Chapter 5

Interrupts and device drivers

A *driver* is the code in an operating system that manages a particular device: it configures the device hardware, tells the device to perform operations, handles the resulting interrupts, and interacts with processes that may be waiting for I/O from the device. Driver code can be tricky because a driver executes concurrently with the device that it manages. In addition, the driver must understand the device's hardware interface, which can be complex and poorly documented.

Devices that need attention from the operating system can usually be configured to generate interrupts, which are one type of trap. The kernel trap handling code recognizes when a device has raised an interrupt and calls the driver's interrupt handler; in xv6, this dispatch happens in `devintr` (kernel/trap.c:177).

Many device drivers execute code in two contexts: a *top half* that runs in a process's kernel thread, and a *bottom half* that executes at interrupt time. The top half is called via system calls such as `read` and `write` that want the device to perform I/O. This code may ask the hardware to start an operation (e.g., ask the disk to read a block); then the code waits for the operation to complete. Eventually the device completes the operation and raises an interrupt. The driver's interrupt handler, acting as the bottom half, figures out what operation has completed, wakes up a waiting process if appropriate, and tells the hardware to start work on any waiting next operation.

5.1 Code: Console input

The console driver (kernel/console.c) is a simple illustration of driver structure. The console driver accepts characters typed by a human, via the *UART* serial-port hardware attached to the RISC-V. The console driver accumulates a line of input at a time, processing special input characters such as backspace and control-u. User processes, such as the shell, use the `read` system call to fetch lines of input from the console. When you type input to xv6 in QEMU, your keystrokes are delivered to xv6 by way of QEMU's simulated UART hardware.

The UART hardware that the driver talks to is a 16550 chip [12] emulated by QEMU. On a real computer, a 16550 would manage an RS232 serial link connecting to a terminal or other computer. When running QEMU, it's connected to your keyboard and display.

The UART hardware appears to software as a set of *memory-mapped* control registers. That

is, there are some physical addresses that RISC-V hardware connects to the UART device, so that loads and stores interact with the device hardware rather than RAM. The memory-mapped addresses for the UART start at 0x10000000, or `UART0` (kernel/memlayout.h:21). There are a handful of UART control registers, each the width of a byte. Their offsets from `UART0` are defined in (kernel/uart.c:22). For example, the `LSR` register contain bits that indicate whether input characters are waiting to be read by the software. These characters (if any) are available for reading from the `RHR` register. Each time one is read, the UART hardware deletes it from an internal FIFO of waiting characters, and clears the "ready" bit in `LSR` when the FIFO is empty. The UART transmit hardware is largely independent of the receive hardware; if software writes a byte to the `THR`, the UART transmit that byte.

Xv6's `main` calls `consoleinit` (kernel/console.c:182) to initialize the UART hardware. This code configures the UART to generate a receive interrupt when the UART receives each byte of input, and a *transmit complete* interrupt each time the UART finishes sending a byte of output (kernel/uart.c:53).

The xv6 shell reads from the console by way of a file descriptor opened by `init.c` (user/init.c:19). Calls to the `read` system call make their way through the kernel to `consoleread` (kernel/console.c:80). `consoleread` waits for input to arrive (via interrupts) and be buffered in `cons.buf`, copies the input to user space, and (after a whole line has arrived) returns to the user process. If the user hasn't typed a full line yet, any reading processes will wait in the `sleep` call (kernel/console.c:96) (Chapter 7 explains the details of `sleep`).

When the user types a character, the UART hardware asks the RISC-V to raise an interrupt, which activates xv6's trap handler. The trap handler calls `devintr` (kernel/trap.c:177), which looks at the RISC-V `scause` register to discover that the interrupt is from an external device. Then it asks a hardware unit called the PLIC [1] to tell it which device interrupted (kernel/trap.c:186). If it was the UART, `devintr` calls `uartintr`.

`uartintr` (kernel/uart.c:180) reads any waiting input characters from the UART hardware and hands them to `consoleintr` (kernel/console.c:136); it doesn't wait for characters, since future input will raise a new interrupt. The job of `consoleintr` is to accumulate input characters in `cons.buf` until a whole line arrives. `consoleintr` treats backspace and a few other characters specially. When a newline arrives, `consoleintr` wakes up a waiting `consoleread` (if there is one).

Once woken, `consoleread` will observe a full line in `cons.buf`, copy it to user space, and return (via the system call machinery) to user space.

5.2 Code: Console output

A `write` system call on a file descriptor connected to the console eventually arrives at `uartputc` (kernel/uart.c:87). The device driver maintains an output buffer (`uart_tx_buf`) so that writing processes do not have to wait for the UART to finish sending; instead, `uartputc` appends each character to the buffer, calls `uartstart` to start the device transmitting (if it isn't already), and returns. The only situation in which `uartputc` waits is if the buffer is already full.

Each time the UART finishes sending a byte, it generates an interrupt. `uartintr` calls `uartstart`,

which checks that the device really has finished sending, and hands the device the next buffered output character. Thus if a process writes multiple bytes to the console, typically the first byte will be sent by `uartputc`'s call to `uartstart`, and the remaining buffered bytes will be sent by `uartstart` calls from `uartintr` as transmit complete interrupts arrive.

A general pattern to note is the decoupling of device activity from process activity via buffering and interrupts. The console driver can process input even when no process is waiting to read it; a subsequent read will see the input. Similarly, processes can send output without having to wait for the device. This decoupling can increase performance by allowing processes to execute concurrently with device I/O, and is particularly important when the device is slow (as with the UART) or needs immediate attention (as with echoing typed characters). This idea is sometimes called *I/O concurrency*.

