

Chapter 8

Exceptional Control Flow

From the time you first apply power to a processor until the time you shut it off, the program counter assumes a sequence of values

$$a_0, a_1, \dots, a_{n-1}$$

where each a_k is the address of some corresponding instruction I_k . Each transition from a_k to a_{k+1} is called a *control transfer*. A sequence of such control transfers is called the *flow of control*, or *control flow* of the processor.

The simplest kind of control flow is a “smooth” sequence where each I_k and I_{k+1} are adjacent in memory. Typically, abrupt changes to this smooth flow, where I_{k+1} is not adjacent to I_k , are caused by familiar program instructions such as jumps, calls, and returns. Such instructions are necessary mechanisms that allow programs to react to changes in internal program state represented by program variables.

But systems must also be able to react to changes in system state that are not captured by internal program variables and are not necessarily related to the execution of the program. For example, a hardware timer goes off at regular intervals and must be dealt with. Packets arrive at the network adapter and must be stored in memory. Programs request data from a disk and then sleep until they are notified that the data are ready. Parent processes that create child processes must be notified when their children terminate.

Modern systems react to these situations by making abrupt changes in the control flow. In general, we refer to these abrupt changes as *exceptional control flow* (ECF). Exceptional control flow occurs at all levels of a computer system. For example, at the hardware level, events detected by the hardware trigger abrupt control transfers to exception handlers. At the operating systems level, the kernel transfers control from one user process to another via context switches. At the application level, a process can send a *signal* to another process that abruptly transfers control to a signal handler in the recipient. An individual program can react to errors by sidestepping the usual stack discipline and making nonlocal jumps to arbitrary locations in other functions.

As programmers, there are a number of reasons why it is important for you to understand ECF:

- *Understanding ECF will help you understand important systems concepts.* ECF is the basic mechanism that operating systems use to implement I/O, processes, and virtual memory. Before you can really understand these important ideas, you need to understand ECF.

- *Understanding ECF will help you understand how applications interact with the operating system.* Applications request services from the operating system by using a form of ECF known as a *trap* or *system call*. For example, writing data to a disk, reading data from a network, creating a new process, and terminating the current process are all accomplished by application programs invoking system calls. Understanding the basic system call mechanism will help you understand how these services are provided to applications.
- *Understanding ECF will help you write interesting new application programs.* The operating system provides application programs with powerful ECF 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. 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 are all examples of concurrency in action. 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

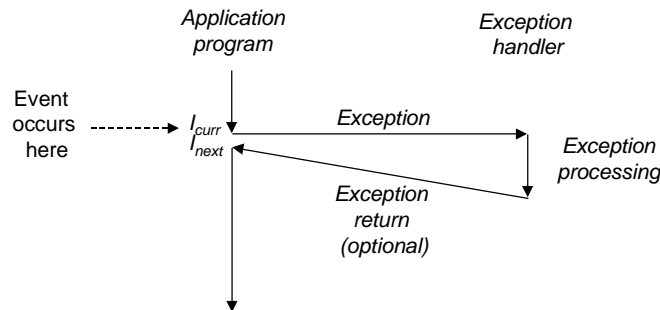


Figure 8.1: **Anatomy of an exception.** A change in the processor's state (event) triggers an abrupt control transfer (an exception) from the application program to an exception handler. After it finishes processing, the handler either returns control to the interrupted program or aborts.

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*. 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.

Aside: Hardware vs. 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. **End Aside.**

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 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.

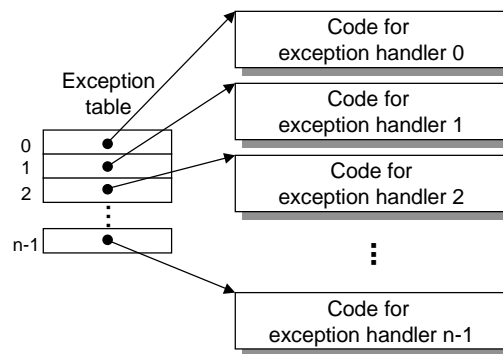


Figure 8.2: **Exception table.** The exception table is a jump table where entry k contains the address of the handler code for exception k .

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*.

