

## 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.

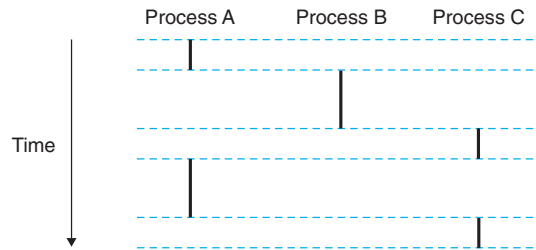
### 8.2.1 Logical Control Flow

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

**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.



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 a while, followed by A, which runs to completion. Finally, C is able to run to completion.

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** (solution page 831)

Consider three processes with the following starting and ending times:

Process	Start time	End time
A	1	3
B	2	5
C	4	6

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

Process pair	Concurrent?
AB	<input type="text"/>
AC	<input type="text"/>
BC	<input type="text"/>

**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.

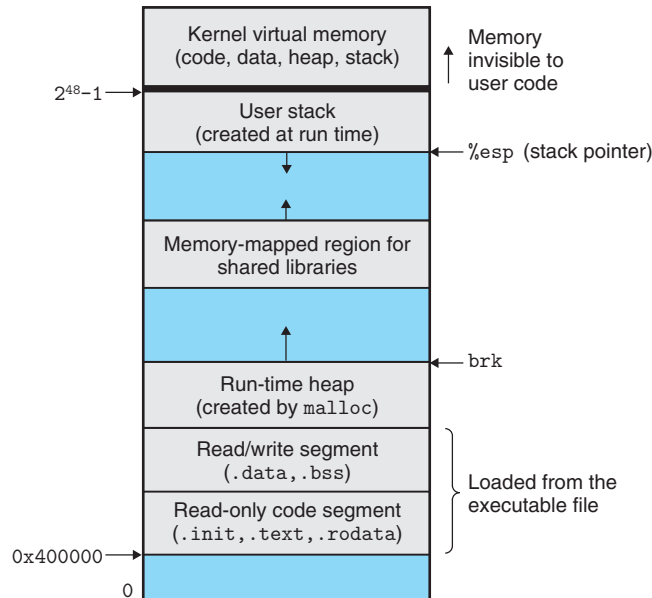
Although the contents of the memory associated with each private address space is different in general, each such space has the same general organization. For example, Figure 8.13 shows the organization of the address space for an x86-64 Linux process.

The bottom portion of the address space is reserved for the user program, with the usual code, data, heap, and stack segments. The code segment always begins at address `0x400000`. The top portion of the address space is reserved for the kernel (the memory-resident part of the operating system). 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

**Figure 8.13**  
**Process address space.**



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* (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.

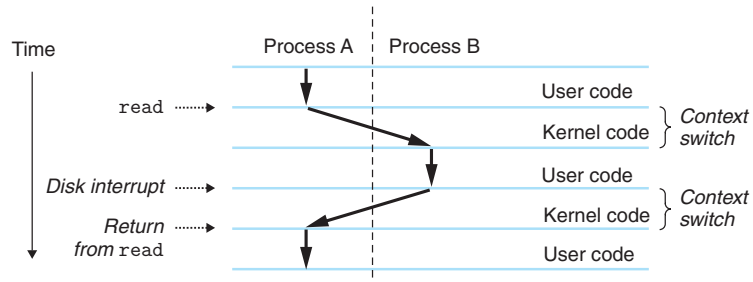
At certain points during the execution of a process, the kernel can decide to preempt the current process and restart a previously preempted process. This decision is known as *scheduling* and is handled by code in the kernel, called the *scheduler*. When the kernel selects a new process to run, we say 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

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



processor after the disk controller has finished transferring the data from disk to memory.

The disk will take a relatively long time to fetch the data (on the order of tens of milliseconds), so instead of waiting and doing nothing in the interim, the kernel performs a context switch from process A to B. Note that, before the switch, the kernel is executing instructions in user mode on behalf of process A (i.e., there is no separate kernel process). During the first part of the switch, the kernel is executing instructions in kernel mode on behalf of process A. Then at some point it begins executing instructions (still in kernel mode) on behalf of process B. And after the switch, the kernel is executing instructions in user mode on behalf of process B.

Process B then runs for a while in user mode until the disk sends an interrupt to signal that data have been transferred from disk to memory. The kernel decides that process B has run long enough and performs a context switch from process B to A, returning control in process A to the instruction immediately following the read system call. Process A continues to run until the next exception occurs, and so on.

### 8.3 System Call Error Handling

When Unix system-level functions encounter an error, they typically return `-1` and set the global integer variable `errno` to indicate what went wrong. Programmers should *always* check for errors, but unfortunately, many skip error checking because it bloats the code and makes it harder to read. For example, here is how we might check for errors when we call the Linux `fork` function:

```
1     if ((pid = fork()) < 0) {
2         fprintf(stderr, "fork error: %s\n", strerror(errno));
3         exit(0);
4     }
```

The `strerror` function returns a text string that describes the error associated with a particular value of `errno`. We can simplify this code somewhat by defining the following *error-reporting function*: