exception
interrupt

Chapter 3

Traps, interrupts, and drivers

When running a process, a CPU executes the normal processor loop: read an instruction, advance the program counter, execute the instruction, repeat. But there are events on which control from a user program must transfer back to the kernel instead of executing the next instruction. These events include a device signaling that it wants attention, a user program doing something illegal (e.g., references a virtual address for which there is no PTE), or a user program asking the kernel for a service with a system call. There are three main challenges in handling these events: 1) the kernel must arrange that a processor switches from user mode to kernel mode (and back); 2) the kernel and devices must coordinate their parallel activities; and 3) the kernel must understand the interface of the devices. Addressing these 3 challenges requires detailed understanding of hardware and careful programming, and can result in opaque kernel code. This chapter explains how xv6 addresses these three challenges.

Systems calls, exceptions, and interrupts

There are three cases when control must be transferred from a user program to the kernel. First, a system call: when a user program asks for an operating system service, as we saw at the end of the last chapter. Second, an `exception`: when a program performs an illegal action. Examples of illegal actions include divide by zero, attempt to access memory for a PTE that is not present, and so on. Third, an `interrupt`: when a device generates a signal to indicate that it needs attention from the operating system. For example, a clock chip may generate an interrupt every 100 msec to allow the kernel to implement time sharing. As another example, when the disk has read a block from disk, it generates an interrupt to alert the operating system that the block is ready to be retrieved.

The kernel handles all interrupts, rather than processes handling them, because in most cases only the kernel has the required privilege and state. For example, in order to time-slice among processes in response the clock interrupts, the kernel must be involved, if only to force uncooperative processes to yield the processor.

In all three cases, the operating system design must arrange for the following to happen. The system must save the processor's registers for future transparent resume. The system must be set up for execution in the kernel. The system must chose a place for the kernel to start executing. The kernel must be able to retrieve information about the event, e.g., system call arguments. It must all be done securely; the system must maintain isolation of user processes and the kernel.

To achieve this goal the operating system must be aware of the details of how the hardware handles system calls, exceptions, and interrupts. In most processors these three events are handled by a single hardware mechanism. For example, on the x86, a program invokes a system call by generating an interrupt using the `int` instruction. Similarly, exceptions generate an interrupt too. Thus, if the operating system has a plan for interrupt handling, then the operating system can handle system calls and exceptions too.

int+code
interrupt handler
traps
kernel mode
user mode

The basic plan is as follows. An interrupts stops the normal processor loop and starts executing a new sequence called an `interrupt handler`. Before starting the interrupt handler, the processor saves its registers, so that the operating system can restore them when it returns from the interrupt. A challenge in the transition to and from the interrupt handler is that the processor should switch from user mode to kernel mode, and back.

A word on terminology: Although the official x86 term is interrupt, xv6 refers to all of these as `traps`, largely because it was the term used by the PDP11/40 and therefore is the conventional Unix term. This chapter uses the terms trap and interrupt interchangeably, but it is important to remember that traps are caused by the current process running on a processor (e.g., the process makes a system call and as a result generates a trap), and interrupts are caused by devices and may not be related to the currently running process. For example, a disk may generate an interrupt when it is done retrieving a block for one process, but at the time of the interrupt some other process may be running. This property of interrupts makes thinking about interrupts more difficult than thinking about traps, because interrupts happen concurrently with other activities. Both rely, however, on the same hardware mechanism to transfer control between user and kernel mode securely, which we will discuss next.

X86 protection

The x86 has 4 protection levels, numbered 0 (most privilege) to 3 (least privilege). In practice, most operating systems use only 2 levels: 0 and 3, which are then called `kernel mode` and `user mode`, respectively. The current privilege level with which the x86 executes instructions is stored in `%cs` register, in the field CPL.

On the x86, interrupt handlers are defined in the interrupt descriptor table (IDT). The IDT has 256 entries, each giving the `%cs` and `%eip` to be used when handling the corresponding interrupt.

