬件接口，但硬件接口可能是很复杂的，而且文档不够完善。

需要操作系统关注的设备通常可以被配置为产生中断，这是trap的一种类型。内核trap处理代码可以知道设备何时引发了中断，并调用驱动的中断处理程序；在xv6中，这个处理发生在**devintr**(kernel/trap.c:177)中。

许多设备驱动程序在两个context中执行代码：上半部分（***top half***）在进程的内核线程中运行，下半部分（***bottom half***）在中断时执行。上半部分是通过系统调用，如希望执行I/O的read和write。这段代码可能会要求硬件开始一个操作（比如要求磁盘读取一个块）；然后代码等待操作完成。最终设备完成操作并引发一个中断。驱动程序的中断处理程序，作为**下半部分**，找出什么操作已经完成，如果合适的话，唤醒一个等待该操作的进程，并告诉硬件执行下一个操作。

5.1 Code: Console input

控制台驱动(console.c)是驱动结构的一个简单说明。控制台驱动通过连接到RISC-V上的UART串行端口硬件，接受输入的字符。控制台驱动程序每次累计一行输入，处理特殊的输入字符，如退格键和control-u。用户进程，如shell，使用**read**系统调用从控制台获取输入行。当你在QEMU中向xv6输入时，你的按键会通过QEMU的模拟UART硬件传递给xv6。

与驱动交互的UART硬件是由QEMU仿真的16550芯片[11]。在真实的计算机上，16550将管理一个连接到终端或其他计算机的RS232串行链接。当运行QEMU时，它连接到你的键盘和显示器上。

UART硬件在软件看来是一组**内存映射**的控制寄存器。也就是说，有一些RISC-V硬件的物理内存地址会关联到UART设备，因此加载和存储与设备硬件而不是RAM交互。UART的内存映射地址从0x10000000开始，即**UART0**（kernel/memlayout.h:21）。这里有一些UART控制寄存器，每个寄存器的宽度是一个字节。它们与UART0的偏移量定义在(kernel/uart.c:22)。例如，**LSR**寄存器中一些位表示是否有输入字符在等待软件读取。这些字符（如果有的话）可以从**RHR**寄存器中读取。每次读取一个字符，UART硬件就会将其从内部等待字符的FIFO中删除，并在FIFO为空时清除**LSR**中的就绪位。UART传输硬件在很大程度上是独立于接收硬件

的，如果软件向**THR**写入一个字节，UART就会发送该字节。

Xv6的**main**调用**consoleinit**（kernel/console.c:184）来初始化UART硬件。这段代码配置了UART，当UART接收到一个字节的输入时，就产生一个接收中断，当UART每次完成发送一个字节的输出时，产生一个***传输完成(transmit complete)***中断(kernel/uart.c:53)。

xv6 shell通过**init.c**(user/init.c:19)打开的文件描述符从控制台读取。对**read**的系统调用通过内核到达**consoleread**（kernel/console.c:82） 。**consoleread**等待输入的到来(通过中断)，输入会被缓冲在**cons.buf**，然后将输入复制到用户空间，再然后(在一整行到达后)返回到用户进程。如果用户还没有输入完整的行，任何调用了**read**进程将在**sleep**中等待(kernel/console.c:98)(第7章解释了sleep的细节)。

当用户键入一个字符时，UART硬件向RISC-V抛出一个中断，从而激活xv6的**trap**处理程序。trap处理程序调用devintr(kernel/trap.c:177)，它查看RISC-V的**scause**寄存器，发现中断来自一个外部设备。然后它向一个叫做PLIC[1]的硬件单元询问哪个设备中断了(kernel/trap.c:186)。如果是UART，**devintr**调用**uartintr**。

uartintr (kernel/uart.c:180) 从**UART**硬件中读取在等待的输入字符，并将它们交给**consoleintr** (kernel/console.c:138)；它不会等待输入字符，因为以后的输入会引发一个新的中断。**consoleintr**的工作是将中输入字符积累**cons.buf**中，直到有一行字符。 **consoleintr**会特别处理退格键和其他一些字符。当一个新行到达时，**consoleintr**会唤醒一个等待的**consoleread**（如果有的话）。

一旦被唤醒，**consoleread**将会注意到**cons.buf**中的完整行，并将其将其复制到用户空间，并返回（通过系统调用机制）到用户空间。

5.2 Code: Console output

向控制台写数据的**write**系统调用最终会到达**uartputc**(kernel/uart.c:87)。设备驱动维护了一个输出缓冲区(**uart_tx_buf**)，这样写进程就不需要等待UART完成发送；相反，**uartputc**将每个字符追加到缓冲区，调用**uartstart**来启动设备发送(如果还没有的话)，然后返回。**Uartputc**只有在缓冲区满的时候才会等待。

每次UART发送完成一个字节，它都会产生一个中断。**uartintr**调用**uartstart**，**uartintr**检查设备是否真的发送完毕，并将下一个缓冲输出字符交给设备，每当UART发送完一个字节，就会产生一个中断。因此，如果一个进程向控制台写入多个字节，通常第一个字节将由uartputcs调用uartstart发送，其余的缓冲字节将由uartintr调用uartstart发送，因为发送完成中断到

来。

uartintr调用**uartstart**，**uartintr**查看设备是否真的发送完成，并将下一个缓冲输出字符交给设备，每当UART发送完一个字节，就会产生一个中断。因此，如果一个进程向控制台写入多个字节，通常第一个字节将由**uartputc**对**uartstart**的调用发送，其余的缓冲字节将随着发送完成中断的到来由**uartintr**的**uartstart**调用发送。