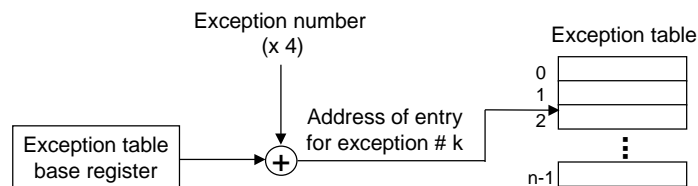


Figure 8.3: **Generating the address of an exception handler.** The exception number is an index into the exception table.

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 IA32 system pushes the EFLAGS register containing, among other things, the current condition codes, onto the stack.
- If 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.

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.

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.

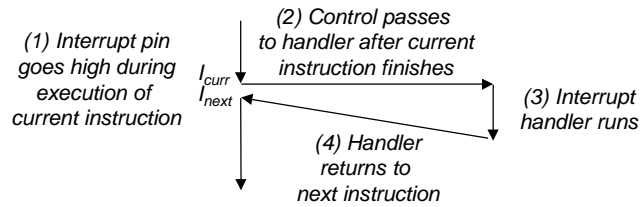


Figure 8.5: **Interrupt handling.** The interrupt handler returns control to the next instruction in the application program's control flow.

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`), or 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

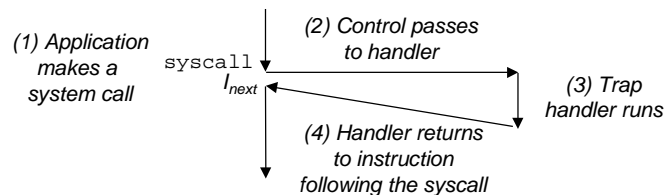


Figure 8.6: **Trap handling.** The trap handler returns control to the next instruction in the application program's control flow.

to a regular function call. However, their implementations are quite different. Regular functions 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 instructions, and accesses 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 reexecuting 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.

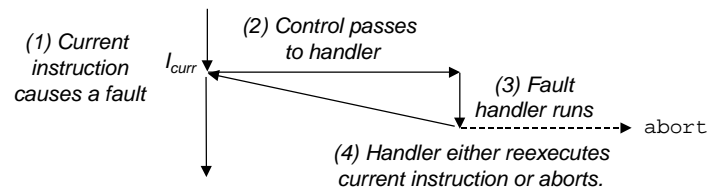


Figure 8.7: **Fault handling.** Depending on whether the fault can be repaired or not, the fault handler either re-executes the faulting instruction or aborts.

A classic example of a fault is the page fault exception, which occurs when an instruction references a virtual address whose corresponding physical 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 physical page is 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.

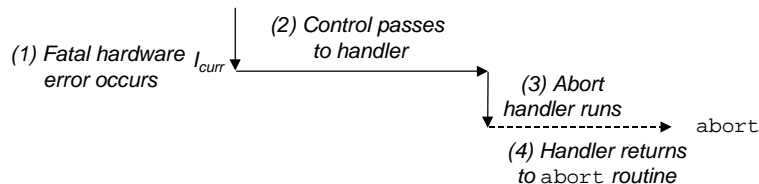


Figure 8.8: **Abort handling.** The abort handler passes control to a kernel `abort` routine that terminates the application program.

8.1.3 Exceptions in Linux/IA32 Systems

To help make things more concrete, let's look at some of the exceptions defined for IA32 systems. There are up to 256 different exception types [27]. Numbers in the range from 0 to 31 correspond to exceptions

that are defined by the Intel architects, and thus are identical for any IA32 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.

Exception Number	Description	Exception Class
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32–127	OS-defined exceptions	Interrupt or trap
128 (0x80)	System call	Trap
129–255	OS-defined exceptions	Interrupt or trap

Figure 8.9: **Examples of exceptions in IA32 systems.**

Linux/IA32 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 physical memory on disk into a page of virtual 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/IA32 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, or creating a new process. Figure 8.10 shows some of the more popular Linux system calls. Each system call has a unique integer number that corresponds to an offset in a jump table in the kernel.

System calls are provided on IA32 systems via a trapping instruction called `int n`, where n can be the index of any of the 256 entries in the IA32 exception table. Historically, system calls are provided through exception 128 (0x80).

