Chapter 1 A Tour of Computer Systems

CPU registers hold words retrieved from cache memory.

L1 cache holds cache lines retrieved from L2 cache.

L2 cache holds cache lines retrieved from L3 cache.

L3 cache holds cache lines retrieved from memory.

Main memory holds disk blocks retrieved from local disks.

Local disks hold files retrieved from disks on remote network server.

Figure 1.9 An example of a memory hierarchy.

Just as programmers can exploit knowledge of the different caches to improve performance, programmers can exploit their understanding of the entire memory hierarchy. Chapter 6 will have much more to say about this.

1.7 The Operating System Manages the Hardware

Back to our hello example. When the shell loaded and ran the hello program, and when the hello program printed its message, neither program accessed the keyboard, display, disk, or main memory directly. Rather, they relied on the services provided by the operating system. We can think of the operating system as a layer of software interposed between the application program and the hardware, as shown in Figure 1.10. All attempts by an application program to manipulate the hardware must go through the operating system.

The operating system has two primary purposes: (1) to protect the hardware from misuse by runaway applications, and (2) to provide applications with simple and uniform mechanisms for manipulating complicated and often wildly different low-level hardware devices. The operating system achieves both goals via the

Figure 1.10 Layered view of a computer system.
fundamental abstractions shown in Figure 1.11: processes, virtual memory, and files. As this figure suggests, files are abstractions for I/O devices, virtual memory is an abstraction for both the main memory and disk I/O devices, and processes are abstractions for the processor, main memory, and I/O devices. We will discuss each in turn.

Aside  Unix and Posix

The 1960s was an era of huge, complex operating systems, such as IBM’s OS/360 and Honeywell’s Multics systems. While OS/360 was one of the most successful software projects in history, Multics dragged on for years and never achieved wide-scale use. Bell Laboratories was an original partner in the Multics project, but dropped out in 1969 because of concern over the complexity of the project and the lack of progress. In reaction to their unpleasant Multics experience, a group of Bell Labs researchers—Ken Thompson, Dennis Ritchie, Doug McIlroy, and Joe Ossanna—began work in 1969 on a simpler operating system for a DEC PDP-7 computer, written entirely in machine language. Many of the ideas in the new system, such as the hierarchical file system and the notion of a shell as a user-level process, were borrowed from Multics but implemented in a smaller, simpler package. In 1970, Brian Kernighan dubbed the new system “Unix” as a pun on the complexity of “Multics.” The kernel was rewritten in C in 1973, and Unix was announced to the outside world in 1974 [89].

Because Bell Labs made the source code available to schools with generous terms, Unix developed a large following at universities. The most influential work was done at the University of California at Berkeley in the late 1970s and early 1980s, with Berkeley researchers adding virtual memory and the Internet protocols in a series of releases called Unix 4.xBSD (Berkeley Software Distribution). Concurrently, Bell Labs was releasing their own versions, which became known as System V Unix. Versions from other vendors, such as the Sun Microsystems Solaris system, were derived from these original BSD and System V versions.

Trouble arose in the mid 1980s as Unix vendors tried to differentiate themselves by adding new and often incompatible features. To combat this trend, IEEE (Institute for Electrical and Electronics Engineers) sponsored an effort to standardize Unix, later dubbed “Posix” by Richard Stallman. The result was a family of standards, known as the Posix standards, that cover such issues as the C language interface for Unix system calls, shell programs and utilities, threads, and network programming. As more systems comply more fully with the Posix standards, the differences between Unix versions are gradually disappearing.
1.7.1 Processes

When a program such as hello runs on a modern system, the operating system provides the illusion that the program is the only one running on the system. The program appears to have exclusive use of both the processor, main memory, and I/O devices. The processor appears to execute the instructions in the program, one after the other, without interruption. And the code and data of the program appear to be the only objects in the system’s memory. These illusions are provided by the notion of a process, one of the most important and successful ideas in computer science.

A process is the operating system’s abstraction for a running program. Multiple processes can run concurrently on the same system, and each process appears to have exclusive use of the hardware. By concurrently, we mean that the instructions of one process are interleaved with the instructions of another process. In most systems, there are more processes to run than there are CPUs to run them. Traditional systems could only execute one program at a time, while newer multicore processors can execute several programs simultaneously. In either case, a single CPU can appear to execute multiple processes concurrently by having the processor switch among them. The operating system performs this interleaving with a mechanism known as context switching. To simplify the rest of this discussion, we consider only a uniprocessor system containing a single CPU. We will return to the discussion of multiprocessor systems in Section 1.9.1.

The operating system keeps track of all the state information that the process needs in order to run. This state, which is known as the context, includes information such as the current values of the PC, the register file, and the contents of main memory. At any point in time, a uniprocessor system can only execute the code for a single process. When the operating system decides to transfer control from the current process to some new process, it performs a context switch by saving the context of the current process, restoring the context of the new process, and then passing control to the new process. The new process picks up exactly where it left off. Figure 1.12 shows the basic idea for our example hello scenario.

