# Chapter 1

# A Tour of Computer Systems

A *computer system* consists of hardware and systems software that work together to run application programs. Specific implementations of systems change over time, but the underlying concepts do not. All computer systems have similar hardware and software components that perform similar functions. This book is written for programmers who want to get better at their craft by understanding how these components work and how they affect the correctness and performance of their programs.

You are poised for an exciting journey. If you dedicate yourself to learning the concepts in this book, then you will be on your way to becoming a rare "power programmer," enlightened by an understanding of the underlying computer system and its impact on your application programs.

You are going to learn practical skills such as how to avoid strange numerical errors caused by the way that computers represent numbers. You will learn how to optimize your C code by using clever tricks that exploit the designs of modern processors and memory systems. You will learn how the compiler implements procedure calls and how to use this knowledge to avoid the security holes from buffer overflow bugs that plague network and Internet software. You will learn how to recognize and avoid the nasty errors during linking that confound the average programmer. You will learn how to write your own Unix shell, your own dynamic storage allocation package, and even your own Web server!

In their classic text on the C programming language [40], Kernighan and Ritchie introduce readers to C using the `hello` program shown in Figure 1.1. Although `hello` is a very simple program, every major

*code/intro/hello.c*

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("hello, world\n");
6 }
```

*code/intro/hello.c*

Figure 1.1: **The `hello` program.**

part of the system must work in concert in order for it to run to completion. In a sense, the goal of this book is to help you understand what happens and why, when you run `hello` on your system.

We begin our study of systems by tracing the lifetime of the `hello` program, from the time it is created by a programmer, until it runs on a system, prints its simple message, and terminates. As we follow the lifetime of the program, we will briefly introduce the key concepts, terminology, and components that come into play. Later chapters will expand on these ideas.

## 1.1 Information is Bits + Context

Our `hello` program begins life as a *source program* (or *source file*) that the programmer creates with an editor and saves in a text file called `hello.c`. The source program is a sequence of bits, each with a value of 0 or 1, organized in 8-bit chunks called *bytes*. Each byte represents some text character in the program.

Most modern systems represent text characters using the ASCII standard that represents each character with a unique byte-sized integer value. For example, Figure 1.2 shows the ASCII representation of the `hello.c` program.

```
 #   i    n     c   l    u    d    e   <sp>   <   s    t    d    i    o     .
 35  105  110   99  108  117  100  101   32   60  115  116  100  105  111   46

h     >    \n   \n  i    n    t   <sp>  m     a   i    n     (    )    \n  {
104   62   10   10  105  110  116   32  109   97  105  110   40   41   10  123

 \n <sp> <sp> <sp> <sp>  p    r    i    n    t    f     (    "   h    e    l
 10   32   32   32   32  112  114  105  110  116  102   40   34  104  101  108

l    o     ,  <sp>  w    o    r    l    d     \   n     "    )    ;    \n  }
108  111   44   32  119  111  114  108  100   92  110   34   41   59   10  125
```

Figure 1.2: **The ASCII text representation of `hello.c`.**

The `hello.c` program is stored in a file as a sequence of bytes. Each byte has an integer value that corresponds to some character. For example, the first byte has the integer value 35, which corresponds to the character '`#`'. The second byte has the integer value 105, which corresponds to the character '`i`', and so on. Notice that each text line is terminated by the invisible *newline* character '`\n`', which is represented by the integer value 10. Files such as `hello.c` that consist exclusively of ASCII characters are known as *text files*. All other files are known as *binary files*.

The representation of `hello.c` illustrates a fundamental idea: All information in a system — including disk files, programs stored in memory, user data stored in memory, and data transferred across a network — is represented as a bunch of bits. The only thing that distinguishes different data objects is the context in which we view them. For example, in different contexts, the same sequence of bytes might represent an integer, floating-point number, character string, or machine instruction.

As programmers, we need to understand machine representations of numbers because they are not the same

as integers and real numbers. They are finite approximations that can behave in unexpected ways. This fundamental idea is explored in detail in Chapter 2.

**Aside: The C programming language.**
C was developed from 1969 to 1973 by Dennis Ritchie of Bell Laboratories. The American National Standards Institute (ANSI) ratified the ANSI C standard in 1989. The standard defines the C language and a set of library functions known as the *C standard library*. Kernighan and Ritchie describe ANSI C in their classic book, which is known affectionately as "K&R" [40]. In Ritchie's words [64], C is "quirky, flawed, and an enormous success." So why the success?

- *C was closely tied with the Unix operating system.* C was developed from the beginning as the system programming language for Unix. Most of the Unix kernel, and all of its supporting tools and libraries, were written in C. As Unix became popular in universities in the late 1970s and early 1980s, many people were exposed to C and found that they liked it. Since Unix was written almost entirely in C, it could be easily ported to new machines, which created an even wider audience for both C and Unix.
- *C is a small, simple language.* The design was controlled by a single person, rather than a committee, and the result was a clean, consistent design with little baggage. The K&R book describes the complete language and standard library, with numerous examples and exercises, in only 261 pages. The simplicity of C made it relatively easy to learn and to port to different computers.
- *C was designed for a practical purpose.* C was designed to implement the Unix operating system. Later, other people found that they could write the programs they wanted, without the language getting in the way.

C is the language of choice for system-level programming, and there is a huge installed base of application-level programs as well. However, it is not perfect for all programmers and all situations. C pointers are a common source of confusion and programming errors. C also lacks explicit support for useful abstractions such as classes, objects, and exceptions. Newer languages such as C++ and Java address these issues for application-level programs. **End Aside.**

## 1.2 Programs Are Translated by Other Programs into Different Forms

The `hello` program begins life as a high-level C program because it can be read and understood by human beings in that form. However, in order to run `hello.c` on the system, the individual C statements must be translated by other programs into a sequence of low-level *machine-language* instructions. These instructions are then packaged in a form called an *executable object program* and stored as a binary disk file. Object programs are also referred to as *executable object files*.

On a Unix system, the translation from source file to object file is performed by a *compiler driver*:

