Chapter 4

Traps and system calls

There are three kinds of event which cause the CPU to set aside ordinary execution of instructions and force a transfer of control to special code that handles the event. One situation is a system call, when a user program executes the `ecall` instruction to ask the kernel to do something for it. Another situation is an *exception*: an instruction (user or kernel) does something illegal, such as divide by zero or use an invalid virtual address. The third situation is a device *interrupt*, when a device signals that it needs attention, for example when the disk hardware finishes a read or write request.

This book uses *trap* as a generic term for these situations. Typically whatever code was executing at the time of the trap will later need to resume, and shouldn't need to be aware that anything special happened. That is, we often want traps to be transparent; this is particularly important for interrupts, which the interrupted code typically doesn't expect. The usual sequence is that a trap forces a transfer of control into the kernel; the kernel saves registers and other state so that execution can be resumed; the kernel executes appropriate handler code (e.g., a system call implementation or device driver); the kernel restores the saved state and returns from the trap; and the original code resumes where it left off.

The xv6 kernel handles all traps. This is natural for system calls. It makes sense for interrupts since isolation demands that user processes not directly use devices, and because only the kernel has the state needed for device handling. It also makes sense for exceptions since xv6 responds to all exceptions from user space by killing the offending program.

Xv6 trap handling proceeds in four stages: hardware actions taken by the RISC-V CPU, an assembly "vector" that prepares the way for kernel C code, a C trap handler that decides what to do with the trap, and the system call or device-driver service routine. While commonality among the three trap types suggests that a kernel could handle all traps with a single code path, it turns out to be convenient to have separate assembly vectors and C trap handlers for three distinct cases: traps from user space, traps from kernel space, and timer interrupts.

4.1 RISC-V trap machinery

Each RISC-V CPU has a set of control registers that the kernel writes to tell the CPU how to handle traps, and that the kernel can read to find out about a trap that has occured. The RISC-V documents contain the full story [1]. `riscv.h` (kernel/riscv.h:1) contains definitions that xv6 uses. Here's an outline of the most important registers:

- `stvec`: The kernel writes the address of its trap handler here; the RISC-V jumps here to handle a trap.
- `sepc`: When a trap occurs, RISC-V saves the program counter here (since the `pc` is then overwritten with `stvec`). The `sret` (return from trap) instruction copies `sepc` to the `pc`. The kernel can write to `sepc` to control where `sret` goes.
- `scause`: The RISC-V puts a number here that describes the reason for the trap.
- `sscratch`: The kernel places a value here that comes in handy at the very start of a trap handler.
- `sstatus`: The SIE bit in `sstatus` controls whether device interrupts are enabled. If the kernel clears SIE, the RISC-V will defer device interrupts until the kernel sets SIE. The SPP bit indicates whether a trap came from user mode or supervisor mode, and controls to what mode `sret` returns.

The above registers relate to traps handled in supervisor mode, and they cannot be read or written in user mode. There is an equivalent set of control registers for traps handled in machine mode; xv6 uses them only for the special case of timer interrupts.

Each CPU on a multi-core chip has its own set of these registers, and more than one CPU may be handling a trap at any given time.

When it needs to force a trap, the RISC-V hardware does the following for all trap types (other than timer interrupts):

1. If the trap is a device interrupt, and the `sstatus` SIE bit is clear, don't do any of the following.
2. Disable interrupts by clearing SIE.
3. Copy the `pc` to `sepc`.
4. Save the current mode (user or supervisor) in the SPP bit in `sstatus`.
5. Set `scause` to reflect the trap's cause.
6. Set the mode to supervisor.
7. Copy `stvec` to the `pc`.

8. Start executing at the new `pc`.

Note that the CPU doesn't switch to the kernel page table, doesn't switch to a stack in the kernel, and doesn't save any registers other than the `pc`. Kernel software must perform these tasks. One reason that the CPU does minimal work during a trap is to provide flexibility to software; for example, some operating systems don't require a page table switch in some situations, which can increase performance.

You might wonder whether the CPU hardware's trap handling sequence could be further simplified. For example, suppose that the CPU didn't switch program counters. Then a trap could switch to supervisor mode while still running user instructions. Those user instructions could break the user/kernel isolation, for example by modifying the `satp` register to point to a page table that allowed accessing all of physical memory. It is thus important that the CPU switch to a kernel-specified instruction address, namely `stvec`.

4.2 Traps from user space

A trap may occur while executing in user space if the user program makes a system call (`ecall` instruction), or does something illegal, or if a device interrupts. The high-level path of a trap from user space is `uservec` (kernel/trampoline.S:16), then `usertrap` (kernel/trap.c:37); and when returning, `usertrapret` (kernel/trap.c:90) and then `userret` (kernel/trampoline.S:16).