There are two concurrent processes in our example scenario: the shell process and the hello process. Initially, the shell process is running alone, waiting for input on the command line. When we ask it to run the hello program, the shell carries

![Figure 1.12 Process context switching.](image-url)
out our request by invoking a special function known as a system call that passes control to the operating system. The operating system saves the shell’s context, creates a new hello process and its context, and then passes control to the new hello process. After hello terminates, the operating system restores the context of the shell process and passes control back to it, where it waits for the next command line input.

Implementing the process abstraction requires close cooperation between both the low-level hardware and the operating system software. We will explore how this works, and how applications can create and control their own processes, in Chapter 8.

1.7.2 Threads

Although we normally think of a process as having a single control flow, in modern systems a process can actually consist of multiple execution units, called threads, each running in the context of the process and sharing the same code and global data. Threads are an increasingly important programming model because of the requirement for concurrency in network servers, because it is easier to share data between multiple threads than between multiple processes, and because threads are typically more efficient than processes. Multi-threading is also one way to make programs run faster when multiple processors are available, as we will discuss in Section 1.9.1. You will learn the basic concepts of concurrency, including how to write threaded programs, in Chapter 12.

1.7.3 Virtual Memory

Virtual memory is an abstraction that provides each process with the illusion that it has exclusive use of the main memory. Each process has the same uniform view of memory, which is known as its virtual address space. The virtual address space for Linux processes is shown in Figure 1.13. (Other Unix systems use a similar layout.) In Linux, the topmost region of the address space is reserved for code and data in the operating system that is common to all processes. The lower region of the address space holds the code and data defined by the user’s process. Note that addresses in the figure increase from the bottom to the top.

The virtual address space seen by each process consists of a number of well-defined areas, each with a specific purpose. You will learn more about these areas later in the book, but it will be helpful to look briefly at each, starting with the lowest addresses and working our way up:

- Program code and data. Code begins at the same fixed address for all processes, followed by data locations that correspond to global C variables. The code and data areas are initialized directly from the contents of an executable object file, in our case the hello executable. You will learn more about this part of the address space when we study linking and loading in Chapter 7.
• **Heap.** The code and data areas are followed immediately by the run-time heap. Unlike the code and data areas, which are fixed in size once the process begins running, the heap expands and contracts dynamically at run time as a result of calls to C standard library routines such as `malloc` and `free`. We will study heaps in detail when we learn about managing virtual memory in Chapter 9.

• **Shared libraries.** Near the middle of the address space is an area that holds the code and data for shared libraries such as the C standard library and the math library. The notion of a shared library is a powerful but somewhat difficult concept. You will learn how they work when we study dynamic linking in Chapter 7.

• **Stack.** At the top of the user's virtual address space is the user stack that the compiler uses to implement function calls. Like the heap, the user stack expands and contracts dynamically during the execution of the program. In particular, each time we call a function, the stack grows. Each time we return from a function, it contracts. You will learn how the compiler uses the stack in Chapter 3.

• **Kernel virtual memory.** The kernel is the part of the operating system that is always resident in memory. The top region of the address space is reserved for the kernel. Application programs are not allowed to read or write the contents of this area or to directly call functions defined in the kernel code.

For virtual memory to work, a sophisticated interaction is required between the hardware and the operating system software, including a hardware translation of every address generated by the processor. The basic idea is to store the contents...
of a process’s virtual memory on disk, and then use the main memory as a cache for the disk. Chapter 9 explains how this works and why it is so important to the operation of modern systems.

1.7.4 Files

A file is a sequence of bytes, nothing more and nothing less. Every I/O device, including disks, keyboards, displays, and even networks, is modeled as a file. All input and output in the system is performed by reading and writing files, using a small set of system calls known as Unix I/O.

This simple and elegant notion of a file is nonetheless very powerful because it provides applications with a uniform view of all of the varied I/O devices that might be contained in the system. For example, application programmers who manipulate the contents of a disk file are blissfully unaware of the specific disk technology. Further, the same program will run on different systems that use different disk technologies. You will learn about Unix I/O in Chapter 10.

Aside  The Linux project

In August 1991, a Finnish graduate student named Linus Torvalds modestly announced a new Unix-like operating system kernel:

From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Date: 25 Aug 91 20:57:08 GMT

Hello everybody out there using minix -
I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since April, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus (torvalds@kruuna.helsinki.fi)
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
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. 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.

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.
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:

<table>
<thead>
<tr>
<th>Process</th>
<th>Start time</th>
<th>End time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Process pair</th>
<th>Concurrent?</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td></td>
</tr>
<tr>
<td>AC</td>
<td></td>
</tr>
<tr>
<td>BC</td>
<td></td>
</tr>
</tbody>
</table>

**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, $0, 1, \ldots, 2^n − 1$. 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 0x08048000 for 32-bit processes, and at address 0x00400000 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 (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.

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. 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 has 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.

Aside  Cache pollution and exceptional control flow

In general, hardware cache memories do not interact well with exceptional control flows such as
interrupts and context switches. If the current process is interrupted briefly by an interrupt, then the
cache is cold for the interrupt handler. If the handler accesses enough items from main memory, then
the cache will also be cold for the interrupted process when it resumes. In this case, we say that the
handler has polluted the cache. A similar phenomenon occurs with context switches. When a process
resumes after a context switch, the cache is cold for the application program and must be warmed up
again.

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. Program-
mers 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     }
```