```
unix> gcc -o hello hello.c
```

Here, the GCC compiler driver reads the source file `hello.c` and translates it into an executable object file `hello`. The translation is performed in the sequence of four phases shown in Figure 1.3. The programs that perform the four phases (*preprocessor*, *compiler*, *assembler*, and *linker*) are known collectively as the *compilation system*.

- *Preprocessing phase.* The preprocessor (`cpp`) modifies the original C program according to directives that begin with the # character. For example, the `#include <stdio.h>` command in line 1 of `hello.c` tells the preprocessor to read the contents of the system header file `stdio.h` and insert it directly into the program text. The result is another C program, typically with the `.i` suffix.

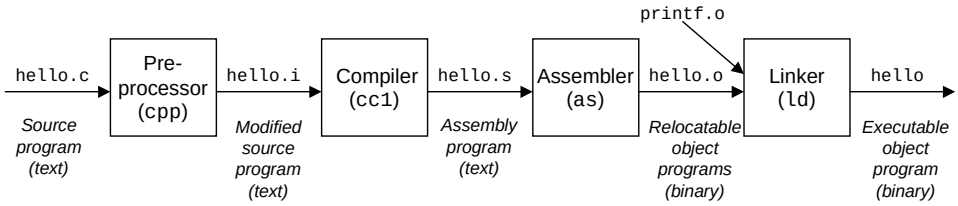


Figure 1.3: **The compilation system.**

- *Compilation phase.* The compiler (`cc1`) translates the text file `hello.i` into the text file `hello.s`, which contains an *assembly-language program*. Each statement in an assembly-language program exactly describes one low-level machine-language instruction in a standard text form. Assembly language is useful because it provides a common output language for different compilers for different high-level languages. For example, C compilers and Fortran compilers both generate output files in the same assembly language.

- *Assembly phase.* Next, the assembler (`as`) translates `hello.s` into machine-language instructions, packages them in a form known as a *relocatable object program*, and stores the result in the object file `hello.o`. The `hello.o` file is a binary file whose bytes encode machine language instructions rather than characters. If we were to view `hello.o` with a text editor, it would appear to be gibberish.

- *Linking phase.* Notice that our `hello` program calls the `printf` function, which is part of the *standard C library* provided by every C compiler. The `printf` function resides in a separate precompiled object file called `printf.o`, which must somehow be merged with our `hello.o` program. The linker (`ld`) handles this merging. The result is the `hello` file, which is an *executable object file* (or simply *executable*) that is ready to be loaded into memory and executed by the system.

**Aside: The GNU project.**
GCC is one of many useful tools developed by the GNU (short for GNU's Not Unix) project. The GNU project is a tax-exempt charity started by Richard Stallman in 1984, with the ambitious goal of developing a complete Unix-like system whose source code is unencumbered by restrictions on how it can be modified or distributed. As of 2002, the GNU project has developed an environment with all the major components of a Unix operating system, except for the kernel, which was developed separately by the Linux project. The GNU environment includes the EMACS editor, GCC compiler, GDB debugger, assembler, linker, utilities for manipulating binaries, and other components.

The GNU project is a remarkable achievement, and yet it is often overlooked. The modern open-source movement (commonly associated with Linux) owes its intellectual origins to the GNU project's notion of *free software* ("free" as in "free speech" not "free beer"). Further, Linux owes much of its popularity to the GNU tools, which provide the environment for the Linux kernel. **End Aside.**

## 1.3 It Pays to Understand How Compilation Systems Work

For simple programs such as `hello.c`, we can rely on the compilation system to produce correct and efficient machine code. However, there are some important reasons why programmers need to understand how compilation systems work:

- *Optimizing program performance.* Modern compilers are sophisticated tools that usually produce good code. As programmers, we do not need to know the inner workings of the compiler in order to write efficient code. However, in order to make good coding decisions in our C programs, we do need a basic understanding of assembly language and how the compiler translates different C statements into assembly language. For example, is a `switch` statement always more efficient than a sequence of `if-then-else` statements? Just how expensive is a function call? Is a `while` loop more efficient than a `do` loop? Are pointer references more efficient than array indexes? Why does our loop run so much faster if we sum into a local variable instead of an argument that is passed by reference? Why do two functionally equivalent loops have such different running times?

  In Chapter 3, we will introduce the Intel IA32 machine language and describe how compilers translate different C constructs into that language. In Chapter 5 you will learn how to tune the performance of your C programs by making simple transformations to the C code that help the compiler do its job. And in Chapter 6 you will learn about the hierarchical nature of the memory system, how C compilers store data arrays in memory, and how your C programs can exploit this knowledge to run more efficiently.

- *Understanding link-time errors.* In our experience, some of the most perplexing programming errors are related to the operation of the linker, especially when you are trying to build large software systems. For example, what does it mean when the linker reports that it cannot resolve a reference? What is the difference between a static variable and a global variable? What happens if you define two global variables in different C files with the same name? What is the difference between a static library and a dynamic library? Why does it matter what order we list libraries on the command line? And scariest of all, why do some linker-related errors not appear until run time? You will learn the answers to these kinds of questions in Chapter 7

- *Avoiding security holes.* For many years now, *buffer overflow bugs* have accounted for the majority of security holes in network and Internet servers. These bugs exist because too many programmers are ignorant of the stack discipline that compilers use to generate code for functions. We will describe the stack discipline and buffer overflow bugs in Chapter 3 as part of our study of assembly language.

## 1.4 Processors Read and Interpret Instructions Stored in Memory

At this point, our `hello.c` source program has been translated by the compilation system into an executable object file called `hello` that is stored on disk. To run the executable file on a Unix system, we type its name to an application program known as a *shell*:

