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 vulnerabilities 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. You will learn the promises and pitfalls of concurrency, a topic of increasing importance as multiple processor cores are integrated onto single chips.

In their classic text on the C programming language [61], 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 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.

code/intro/hello.c

```
1  #include <stdio.h>
2
3  int main()
4  {
5      printf("hello, world\n");
6      return 0;
7  }
```

code/intro/hello.c

Figure 1.1 The `hello` program. (Source: [60])

이 책은 시스템 응용 프로그램을 실행하고자 하는 프로그래머들을 위해 쓰여졌습니다. 특정 구현은 시간이 지나면 변할지라도, 그 아래 개념들은 변하지 않습니다. 모든 컴퓨터 시스템은 유사한 기능을 하는 유사한 하드웨어와 소프트웨어 구성 요소를 가지고 있습니다. 이 책을 통해 프로그래머들은 이러한 구성 요소들이 어떻게 작동하며 그들의 프로그램의 정확성과 성능에 어떤 영향을 미치는지를 이해하며 자신의 기술을 향상시킬 수 있습니다. 여러분들은 흥미진진한 여정을 앞두고 있습니다. 이 책의 개념을 배우기 위해 헌신한다면, 여러분은 희귀한 "파워 프로그래머"가 되기 위한 길을 나아가게 될 것입니다. 컴퓨터 시스템과 응용 프로그램에 미치는 영향을 이해하는 것으로 깨어 있는 지식을 얻게 될 것입니다. 여러분은 컴퓨터가 숫자를 표현하는 방식에서 발생하는 이상한 수치 오류를 피하는 방법과 같은 실용적인 기술들을 배울 것입니다. 또한 현대 프로세서와 메모리 시스템의 설계를 활용한 똑똑한 트릭을 사용해 C 코드를 최적화하는 법을 배울 것입니다. 여러분은 컴파일러가 절차 호출을 어떻게 실행하며 이 지식을 활용해 네트워크 및 인터넷 소프트웨어를 괴롭히는 버퍼 오버플로 취약성을 피하는 방법을 배울 것입니다. 또한 일반적인 프로그래머를 이해에 혼란을 주는 링킹 중에 발생하는 끔찍한 오류를 인식하고 피하는 법을 배울 것입니다. 여러분은 여러분만의 Unix 셸, 동적 저장 공간 할당 패키지, 심지어 여러분만의 웹 서버를 쓰는 법을 배울 것입니다. 이제 여러 프로세서 코어가 단일 칩에 통합되는 중요한 주제로 인해, 동시성의 약속과 함정을 배울 것입니다. C 프로그래밍 언어에 관한 걸작 텍스트 [61]에서 Kernighan과 Ritchie는 그림 1.1에서 보여진 hello 프로그램을 사용하여 C에 입문하는 독자들을 소개합니다. hello는 매우 간단한 프로그램이지만, 그것이 정상적으로 실행되기 위해서는 시스템의 모든 주요 부분이 함께 작동해야 합니다. 어떤 의미에서 이 책의 목표는 여러분의 시스템에서 hello를 실행했을 때 무슨 일이 일어나는지 이해하도록 돕는 것입니다. 우리는 프로그래머에 의해 생성된 hello 프로그램이 시스템에 실행되어 간단한 메시지를 출력하고 종료될 때까지의 수명을 추적함으로써 시스템의 연구를 시작합니다. 프로그램의 수명을 따라가면서 중요한 개념, 용어 및 구성 요소들을 간단히 소개할 것입니다. 나중 장에서는 이러한 아이디어를 확장할 것입니다.

```
 #    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   SP
108  111   44   32  119  111  114  108  100   92  110   34   41   59   10   32

SP   SP   SP    r    e    t    u    r    n   SP    0    ;   \n    }   \n
32   32   32  114  101  116  117  114  110   32   48   59   10  125   10
```

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

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 computer systems represent text characters using the ASCII standard that represents each character with a unique byte-size integer value.[1] For example, Figure 1.2 shows the ASCII representation of the `hello.c` program.

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

1. Other encoding methods are used to represent text in non-English languages. See the aside on page 86 for a discussion on this.

저희 hello 프로그램은 소스 프로그램(또는 소스 파일)으로 시작합니다. 프로그래머가 편집기로 작성하고 hello.c 라는 텍스트 파일에 저장하는 것입니다. 소스 프로그램은 각각 0 또는 1의 값을 갖는 일련의 비트로 구성된 8비트 덩어리인 바이트로 구성되어 있습니다. 각 바이트는 프로그램 안의 일부 텍스트 문자를 나타냅니다. 대부분의 컴퓨터 시스템은 ASCII 표준을 사용하여 텍스트 문자를 표현합니다. ASCII는 각 문자를 고유한 바이트 크기의 정수 값으로 나타냅니다. 예를 들어 1.2 그림은 hello.c 프로그램의 ASCII 표현을 보여줍니다. hello.c 프로그램은 바이트의 연속으로 파일에 저장됩니다. 각 바이트는 문자에 대응하는 정수 값을 갖습니다. 예를 들어, 첫 번째 바이트는 문자 '#'에 대응하는 정수 값 35를 가지고 있고, 두 번째 바이트는 문자 'i'에 대응하는 정수 값 105를 가집니다. 각 텍스트 라인은 보이지 않는 새 줄 문자 '₩n'에 의해 종결됩니다. 이는 정수 값 10으로 표현됩니다. hello.c와 같이 ASCII 문자로만 구성된 파일을 텍스트 파일이라고 합니다. 그 이외의 파일들은 바이너리 파일로 알려져 있습니다. hello.c의 표현은 기본 아이디어를 보여줍니다: 시스템 안에 있는 모든 정보(디스크 파일, 메모리에 저장된 프로그램, 메모리에 저장된 사용자 데이터, 네트워크를 통해 전달되는 데이터 포함)는 일련의 비트로 나타낼 수 있습니다. 다른 데이터 객체를 구별하는 것은 우리가 그것들을 보는 맥락 뿐입니다. 예를 들어, 다른 맥락에서 같은 바이트의 연속이 정수, 부동 소수점 수, 문자열 또는 기계 명령어를 나타낼 수 있습니다. 프로그래머로써, 정수와 실수와는 다르게 숫자의 기계적 표현을 이해해야 합니다. 그것들은 유한하기 때문입니다.

다른 부호화 방법들은 영어가 아닌 언어의 텍스트를 표현하는 데 사용됩니다. 이에 대한 토론은 86페이지의 쪽지를 참조하세요.

Aside Origins of 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, and this standardization later became the responsibility of the International Standards Organization (ISO). The standards define 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" [61]. In Ritchie's words [92], 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 (the core part of the operating system), 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.

C는 1969년부터 1973년까지 벨 연구소의 데니스 리치에에 의해 개발되었습니다. 미국 국가 표준 협회(ANSI)는 1989년에 ANSI C 표준을 승인했으며, 이 표준화는 나중에 국제 표준 기구(ISO)의 책임이 되었습니다. 이 표준은 C언어와 C 표준 라이브러리라고 불리는 라이브러리 함수 집합을 정의합니다. Kernighan과 Ritchie는 그들의 고전적인 책에서 ANSI C를 묘사했으며, 이 책은 애정을 담아 "K&R;"이라고합니다[61]. 리치의 말에 따르면 [92], C는 "변덕스럽고, 흠잡을 데 없으며 거대한 성공"입니다. 그래서 왜 성공일까요?

C는 Unix 운영 체제와 밀접한 연관이 있었습니다. C는 처음부터 Unix의 시스템 프로그래밍 언어로 개발되었습니다. Unix의 대부분의 커널(운영 체제의 핵심 부분) 및 그 지원 도구와 라이브러리는 모두 C로 작성되었습니다. 1970년대 후반과 1980년대 초에 대학들에서 Unix가 인기를 얻으면서 많은 사람들이 C에 노출되었고 이를 선호했습니다. Unix가 거의 완전히 C로 작성되었기 때문에 새로운 기계로 쉽게 이식될 수 있었으며, 이는 C와 Unix에 대한 더 넓은 사용자층을 만들었습니다.

C 언어는 작고 간단한 언어입니다. 이 디자인은 위원회가 아닌 한 사람에 의해 통제되었고, 결과는 불필요한 것이 적은 깔끔하고 일관된 디자인이었습니다. K&R; 책은 단 261 페이지만에 완전한 언어와 표준 라이브러리를 다양한 예제와 연습문제와 함께 설명합니다. C의 간결함 덕분에 상대적으로 쉽게 배우고 다른 컴퓨터로 이식할 수 있었습니다.

C 언어는 실용적인 목적으로 설계되었습니다. C 언어는 유닉스 운영 체제를 구현하기 위해 설계되었습니다. 나중에 다른 사람들은 언어를 방해받지 않고 원하는 프로그램을 작성할 수 있다는 것을 발견했습니다.

C는 시스템 수준의 프로그래밍을 위한 언어로 선택되며, 응용 프로그램 수준의 프로그램 또한 방대한 설치된 기반을 가지고 있습니다. 하지만, 모든 프로그래머와 모든 상황에 완벽한 것은 아닙니다. C 포인터는 혼란과 프로그래밍 오류의 일반적인 원인입니다. 또한, C는 클래스, 객체, 예외와 같은 유용한 추상화를 명시적으로 지원하지 않습니다. C++ 및 Java와 같은 최신 언어는 응용 프로그램 수준의 프로그램을 위해 이러한 문제를 해결합니다.

approximations that can behave in unexpected ways. This fundamental idea is explored in detail in Chapter 2.

예상치 못한 방식으로 작용할 수 있는 근사치에 대한 이 기본적인 아이디어는 2장에서 자세히 다루어집니다.