To make a system call on the x86, a program invokes the `int` *n* instruction, where *n* specifies the index into the IDT. The `int` instruction performs the following steps:

- Fetch the *n*'th descriptor from the IDT, where *n* is the argument of `int`.
- Check that CPL in `%cs` is <= DPL, where DPL is the privilege level in the descriptor.
- Save `%esp` and `%ss` in CPU-internal registers, but only if the target segment selector's PL < CPL.
- Load `%ss` and `%esp` from a task segment descriptor.
- Push `%ss`.

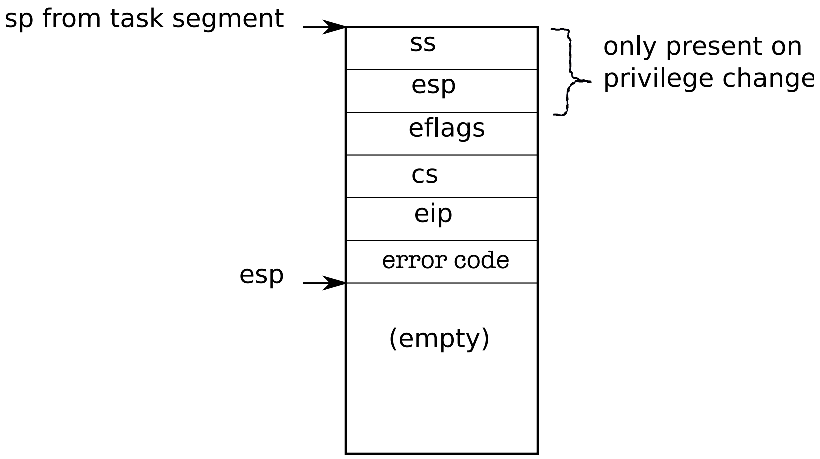

Figure 3-1. Kernel stack after an int instruction.

- Push %esp.
- Push %eflags.
- Push %cs.
- Push %eip.
- Clear some bits of %eflags.
- Set %cs and %eip to the values in the descriptor.

The int instruction is a complex instruction, and one might wonder whether all these actions are necessary. For example, the check CPL <= DPL allows the kernel to forbid int calls to inappropriate IDT entries such as device interrupt routines. For a user program to execute int, the IDT entry's DPL must be 3. If the user program doesn't have the appropriate privilege, then int will result in int 13, which is a general protection fault. As another example, the int instruction cannot use the user stack to save values, because the process may not have a valid stack pointer; instead, the hardware uses the stack specified in the task segment, which is set by the kernel.

int+code
iret+code
int+code
initcode.S+code

Figure 3-1 shows the stack after an int instruction completes and there was a privilege-level change (the privilege level in the descriptor is lower than CPL). If the int instruction didn't require a privilege-level change, the x86 won't save %ss and %esp. After both cases, %eip is pointing to the address specified in the descriptor table, and the instruction at that address is the next instruction to be executed and the first instruction of the handler for int *n*. It is job of the operating system to implement these handlers, and below we will see what xv6 does.

An operating system can use the iret instruction to return from an int instruction. It pops the saved values during the int instruction from the stack, and resumes execution at the saved %eip.

Code: The first system call

Chapter 1 ended with initcode.S invoking a system call. Let's look at that again

(8414). The process pushed the arguments for an exec call on the process's stack, and put the system call number in %eax. The system call numbers match the entries in the syscalls array, a table of function pointers (3600). We need to arrange that the int instruction switches the processor from user mode to kernel mode, that the kernel invokes the right kernel function (i.e., sys_exec), and that the kernel can retrieve the arguments for sys_exec. The next few subsections describe how xv6 arranges this for system calls, and then we will discover that we can reuse the same code for interrupts and exceptions.

exec+code
sys_exec+code
int+code
tvinit
main+code
idt+code
vectors[i]+code
T_SYSCALL+code
IF+code
DPL_USER+code
switchuvm+code
alltraps+code

Code: Assembly trap handlers

Xv6 must set up the x86 hardware to do something sensible on encountering an int instruction, which causes the processor to generate a trap. The x86 allows for 256 different interrupts. Interrupts 0-31 are defined for software exceptions, like divide errors or attempts to access invalid memory addresses. Xv6 maps the 32 hardware interrupts to the range 32-63 and uses interrupt 64 as the system call interrupt.