```
unix> ./hello
hello, world
unix>
```

The shell is a command-line interpreter that prints a prompt, waits for you to type a command line, and then performs the command. If the first word of the command line does not correspond to a built-in shell command, then the shell assumes that it is the name of an executable file that it should load and run. So in this case, the shell loads and runs the `hello` program and then waits for it to terminate. The `hello`

program prints its message to the screen and then terminates. The shell then prints a prompt and waits for the next input command line.

### 1.4.1 Hardware Organization of a System

To understand what happens to our `hello` program when we run it, we need to understand the hardware organization of a typical system, which is shown in Figure 1.4. This particular picture is modeled after the family of Intel Pentium systems, but all systems have a similar look and feel. Don't worry about the complexity of this figure just now. We will get to its various details in stages throughout the course of the book.

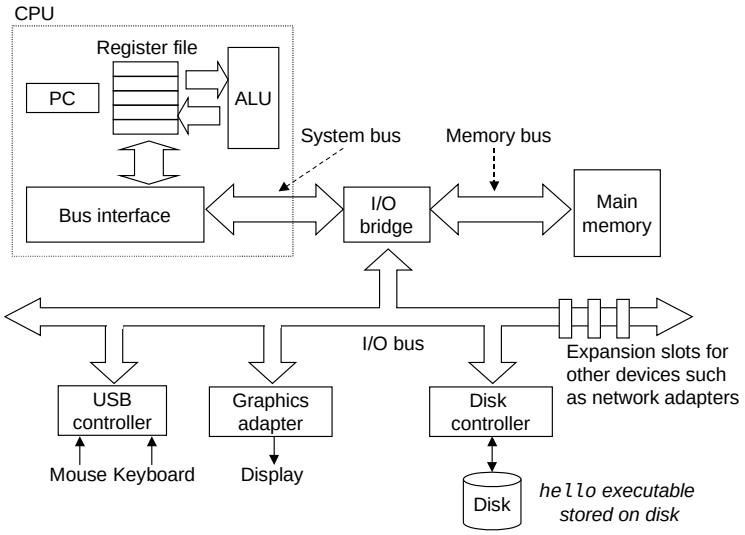


Figure 1.4: **Hardware organization of a typical system.** CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program counter, USB: Universal Serial Bus.

#### Buses

Running throughout the system is a collection of electrical conduits called *buses* that carry bytes of information back and forth between the components. Buses are typically designed to transfer fixed-sized chunks of bytes known as *words*. The number of bytes in a word (the *word size*) is a fundamental system parameter that varies across systems. For example, Intel Pentium systems have a word size of 4 bytes, while server-class systems such as Intel Itaniums and high-end Sun SPARCS have word sizes of 8 bytes. Smaller systems that are used as embedded controllers in automobiles and factories can have word sizes of 1 or 2 bytes. For simplicity, we will assume a word size of 4 bytes, and we will assume that buses transfer only one word at a time.

### I/O Devices

Input/output (I/O) devices are the system's connection to the external world. Our example system has four I/O devices: a keyboard and mouse for user input, a display for user output, and a disk drive (or simply disk) for long-term storage of data and programs. Initially, the executable `hello` program resides on the disk.

Each I/O device is connected to the I/O bus by either a *controller* or an *adapter*. The distinction between the two is mainly one of packaging. Controllers are chip sets in the device itself or on the system's main printed circuit board (often called the *motherboard*). An adapter is a card that plugs into a slot on the motherboard. Regardless, the purpose of each is to transfer information back and forth between the I/O bus and an I/O device.