Number	Name	Description	Number	Name	Description
1	exit	Terminate process	27	alarm	Set signal delivery alarm clock
2	fork	Create new process	29	pause	Suspend process until signal arrives
3	read	Read file	37	kill	Send signal to another process
4	write	Write file	48	signal	Install signal handler
5	open	Open file	63	dup2	Copy file descriptor
6	close	Close file	64	getppid	Get parent's process ID
7	waitpid	Wait for child to terminate	65	getpgrp	Get process group
11	execve	Load and run program	67	sigaction	Install portable signal handler
19	lseek	Go to file offset	90	mmap	Map memory page to file
20	getpid	Get process ID	106	stat	Get information about file

Figure 8.10: **Examples of popular system calls in Linux/IA32 systems.** Linux provides hundreds of system calls. Source: `/usr/include/sys/syscall.h`.

C programs can invoke any system call directly by using the `syscall` function. However, this is rarely necessary in practice. The standard C 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 number, 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*.

It is quite interesting to study how programs can use the `int` instruction to invoke Linux systems calls directly. All parameters to Linux system calls are passed through general purpose registers rather than the stack. By convention, register `%eax` contains the syscall number, and registers `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, and `%ebp` contain up to six arbitrary arguments. The stack pointer `%esp` cannot be used because it is overwritten by the kernel when it enters kernel mode.

For example, consider the following version of the familiar `hello` program, written using the `write` system-level function:

```

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.

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

Aside: A note on terminology.

The terminology for the various classes of exceptions varies from system to system. Processor macroarchitecture specifications often distinguish between asynchronous “interrupts” and synchronous “exceptions,” yet provide no

code/ecf/hello-asm.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   movl $4, %eax           System call number 4
10  movl $1, %ebx           stdout has descriptor 1
11  movl $string, %ecx      Hello world string
12  movl $len, %edx        String length
13  int $0x80              System call code

   Next, call exit(0)
14  movl $1, %eax           System call number 0
15  movl $0, %ebx          Argument is 0
16  int $0x80              System call code
```

code/ecf/hello-asm.sa

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

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. **End Aside.**

8.2 Processes

Exceptions are the basic building blocks that allow the operating system 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 a process. In the example, the execution of the three logical flows is interleaved. Process A runs for a while, followed by B, which runs to completion. Process C then runs for awhile, followed by A, which runs to completion. Finally, C is able to run to completion.

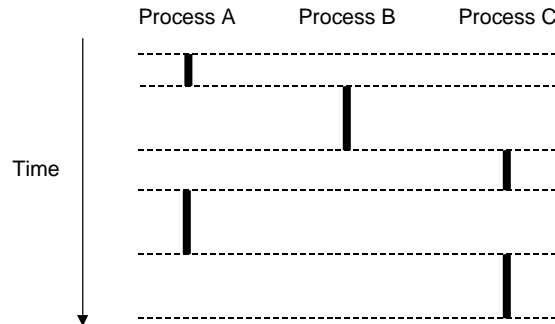


Figure 8.12: **Logical control flows.** Processes provide each program with the illusion that it has exclusive use of the processor. Each vertical bar represents a portion of the logical control flow for a process.

The key point in Figure 8.12 is that processes take turns using the processor. Each process executes a portion of its flow and then is *preempted* (temporarily suspended) while other processes take their turns. To a program running in the context of one of these processes, it appears to have exclusive use of the processor. The only evidence to the contrary is that if we were to precisely measure the elapsed time of each instruction, we would notice that the CPU appears to periodically stall between the execution of some of the instructions in our program. However, each time the processor stalls, it subsequently resumes execution of our program without any change to the contents of the program's memory locations or registers.

8.2.2 Concurrent Flows

Logical flows take many different forms in computer systems. Exception handlers, processes, signal handlers, threads, and Java processes are all examples of logical flows.

A logical flow whose execution overlaps in time with another flow is called a *concurrent flow*, and the two flows are said to *run concurrently*. More precisely, flows X and Y are concurrent with respect to each other if and only if X begins after Y begins and before Y finishes, or Y begins after X begins and before X finishes. For example, in Figure 8.12, processes A and B run concurrently, as do A and C. On the other hand, B and C do not run concurrently, because the last instruction of B executes before the first instruction of C.