Tvinit (3317), called from main, sets up the 256 entries in the table idt. Interrupt i is handled by the code at the address in vectors[i]. Each entry point is different, because the x86 does not provide the trap number to the interrupt handler. Using 256 different handlers is the only way to distinguish the 256 cases.

Tvinit handles T_SYSCALL, the user system call trap, specially: it specifies that the gate is of type "trap" by passing a value of 1 as second argument. Trap gates don't clear the IF flag, allowing other interrupts during the system call handler.

The kernel also sets the system call gate privilege to DPL_USER, which allows a user program to generate the trap with an explicit int instruction. xv6 doesn't allow processes to raise other interrupts (e.g., device interrupts) with int; if they try, they will encounter a general protection exception, which goes to vector 13.

When changing protection levels from user to kernel mode, the kernel shouldn't use the stack of the user process, because it may not be valid. The user process may be malicious or contain an error that causes the user %esp to contain an address that is not part of the process's user memory. Xv6 programs the x86 hardware to perform a stack switch on a trap by setting up a task segment descriptor through which the hardware loads a stack segment selector and a new value for %esp. The function switchuvm (1873) stores the address of the top of the kernel stack of the user process into the task segment descriptor.

When a trap occurs, the processor hardware does the following. If the processor was executing in user mode, it loads %esp and %ss from the task segment descriptor, pushes the old user %ss and %esp onto the new stack. If the processor was executing in kernel mode, none of the above happens. The processor then pushes the %eflags, %cs, and %eip registers. For some traps, the processor also pushes an error word. The processor then loads %eip and %cs from the relevant IDT entry.

xv6 uses a Perl script (3200) to generate the entry points that the IDT entries point to. Each entry pushes an error code if the processor didn't, pushes the interrupt number, and then jumps to alltraps.

Alltraps (3254) continues to save processor registers: it pushes %ds, %es, %fs,

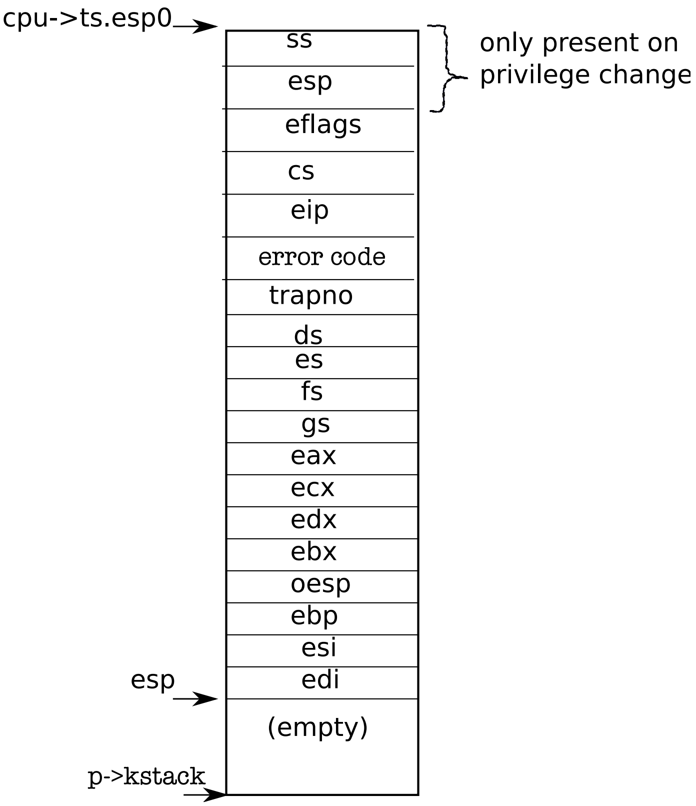

Figure 3-2. The trapframe on the kernel stack

%gs, and the general-purpose registers (3255-3260). The result of this effort is that the kernel stack now contains a struct trapframe (0602) containing the processor registers at the time of the trap (see Figure 3-2). The processor pushes %ss, %esp, %eflags, %cs, and %eip. The processor or the trap vector pushes an error number, and alltraps pushes the rest. The trap frame contains all the information necessary to restore the user mode processor registers when the kernel returns to the current process, so that the processor can continue exactly as it was when the trap started. Recall from Chapter 2, that userinit built a trapframe by hand to achieve this goal (see Figure 1-4).