Chapter 6 has more to say about how I/O devices such as disks work. In Chapter 11, you will learn how to use the Unix I/O interface to access devices from your application programs. We focus on the especially interesting class of devices known as networks, but the techniques generalize to other kinds of devices as well.

### Main Memory

The *main memory* is a temporary storage device that holds both a program and the data it manipulates while the processor is executing the program. Physically, main memory consists of a collection of *Dynamic Random Access Memory (DRAM)* chips. Logically, memory is organized as a linear array of bytes, each with its own unique address (array index) starting at zero. In general, each of the machine instructions that constitute a program can consist of a variable number of bytes. The sizes of data items that correspond to C program variables vary according to type. For example, on an Intel machine running Linux, data of type `short` requires two bytes, types `int`, `float`, and `long` four bytes, and type `double` eight bytes.

Chapter 6 has more to say about how memory technologies such as DRAM chips work, and how they are combined to form main memory.

### Processor

The *central processing unit* (CPU), or simply *processor*, is the engine that interprets (or *executes*) instructions stored in main memory. At its core is a word-sized storage device (or *register*) called the *program counter* (PC). At any point in time, the PC points at (contains the address of) some machine-language instruction in main memory. [1]

From the time that power is applied to the system, until the time that the power is shut off, the processor blindly and repeatedly performs the same basic task, over and over again: It reads the instruction from memory pointed at by the program counter (PC), interprets the bits in the instruction, performs some simple *operation* dictated by the instruction, and then updates the PC to point to the *next* instruction, which may or may not be contiguous in memory to the instruction that was just executed.

There are only a few of these simple operations, and they revolve around main memory, the *register file*, and

[1] PC is also a commonly nused acronym for "personal computer". However, the distinction between the two should be clear from the context.

the *arithmetic/logic unit* (ALU). The register file is a small storage device that consists of a collection of word-sized registers, each with its own unique name. The ALU computes new data and address values. Here are some examples of the simple operations that the CPU might carry out at the request of an instruction:

- *Load:* Copy a byte or a word from main memory into a register, overwriting the previous contents of the register.
- *Store:* Copy a byte or a word from a register to a location in main memory, overwriting the previous contents of that location.
- *Update:* Copy the contents of two registers to the ALU, which adds the two words together and stores the result in a register, overwriting the previous contents of that register.
