mechanisms for creating new processes, waiting for processes to terminate, notifying other processes of exceptional events in the system, and detecting and responding to these events. If you understand these ECF mechanisms, then you can use them to write interesting programs such as Unix shells and Web servers.

- *Understanding ECF will help you understand concurrency.* ECF is a basic mechanism for implementing concurrency in computer systems. The following are all examples of concurrency in action: an exception handler that interrupts the execution of an application program; processes and threads whose execution overlap in time; and a signal handler that interrupts the execution of an application program. Understanding ECF is a first step to understanding concurrency. We will return to study it in more detail in Chapter 12.
- *Understanding ECF will help you understand how software exceptions work.* Languages such as C++ and Java provide software exception mechanisms via `try`, `catch`, and `throw` statements. Software exceptions allow the program to make *nonlocal* jumps (i.e., jumps that violate the usual call/return stack discipline) in response to error conditions. Nonlocal jumps are a form of application-level ECF and are provided in C via the `setjmp` and `longjmp` functions. Understanding these low-level functions will help you understand how higher-level software exceptions can be implemented.

Up to this point in your study of systems, you have learned how applications interact with the hardware. This chapter is pivotal in the sense that you will begin to learn how your applications interact with the operating system. Interestingly, these interactions all revolve around ECF. We describe the various forms of ECF that exist at all levels of a computer system. We start with exceptions, which lie at the intersection of the hardware and the operating system. We also discuss system calls, which are exceptions that provide applications with entry points into the operating system. We then move up a level of abstraction and describe processes and signals, which lie at the intersection of applications and the operating system. Finally, we discuss nonlocal jumps, which are an application-level form of ECF.

8.1 Exceptions

Exceptions are a form of exceptional control flow that are implemented partly by the hardware and partly by the operating system. Because they are partly implemented in hardware, the details vary from system to system. However, the basic ideas are the same for every system. Our aim in this section is to give you a general understanding of exceptions and exception handling and to help demystify what is often a confusing aspect of modern computer systems.

An *exception* is an abrupt change in the control flow in response to some change in the processor's state. Figure 8.1 shows the basic idea.

In the figure, the processor is executing some current instruction I_{curr} when a significant change in the processor's *state* occurs. The state is encoded in various bits and signals inside the processor. The change in state is known as an *event*.

Aside Hardware versus software exceptions

C++ and Java programmers will have noticed that the term "exception" is also used to describe the application-level ECF mechanism provided by C++ and Java in the form of `catch`, `throw`, and `try` statements. If we wanted to be perfectly clear, we might distinguish between "hardware" and "software" exceptions, but this is usually unnecessary because the meaning is clear from the context.

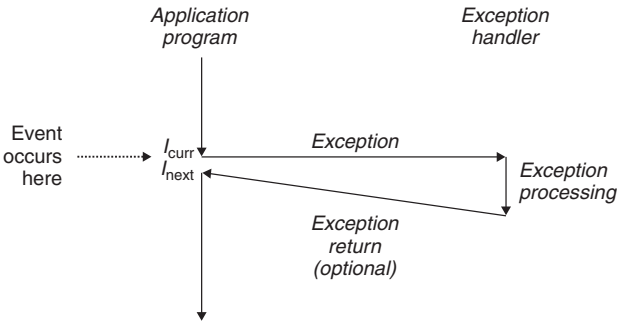

Figure 8.1
Anatomy of an exception. A change in the processor's state (an event) triggers an abrupt control transfer (an exception) from the application program to an exception handler. After it finishes processing, the handler either returns control to the interrupted program or aborts.

The event might be directly related to the execution of the current instruction. For example, a virtual memory page fault occurs, an arithmetic overflow occurs, or an instruction attempts a divide by zero. On the other hand, the event might be unrelated to the execution of the current instruction. For example, a system timer goes off or an I/O request completes.

In any case, when the processor detects that the event has occurred, it makes an indirect procedure call (the exception), through a jump table called an *exception table*, to an operating system subroutine (the *exception handler*) that is specifically designed to process this particular kind of event. When the exception handler finishes processing, one of three things happens, depending on the type of event that caused the exception:

1. The handler returns control to the current instruction I_{curr}, the instruction that was executing when the event occurred.
2. The handler returns control to I_{next}, the instruction that would have executed next had the exception not occurred.
3. The handler aborts the interrupted program.