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:*

"Hello" 프로그램은 사람이 그 형태에서 읽고 이해할 수 있기 때문에 고수준 C 프로그램으로 생성됩니다. 그러나 시스템에서 hello.c를 실행하기 위해서는 각각의 C 문장을 다른 프로그램들에 의해 저수준의 기계어 명령어의 일련의 순서로 번역해야 합니다. 이러한 명령어들은 그 후에 실행 가능한 객체 프로그램이라는 형태로 묶여 이진 디스크 파일로 저장됩니다. 객체 프로그램은 실행 가능한 객체 파일로도 불립니다.

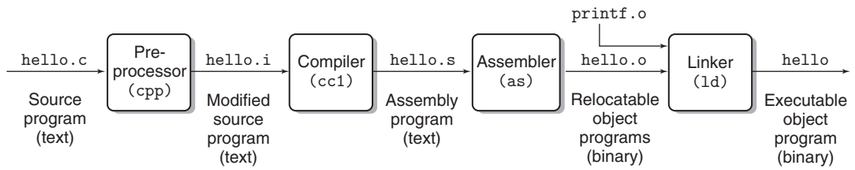


Figure 1.3 The compilation system.

```
linux> 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*.

여기서 gcc 컴파일러 드라이버는 소스 파일 hello.c를 읽고 실행 가능한 객체 파일 hello로 번역합니다. 번역은 그림 1.3에 표시된 네 단계의 순서대로 수행됩니다. 네 단계를 수행하는 프로그램(전처리기, 컴파일러, 어셈블러 및 링커)은 통틀어 컴파일 시스템이라고 합니다.

- *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.

전처리 단계. 프리프로세서(cpp)는 '# '문자로 시작하는 지시문에 따라 원래의 C 프로그램을 수정합니다. 예를 들어, hello.c의 1번 라인의 #include 명령은 프리프로세서에게 시스템 헤더 파일 stdio.h의 내용을 읽고 프로그램 텍스트에 직접 삽입하도록 지시합니다. 그 결과로 일반적으로 .i 접미사가 붙은 다른 C 프로그램이 생성됩니다.

- *Compilation phase.* The compiler (`cc1`) translates the text file `hello.i` into the text file `hello.s`, which contains an *assembly-language program*. This program includes the following definition of function `main`:

컴파일 단계. 컴파일러(cc1)는 텍스트 파일 hello.i를 어셈블리 언어 프로그램인 텍스트 파일 hello.s로 번역합니다. 이 프로그램에는 main 함수의 다음과 같은 정의가 포함되어 있습니다:

```
1  main:
2    subq    $8, %rsp
3    movl    $.LC0, %edi
4    call    puts
5    movl    $0, %eax
6    addq    $8, %rsp
7    ret
```

Each of lines 2–7 in this definition describes one low-level machine-language instruction in a textual 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.

이 정의의 2~7번 라인 각각은 텍스트 형태로 저수준 기계어 명령어를 하나씩 설명합니다. 어셈블리 언어는 다른 고급 레벨 언어용 컴파일러들에 대해 공통된 출력 언어를 제공하기 때문에 유용합니다. 예를 들어, C 컴파일러와 포트란 컴파일러는 둘 다 동일한 어셈블리 언어로 출력 파일을 생성합니다.

- *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`. This file is a binary file containing 17 bytes to encode the instructions for function `main`. If we were to view `hello.o` with a text editor, it would appear to be gibberish.

. Assembly phase. 그 다음, 어셈블러(as)는 hello.s를 기계어 명령어로 번역하고, 이를 relocatable object program 형식으로 패키지하여 object 파일인 hello.o에 저장합니다. 이 파일은 main 함수의 명령어를 인코딩하기 위한 17바이트를 포함한 2진 파일입니다. 만약 우리가 텍스트 편집기로 hello.o를 보면, 이상한 문자들로 보일 것입니다.

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. 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 GCC compiler has grown to support many different languages, with the ability to generate code for many different machines. Supported languages include C, C++, Fortran, Java, Pascal, Objective-C, and Ada.

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.

Gcc는 GNU(GNU's Not Unix의 약자) 프로젝트에서 개발된 많은 유용한 도구 중 하나입니다. GNU 프로젝트는 1984년 리처드 스톨만에 의해 시작된 비영리 단체로, 수정 또는 배포에 제한이 없는 소스 코드로 완전한 Unix와 유사한 시스템을 개발하는 것을 목표로 하고 있습니다. GNU 프로젝트는 GNU 환경을 개발했으며, 커널을 제외한 Unix 운영 체제의 주요 구성 요소를 포함합니다. 커널은 Linux 프로젝트에서 별도로 개발되었습니다. GNU 환경에는 emacs 편집기, gcc 컴파일러, gdb 디버거, 어셈블러, 링커, 이진 파일 조작을 위한 유틸리티 및 기타 구성 요소가 포함되어 있습니다. gcc 컴파일러는 다양한 언어를 지원하며 다양한 기기용 코드를 생성할 수 있는 능력을 갖추고 있습니다. C, C++, 포트란, 자바, 파스칼, Objective-C, 아다 등의 언어를 지원합니다. GNU 프로젝트는 주목할 만한 성취이지만 종종 간과되고 있습니다. 현대의 오픈소스 운동(Linux와 일반적으로 연결)은 GNU 프로젝트의 자유 소프트웨어 개념("자유"는 "무료 맥주"가 아닌 "언어의 자유")에 그 정신적 기원을 찾을 수 있습니다. 또한 GNU 도구는 Linux 커널을 위한 환경을 제공하여 Linux가 큰 인기를 얻을 수 있었습니다.

- *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.

링킹 단계. 우리의 hello 프로그램은 printf 함수를 호출한다는 것에 주목하십시오. printf 함수는 모든 C 컴파일러에서 제공되는 표준 C 라이브러리의 일부입니다. printf 함수는 printf.o라는 별도의 미리 컴파일된 객체 파일에 포함되어 있으며, 이 파일은 어떤 식으로든 우리의 hello.o 프로그램과 병합되어야 합니다. 링커(ld)가 이 병합을 처리합니다. 그 결과물은 메모리로 로딩되어 시스템에 의해 실행될 준비가 된 실행 가능한 객체 파일인 hello 파일입니다.

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:

"hello.c"와 같은 간단한 프로그램의 경우, 우리는 컴파일 시스템이 정확하고 효율적인 기계 코드를 생성할 것으로 의지할 수 있습니다. 그러나 프로그래머들이 컴파일 시스템이 어떻게 작동하는지 이해해야 하는 중요한 이유가 몇 가지 있습니다:

- *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 machine-level code and how the compiler translates different C statements into machine code. For example, is a `switch` statement always more efficient than a sequence of `if-else` statements? How much overhead is incurred by a function call? Is a `while` loop more efficient than a `for` 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? How can a function run faster when we simply rearrange the parentheses in an arithmetic expression?