- *I/O Read:* Copy a byte or a word from an I/O device into a register.
- *I/O Write:* Copy a byte or a word from a register to an I/O device.
- *Jump:* Extract a word from the instruction itself and copy that word into the program counter (PC), overwriting the previous value of the PC.

Chapter 4 has much more to say about how processors work.

### 1.4.2 Running the `hello` Program

Given this simple view of a system's hardware organization and operation, we can begin to understand what happens when we run our example program. We must omit a lot of details here that will be filled in later, but for now we will be content with the big picture.

Initially, the shell program is executing its instructions, waiting for us to type a command. As we type the characters "`./hello`" at the keyboard, the shell program reads each one into a register, and then stores it in memory, as shown in Figure 1.5.

When we hit the `enter` key on the keyboard, the shell knows that we have finished typing the command. The shell then loads the executable `hello` file by executing a sequence of instructions that copies the code and data in the `hello` object file from disk to main memory. The data include the string of characters "`hello, world\n`" that will eventually be printed out.

Using a technique known as *direct memory access* (DMA, discussed in Chapter 6), the data travels directly from disk to main memory, without passing through the processor. This step is shown in Figure 1.6.

Once the code and data in the `hello` object file are loaded into memory, the processor begins executing the machine-language instructions in the `hello` program's `main` routine. These instruction copy the bytes in the "`hello, world\n`" string from memory to the register file, and from there to the display device, where they are displayed on the screen. This step is shown in Figure 1.7.

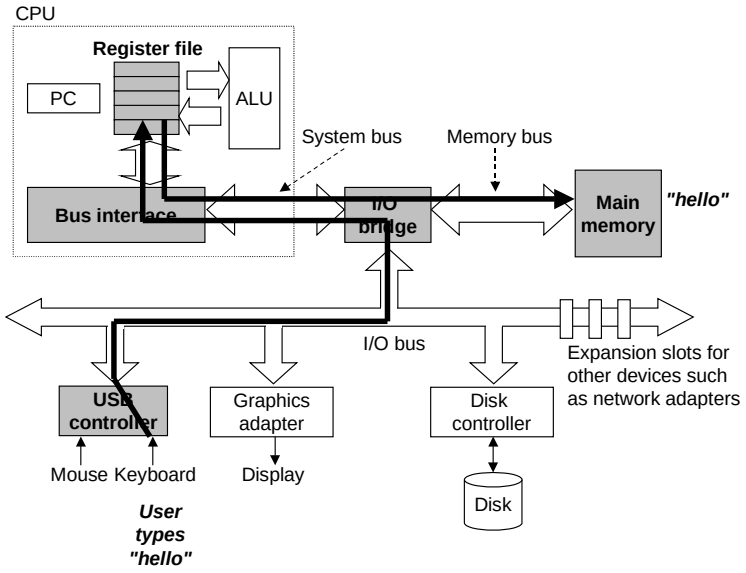


Figure 1.5: **Reading the `hello` command from the keyboard.**

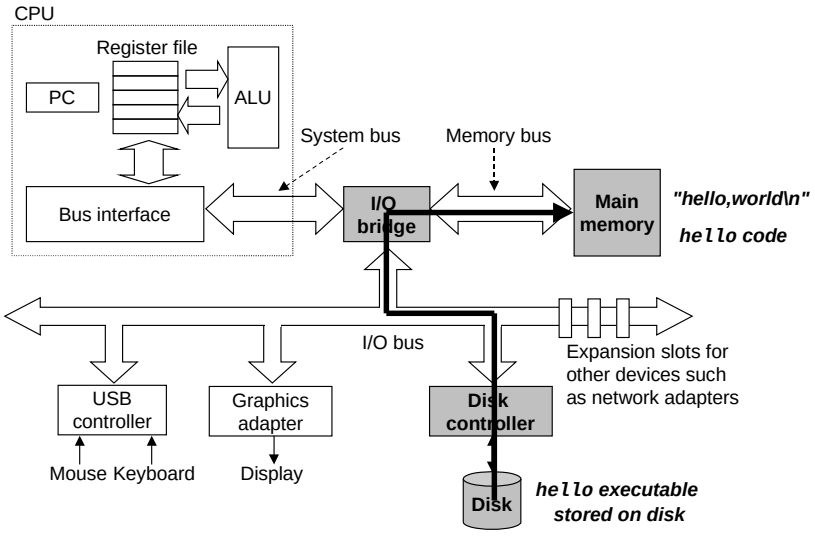


Figure 1.6: **Loading the executable from disk into main memory.**

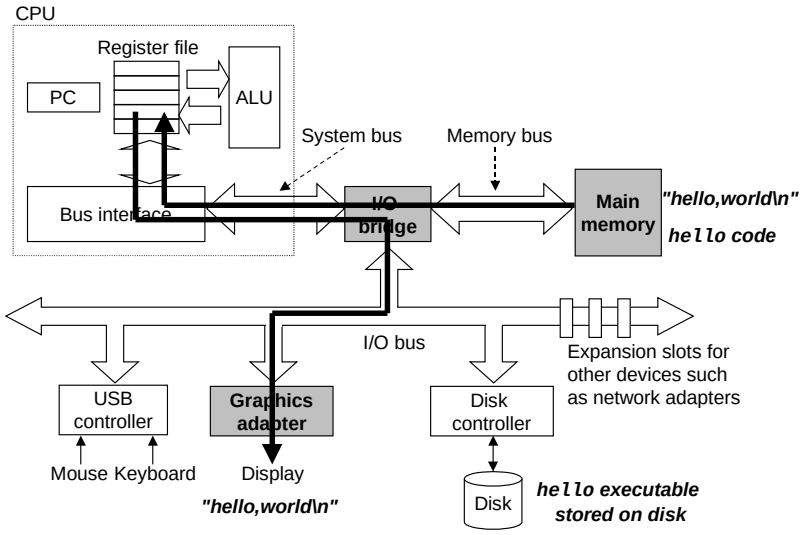


Figure 1.7: **Writing the output string from memory to the display.**

## 1.5 Caches Matter

An important lesson from this simple example is that a system spends a lot of time moving information from one place to another. The machine instructions in the `hello` program are originally stored on disk. When the program is loaded, they are copied to main memory. As the processor runs the program, instructions are copied from main memory into the processor. Similarly, the data string "`hello,world\n`", originally on disk, is copied to main memory, and then copied from main memory to the display device. From a programmer's perspective, much of this copying is overhead that slows down the "real work" of the program. Thus, a major goal for system designers is make these copy operations run as fast as possible.