alltraps+code
alltraps+code
SEG_KCPU+code
alltraps+code
trap+code
alltraps+code

In the case of the first system call, the saved %eip is the address of the instruction right after the int instruction. %cs is the user code segment selector. %eflags is the content of the eflags register at the point of executing the int instruction. As part of saving the general-purpose registers, alltraps also saves %eax, which contains the system call number for the kernel to inspect later.

Now that the user mode processor registers are saved, alltraps can finishing setting up the processor to run kernel C code. The processor set the selectors %cs and %ss before entering the handler; alltraps sets %ds and %es (3263-3265). It sets %fs and %gs to point at the SEG_KCPU per-CPU data segment (3266-3268).

Once the segments are set properly, alltraps can call the C trap handler trap. It pushes %esp, which points at the trap frame it just constructed, onto the stack as an argument to trap (3271). Then it calls trap (3272). After trap returns, alltraps pops the argument off the stack by adding to the stack pointer (3273) and then starts executing the code at label `trapret`. We traced through this code in Chapter 2 when the first user process ran it to exit to user space. The same sequence happens here: popping through the trap frame restores the user mode registers and then `iret` jumps back into user space.

The discussion so far has talked about traps occurring in user mode, but traps can also happen while the kernel is executing. In that case the hardware does not switch stacks or save the stack pointer or stack segment selector; otherwise the same steps occur as in traps from user mode, and the same xv6 trap handling code executes. When `iret` later restores a kernel mode `%cs,` the processor continues executing in kernel mode.

trapret+code
iret+code
trap+code
tf->trapno+code
T_SYSCALL+code
syscall+code
proc->killed+code
trap+code
panic+code
trap+code
syscall+code
SYS_exec+code
cp->tf+code
syscall+code

Code: C trap handler

We saw in the last section that each handler sets up a trap frame and then calls the C function `trap`. `Trap` (3351) looks at the hardware trap number `tf->trapno` to decide why it has been called and what needs to be done. If the trap is `T_SYSCALL`, `trap` calls the system call handler `syscall`. We'll revisit the `proc->killed` checks in Chapter 5.

After checking for a system call, trap looks for hardware interrupts (which we discuss below). In addition to the expected hardware devices, a trap can be caused by a spurious interrupt, an unwanted hardware interrupt.

If the trap is not a system call and not a hardware device looking for attention, `trap` assumes it was caused by incorrect behavior (e.g., divide by zero) as part of the code that was executing before the trap. If the code that caused the trap was a user program, xv6 prints details and then sets `proc->killed` to remember to clean up the user process. We will look at how xv6 does this cleanup in Chapter 5.

If it was the kernel running, there must be a kernel bug: `trap` prints details about the surprise and then calls `panic`.

Code: System calls

For system calls, `trap` invokes `syscall` (3625). `Syscall` loads the system call number from the trap frame, which contains the saved `%eax,` and indexes into the system call tables. For the first system call, `%eax` contains the value `SYS_exec` (3457), and `syscall` will invoke the `SYS_exec`'th entry of the system call table, which corresponds to invoking `sys_exec`.

`Syscall` records the return value of the system call function in `%eax.` When the trap returns to user space, it will load the values from `cp->tf` into the machine registers. Thus, when `exec` returns, it will return the value that the system call handler returned (3631). System calls conventionally return negative numbers to indicate errors, positive numbers for success. If the system call number is invalid, `syscall` prints an error and returns –1.