The general phenomenon of multiple flows executing concurrently is known as *concurrency*. The notion of a process taking turns with other processes is also known as *multitasking*. Each time period that a process executes a portion of its flow is called a *time slice*. Thus, multitasking is also referred to as *time slicing*. For example, in Figure 8.12, the flow for process A consists of two time slices.

Notice that the idea of concurrent flows is independent of the number of processor cores or computers that the flows are running on. If two flows overlap in time, then they are concurrent, even if they are running on the same processor. However, we will sometimes find it useful to identify a proper subset of concurrent flows

known as *parallel flows*. If two flows are running concurrently on different processor cores or computers, then we say that they are *parallel flows*, that they are *running in parallel*, and have *parallel execution*.

Practice Problem 8.1:

Consider three processes with the following starting and ending times:

Process	Start time	End time
A	0	2
B	1	4
C	3	5

For each pair of processes, indicate whether they run concurrently (y) or not (n):

Process pair	Concurrent?
AB	
AC	
BC	

8.2.3 Private Address Space

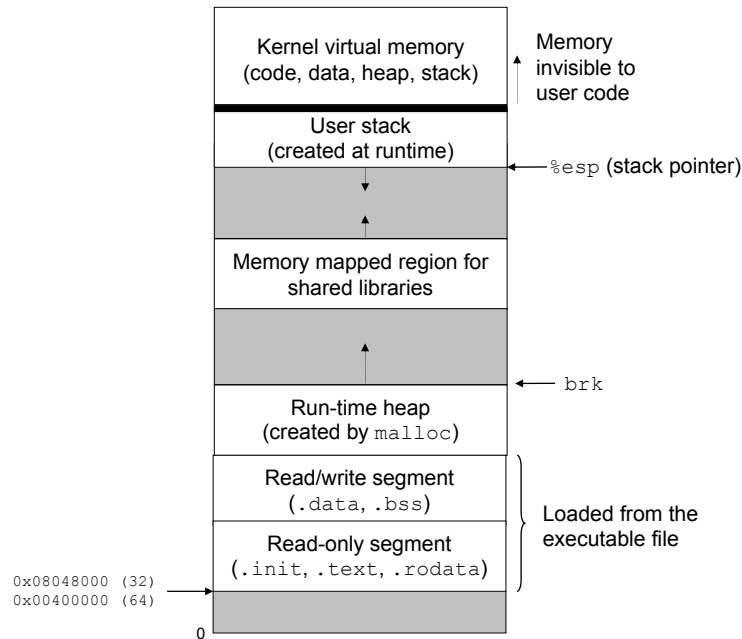
A process provides each program with the illusion that it has exclusive use of the system's address space. On a machine with n -bit addresses, the *address space* is the set of 2^n possible addresses, $0, 1, \dots, 2^n - 1$. A process provides each program with its own *private address space*. This space is private in the sense that a byte of memory associated with a particular address in the space cannot in general be read or written by any other process.

Although the contents of the memory associated with each private address space is different in general, each such space has the same general organization. For example, Figure 8.13 shows the organization of the address space for an x86 Linux process. The bottom portion of the address space is reserved for the user program, with the usual text, data, heap, and stack segments. Code segments begin at address 0×08048000 for 32-bit processes, and at address 0×00400000 for 64-bit processes. The top portion of the address space is reserved for the kernel. This part of the address space contains the code, data, and stack that the kernel uses when it executes instructions on behalf of the process (e.g., when the application program executes a system call).

8.2.4 User and Kernel Modes

In order for the operating system kernel to provide an airtight process abstraction, the processor must provide a mechanism that restricts the instructions that an application can execute, as well as the portions of the address space that it can access.

Processors typically provide this capability with a *mode bit* in some control register that characterizes the privileges that the process currently enjoys. When the mode bit is set, the process is running in *kernel mode*

Figure 8.13: **Process address space.**

(sometimes called *supervisor mode*). A process running in kernel mode can execute any instruction in the instruction set and access any memory location in the system.

When the mode bit is not set, the process is running in *user mode*. A process in user mode is not allowed to execute *privileged instructions* that do things such as halt the processor, change the mode bit, or initiate an I/O operation. Nor is it allowed to directly reference code or data in the kernel area of the address space. Any such attempt results in a fatal protection fault. User programs must instead access kernel code and data indirectly via the system call interface.

A process running application code is initially in user mode. The only way for the process to change from user mode to kernel mode is via an exception such as an interrupt, a fault, or a trapping system call. When the exception occurs, and control passes to the exception handler, the processor changes the mode from user mode to kernel mode. The handler runs in kernel mode. When it returns to the application code, the processor changes the mode from kernel mode back to user mode.

Linux provides a clever mechanism, called the `/proc` filesystem, that allows user mode processes to access the contents of kernel data structures. The `/proc` filesystem exports the contents of many kernel data structures as a hierarchy of text files that can be read by user programs. For example, you can use the `/proc` filesystem to find out general system attributes such as CPU type (`/proc/cpuinfo`), or the memory segments used by a particular process (`/proc/<process id>/maps`). The 2.6 version of the Linux kernel introduced a `/sys` filesystem, which exports additional low-level information about system buses and devices.

8.2.5 Context Switches

The operating system kernel implements multitasking using a higher-level form of exceptional control flow known as a *context switch*. The context switch mechanism is built on top of the lower-level exception mechanism that we discussed in Section 8.1.

The kernel maintains a *context* for each process. The context is the state that the kernel needs to restart a preempted process. It consists of the values of objects such as the general purpose registers, the floating-point registers, the program counter, user's stack, status registers, kernel's stack, and various kernel data structures such as a *page table* that characterizes the address space, a *process table* that contains information about the current process, and a *file table* that contains information about the files that the process has opened.

At certain points during the execution of a process, the kernel can decide to preempt the current process and restart a previously preempted process. This decision is known as *scheduling*, and is handled by code in the kernel called the *scheduler*. When the kernel selects a new process to run, we say that the kernel has *scheduled* that process. After the kernel has scheduled a new process to run, it preempts the current process and transfers control to the new process using a mechanism called a *context switch* that (1) saves the context of the current process, (2) restores the saved context of some previously preempted process, and (3) passes control to this newly restored process.

A context switch can occur while the kernel is executing a system call on behalf of the user. If the system call blocks because it is waiting for some event to occur, then the kernel can put the current process to sleep and switch to another process. For example, if a `read` system call requires a disk access, the kernel can opt to perform a context switch and run another process instead of waiting for the data to arrive from the disk. Another example is the `sleep` system call, which is an explicit request to put the calling process to sleep. In general, even if a system call does not block, the kernel can decide to perform a context switch rather than return control to the calling process.

A context switch can also occur as a result of an interrupt. For example, all systems have some mechanism for generating periodic timer interrupts, typically every 1 ms or 10 ms. Each time a timer interrupt occurs, the kernel can decide that the current process has run long enough and switch to a new process.

Figure 8.14 shows an example of context switching between a pair of processes A and B. In this example, initially process A is running in user mode until it traps to the kernel by executing a `read` system call. The trap handler in the kernel requests a DMA transfer from the disk controller and arranges for the disk to interrupt the processor after the disk controller has finished transferring the data from disk to memory.

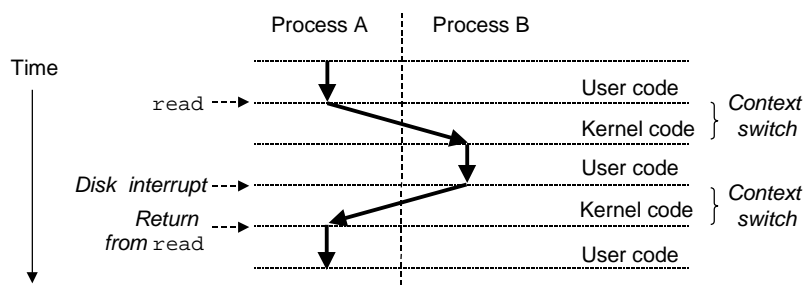


Figure 8.14: **Anatomy of a process context switch.**