Because of physical laws, larger storage devices are slower than smaller storage devices. And faster devices are more expensive to build than their slower counterparts. For example, the disk drive on a typical system might be 100 times larger than the main memory, but it might take the processor 10,000,000 times longer to read a word from disk than from memory.

Similarly, a typical register file stores only a few hundred bytes of information, as opposed to millions of bytes in the main memory. However, the processor can read data from the register file almost 100 times faster than from memory. Even more troublesome, as semiconductor technology progresses over the years, this *processor-memory gap* continues to increase. It is easier and cheaper to make processors run faster than it is to make main memory run faster.

To deal with the processor-memory gap, system designers include smaller faster storage devices called *cache memories* (or simply caches) that serve as temporary staging areas for information that the processor is likely to need in the near future. Figure 1.8 shows the cache memories in a typical system. An *L1 cache* on the processor chip holds tens of thousands of bytes and can be accessed nearly as fast as the register file. A larger *L2 cache* with hundreds of thousands to millions of bytes is connected to the processor by a special bus. It might take 5 times longer for the process to access the L2 cache than the L1 cache, but this is still 5

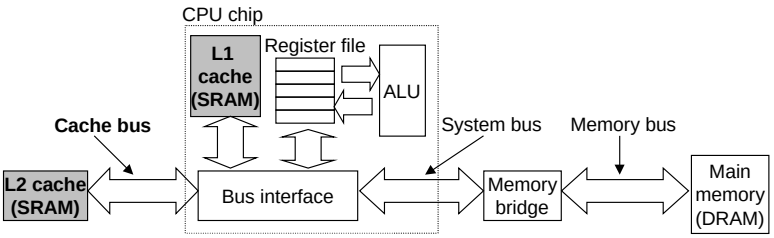


Figure 1.8: **Cache memories.**

to 10 times faster than accessing the main memory. The L1 and L2 caches are implemented with a hardware technology known as *Static Random Access Memory* (SRAM).

One of the most important lessons in this book is that application programmers who are aware of cache memories can exploit them to improve the performance of their programs by an order of magnitude. You will learn more about these important devices and how to exploit them in Chapter 6.

## 1.6 Storage Devices Form a Hierarchy

This notion of inserting a smaller, faster storage device (e.g., cache memory) between the processor and a larger slower device (e.g., main memory) turns out to be a general idea. In fact, the storage devices in every computer system are organized as a *memory hierarchy* similar to Figure 1.9. As we move from the top of

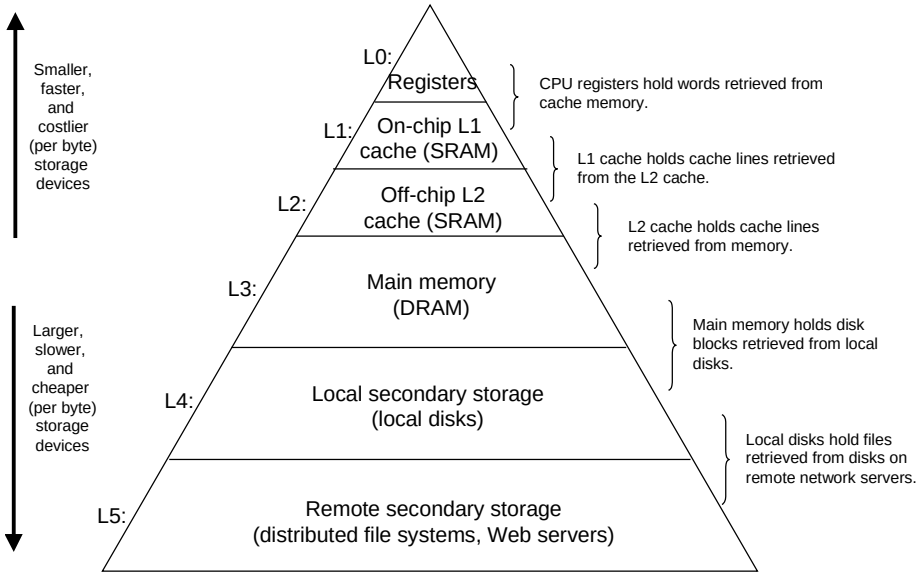


Figure 1.9: **An example of a memory hierarchy.**

the hierarchy to the bottom, the devices become slower, larger, and less costly per byte. The register file

occupies the top level in the hierarchy, which is known as level 0 or L0. The L1 cache occupies level 1 (hence the term L1). The L2 cache occupies level 2. Main memory occupies level 3, and so on.