Later chapters will examine the implementation of particular system calls. This chapter is concerned with the mechanisms for system calls. There is one bit of mechanism left: finding the system call arguments. The helper functions argint, argptr, argstr, and argfd retrieve the *n*'th system call argument, as either an integer, pointer, a string, or a file descriptor. `argint` uses the user-space `%esp` register to locate the *n'th* argument: `%esp` points at the return address for the system call stub. The arguments are right above it, at `%esp+4`. Then the nth argument is at `%esp+4+4*n`.

argint+code
fetchint+code
p->sz+code
argptr+code
argstr+code
argfd+code

`argint` calls `fetchint` to read the value at that address from user memory and write it to `*ip`. `fetchint` can simply cast the address to a pointer, because the user and the kernel share the same page table, but the kernel must verify that the pointer lies within the user part of the address space. The kernel has set up the page-table hardware to make sure that the process cannot access memory outside its local private memory: if a user program tries to read or write memory at an address of `p->sz` or above, the processor will cause a segmentation trap, and trap will kill the process, as we saw above. The kernel, however, can derefence any address that the user might have passed, so it must check explicitly that the address is below `p->sz`.

`argptr` fetches the *n*th system call argument and checks that this argument is a valid user-space pointer. Note that two checks occur during a call to `argptr`. First, the user stack pointer is checked during the fetching of the argument. Then the argument, itself a user pointer, is checked.

`argstr` interprets the *n*th argument as a pointer. It ensures that the pointer points at a NUL-terminated string and that the complete string is located below the end of the user part of the address space.

Finally, `argfd` (5819) uses `argint` to retrieve a file descriptor number, checks if it is valid file descriptor, and returns the corresponding `struct file`.

The system call implementations (for example, sysproc.c and sysfile.c) are typically wrappers: they decode the arguments using `argint`, `argptr`, and `argstr` and then call the real implementations. In chapter 2, `sys_exec` uses these functions to get at its arguments.

Code: Interrupts

Devices on the motherboard can generate interrupts, and xv6 must set up the hardware to handle these interrupts. Devices usually interrupt in order to tell the kernel that some hardware event has occured, such as I/O completion. Interrupts are usually optional in the sense that the kernel could instead periodically check (or "poll") the device hardware to check for new events. Interrupts are preferable to polling if the events are relatively rare, so that polling would waste CPU time. Interrupt handling shares some of the code already needed for system calls and exceptions.

Interrupts are similar to system calls, except devices generate them at any time. There is hardware on the motherboard to signal the CPU when a device needs attention (e.g., the user has typed a character on the keyboard). We must program the device to generate an interrupt, and arrange that a CPU receives the interrupt.

Let's look at the timer device and timer interrupts. We would like the timer hardware to generate an interrupt, say, 100 times per second so that the kernel can track the passage of time and so the kernel can time-slice among multiple running processes. The choice of 100 times per second allows for decent interactive performance while not swamping the processor with handling interrupts.

Like the x86 processor itself, PC motherboards have evolved, and the way interrupts are provided has evolved too. The early boards had a simple programmable interrupt controler (called the PIC), and you can find the code to manage it in `picirq.c`.

programmable interrupt controler (PIC)
inb+code
outb+code
timer.c+code
lapicinit+code
IRQ_TIMER,+code
IF+code
cli+code
sti+code
switchuvm+code
idtinit+code

With the advent of multiprocessor PC boards, a new way of handling interrupts was needed, because each CPU needs an interrupt controller to handle interrupts sent to it, and there must be a method for routing interrupts to processors. This way consists of two parts: a part that is in the I/O system (the IO APIC, `ioapic.c`), and a part that is attached to each processor (the local APIC, `lapic.c`). Xv6 is designed for a board with multiple processors, and each processor must be programmed to receive interrupts.

To also work correctly on uniprocessors, Xv6 programs the programmable interrupt controler (PIC) (7582). Each PIC handles a maximum of 8 interrupts (i.e., devices) and multiplexes them onto the interrupt pin of the processor. To allow for more than 8 devices, PICs can be cascaded and typically boards have at least two. Using `inb` and `outb` instructions Xv6 programs the master to generate IRQ 0 through 7 and the slave to generate IRQ 8 through 16. Initially xv6 programs the PIC to mask all interrupts. The code in `timer.c` sets timer 1 and enables the timer interrupt on the PIC (8274). This description omits some of the details of programming the PIC. These details of the PIC (and the IOAPIC and LAPIC) are not important to this text but the interested reader can consult the manuals for each device, which are referenced in the source files.

