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 device 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.

Xv6 handles all traps in the kernel; traps are not delivered to user code. Handling traps in the kernel is natural for system calls. It makes sense for interrupts since isolation demands that only the kernel be allowed to use devices, and because the kernel is a convenient mechanism with which to share devices among multiple processes. 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, some assembly instructions that prepare the way for kernel C code, a C function 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 code for three distinct cases: traps from user space, traps from kernel space, and timer interrupts. Kernel code (assembler or C) that processes a trap is often called a *handler*; the first handler instructions are usually written in assembler (rather than C) and are sometimes called a *vector*.

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 to the address in `stvec` to handle a trap.
- `sepc`: When a trap occurs, RISC-V saves the program counter here (since the `pc` is then overwritten with the value in `stvec`). The `sret` (return from trap) instruction copies `sepc` to the `pc`. The kernel can write `sepc` to control where `sret` goes.
- `scause`: 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 a similar 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 the SIE bit in `sstatus`.
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 omit a page table switch in some situations to increase trap performance.

It's worth thinking about whether any of the steps listed above could be omitted, perhaps in search of faster traps. Though there are situations in which a simpler sequence can work, many of the steps would be dangerous to omit in general. For example, suppose that the CPU didn't switch program counters. Then a trap from user space could switch to supervisor mode while still running user instructions. Those user instructions could break 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

Xv6 handles traps differently depending on whether it is executing in the kernel or in user code. Here is the story for traps from user code; Section 4.5 describes traps from kernel code.

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:88).

A major constraint on the design of xv6's trap handling is the fact that the RISC-V hardware does not switch page tables when it forces a trap. This means that the trap handler address in `stvec` must have a valid mapping in the user page table, since that's the page table in force when the trap handling code starts executing. Furthermore, xv6's trap handling code needs to switch to the kernel page table; in order to be able to continue executing after that switch, the kernel page table must also have a mapping for the handler pointed to by `stvec`.

Xv6 satisfies these requirements using a *trampoline* page. The trampoline page contains `uservec`, the xv6 trap handling code that `stvec` points to. The trampoline page is mapped in every process's page table at address `TRAMPOLINE`, which is at the end of the virtual address space so that it will be above memory that programs use for themselves. The trampoline page is also mapped at address `TRAMPOLINE` in the kernel page table. See Figure 2.3 and Figure 3.3. Because the trampoline page is mapped in the user page table, with the `PTE_U` flag, traps can start executing there in supervisor mode. Because the trampoline page is mapped at the same address in the kernel address space, the trap handler can continue to execute after it switches to the kernel page table.

The code for the `uservec` trap handler is in `trampoline.S` (kernel/trampoline.S:16). When `uservec` starts, all 32 registers contain values owned by the interrupted user code. These 32 values need to be saved somewhere in memory, so that they can be restored when the trap returns to user space. Storing to memory requires use of a register to hold the address, but at this point there are no general-purpose registers available! Luckily 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 in sscratch; 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 32 user registers. Before entering user space, the kernel set sscratch to point to a per-process trapframe structure that (among other things) has space to save the 32 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 the address of the current process's kernel stack, the current CPU's hartid, the address of the usertrap function, 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). It first changes stvec so that a trap while in the kernel will be handled by kernelvec rather than uservec. It saves the sepc register (the saved user program counter), because usertrap might call yield to switch to another process's kernel thread, and that process might return to user space, in the process of which it will modify sepc. If the trap is a system call, usertrap calls syscall to handle 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 program counter because RISC-V, in the case of a system call, leaves the program pointer pointing to the ecall instruction but user code needs to resume executing at the subsequent 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 TRAPFRAME in a0 and a pointer to the process's user page table 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. 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 executes `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.

`initcode.S` 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 causes `uservec`, `usertrap`, and then `syscall` to execute, 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 `sys_exec` 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 kernel 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:398) copies up to `max` bytes to `dst` from virtual address `srcva` in the user page table `pagetable`. Since `pagetable` is *not* the current page table, `copyinstr` uses `walkaddr` (which calls `walk`) to look up `srcva` in `pagetable`, yielding physical address

pa0. The kernel maps each physical RAM address to the corresponding kernel virtual address, so copyinstr can directly copy string bytes from pa0 to dst. walkaddr (kernel/vm.c:104) 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 pushes all 32 registers onto the stack, from which it will later restore them so that the interrupted kernel code can 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 from 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 (as opposed to 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 sepc and the 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 has started executing but stvec is still set to uservec, and it's crucial that no device interrupt occur 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's initial memory content to be the same as the parent's at the time of the fork. xv6 implements fork with `uvmcopy` (kernel/vm.c:301), which allocates physical memory for the child and copies 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 physical memory by appropriate use of page-table permissions and page faults. The CPU raises a *page-fault exception* when a virtual address is used that has no mapping in the page table, or has a mapping whose `PTE_V` flag is clear, or a mapping whose permission bits (`PTE_R`, `PTE_W`, `PTE_X`, `PTE_U`) forbid the operation being attempted. RISC-V distinguishes three 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 in the program counter doesn't translate). 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 for each to map them read-only (with the `PTE_W` flag clear). Parent and child can read from the shared physical memory. If either writes a given page, the RISC-V CPU raises a page-fault exception. The kernel's trap handler responds by allocating a new page of physical memory and copying into it the physical page that the faulted address maps to. The kernel changes the relevant PTE in the faulting process's page table to point to the copy and to allow writes as well as reads, and then resumes the faulting process at the instruction that caused the fault. Because the PTE allows writes, the re-executed instruction will now execute without a fault. Copy-on-write requires book-keeping to help decide when physical pages can be freed, since each page can be referenced by a varying number of page tables depending on the history of forks, page faults, execs, and exits. This book-keeping allows an important optimization: if a process incurs a store page fault and the physical page is only referred to from that process's page table, no copy is needed.