The main idea of a memory hierarchy is that storage at one level serves as a cache for storage at the next lower level. Thus, the register file is a cache for the L1 cache, which is a cache for the L2 cache, which is a cache for the main memory, which is a cache for the disk. On some networked systems with distributed file systems, the local disk serves as a cache for data stored on the disks of other systems.

Just as programmers can exploit knowledge of the L1 and L2 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.

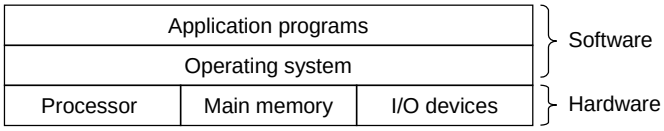


Figure 1.10: **Layered view of a computer system.**

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 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.

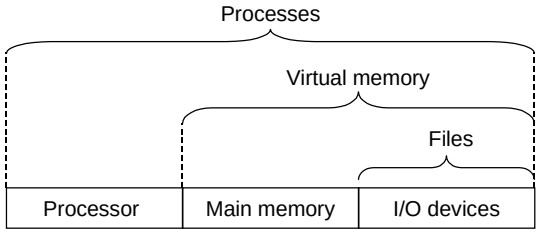


Figure 1.11: **Abstractions provided by an operating system.**

**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 [65].

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 become 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. **End Aside.**

### 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. The operating system performs this interleaving with a mechanism known as *context switching*.

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, exactly one process is running on the system. When the operating system decides to transfer control from the current process to a 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 out our request by invoking a special function known as a *system*

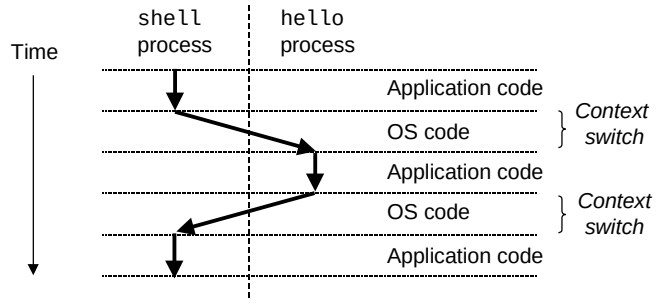


Figure 1.12: **Process context switching.**

*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.

One of the implications of the process abstraction is that by interleaving different processes, it distorts the notion of time, making it difficult for programmers to obtain accurate and repeatable measurements of running time. Chapter 9 discusses the various notions of time in a modern system and describes techniques for obtaining accurate measurements.

### 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. You will learn the basic concepts of concurrency, including threading, in Chapter 13.

### 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 one-fourth of the address space is reserved for code and data in the operating system that is common to all processes. The bottommost three-quarters of the address space holds the code and data defined by the user's process. Note that addresses in the figure increase from bottom to the top.

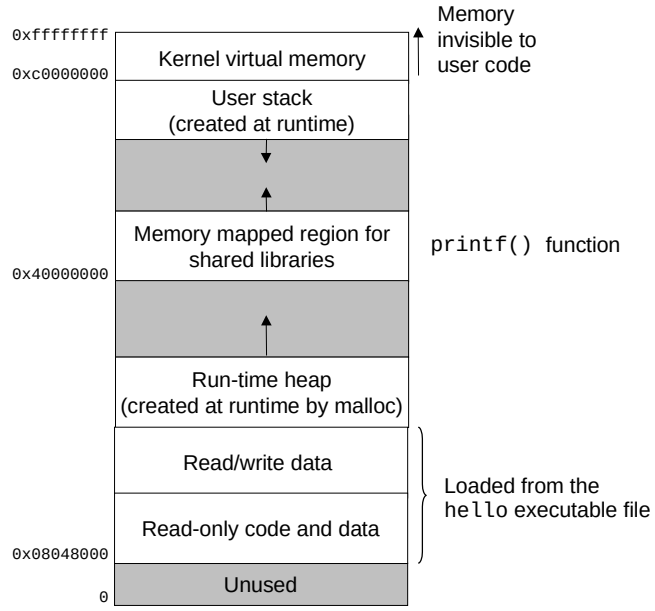


Figure 1.13: **Process virtual address space.**

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, 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 10.

- *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 1/4 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 10 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 11.

**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)
```