Section 8.1.2 says more about these possibilities.

8.1.1 Exception Handling

Exceptions can be difficult to understand because handling them involves close cooperation between hardware and software. It is easy to get confused about

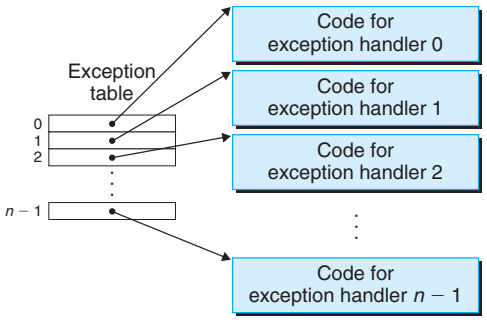

Figure 8.2
Exception table. The exception table is a jump table where entry k contains the address of the handler code for exception k.

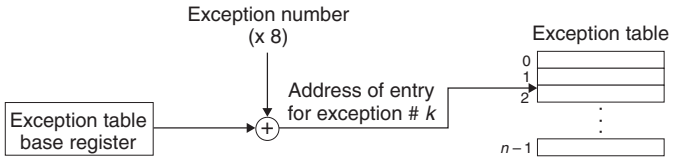

Figure 8.3
Generating the address of an exception handler. The exception number is an index into the exception table.

which component performs which task. Let's look at the division of labor between hardware and software in more detail.

Each type of possible exception in a system is assigned a unique nonnegative integer *exception number*. Some of these numbers are assigned by the designers of the processor. Other numbers are assigned by the designers of the operating system *kernel* (the memory-resident part of the operating system). Examples of the former include divide by zero, page faults, memory access violations, breakpoints, and arithmetic overflows. Examples of the latter include system calls and signals from external I/O devices.

At system boot time (when the computer is reset or powered on), the operating system allocates and initializes a jump table called an *exception table*, so that entry k contains the address of the handler for exception k. Figure 8.2 shows the format of an exception table.

At run time (when the system is executing some program), the processor detects that an event has occurred and determines the corresponding exception number k. The processor then triggers the exception by making an indirect procedure call, through entry k of the exception table, to the corresponding handler. Figure 8.3 shows how the processor uses the exception table to form the address of the appropriate exception handler. The exception number is an index into the exception table, whose starting address is contained in a special CPU register called the *exception table base register*.

An exception is akin to a procedure call, but with some important differences:

- As with a procedure call, the processor pushes a return address on the stack before branching to the handler. However, depending on the class of exception, the return address is either the current instruction (the instruction that

was executing when the event occurred) or the next instruction (the instruction that would have executed after the current instruction had the event not occurred).

- The processor also pushes some additional processor state onto the stack that will be necessary to restart the interrupted program when the handler returns. For example, an x86-64 system pushes the EFLAGS register containing the current condition codes, among other things, onto the stack.
- When control is being transferred from a user program to the kernel, all of these items are pushed onto the kernel's stack rather than onto the user's stack.
- Exception handlers run in *kernel mode* (Section 8.2.4), which means they have complete access to all system resources.

Once the hardware triggers the exception, the rest of the work is done in software by the exception handler. After the handler has processed the event, it optionally returns to the interrupted program by executing a special "return from interrupt" instruction, which pops the appropriate state back into the processor's control and data registers, restores the state to *user mode* (Section 8.2.4) if the exception interrupted a user program, and then returns control to the interrupted program.

8.1.2 Classes of Exceptions

Exceptions can be divided into four classes: *interrupts*, *traps*, *faults*, and *aborts*. The table in Figure 8.4 summarizes the attributes of these classes.

Interrupts

Interrupts occur *asynchronously* as a result of signals from I/O devices that are external to the processor. Hardware interrupts are asynchronous in the sense that they are not caused by the execution of any particular instruction. Exception handlers for hardware interrupts are often called *interrupt handlers*.

Figure 8.5 summarizes the processing for an interrupt. I/O devices such as network adapters, disk controllers, and timer chips trigger interrupts by signaling a pin on the processor chip and placing onto the system bus the exception number that identifies the device that caused the interrupt.

Class	Cause	Async/sync	Return behavior
Interrupt	Signal from I/O device	Async	Always returns to next instruction
Trap	Intentional exception	Sync	Always returns to next instruction
Fault	Potentially recoverable error	Sync	Might return to current instruction
Abort	Nonrecoverable error	Sync	Never returns

Figure 8.4 Classes of exceptions. Asynchronous exceptions occur as a result of events in I/O devices that are external to the processor. Synchronous exceptions occur as a direct result of executing an instruction.