5.3 Concurrency in drivers

You may have noticed calls to `acquire` in consoleread and in `consoleintr`. These calls acquire a lock, which protects the console driver's data structures from concurrent access. There are three concurrency dangers here: two processes on different CPUs might call `consoleread` at the same time; the hardware might ask a CPU to deliver a console (really UART) interrupt while that CPU is already executing inside `consoleread`; and the hardware might deliver a console interrupt on a different CPU while `consoleread` is executing. These dangers may result in race conditions or deadlocks. Chapter 6 explores these problems and how locks can address them.

Another way in which concurrency requires care in drivers is that one process may be waiting for input from a device, but the interrupt signaling arrival of the input may arrive when a different process (or no process at all) is running. Thus interrupt handlers are not allowed to think about the process or code that they have interrupted. For example, an interrupt handler cannot safely call `copyout` with the current process's page table. Interrupt handlers typically do relatively little work (e.g., just copy the input data to a buffer), and wake up top-half code to do the rest.

5.4 Timer interrupts

Xv6 uses timer interrupts to maintain its clock and to enable it to switch among compute-bound processes; the `yield` calls in `usertrap` and `kerneltrap` cause this switching. Timer interrupts come from clock hardware attached to each RISC-V CPU. Xv6 programs this clock hardware to interrupt each CPU periodically.

RISC-V requires that timer interrupts be taken in machine mode, not supervisor mode. RISC-V machine mode executes without paging, and with a separate set of control registers, so it's not practical to run ordinary xv6 kernel code in machine mode. As a result, xv6 handles timer interrupts completely separately from the trap mechanism laid out above.

Code executed in machine mode in `start.c`, before `main`, sets up to receive timer interrupts (kernel/start.c:62). Part of the job is to program the CLINT hardware (core-local interruptor) to generate an interrupt after a certain delay. Another part is to set up a scratch area, analogous to the

trapframe, to help the timer interrupt handler save registers and the address of the CLINT registers. Finally, `start` sets `mtvec` to `timervec` and enables timer interrupts.

A timer interrupt can occur at any point when user or kernel code is executing; there's no way for the kernel to disable timer interrupts during critical operations. Thus the timer interrupt handler must do its job in a way guaranteed not to disturb interrupted kernel code. The basic strategy is for the handler to ask the RISC-V to raise a "software interrupt" and immediately return. The RISC-V delivers software interrupts to the kernel with the ordinary trap mechanism, and allows the kernel to disable them. The code to handle the software interrupt generated by a timer interrupt can be seen in `devintr` (kernel/trap.c:204).

The machine-mode timer interrupt handler is `timervec` (kernel/kernelvec.S:93). It saves a few registers in the scratch area prepared by `start`, tells the CLINT when to generate the next timer interrupt, asks the RISC-V to raise a software interrupt, restores registers, and returns. There's no C code in the timer interrupt handler.

5.5 Real world

Xv6 allows device and timer interrupts while executing in the kernel, as well as when executing user programs. Timer interrupts force a thread switch (a call to `yield`) from the timer interrupt handler, even when executing in the kernel. The ability to time-slice the CPU fairly among kernel threads is useful if kernel threads sometimes spend a lot of time computing, without returning to user space. However, the need for kernel code to be mindful that it might be suspended (due to a timer interrupt) and later resume on a different CPU is the source of some complexity in xv6 (see Section 6.6). The kernel could be made somewhat simpler if device and timer interrupts only occurred while executing user code.

Supporting all the devices on a typical computer in its full glory is much work, because there are many devices, the devices have many features, and the protocol between device and driver can be complex and poorly documented. In many operating systems, the drivers account for more code than the core kernel.

The UART driver retrieves data a byte at a time by reading the UART control registers; this pattern is called *programmed I/O*, since software is driving the data movement. Programmed I/O is simple, but too slow to be used at high data rates. Devices that need to move lots of data at high speed typically use *direct memory access (DMA)*. DMA device hardware directly writes incoming data to RAM, and reads outgoing data from RAM. Modern disk and network devices use DMA. A driver for a DMA device would prepare data in RAM, and then use a single write to a control register to tell the device to process the prepared data.

Interrupts make sense when a device needs attention at unpredictable times, and not too often. But interrupts have high CPU overhead. Thus high speed devices, such networks and disk controllers, use tricks that reduce the need for interrupts. One trick is to raise a single interrupt for a whole batch of incoming or outgoing requests. Another trick is for the driver to disable interrupts entirely, and to check the device periodically to see if it needs attention. This technique is called *polling*. Polling makes sense if the device performs operations very quickly, but it wastes CPU time if the device is mostly idle. Some drivers dynamically switch between polling and interrupts

depending on the current device load.

The UART driver copies incoming data first to a buffer in the kernel, and then to user space. This makes sense at low data rates, but such a double copy can significantly reduce performance for devices that generate or consume data very quickly. Some operating systems are able to directly move data between user-space buffers and device hardware, often with DMA.

As mentioned in Chapter 1, the console appears to applications as a regular file, and applications read input and write output using the `read` and `write` system calls. Applications may want to control aspects of a device that cannot be expressed through the standard file system calls (e.g., enabling/disabling line buffering in the console driver). Unix operating systems support the `ioctl` system call for such cases.

Some usages of computers require that the system must respond in a bounded time. For example, in safety-critical systems missing a deadline can lead to disasters. Xv6 is not suitable for hard real-time settings. Operating systems for hard real-time tend to be libraries that link with the application in a way that allows for an analysis to determine the worst-case response time. Xv6 is also not suitable for soft real-time applications, when missing a deadline occasionally is acceptable, because xv6's scheduler is too simplistic and it has kernel code path where interrupts are disabled for a long time.

5.6 Exercises

1. Modify `uart.c` to not use interrupts at all. You may need to modify `console.c` as well.

2. Add a driver for an Ethernet card.