有一个通用模式需要注意，设备活动和进程活动需要解耦，这将通过缓冲和中断来实现。控制台驱动程序可以处理输入，即使没有进程等待读取它；随后的读取将看到输入。同样，进程可以发送输出字节，而不必等待设备。这种解耦可以通过允许进程与设备I/O并发执行来提高性能，当设备速度很慢（如UART）或需要立即关注（如回显键入的字节）时，这种解耦尤为重要。这个idea有时被称为***I/O并发***。

5.3 Concurrency in drivers

你可能已经注意到在**consoleread**和**consoleintr**中会调用**acquire**。**acquire**调用会获取一个锁，保护控制台驱动的数据结构不被并发访问。这里有三个并发风险：不同CPU上的两个进程可能会同时调用**consoleread**；硬件可能会在一个CPU正在执行consoleread时，向该CPU抛出一个控制台（实际上是UART）中断；硬件可能会在consoleread执行时向另一个CPU抛出一个控制台中断。第6章探讨锁如何在这些情况下提供帮助。

需要关注驱动并发安全的另一个原因是，一个进程可能正在等待来自设备的输入，但是当表明输入到来的中断发生时该进程已经没有在运行（被切换）。因此，中断处理程序不允许知道被中断的进程或代码。例如，一个中断处理程序不能安全地用当前进程的页表调用**copyout**。中断处理程序通常只做相对较少的工作（例如，只是将输入数据复制到缓冲区），并唤醒***上半部分***代码来做剩下的工作。

5.4 Timer interrupts

Xv6使用定时器中断来维护它的时钟，并使它能够切换计算密集型进程；**usertrap**和**kerneltrap**中的**yield**调用会导致这种切换。每个RISC-V CPU的时钟硬件都会抛出时钟中断。Xv6对这个时钟硬件进行编程，使其定期周期性地中断相应的CPU。

RISC-V要求在机器模式下处理定时器中断，而不是监督者模式。RISCV机器模式执行时没有分页，并且有一套单独的控制寄存器，因此在机器模式下运行普通的xv6内核代码是不实用的。因此，xv6对定时器中断的处理与上面谈到的trap机制完全分离了。

在main执行之前的**start.c**，是在机器模式下执行的。它设置了接收定时器中断(kernel/start.c:57)。一部分工作是对**CLINT**硬件（**core-local interruptor**）进行编程，使其每隔一定时间产生一次中断。另一部分是设置一个类似于**trapframe**的暂存区，帮助定时器中断处理程序保存寄存器和**CLINT**寄存器的地址。最后，**start**将**mtvec**设置为**timervec**，启用定时器中断。

定时器中断可能发生在用户或内核代码执行的任何时候；内核没有办法在关键操作中禁用定时器中断。因此，定时器中断处理程序必须以保证不干扰被中断的内核代码的方式进行工作。基本策略是处理程序要求RISC-V引发一个软件中断并立即返回。RISC-V用普通的trap机制将软件中断传递给内核，并允许内核禁用它们。处理定时器中断产生的软件中断的代码可以在**devintr**（kernel/trap.c:204）中看到。

机器模式的定时器中断向量是**timervec**(kernel/kernelvec.S:93)。它在**start**准备的暂存区保存一些寄存器，告诉**CLINT**何时产生下一个定时器中断，使RISC-V产生一个软件中断，恢复寄存器，然后返回。在定时器中断处理程序中没有C代码。

5.5 Real world

Xv6允许在内核和用户程序执行时使用设备和定时器中断。定时器中断可以强制从定时器中断处理程序进行线程切换（调用**yield**），即使是在内核中执行。如果内核线程有时会花费大量的时间进行计算，而不返回用户空间，那么在内核线程之间公平地对CPU进行时间划分的能力是很有用的。然而，内核代码需要注意它可能会被暂停（由于定时器中断），然后在不同的CPU上恢复，这是xv6中一些复杂的根源。如果设备和定时器中断只发生在执行用户代码时，内核可以变得更简单一些。

在一台典型的计算机上支持所有设备的全貌是一件很辛苦的事情，因为设备很多，设备有很多功能，设备和驱动程序之间的协议可能很复杂，而且文档也不完善。在许多操作系统中，驱动程序所占的代码比核心内核还多。

UART驱动器通过读取UART控制寄存器，一次读取一个字节的数据；这种模式被称为编程I/O，因为软件在控制数据移动。程序化I/O简单，但速度太慢，无法在高数据速率下使用。需要高速移动大量数据的设备通常使用***直接内存访问(direct memory access, DMA)***。DMA设备硬件直接将传入数据写入RAM，并从RAM中读取传出数据。现代磁盘和网络设备都使用DMA。DMA设备的驱动程序会在RAM中准备数据，然后使用对控制寄存器的一次写入来告诉设备处理准备好的数据。

当设备在不可预知的时间需要关注,且不那么频繁时，中断是很有用的。但中断对CPU的开销

很大。因此，高速设备，如网络和磁盘控制器，使用了减少对中断需求的技巧。其中一个技巧是对整批传入或传出的请求提出一个单一的中断。另一个技巧是让驱动程序完全禁用中断，并定期检查设备是否需要关注。这种技术称为***轮询（polling）***。如果设备执行操作的速度非常快，轮询是有意义的，但如果设备大部分时间处于空闲状态，则会浪费CPU时间。一些驱动程序会根据当前设备的负载情况，在轮询和中断之间动态切换。

UART驱动首先将输入的数据复制到内核的缓冲区，然后再复制到用户空间。这在低数据速率下是有意义的，但对于那些快速生成或消耗数据的设备来说，这样的双重拷贝会大大降低性能。一些操作系统能够直接在用户空间缓冲区和设备硬件之间移动数据，通常使用DMA。

5.6 Exercises

1. 修改uart.c，使其完全不使用中断。你可能还需要修改 console.c。
2. 添加一个网卡驱动。