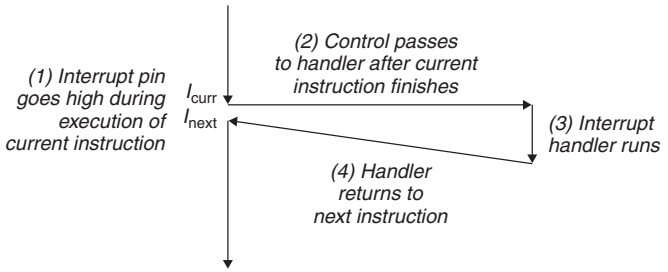

Figure 8.5
Interrupt handling. The interrupt handler returns control to the next instruction in the application program's control flow.

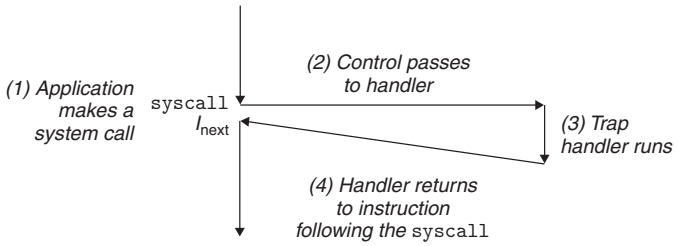

Figure 8.6
Trap handling. The trap handler returns control to the next instruction in the application program's control flow.

After the current instruction finishes executing, the processor notices that the interrupt pin has gone high, reads the exception number from the system bus, and then calls the appropriate interrupt handler. When the handler returns, it returns control to the next instruction (i.e., the instruction that would have followed the current instruction in the control flow had the interrupt not occurred). The effect is that the program continues executing as though the interrupt had never happened.

The remaining classes of exceptions (traps, faults, and aborts) occur *synchronously* as a result of executing the current instruction. We refer to this instruction as the *faulting instruction*.

Traps and System Calls

Traps are *intentional* exceptions that occur as a result of executing an instruction. Like interrupt handlers, trap handlers return control to the next instruction. The most important use of traps is to provide a procedure-like interface between user programs and the kernel, known as a *system call*.

User programs often need to request services from the kernel such as reading a file (`read`), creating a new process (`fork`), loading a new program (`execve`), and terminating the current process (`exit`). To allow controlled access to such kernel services, processors provide a special `syscall` *n* instruction that user programs can execute when they want to request service *n*. Executing the `syscall` instruction causes a trap to an exception handler that decodes the argument and calls the appropriate kernel routine. Figure 8.6 summarizes the processing for a system call.

From a programmer's perspective, a system call is identical to a regular function call. However, their implementations are quite different. Regular functions

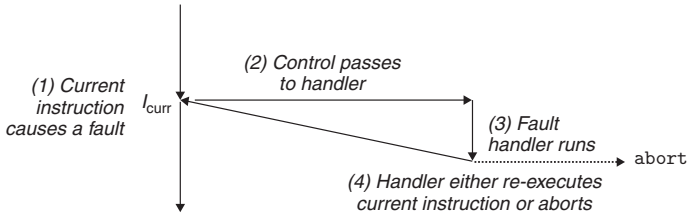

Figure 8.7
Fault handling. Depending on whether the fault can be repaired or not, the fault handler either re-executes the faulting instruction or aborts.

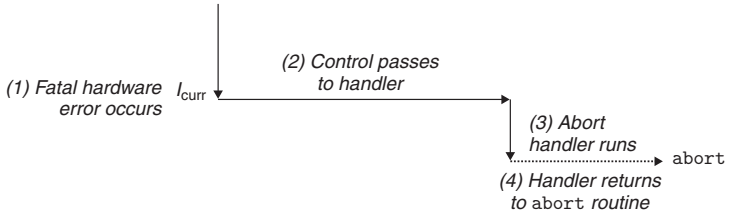

Figure 8.8
Abort handling. The abort handler passes control to a kernel `abort` routine that terminates the application program.

run in *user mode*, which restricts the types of instructions they can execute, and they access the same stack as the calling function. A system call runs in *kernel mode*, which allows it to execute privileged instructions and access a stack defined in the kernel. Section 8.2.4 discusses user and kernel modes in more detail.