. 프로그램 성능 최적화. 현대 컴파일러는 일반적으로 좋은 코드를 생성하는 정교한 도구입니다. 우리 프로그래머로서는 효율적인 코드를 작성하기 위해 컴파일러의 내부 작업을 알 필요가 없습니다. 그러나 C 프로그램에서 좋은 코딩 결정을 내리기 위해서는 기계 수준 코드와 컴파일러가 다른 C 문을 기계 코드로 어떻게 번역하는지에 대한 기본적인 이해가 필요합니다. 예를 들어, switch 문은 항상 if-else 문의 순서보다 더 효율적인가요? 함수 호출에는 얼마나 많은 오버헤드가 발생하나요? while 루프가 for 루프보다 더 효율적인가요? 포인터 참조가 배열 인덱스보다 더 효율적인가요? 왜 루프가 매개변수가 참조로 전달되는 것보다 로컬 변수에 더하면 훨씬 빠르게 실행되나요? 수식의 괄호를 단순히 재배열할 때 함수가 왜 더 빨리 실행되나요?

In Chapter 3, we introduce x86-64, the machine language of recent generations of Linux, Macintosh, and Windows computers. We describe how compilers translate different C constructs into this 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 better. 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.

제3장에서는 최근 Linux, Macintosh 및 Windows 컴퓨터의 기계어 인 x86-64를 소개합니다. 컴파일러가 다른 C 구조를 이 언어로 번역하는 방법을 설명합니다. 제5장에서는 C 프로그램의 성능을 튜닝하는 방법을 배우게 됩니다. 이는 C 코드를 간단히 변환하여 컴파일러가 더 잘 작동하도록 돕는 내용입니다. 제6장에서는 메모리 시스템의 계층적 성격, C 컴파일러가 데이터 배열을 메모리에 저장하는 방법, 그리고 C 프로그램이 이 지식을 활용하여 보다 효율적으로 실행되는 방법에 대해 배우게 됩니다.

- *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.

이해 링크 시간 오류. 우리의 경험 상, 가장 난해한 프로그래밍 오류 중 일부는 링커의 작동과 관련이 있습니다, 특히 대규모 소프트웨어 시스템을 빌드하려고 할 때. 예를 들어, 링커가 참조를 해결할 수 없다고 보고할 때 무슨 의미입니까? 정적 변수와 전역 변수의 차이는 무엇입니까? 다른 C 파일에서 같은 이름으로 두 개의 전역 변수를 정의하면 어떻게 됩니까? 정적 라이브러리와 동적 라이브러리의 차이는 무엇입니까? 커맨드 라인에서 라이브러리를 나열하는 순서가 왜 중요한가요? 그리고 가장 무서운 건, 어떤 링크 관련 오류는 실행 시간까지 나타나지 않는 이유는 무엇입니까? 이러한 종류의 질문에 대한 답을 7장에서 배우게 될 것입니다.

- *Avoiding security holes.* For many years, *buffer overflow vulnerabilities* have accounted for many of the security holes in network and Internet servers. These vulnerabilities exist because too few programmers understand the need to carefully restrict the quantity and forms of data they accept from untrusted sources. A first step in learning secure programming is to understand the consequences of the way data and control information are stored on the program stack. We cover the stack discipline and buffer overflow vulnerabilities in Chapter 3 as part of our study of assembly language. We will also learn about methods that can be used by the programmer, compiler, and operating system to reduce the threat of attack.

보안 구멍 피하기. 오랫동안 버퍼 오버플로우 취약점이 네트워크 및 인터넷 서버의 많은 보안 구멍을 차지해왔습니다. 이러한 취약점들은 너무 적은 프로그래머들이 신뢰할 수 없는 소스로부터 받는 데이터의 양과 형태를 신중하게 제한해야 한다는 필요성을 이해하지 못하기 때문에 존재합니다. 안전한 프로그래밍을 배우는 첫 번째 단계는 프로그램 스택에 데이터와 제어 정보가 저장되는 방식의 결과를 이해하는 것입니다. 저희는 어셈블리 언어 공부의 일환으로 제3장에서 스택 규칙과 버퍼 오버플로우 취약점을 다룰 것입니다. 또한, 프로그래머, 컴파일러 및 운영 체제가 사용할 수 있는 공격 위협을 줄이는 방법에 대해서도 배울 것입니다.

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:*

이 시점에서, 우리의 hello.c 소스 프로그램은 컴파일 시스템에 의해 hello라는 실행 가능한 객체 파일로 변환되었으며 디스크에 저장되어 있습니다. Unix 시스템에서 실행 가능한 파일을 실행하려면 셸이라고 하는 응용 프로그램에 파일 이름을 입력합니다:

```
linux> ./hello
hello, world
linux>
```

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

shell은 프롬프트를 출력하고 사용자가 명령 줄을 입력할 때까지 기다린 후 해당 명령을 실행하는 명령줄 해석기입니다. 명령 줄의 첫 번째 단어가 내장된 셸 명령에 해당되지 않는 경우에는 소셜

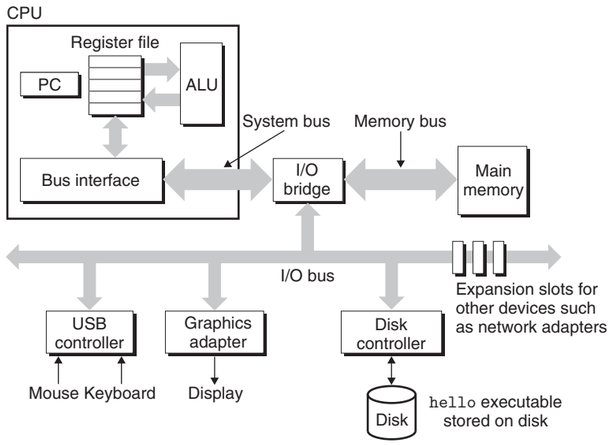


Figure 1.4
Hardware organization of a typical system. CPU: central processing unit, ALU: arithmetic/logic unit, PC: program counter, USB: Universal Serial Bus.

도 1.4 전형적인 시스템의 하드웨어 구성. CPU: 중앙처리장치, ALU: 산술/논리 장치, PC: 프로그램 카운터, USB: 유니버설 시리얼 버스.

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.

가정합니다 실행 파일의 이름을로드하고 실행해야한다. 그러므로 이 경우에는 쉘이 hello프로그램을로드하고 실행하고 종료를 기다립니다. hello프로그램은 메시지를 화면에 출력한 다음 종료됩니다. 그런 다음 쉘은 프롬프트를 표시하고 다음 입력 명령 줄을 기다립니다.

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 recent Intel 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.

저희가 hello 프로그램을 실행했을 때 어떤 일이 일어나는지 이해하려면, 보통 시스템의 하드웨어 구성을 이해해야 합니다. 이는 1.4번 그림에서 보여지고 있습니다. 이 특별한 그림은 최근 Intel 시스템과 비슷하게 모델링되었지만, 모든 시스템은 비슷한 모습과 느낌을 가지고 있습니다. 이 그림의 복잡성에 대해 지금은 걱정하지 마십시오. 책의 여러 단계를 통해 이에 대한 다양한 세부 사항을 다룰 것입니다.

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-size 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. Most machines today have word sizes of either 4 bytes (32 bits) or 8 bytes (64 bits). In this book, we do not assume any fixed definition of word size. Instead, we will specify what we mean by a "word" in any context that requires this to be defined.

시스템 전체에 걸쳐 바이트 정보를 오가게 하는 전자적 도관들의 모음인 버스가 동작하고 있다. 버스는 일반적으로 워드로 알려진 고정 크기의 바이트 청크를 전송하기 위해 설계된다. 워드 안의 바이트 숫자(워드 사이즈)는 시스템에 따라 변하는 기본 매개 변수이다. 현재 대부분의 기계는 4바이트(32비트) 또는 8바이트(64비트)의 워드 사이즈를 가지고 있다. 이 책에서는 워드 사이즈의 고정된 정의를 가정하지 않는다. 그 대신, 우리는 "워드"가 정의되어야 하는 모든 맥락에서 어떤 의미로 사용되는지를 명시할 것이다.

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 10, 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.

입출력(I/O) 장치는 시스템과 외부 세계의 연결이다. 예시 시스템은 키보드와 마우스로 사용자 입력, 사용자 출력을 위한 디스플레이, 그리고 데이터와 프로그램의 장기 보관을 위한 디스크 드라이브(또는 단순히 디스크)까지 4개의 I/O 장치를 가지고 있다. 최초에 실행 가능한 hello 프로그램은 디스크에 저장되어 있다. 각 I/O 장치는 컨트롤러나 어댑터 중 하나로 I/O 버스에 연결되어 있다. 두 가지의 차이점은 주로 구성에 있다. 컨트롤러는 장치 자체나 시스템의 주요 프린트 회로 기판(보통 모더보드라고 불림)에 있는 칩셋이다. 어댑터는 모더보드의 슬롯에 꽂히는 카드이다. 그러나 각각의 목적은 I/O 버스와 I/O 장치 사이에 정보를 오고가게 하는 것이다. 제 6 장에서 디스크와 같은 I/O 장치가 어떻게 작동하는지에 대해 더 말한다. 제 10 장에서는 Unix I/O 인터페이스를 통해 응용 프로그램에서 장치에 접근하는 방법을 배울 것이다. 우리는 특히 네트워크라고 불리는 흥미로운 종류의 장치에 중점을 두지만, 이 기술은 다른 종류의 장치에도 일반화될 수 있다.

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 x86-64 machine running Linux, data of type `short` require 2 bytes, types `int` and `float` 4 bytes, and types `long` and `double` 8 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.

주 기억장치는 프로세서가 프로그램을 실행하는 동안 프로그램과 데이터를 임시로 저장하는 장치입니다. 물리적으로 주 기억장치는 동적 랜덤 접근 메모리(DRAM) 칩의 집합으로 구성됩니다. 논리적으로, 기억장치는 고유한 주소(배열 인덱스)를 가진 바이트의 선형 배열로 구성됩니다. 일반적으로 프로그램을 구성하는 각 기계 명령은 변수 길이의 바이트로 구성될 수 있습니다. C 프로그램 변수에 해당하는 데이터 항목의 크기는 유형에 따라 다릅니다. 예를 들어, x86-64 기계에서 Linux를 실행하는 경우, short 유형의 데이터는 2바이트를 필요로 하며, int와 float 유형은 4바이트, long과 double 유형은 8바이트를 필요로 합니다. 제6장에서 DRAM과 같은 기억장치 기술에 대해 더 많이 다룹니다.

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-size 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.[2]