On multiprocessors, xv6 must program the IOAPIC, and the LAPIC on each processor. The IO APIC has a table and the processor can program entries in the table through memory-mapped I/O, instead of using `inb` and `outb` instructions. During initialization, xv6 programs to map interrupt 0 to IRQ 0, and so on, but disables them all. Specific devices enable particular interrupts and say to which processor the interrupt should be routed. For example, xv6 routes keyboard interrupts to processor 0 (8224). Xv6 routes disk interrupts to the highest numbered processor on the system, as we will see below.

The timer chip is inside the LAPIC, so that each processor can receive timer interrupts independently. Xv6 sets it up in `lapicinit` (7201). The key line is the one that programs the timer (7214). This line tells the LAPIC to periodically generate an interrupt at `IRQ_TIMER`, which is IRQ 0. Line (7243) enables interrupts on a CPU's LAPIC, which will cause it to deliver interrupts to the local processor.

A processor can control if it wants to receive interrupts through the `IF` flag in the eflags register. The instruction `cli` disables interrupts on the processor by clearing `IF`, and `sti` enables interrupts on a processor. Xv6 disables interrupts during booting of the main cpu (9112) and the other processors (1224). The scheduler on each processor enables interrupts (2714). To control that certain code fragments are not interrupted, xv6 disables interrupts during these code fragments (e.g., see `switchuvm` (1873)).

The timer interrupts through vector 32 (which xv6 chose to handle IRQ 0), which xv6 setup in `idtinit` (1365). The only difference between vector 32 and vector 64 (the one for system calls) is that vector 32 is an interrupt gate instead of a trap gate. Interrupt gates clear IF, so that the interrupted processor doesn't receive interrupts while it is handling the current interrupt. From here on until trap, interrupts follow the same code path as system calls and exceptions, building up a trap frame.

trap+code
wakeup+code
driver
block
sector
struct buf+code
B_VALID+code
B_DIRTY+code
B_BUSY+code

Trap when it's called for a time interrupt, does just two things: increment the ticks variable (3367), and call wakeup. The latter, as we will see in Chapter 5, may cause the interrupt to return in a different process.

Drivers

A driver is the code in an operating system that manages a particular device: it tells the device hardware to perform operations, configures the device to generate interrupts when done, and handles the resulting interrupts. Driver code can be tricky to write because a driver executes concurrently with the device that it manages. In addition, the driver must understand the device's interface (e.g., which I/O ports do what), and that interface can be complex and poorly documented.

The disk driver provides a good example. The disk driver copies data from and back to the disk. Disk hardware traditionally presents the data on the disk as a numbered sequence of 512-byte *blocks* (also called *sectors*): sector 0 is the first 512 bytes, sector 1 is the next, and so on. The block size that an operating system uses for its file system maybe different than the sector size that a disk uses, but typically the block size is a multiple of the sector size. Xv6's block size is identical to the disk's sector size. To represent a block xv6 has a structure struct buf (3750). The data stored in this structure is often out of sync with the disk: it might have not yet been read in from disk (the disk is working on it but hasn't returned the sector's content yet), or it might have been updated but not yet written out. The driver must ensure that the rest of xv6 doesn't get confused when the structure is out of sync with the disk.

Code: Disk driver

The IDE device provides access to disks connected to the PC standard IDE controller. IDE is now falling out of fashion in favor of SCSI and SATA, but the interface is simple and lets us concentrate on the overall structure of a driver instead of the details of a particular piece of hardware.

Xv6 represent file system blocks using struct buf (3750). BSIZE (3905) is identical to the IDE's sector size and thus each buffer represents the contents of one sector on a particular disk device. The dev and sector fields give the device and sector number and the data field is an in-memory copy of the disk sector. Although the xv6 file system chooses BSIZE to be identical to the IDE's sector size, the driver can handle a BSIZE that is a multiple of the sector size. Operating systems often use bigger blocks than 512 bytes to obtain higher disk throughput.