Traps from user code are more challenging than from the kernel, since `satp` points to a user page table that doesn't map the kernel, and the stack pointer may contain an invalid or even malicious value.

Because the RISC-V hardware doesn't switch page tables during a trap, the user page table must include a mapping for `uservec`, the trap vector instructions that `stvec` points to. `uservec` must switch `satp` to point to the kernel page table; in order to continue executing instructions after the switch, `uservec` must be mapped at the same address in the kernel page table as in the user page table.

Xv6 satisfies these constraints with a *trampoline* page that contains `uservec`. Xv6 maps the trampoline page at the same virtual address in the kernel page table and in every user page table. This virtual address is `TRAMPOLINE` (as we saw in Figure 2.3 and in Figure 3.3). The trampoline contents are set in `trampoline.S`, and (when executing user code) `stvec` is set to `uservec` (kernel/trampoline.S:16).

When `uservec` starts, all 32 registers contain values owned by the interrupted code. But `uservec` needs to be able to modify some registers in order to set `satp` and generate addresses at which to save the registers. RISC-V provides a helping hand in the form of the `sscratch` register. The `csrrw` instruction at the start of `uservec` swaps the contents of `a0` and `sscratch`. Now the user code's `a0` is saved; `uservec` has one register (`a0`) to play with; and `a0` contains the value the kernel previously placed in `sscratch`.

`uservec`'s next task is to save the user registers. Before entering user space, the kernel previously set `sscratch` to point to a per-process `trapframe` that (among other things) has space to save all the user registers (kernel/proc.h:44). Because `satp` still refers to the user page

table, uservec needs the trapframe to be mapped in the user address space. When creating each process, xv6 allocates a page for the process's trapframe, and arranges for it always to be mapped at user virtual address TRAPFRAME, which is just below TRAMPOLINE. The process's p->trapframe also points to the trapframe, though at its physical address so the kernel can use it through the kernel page table.

Thus after swapping a0 and sscratch, a0 holds a pointer to the current process's trapframe. uservec now saves all user registers there, including the user's a0, read from sscratch.

The trapframe contains pointers to the current process's kernel stack, the current CPU's hartid, the address of usertrap, and the address of the kernel page table. uservec retrieves these values, switches satp to the kernel page table, and calls usertrap.

The job of usertrap is to determine the cause of the trap, process it, and return (kernel/trap.c:37). As mentioned above, it first changes stvec so that a trap while in the kernel will be handled by kernelvec. It saves the sepc (the saved user program counter), again because there might be a process switch in usertrap that could cause sepc to be overwritten. If the trap is a system call, syscall handles it; if a device interrupt, devintr; otherwise it's an exception, and the kernel kills the faulting process. The system call path adds four to the saved user pc because RISC-V, in the case of a system call, leaves the program pointer pointing to the ecall instruction. On the way out, usertrap checks if the process has been killed or should yield the CPU (if this trap is a timer interrupt).

The first step in returning to user space is the call to usertrapret (kernel/trap.c:90). This function sets up the RISC-V control registers to prepare for a future trap from user space. This involves changing stvec to refer to uservec, preparing the trapframe fields that uservec relies on, and setting sepc to the previously saved user program counter. At the end, usertrapret calls userret on the trampoline page that is mapped in both user and kernel page tables; the reason is that assembly code in userret will switch page tables.

usertrapret's call to userret passes a pointer to the process's user page table in a0 and TRAPFRAME in a1 (kernel/trampoline.S:88). userret switches satp to the process's user page table. Recall that the user page table maps both the trampoline page and TRAPFRAME, but nothing else from the kernel. Again, the fact that the trampoline page is mapped at the same virtual address in user and kernel page tables is what allows uservec to keep executing after changing satp. userret copies the trapframe's saved user a0 to sscratch in preparation for a later swap with TRAPFRAME. From this point on, the only data userret can use is the register contents and the content of the trapframe. Next userret restores saved user registers from the trapframe, does a final swap of a0 and sscratch to restore the user a0 and save TRAPFRAME for the next trap, and uses sret to return to user space.

4.3 Code: Calling system calls

Chapter 2 ended with initcode.S invoking the exec system call (user/initcode.S:11). Let's look at how the user call makes its way to the exec system call's implementation in the kernel.

The user code places the arguments for exec in registers a0 and a1, and puts the system call number in a7. System call numbers match the entries in the syscalls array, a table of function

pointers (kernel/syscall.c:108). The `ecall` instruction traps into the kernel and executes `uservec`, `usertrap`, and then `syscall`, as we saw above.