From the time that power is applied to the system until the time that the power is shut off, a processor repeatedly executes the instruction pointed at by the program counter and updates the program counter to point to the next instruction. A processor *appears* to operate according to a very simple instruction execution model, defined by its *instruction set architecture*. In this model, instructions execute

중앙 처리 장치 (CPU) 또는 단순히 프로세서는 주 메모리에 저장된 명령을 해석 (또는 실행)하는 엔진입니다. 그 핵심에는 프로그램 카운터 (PC)라고 불리는 워드 사이즈의 저장 장치 (또는 레지스터)가 있습니다. 어느 순간에도 PC는 주메모리에 있는 어떤 기계어 명령을 가리킵니다 (또는 그 주소를 가지고 있습니다). 시스템에 전원이 공급되는 순간부터 전원이 차단될 때까지 프로세서는 반복적으로 프로그램 카운터가 가리키는 명령을 실행하고 프로그램 카운터를 다음 명령을 가리킬 수 있도록 업데이트합니다. 프로세서는 매우 간단한 명령 실행 모델에 따라 작동하는 것으로 보입니다. 이 모델은 명령 집합 아키텍처에 의해 정의됩니다. 여기에선 명령들이 실행됩니다.

2. PC is also a commonly used acronym for "personal computer." However, the distinction between the two should be clear from the context.

PC는 "개인용 컴퓨터"를 나타내는 일반적으로 사용되는 약어입니다. 그러나 둘 사이의 구별은 문맥에서 명확해야 합니다.

in strict sequence, and executing a single instruction involves performing a series of steps. The processor 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 the *arithmetic/logic unit* (ALU). The register file is a small storage device that consists of a collection of word-size 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:

엄격한 순서대로, 단일 명령을 실행하는 것은 일련의 단계를 수행하는 것을 포함합니다. 프로세서는 프로그램 카운터(PC)가 가리키는 메모리에서 명령을 읽고, 명령 안의 비트를 해석하며, 명령에 의해 지시된 간단한 작업을 수행한 후, 다음 명령을 가리키기 위해 PC를 업데이트합니다. 이때 다음 명령이 방금 실행된 명령과 연속된 메모리에 있을 수도 있고 없을 수도 있습니다. 이러한 간단한 작업은 몇 가지뿐이며, 주 기억장치, 레지스터 파일 및 산술/논리 장치(ALU)와 관련이 있습니다. 레지스터 파일은 단어 크기의 레지스터 모음으로 구성된 소형 저장장치로, 각각에 고유한 이름이 있습니다. ALU는 새로운 데이터 및 주소 값들을 계산합니다. CPU가 명령에 따라 수행할 수 있는 간단한 작업의 몇 가지 예시는 다음과 같습니다:

- *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.
- *Operate:* Copy the contents of two registers to the ALU, perform an arithmetic operation on the two words, and store the result in a register, overwriting the previous contents of that register.
- *Jump:* Extract a word from the instruction itself and copy that word into the program counter (PC), overwriting the previous value of the PC.

운영 : 두 레지스터의 내용을 ALU에 복사하고, 두 워드에 대한 산술 연산을 수행하여 결과를 레지스터에 저장하여 해당 레지스터의 이전 내용을 덮어쓰기합니다.

We say that a processor appears to be a simple implementation of its instruction set architecture, but in fact modern processors use far more complex mechanisms to speed up program execution. Thus, we can distinguish the processor's instruction set architecture, describing the effect of each machine-code instruction, from its *microarchitecture*, describing how the processor is actually implemented. When we study machine code in Chapter 3, we will consider the abstraction provided by the machine's instruction set architecture. Chapter 4 has more to say about how processors are actually implemented. Chapter 5 describes a model of how modern processors work that enables predicting and optimizing the performance of machine-language programs.