Faults

Faults result from error conditions that a handler might be able to correct. When a fault occurs, the processor transfers control to the fault handler. If the handler is able to correct the error condition, it returns control to the faulting instruction, thereby re-executing it. Otherwise, the handler returns to an `abort` routine in the kernel that terminates the application program that caused the fault. Figure 8.7 summarizes the processing for a fault.

A classic example of a fault is the page fault exception, which occurs when an instruction references a virtual address whose corresponding page is not resident in memory and must therefore be retrieved from disk. As we will see in Chapter 9, a page is a contiguous block (typically 4 KB) of virtual memory. The page fault handler loads the appropriate page from disk and then returns control to the instruction that caused the fault. When the instruction executes again, the appropriate page is now resident in memory and the instruction is able to run to completion without faulting.

Aborts

Aborts result from unrecoverable fatal errors, typically hardware errors such as parity errors that occur when DRAM or SRAM bits are corrupted. Abort handlers never return control to the application program. As shown in Figure 8.8, the handler returns control to an `abort` routine that terminates the application program.

Exception number	Description	Exception class
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32–255	OS-defined exceptions	Interrupt or trap

Figure 8.9 Examples of exceptions in x86-64 systems.

8.1.3 Exceptions in Linux/x86-64 Systems

To help make things more concrete, let's look at some of the exceptions defined for x86-64 systems. There are up to 256 different exception types [50]. Numbers in the range from 0 to 31 correspond to exceptions that are defined by the Intel architects and thus are identical for any x86-64 system. Numbers in the range from 32 to 255 correspond to interrupts and traps that are defined by the operating system. Figure 8.9 shows a few examples.

Linux/x86-64 Faults and Aborts

Divide error. A divide error (exception 0) occurs when an application attempts to divide by zero or when the result of a divide instruction is too big for the destination operand. Unix does not attempt to recover from divide errors, opting instead to abort the program. Linux shells typically report divide errors as "Floating exceptions."

General protection fault. The infamous general protection fault (exception 13) occurs for many reasons, usually because a program references an undefined area of virtual memory or because the program attempts to write to a read-only text segment. Linux does not attempt to recover from this fault. Linux shells typically report general protection faults as "Segmentation faults."

Page fault. A page fault (exception 14) is an example of an exception where the faulting instruction is restarted. The handler maps the appropriate page of virtual memory on disk into a page of physical memory and then restarts the faulting instruction. We will see how page faults work in detail in Chapter 9.

Machine check. A machine check (exception 18) occurs as a result of a fatal hardware error that is detected during the execution of the faulting instruction. Machine check handlers never return control to the application program.

Linux/x86-64 System Calls

Linux provides hundreds of system calls that application programs use when they want to request services from the kernel, such as reading a file, writing a file, and

Number	Name	Description	Number	Name	Description
0	`read`	Read file	33	`pause`	Suspend process until signal arrives
1	`write`	Write file	37	`alarm`	Schedule delivery of alarm signal
2	`open`	Open file	39	`getpid`	Get process ID
3	`close`	Close file	57	`fork`	Create process
4	`stat`	Get info about file	59	`execve`	Execute a program
9	`mmap`	Map memory page to file	60	`_exit`	Terminate process
12	`brk`	Reset the top of the heap	61	`wait4`	Wait for a process to terminate
32	`dup2`	Copy file descriptor	62	`kill`	Send signal to a process

Figure 8.10 Examples of popular system calls in Linux x86-64 systems.

creating a new process. Figure 8.10 lists some popular Linux system calls. Each system call has a unique integer number that corresponds to an offset in a jump table in the kernel. (Notice that this jump table is not the same as the exception table.)

C programs can invoke any system call directly by using the `syscall` function. However, this is rarely necessary in practice. The C standard library provides a set of convenient wrapper functions for most system calls. The wrapper functions package up the arguments, trap to the kernel with the appropriate system call instruction, and then pass the return status of the system call back to the calling program. Throughout this text, we will refer to system calls and their associated wrapper functions interchangeably as *system-level functions*.