The rest, as they say, is history. Linux has evolved into a technical and cultural phenomenon. By combining forces with the GNU project, the Linux project has developed a complete, Posix-compliant version of the Unix operating system, including the kernel and all of the supporting infrastructure. Linux is available on a wide array of computers, from hand-held devices to mainframe computers. A group at IBM has even ported Linux to a wristwatch! **End Aside.**

## 1.8 Systems Communicate With Other Systems Using Networks

Up to this point in our tour of systems, we have treated a system as an isolated collection of hardware and software. In practice, modern systems are often linked to other systems by networks. From the point of view of an individual system, the network can be viewed as just another I/O device, as shown in Figure 1.14. When the system copies a sequence of bytes from main memory to the network adapter, the data flows across

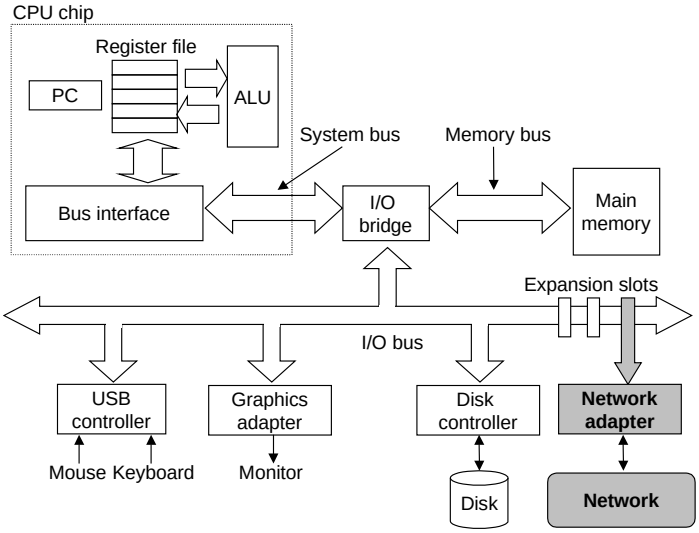


Figure 1.14: **A network is another I/O device.**

the network to another machine, instead of, say, to a local disk drive. Similarly, the system can read data sent from other machines and copy this data to its main memory.

With the advent of global networks such as the Internet, copying information from one machine to another has become one of the most important uses of computer systems. For example, applications such as email, instant messaging, the World Wide Web, FTP, and telnet are all based on the ability to copy information over a network.

Returning to our `hello` example, we could use the familiar telnet application to run `hello` on a remote machine. Suppose we use a telnet *client* running on our local machine to connect to a telnet *server* on a remote machine. After we log in to the remote machine and run a shell, the remote shell is waiting to receive an input command. From this point, running the `hello` program remotely involves the five basic steps shown in Figure 1.15.

After we type the "`hello`" string to the telnet client and hit the `enter` key, the client sends the string to the telnet server. After the telnet server receives the string from the network, it passes it along to the remote shell program. Next, the remote shell runs the `hello` program, and passes the output line back to the telnet server. Finally, the telnet server forwards the output string across the network to the telnet client, which prints the output string on our local terminal.

This type of exchange between clients and servers is typical of all network applications. In Chapter 12 you will learn how to build network applications, and apply this knowledge to build a simple Web server.

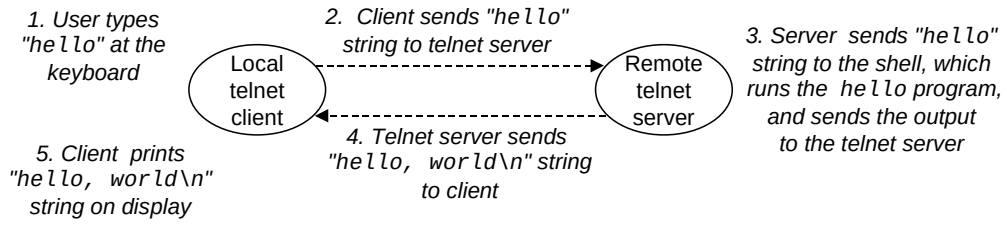


Figure 1.15: **Using telnet to run `hello` remotely over a network.**

## 1.9 The Next Step

This concludes our initial whirlwind tour of systems. An important idea to take away from this discussion is that a system is more than just hardware. It is a collection of intertwined hardware and systems software that must cooperate in order to achieve the ultimate goal of running application programs. The rest of this book will expand on this theme.

## 1.10 Summary

A computer system consists of hardware and systems software that cooperate to run application programs. Information inside the computer is represented as groups of bits that are interpreted in different ways, depending on the context. Programs are translated by other programs into different forms, beginning as ASCII text and then translated by compilers and linkers into binary executable files.

Processors read and interpret binary instructions that are stored in main memory. Since computers spend most of their time copying data between memory, I/O devices, and the CPU registers, the storage devices in a system are arranged in a hierarchy, with the CPU registers at the top, followed by multiple levels of hardware cache memories, DRAM main memory, and disk storage. Storage devices that are higher in the hierarchy are faster and more costly per bit than those lower in the hierarchy. Storage devices that are higher in the hierarchy serve as caches for devices that are lower in the hierarchy. Programmers can optimize the performance of their C programs by understanding and exploiting the memory hierarchy.

The operating system kernel serves an intermediary between the application and the hardware. It provides three fundamental abstractions: (1) Files are abstractions for I/O devices. (2) Virtual memory is an abstraction for both main memory and disks. (3) Processes are abstractions for the processor, main memory, and I/O devices.

Finally, networks provide ways for computer systems to communicate with one another. From the viewpoint of a particular system, the network is just another I/O device.

## Bibliographic Notes

Ritchie has written interesting first hand accounts of the early days of C and Unix [63, 64]. Ritchie and Thompson presented the first published account of Unix [65]. Silberschatz and Gavin [70] provide a compre-

hensive history of the different flavors of Unix. The GNU (`www.gnu.org`) and Linux (`www.linux.org`) Web pages have loads of current and historical information. Unfortunately, the Posix standards are not available online. They must be ordered for a fee from IEEE (`standards.ieee.org`).