`syscall` (kernel/syscall.c:133) retrieves the system call number from the saved `a7` in the trapframe and uses it to index into `syscalls`. For the first system call, `a7` contains `SYS_exec` (kernel/syscall.h:8), resulting in a call to the system call implementation function `sys_exec`.

When the system call implementation function returns, `syscall` records its return value in `p->trapframe->a0`. This will cause the original user-space call to `exec()` to return that value, since the C calling convention on RISC-V places return values in `a0`. System calls conventionally return negative numbers to indicate errors, and zero or positive numbers for success. If the system call number is invalid, `syscall` prints an error and returns -1.

4.4 Code: System call arguments

System call implementations in the kernel need to find the arguments passed by user code. Because user code calls system call wrapper functions, the arguments are initially where the RISC-V C calling convention places them: in registers. The kernel trap code saves user registers to the current process's trap frame, where kernel code can find them. The functions `argint`, `argaddr`, and `argfd` retrieve the n 'th system call argument from the trap frame as an integer, pointer, or a file descriptor. They all call `argraw` to retrieve the appropriate saved user register (kernel/syscall.c:35).

Some system calls pass pointers as arguments, and the kernel must use those pointers to read or write user memory. The `exec` system call, for example, passes the kernel an array of pointers referring to string arguments in user space. These pointers pose two challenges. First, the user program may be buggy or malicious, and may pass the kernel an invalid pointer or a pointer intended to trick the kernel into accessing kernel memory instead of user memory. Second, the xv6 kernel page table mappings are not the same as the user page table mappings, so the kernel cannot use ordinary instructions to load or store from user-supplied addresses.

The kernel implements functions that safely transfer data to and from user-supplied addresses. `fetchstr` is an example (kernel/syscall.c:25). File system calls such as `exec` use `fetchstr` to retrieve string file-name arguments from user space. `fetchstr` calls `copyinstr` to do the hard work.

`copyinstr` (kernel/vm.c:406) copies up to `max` bytes to `dst` from virtual address `srcva` in the user page table `pagetable`. It uses `walkaddr` (which calls `walk`) to walk the page table in software to determine the physical address `pa0` for `srcva`. Since the kernel maps all physical RAM addresses to the same kernel virtual address, `copyinstr` can directly copy string bytes from `pa0` to `dst`. `walkaddr` (kernel/vm.c:95) checks that the user-supplied virtual address is part of the process's user address space, so programs cannot trick the kernel into reading other memory. A similar function, `copyout`, copies data from the kernel to a user-supplied address.

4.5 Traps from kernel space

Xv6 configures the CPU trap registers somewhat differently depending on whether user or kernel code is executing. When the kernel is executing on a CPU, the kernel points `stvec` to the assembly code at `kernelvec` (kernel/kernelvec.S:10). Since xv6 is already in the kernel, `kernelvec` can rely on `satp` being set to the kernel page table, and on the stack pointer referring to a valid kernel stack. `kernelvec` saves all registers so that the interrupted code can eventually resume without disturbance.

`kernelvec` saves the registers on the stack of the interrupted kernel thread, which makes sense because the register values belong to that thread. This is particularly important if the trap causes a switch to a different thread – in that case the trap will actually return on the stack of the new thread, leaving the interrupted thread's saved registers safely on its stack.

`kernelvec` jumps to `kerneltrap` (kernel/trap.c:134) after saving registers. `kerneltrap` is prepared for two types of traps: device interrrupts and exceptions. It calls `devintr` (kernel/-trap.c:177) to check for and handle the former. If the trap isn't a device interrupt, it must be an exception, and that is always a fatal error if it occurs in the xv6 kernel; the kernel calls `panic` and stops executing.

If `kerneltrap` was called due to a timer interrupt, and a process's kernel thread is running (rather than a scheduler thread), `kerneltrap` calls `yield` to give other threads a chance to run. At some point one of those threads will yield, and let our thread and its `kerneltrap` resume again. Chapter 7 explains what happens in `yield`.

When `kerneltrap`'s work is done, it needs to return to whatever code was interrupted by the trap. Because a `yield` may have disturbed the saved `sepc` and the saved previous mode in `sstatus`, `kerneltrap` saves them when it starts. It now restores those control registers and returns to `kernelvec` (kernel/kernelvec.S:48). `kernelvec` pops the saved registers from the stack and executes `sret`, which copies `sepc` to `pc` and resumes the interrupted kernel code.

It's worth thinking through how the trap return happens if `kerneltrap` called `yield` due to a timer interrupt.