System calls are provided on x86-64 systems via a trapping instruction called `syscall`. It is quite interesting to study how programs can use this instruction to invoke Linux system calls directly. All arguments to Linux system calls are passed through general-purpose registers rather than the stack. By convention, register `%rax` contains the syscall number, with up to six arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, and `%r9`. The first argument is in `%rdi`, the second in `%rsi`, and so on. On return from the system call, registers `%rcx` and `%r11` are destroyed, and `%rax` contains the return value. A negative return value between −4,095 and −1 indicates an error corresponding to negative `errno`.

For example, consider the following version of the familiar `hello` program, written using the `write` system-level function (Section 10.4) instead of `printf`:

```
1  int main()
2  {
3      write(1, "hello, world\n", 13);
4      _exit(0);
5  }
```

The first argument to `write` sends the output to `stdout`. The second argument is the sequence of bytes to write, and the third argument gives the number of bytes to write.

Aside A note on terminology

The terminology for the various classes of exceptions varies from system to system. Processor ISA specifications often distinguish between asynchronous "interrupts" and synchronous "exceptions" yet provide no umbrella term to refer to these very similar concepts. To avoid having to constantly refer to "exceptions and interrupts" and "exceptions or interrupts," we use the word "exception" as the general term and distinguish between asynchronous exceptions (interrupts) and synchronous exceptions (traps, faults, and aborts) only when it is appropriate. As we have noted, the basic ideas are the same for every system, but you should be aware that some manufacturers' manuals use the word "exception" to refer only to those changes in control flow caused by synchronous events.

code/ecf/hello-asm64.sa

```
1   .section .data
2   string:
3     .ascii "hello, world\n"
4   string_end:
5     .equ len, string_end - string
6   .section .text
7   .globl main
8   main:
      First, call write(1, "hello, world\n", 13)
9     movq $1, %rax          write is system call 1
10    movq $1, %rdi          Arg1: stdout has descriptor 1
11    movq $string, %rsi     Arg2: hello world string
12    movq $len, %rdx        Arg3: string length
13    syscall                Make the system call

      Next, call _exit(0)
14    movq $60, %rax         _exit is system call 60
15    movq $0, %rdi          Arg1: exit status is 0
16    syscall                Make the system call
```

code/ecf/hello-asm64.sa

Figure 8.11 **Implementing the `hello` program directly with Linux system calls.**

Figure 8.11 shows an assembly-language version of `hello` that uses the `syscall` instruction to invoke the `write` and `exit` system calls directly. Lines 9–13 invoke the `write` function. First, line 9 stores the number of the `write` system call in `%rax`, and lines 10–12 set up the argument list. Then, line 13 uses the `syscall` instruction to invoke the system call. Similarly, lines 14–16 invoke the `_exit` system call.

8.2 Processes

Exceptions are the basic building blocks that allow the operating system kernel to provide the notion of a *process*, one of the most profound and successful ideas in computer science.

When we run a program on a modern system, we are presented with the illusion that our program is the only one currently running in the system. Our program appears to have exclusive use of both the processor and the memory. The processor appears to execute the instructions in our program, one after the other, without interruption. Finally, the code and data of our program appear to be the only objects in the system's memory. These illusions are provided to us by the notion of a process.

The classic definition of a process is *an instance of a program in execution*. Each program in the system runs in the *context* of some process. The context consists of the state that the program needs to run correctly. This state includes the program's code and data stored in memory, its stack, the contents of its general-purpose registers, its program counter, environment variables, and the set of open file descriptors.

Each time a user runs a program by typing the name of an executable object file to the shell, the shell creates a new process and then runs the executable object file in the context of this new process. Application programs can also create new processes and run either their own code or other applications in the context of the new process.

A detailed discussion of how operating systems implement processes is beyond our scope. Instead, we will focus on the key abstractions that a process provides to the application:

- An independent logical control flow that provides the illusion that our program has exclusive use of the processor.
- A private address space that provides the illusion that our program has exclusive use of the memory system.

Let's look more closely at these abstractions.

8.2.1 Logical Control Flow

A process provides each program with the illusion that it has exclusive use of the processor, even though many other programs are typically running concurrently on the system. If we were to use a debugger to single-step the execution of our program, we would observe a series of program counter (PC) values that corresponded exclusively to instructions contained in our program's executable object file or in shared objects linked into our program dynamically at run time. This sequence of PC values is known as a *logical control flow*, or simply *logical flow*.

Consider a system that runs three processes, as shown in Figure 8.12. The single physical control flow of the processor is partitioned into three logical flows, one for each process. Each vertical line represents a portion of the logical flow for