The flags track the relationship between memory and disk: the B_VALID flag means that data has been read in, and the B_DIRTY flag means that data needs to be written out. The B_BUSY flag is a lock bit; it indicates that some process is using the buffer and other processes must not. When a buffer has the B_BUSY flag set, we say the buffer is locked.

The kernel initializes the disk driver at boot time by calling `ideinit` (4151) from `main` (1333). `Ideinit` calls `picenable` and `ioapicenable` to enable the `IDE_IRQ` interrupt (4156-4157). The call to `picenable` enables the interrupt on a uniprocessor; `ioapicenable` enables the interrupt on a multiprocessor, but only on the last CPU (`ncpu-1`): on a two-processor system, CPU 1 handles disk interrupts.

Next, `ideinit` probes the disk hardware. It begins by calling `idewait` (4158) to wait for the disk to be able to accept commands. A PC motherboard presents the status bits of the disk hardware on I/O port `0x1f7`. `Idewait` (4137) polls the status bits until the busy bit (`IDE_BSY`) is clear and the ready bit (`IDE_DRDY`) is set.

Now that the disk controller is ready, `ideinit` can check how many disks are present. It assumes that disk 0 is present, because the boot loader and the kernel were both loaded from disk 0, but it must check for disk 1. It writes to I/O port `0x1f6` to select disk 1 and then waits a while for the status bit to show that the disk is ready (4160-4167). If not, `ideinit` assumes the disk is absent.

After `ideinit`, the disk is not used again until the buffer cache calls `iderw`, which updates a locked buffer as indicated by the flags. If `B_DIRTY` is set, `iderw` writes the buffer to the disk; if `B_VALID` is not set, `iderw` reads the buffer from the disk.

Disk accesses typically take milliseconds, a long time for a processor. The boot loader issues disk read commands and reads the status bits repeatedly until the data is ready (see Appendix B). This `polling` or `busy waiting` is fine in a boot loader, which has nothing better to do. In an operating system, however, it is more efficient to let another process run on the CPU and arrange to receive an interrupt when the disk operation has completed. `Iderw` takes this latter approach, keeping the list of pending disk requests in a queue and using interrupts to find out when each request has finished. Although `iderw` maintains a queue of requests, the simple IDE disk controller can only handle one operation at a time. The disk driver maintains the invariant that it has sent the buffer at the front of the queue to the disk hardware; the others are simply waiting their turn.

`Iderw` (4254) adds the buffer `b` to the end of the queue (4267-4271). If the buffer is at the front of the queue, `iderw` must send it to the disk hardware by calling `idestart` (4227-4229); otherwise the buffer will be started once the buffers ahead of it are taken care of.

`Idestart` (4175) issues either a read or a write for the buffer's device and sector, according to the flags. If the operation is a write, `idestart` must supply the data now (4197) and the interrupt will signal that the data has been written to disk. If the operation is a read, the interrupt will signal that the data is ready, and the handler will read it. Note that `idestart` has detailed knowledge about the IDE device, and writes the right values at the right ports. If any of these `outb` statements is wrong, the IDE will do something differently than what we want. Getting these details right is one reason why writing device drivers is challenging.

Having added the request to the queue and started it if necessary, `iderw` must wait for the result. As discussed above, polling does not make efficient use of the CPU. Instead, `iderw` sleeps, waiting for the interrupt handler to record in the buffer's flags that the operation is done (4278-4279). While this process is sleeping, xv6 will

ideinit+code
main+code
picenable+code
ioapicenable+code
IDE_IRQ+code
ideinit+code
idewait+code
IDE_BSY+code
IDE_DRDY+code
iderw+code
B_DIRTY+code
B_VALID+code
iderw+code
polling
busy waiting
iderw+code
idestart+code
idestart+code
iderw+code

schedule other processes to keep the CPU busy.

