Chapter 3

System calls, exceptions, and interrupts

An operating system must handle system calls, exceptions, and interrupts. With a system call a user program can ask for an operating system service, as we saw at the end of the last chapter. *Exceptions* are illegal program actions that generate an interrupt. Examples of illegal programs actions include divide by zero, attempt to access memory outside segment bounds, and so on. *Interrupts* are generated by hardware devices that need attention of 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.

The basic plan is as follows. An interrupts stops the normal processor loop—read an instruction, advance the program counter, execute the instruction, repeat—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, x86 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, and requires the designer to think about parallelism and concurrency. A topic that we will address in Chapter 4.

This chapter examines the xv6 trap handlers, covering hardware interrupts, software exceptions, and system calls.

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" 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 a CPU-internal registers, but only if the target segment selector's PL < CPL.
- Load `%ss` and `%esp` from a task segment descriptor.
- Push `%ss`.
- 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. The check CPL <= DPL allows the kernel to forbid systems for some privilege levels. For example, for a user program to execute `int` instruction succesfully, the DPL must be 3. If the user program doesn't have the appropriate privilege, then `int` instruction 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 user might not have set up an appropriate stack so that hardware uses the stack specified in the task segments, which is setup in kernel mode.

When an int instruction completes and there was a privilege-level change (the privilege level in the descriptor is lower than CPL), the following values are on the stack specified in the task segment:

```
        ss
        esp
        eflags
        cs
        eip
esp ->  error code
```

If the int instruction didn't require a privilege-level change, the following values are on the original stack:

```
         eflags
         cs
         eip
esp ->   error code
```

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

The last chapter ended with initcode.S invoking a system call. Let's look at that again (7212). 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 (3250). We need to arrange that the int instruction switches the processor from user space to kernel space, 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 describes how xv6 arranges this for system calls, and then we will discover that we can reuse the same code for interrupts and exceptions.

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 (2966), 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 provides 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_FL 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 (2622) 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 (2850) 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 (2906) continues to save processor registers: it pushes %ds, %es, %fs, %gs, and the general-purpose registers (2907-2912). 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. 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.

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 (2915-2917). It sets %fs and %gs to point at the SEG_KCPU per-CPU data segment (2918-2920).

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` (2923). Then it calls `trap` (2924). After `trap` returns, `alltraps` pops the argument off the stack by adding to the stack pointer (2925) 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.

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` (3001) 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 two `cp->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 `cp->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`.

[[Sidebar about panic: panic is the kernel's last resort: the impossible has happened and the kernel does not know how to proceed. In xv6, panic does ...]]

Code: System calls

For system calls, trap invokes `syscall` (3275). `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 9, and `syscall` will invoke the 9th 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 (3281). 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 and argptr, argstr retrieve the *n*'th system call argument, as either an integer, pointer, or a string. 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 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 by the user is indeed a pointer in 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. Now though, the kernel is running and it can derefence any address that the user might have passed, so it must check explicitly that the address is below p->sz

argptr is similar in purpose to argint: it interprets the *n*th system call argument. argptr calls argint to fetch the argument as an integer and then checks if the integer as a user pointer is indeed in the user part of the address space. Note that two checks occur during a call to code argptr . First, the user stack pointer is checked during the fetching of the argument. Then the argument, itself a user pointer, is checked.

argstr is the final member of the system call argument trio. It 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.

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 9, sys_exec uses these functions to get at its arguments.

Code: Interrupts

Devices on the motherboard can generate interrupts, and xv6 must setup the hardware to handle these interrupts. Without device support xv6 wouldn't be usable; a user couldn't type on the keyboard, a file system couldn't store data on disk, etc. Fortunately, adding interrupts and support for simple devices doesn't require much additional complexity. As we will see, interrupts can use the same code as for systems 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`.

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 send 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) (6432). Each PIC can handle a maximum of 8 interrupts (i.e., devices) and multiplex them on 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 (7074). 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 (7016). Xv6 routes disk interrupts to the highest numbered processor on the system (3751).

The timer chip is inside the LAPIC, so that each processor can receive timer interrupts independently. Xv6 sets it up in `lapicinit` (6151). The key line is the one that programs the timer (6165). This line tells the LAPIC to periodically generate an interrupt at `IRQ_TIMER,` which is IRQ 0. Line (6194) 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` flags 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 (1015) and the other processors (1129). The scheduler on each processor enables interrupts (1800). To control that certain code fragments are not interrupted, xv6 disables interrupts during these code fragments (e.g., see `switchuvm` (2622)).

The timer interrupts through vector 32 (which xv6 chose to handle IRQ 0), which xv6 setup in `idtinit` (1382). 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 clears `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` when it's called for a time interrupt, does just two things: increment the ticks variable (2962), and call `wakeup`. The latter, as we will see in Chapter 5, may cause the interrupt to return in a different process.

Real world

polling

memory-mapped I/O versus I/O instructions

interrupt handler (trap) table driven.

Interrupt masks. Interrupt routing. On multiprocessor, different hardware but same effect.

interrupts can move.

more complicated routing.

more system calls.

have to copy system call strings.

even harder if memory space can be adjusted.

Supporting all the devices on a PC motherboard in its full glory is much work, because the drivers to manage the devices can get complex.