Copy-on-write makes `fork` faster, since `fork` need not copy memory. Some of the memory will have to be copied later, when written, but it's often the case that most of the memory never has to be copied. A common example is `fork` followed by `exec`: a few pages may be written after the `fork`, but then the child's `exec` releases the bulk of the memory inherited from the parent. Copy-on-write `fork` eliminates the need to ever copy this memory. 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 in addition to COW fork. Another widely-used feature is called *lazy allocation*, which has two parts. First, when an application asks for more memory by calling `sbrk`, the kernel notes the

increase in size, but does not allocate physical memory and does not create PTEs for the new range of virtual addresses. Second, on a page fault on one of those new addresses, the kernel allocates a page of physical memory and maps it into the page table. Like COW fork, the kernel can implement lazy allocation transparently to applications.

Since applications often ask for more memory than they need, lazy allocation is a win: the kernel doesn't have to do any work at all for pages that the application never uses. Furthermore, if the application is asking to grow the address space by a lot, then `sbrk` without lazy allocation is expensive: if an application asks for a gigabyte of memory, the kernel has to allocate and zero 262,144 4096-byte pages. Lazy allocation allows this cost to be spread over time. On the other hand, lazy allocation incurs the extra overhead of page faults, which involve a kernel/user transition. Operating systems can reduce this cost by allocating a batch of consecutive pages per page fault instead of one page and by specializing the kernel entry/exit code for such page-faults.

Yet another widely-used feature that exploits page faults is *demand paging*. In `exec`, xv6 loads all text and data of an application eagerly into memory. Since applications can be large and reading from disk is expensive, this startup cost may be noticeable to users: when the user starts a large application from the shell, it may take a long time before user sees a response. To improve response time, a modern kernel creates the page table for the user address space, but marks the PTEs for the pages invalid. On a page fault, the kernel reads the content of the page from disk and maps it into the user address space. Like COW fork and lazy allocation, the kernel can implement this feature transparently to applications.

The programs running on a computer may need more memory than the computer has RAM. To cope gracefully, the operating system may implement *paging to disk*. The idea is to store only a fraction of user pages in RAM, and to store the rest on disk in a *paging area*. The kernel marks PTEs that correspond to memory stored in the paging area (and thus not in RAM) as invalid. If an application tries to use one of the pages that has been *paged out* to disk, the application will incur a page fault, and the page must be *paged in*: the kernel trap handler will allocate a page of physical RAM, read the page from disk into the RAM, and modify the relevant PTE to point to the RAM.

What happens if a page needs to be paged in, but there is no free physical RAM? In that case, the kernel must first free a physical page by paging it out or *evicting* it to the paging area on disk, and marking the PTEs referring to that physical page as invalid. Eviction is expensive, so paging performs best if it's infrequent: if applications use only a subset of their memory pages and the union of the subsets fits in RAM. This property is often referred to as having good locality of reference. As with many virtual memory techniques, kernels usually implement paging to disk in a way that's transparent to applications.

Computers often operate with little or no *free* physical memory, regardless of how much RAM the hardware provides. For example, cloud providers multiplex many customers on a single machine to use their hardware cost-effectively. As another example, users run many applications on smart phones in a small amount of physical memory. In such settings allocating a page may require first evicting an existing page. Thus, when free physical memory is scarce, allocation is expensive.

Lazy allocation and demand paging are particularly advantageous when free memory is scarce. Eagerly allocating memory in `sbrk` or `exec` incurs the extra cost of eviction to make memory available. Furthermore, there is a risk that the eager work is wasted, because before the application

uses the page, the operating system may have evicted it.

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

4.7 Real world

The trampoline and trapframe may seem excessively complex. A driving force is that the RISC-V intentionally does as little as it can when forcing a trap, to allow the possibility of very fast trap handling, which turns out to be important. As a result, the first few instructions of the kernel trap handler effectively have to execute in the user environment: the user page table, and user register contents. And the trap handler is initially ignorant of useful facts such as the identity of the process that's running or the address of the kernel page table. A solution is possible because RISC-V provides protected places in which the kernel can stash away information before entering user space: the `sscratch` register, and user page table entries that point to kernel memory but are protected by lack of `PTE_U`. Xv6's trampoline and trapframe exploit these RISC-V features.

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.

Production operating systems implement copy-on-write fork, lazy allocation, demand paging, paging to disk, memory-mapped files, etc. Furthermore, production operating systems will try to use all of physical memory, either for applications or caches (e.g., the buffer cache of the file system, which we will cover later in Section 8.2). Xv6 is naïve in this regard: you want your operating system to use the physical memory you paid for, but xv6 doesn't. Furthermore, if xv6 runs out of memory, it returns an error to the running application or kills it, instead of, for example, evicting a page of another application.

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.

4. Is there a way to eliminate the special `TRAPFRAME` page mapping in every user address space? For example, could `uservec` be modified to simply push the 32 user registers onto the kernel stack, or store them in the `proc` structure?

5. Could xv6 be modified to eliminate the special `TRAMPOLINE` page mapping?