Eventually, the disk will finish its operation and trigger an interrupt. `trap` will call `ideintr` to handle it (3374). `Ideintr` (4205) consults the first buffer in the queue to find out which operation was happening. If the buffer was being read and the disk controller has data waiting, `ideintr` reads the data into the buffer with `insl` (4218-4220). Now the buffer is ready: `ideintr` sets `B_VALID`, clears `B_DIRTY`, and wakes up any process sleeping on the buffer (4222-4225). Finally, `ideintr` must pass the next waiting buffer to the disk (4227-4229).

trap+code
ideintr+code
insl+code
B_VALID+code
B_DIRTY+code
batch

Real world

Supporting all the devices on a PC motherboard 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. In many operating systems, the drivers together account for more code in the operating system than the core kernel.

Actual device drivers are far more complex than the disk driver in this chapter, but the basic ideas are the same: typically devices are slower than CPU, so the hardware uses interrupts to notify the operating system of status changes. Modern disk controllers typically accept a `batch` of disk requests at a time and even reorder them to make most efficient use of the disk arm. When disks were simpler, operating system often reordered the request queue themselves.

Many operating systems have drivers for solid-state disks because they provide much faster access to data. But, although a solid-state works very differently from a traditional mechanical disk, both devices provide block-based interfaces and reading/writing blocks on a solid-state disk is still more expensive than reading/writing RAM.

Other hardware is surprisingly similar to disks: network device buffers hold packets, audio device buffers hold sound samples, graphics card buffers hold video data and command sequences. High-bandwidth devices—disks, graphics cards, and network cards—often use direct memory access (DMA) instead of the explicit I/O (`insl`, `outsl`) in this driver. DMA allows the disk or other controllers direct access to physical memory. The driver gives the device the physical address of the buffer's data field and the device copies directly to or from main memory, interrupting once the copy is complete. Using DMA means that the CPU is not involved at all in the transfer, which can be more efficient and is less taxing for the CPU's memory caches.

Most of the devices in this chapter used I/O instructions to program them, which reflects the older nature of these devices. All modern devices are programmed using memory-mapped I/O.

Some drivers dynamically switch between polling and interrupts, because using interrupts can be expensive, but using polling can introduce delay until the driver processes an event. For example, for a network driver that receives a burst of packets, may switch from interrupts to polling since it knows that more packets must be processed and it is less expensive to process them using polling. Once no more packets need to be processed, the driver may switch back to interrupts, so that it will be alerted immediately when a new packet arrives.

The IDE driver routed interrupts statically to a particular processor. Some drivers have a sophisticated algorithm for routing interrupts to processor so that the load of processing packets is well balanced but good locality is achieved too. For example, a network driver might arrange to deliver interrupts for packets of one network connection to the processor that is managing that connection, while interrupts for packets of another connection are delivered to another processor. This routing can get quite sophisticated; for example, if some network connections are short lived while others are long lived and the operating system wants to keep all processors busy to achieve high throughput.

If user process reads a file, the data for that file is copied twice. First, it is copied from the disk to kernel memory by the driver, and then later it is copied from kernel space to user space by the `read` system call. If the user process, then sends the data on the network, then the data is copied again twice: once from user space to kernel space and from kernel space to the network device. To support applications for which low latency is important (e.g., a Web serving static Web pages), operating systems use special code paths to avoid these many copies. As one example, in real-world operating systems, buffers typically match the hardware page size, so that read-only copies can be mapped into a process's address space using the paging hardware, without any copying.

Exercises

1. Set a breakpoint at the first instruction of syscall() to catch the very first system call (e.g., br syscall). What values are on the stack at this point? Explain the output of x/37x $esp at that breakpoint with each value labeled as to what it is (e.g., saved %ebp for trap, trapframe.eip, scratch space, etc.).

2. Add a new system call that returns the uptime (i.e., return the number of ticks since xv6 booted).

3. Write a driver for a disk that supports the SATA standard (search for SATA on the Web). Unlike IDE, SATA isn't obsolete. Use SATA's tagged command queuing to issue many commands to the disk so that the disk internally can reorder commands to obtain high performance.

4. Add simple driver for an Ethernet card.