Xv6 sets a CPU's `stvec` to `kernelvec` when that CPU enters the kernel from user space; you can see this in `usertrap` (kernel/trap.c:29). There's a window of time when the kernel is executing but `stvec` is set to `uservec`, and it's crucial that device interrupts be disabled during that window. Luckily the RISC-V always disables interrupts when it starts to take a trap, and xv6 doesn't enable them again until after it sets `stvec`.

4.6 Page-fault exceptions

Xv6's response to exceptions is quite boring: if an exception happens in user space, the kernel kills the faulting process. If an exception happens in the kernel, the kernel panics. Real operating systems often respond in much more interesting ways.

As an example, many kernels use page faults to implement *copy-on-write (COW) fork*. To explain copy-on-write fork, consider xv6's `fork`, described in Chapter 3. `fork` causes the child to have the same memory content as the parent, by calling `uvmcopy` (kernel/vm.c:309) to allocate

physical memory for the child and copy the parent's memory into it. It would be more efficient if the child and parent could share the parent's physical memory. A straightforward implementation of this would not work, however, since it would cause the parent and child to disrupt each other's execution with their writes to the shared stack and heap.

Parent and child can safely share phyical memory using copy-on-write fork, driven by page faults. When a CPU cannot translate a virtual address to a physical address, the CPU generates a *page-fault exception*. RISC-V has three different kinds of page fault: load page faults (when a load instruction cannot translate its virtual address), store page faults (when a store instruction cannot translate its virtual address), and instruction page faults (when the address for an instruction doesn't translate). The value in the `scause` register indicates the type of the page fault and the `stval` register contains the address that couldn't be translated.

The basic plan in COW fork is for the parent and child to initially share all physical pages, but to map them read-only. Thus, when the child or parent executes a store instruction, the RISC-V CPU raises a page-fault exception. In response to this exception, the kernel makes a copy of the page that contains the faulted address. It maps one copy read/write in the child's address space and the other copy read/write in the parent's address space. After updating the page tables, the kernel resumes the faulting process at the instruction that caused the fault. Because the kernel has updated the relevant PTE to allow writes, the faulting instruction will now execute without a fault.

This COW plan works well for `fork`, because often the child calls `exec` immediately after the fork, replacing its address space with a new address space. In that common case, the child will experience only a few page faults, and the kernel can avoid making a complete copy. Furthermore, COW fork is transparent: no modifications to applications are necessary for them to benefit.

The combination of page tables and page faults opens up a wide-range of interesting possibilities other than COW fork. Another widely-used feature is called *lazy allocation*, which has two parts. First, when an application calls `sbrk`, the kernel grows the address space, but marks the new addresses as not valid in the page table. Second, on a page fault on one of those new addresses, the kernel allocates physical memory and maps it into the page table. Since applications often ask for more memory than they need, lazy allocation is a win: the kernel allocates memory only when the application actually uses it. Like COW fork, the kernel can implement this feature transparently to applications.

Yet another widely-used feature that exploits page faults is *paging from disk*. If applications need more memory than the available physical RAM, the kernel can evict some pages: write them to a storage device such as a disk and mark their PTEs as not valid. If an application reads or writes an evicted page, the CPU will experience a page fault. The kernel can then inspect the faulting address. If the address belongs to a page that is on disk, the kernel allocates a page of physical memory, reads the page from disk to that memory, updates the PTE to be valid and refer to that memory, and resumes the application. To make room for the page, the kernel may have to evict another page. This feature requires no changes to applications, and works well if applications have locality of reference (i.e., they use only a subset of their memory at any given time).

Other features that combine paging and page-fault exceptions include automatically extending stacks and memory-mapped files.

4.7 Real world

The need for special trampoline pages could be eliminated if kernel memory were mapped into every process's user page table (with appropriate PTE permission flags). That would also eliminate the need for a page table switch when trapping from user space into the kernel. That in turn would allow system call implementations in the kernel to take advantage of the current process's user memory being mapped, allowing kernel code to directly dereference user pointers. Many operating systems have used these ideas to increase efficiency. Xv6 avoids them in order to reduce the chances of security bugs in the kernel due to inadvertent use of user pointers, and to reduce some complexity that would be required to ensure that user and kernel virtual addresses don't overlap.

4.8 Exercises

1. The functions `copyin` and `copyinstr` walk the user page table in software. Set up the kernel page table so that the kernel has the user program mapped, and `copyin` and `copyinstr` can use `memcpy` to copy system call arguments into kernel space, relying on the hardware to do the page table walk.

2. Implement lazy memory allocation

3. Implement COW fork

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 (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 [11] 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:184) 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:82). `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:98) (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:138); 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`,