프로세서는 명령 집합 아키텍처(ISA)의 간단한 구현으로 보이지만, 실제로 현대 프로세서는 프로그램 실행 속도를 높이기 위해 훨씬 더 복잡한 메커니즘을 사용합니다. 따라서 우리는 프로세서의 명령 집합 아키텍처와 각 기계어 명령의 효과를 설명하는 것과 실제 프로세서의 구현 방법을 설명하는 마이크로 아키텍처를 구분할 수 있습니다. 제 3 장에서 기계어를 공부할 때 기계의 명령 집합 아키텍처에서 제공되는 추상화를 고려할 것입니다. 제 4 장에서는 프로세서가 실제로 어떻게 구현되는지에 대해 더 많이 다룰 것입니다. 제 5 장에서는 현대 프로세서가 작동하는 방식에 대한 모델을 설명하여 기계어 프로그램의 성능을 예측하고 최적화하는 능력을 제시할 것입니다.

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`

시스템의 하드웨어 조직과 작동에 대한 이 간단한 시각을 고려할 때, 우리는 우리의 예제 프로그램을 실행할 때 무엇이 일어나는지 이해하기 시작할 수 있습니다. 우리는 나중에 채워질 많은 세부 사항을 여기서 생략해야 하지만, 지금은 큰 그림에 만족해야 합니다. 우선, 셸 프로그램은 실행 중에 명령을 입력할 때를 기다리면서 그 명령을 실행합니다. 키보드에서 ./hello 문자를 입력할 때마다 셸 프로그램은 각 문자를 레지스터에 읽은 다음에 메모리에 저장합니다. 우리가 키보드에서 enter 키를 누르면, 셸은 우리가 명령을 입력을 마쳤다는 것을 알고 있습니다. 그런 다음 셸은 hello 파일을 실행하여 hello 코드와 데이터를 복사하는 일련의 명령을 실행합니다. (Figure 1.5에 표시됨)

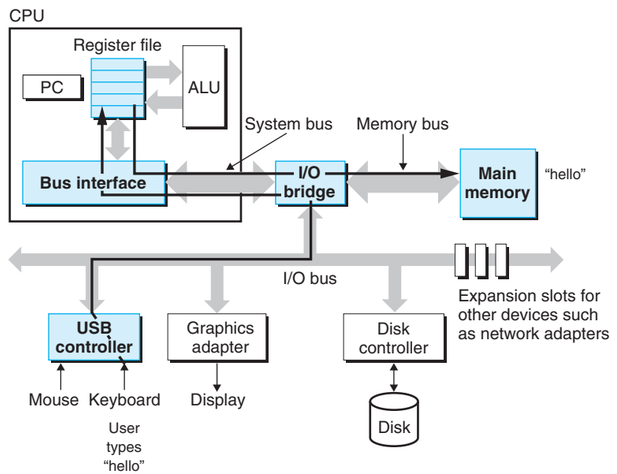


Figure 1.5
Reading the `hello` command from the keyboard.

object file from disk to main memory. The data includes 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 travel 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 instructions 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.

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 to 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

디스크에서 주 기억 장치로 오브젝트 파일을 옮긴다. 해당 데이터에는 문자열 "hello, world₩n"이 포함되어 있으며 이는 최종적으로 출력될 것이다. 직접 기억 장치 접근(DMA) 기술을 사용하여(제6장에서 논의됨), 데이터는 프로세서를 거치지 않고 디스크에서 주 기억 장치로 직접 이동한다. 이 단계는 도표 1.6에 표시되어 있다. hello 오브젝트 파일의 코드와 데이터가 메모리에 로드되면 프로세서는 hello 프로그램의 주 루틴에서 기계어 명령어를 실행하기 시작한다. 이 명령어들은 hello, world₩n 문자열의 바이트들을 메모리에서 레지스터 파일로 복사하고, 거기서 디스플레이 장치로 이동시켜 화면에 표시된다. 이 단계는 도표 1.7에 표시되어 있다.

이 간단한 예에서 중요한 교훈은 시스템이 정보를 한 곳에서 다른 곳으로 이동하는 데 많은 시간을 소비한다는 것입니다. hello 프로그램의 기계 명령어는 원래 디스크에 저장되어 있습니다. 프로그램이 로드되면 이들은 주기억장치로 복사됩니다. 프로세서가 프로그램을 실행하는 동안 명령어는 주기억장치에서 프로세서로 복사됩니다. 마찬가지로, hello,world₩n과 같은 데이터 문자열은 원래 디스크에 있었지만, 주기억장치로 복사되고, 그런 다음 주기억장치에서 디스플레이 장치로 복사됩니다. 프로그래머의 관점에서 이러한 복사 작업은 프로그램의 "실제 작업"을 늦추는 오버헤드입니다. 따라서 시스템 설계자들의 주요 목표는 이러한 복사 작업을 가능한 빠르게 실행하는 것입니다. 물리학적 법칙으로 인해 더 큰 저장 장치는 더 작은 저장 장치보다 느립니다. 그리고 더 빠른 장치는 느린 장치보다 더 비싸게 만들어집니다.

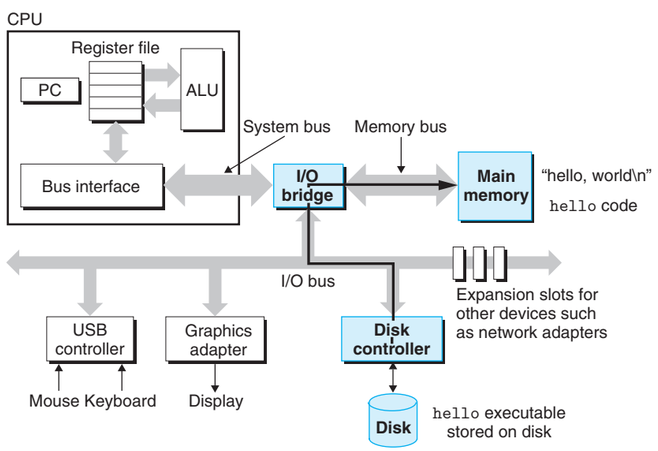


Figure 1.6 **Loading the executable from disk into main memory.**

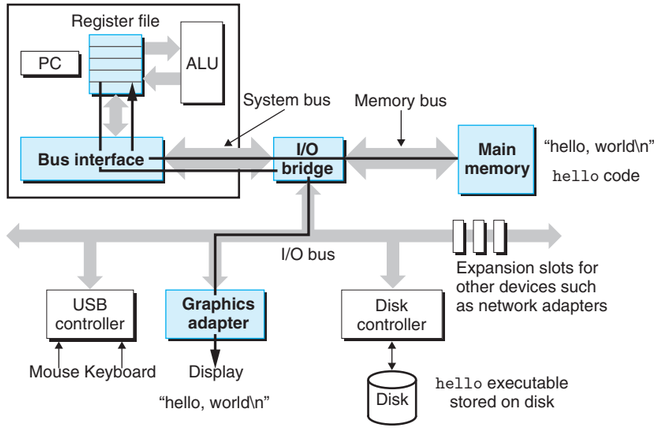
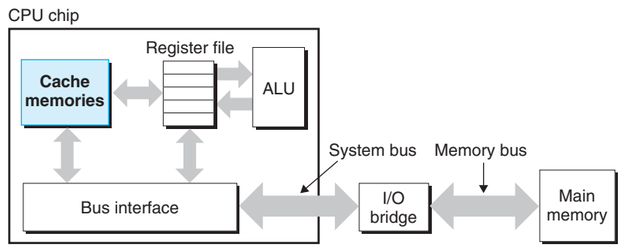


Figure 1.7 **Writing the output string from memory to the display.**

Figure 1.8
Cache memories.



counterparts. For example, the disk drive on a typical system might be 1,000 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 billions 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 processor to access the L2 cache than the L1 cache, but this is still 5 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). Newer and more powerful systems even have three levels of cache: L1, L2, and L3. The idea behind caching is that a system can get the effect of both a very large memory and a very fast one by exploiting *locality*, the tendency for programs to access data and code in localized regions. By setting up caches to hold data that are likely to be accessed often, we can perform most memory operations using the fast caches.

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.

상대적인 개념. 예를 들어, 일반 시스템의 디스크 드라이브는 주 메모리보다 1,000배 크지만, 프로세서가 디스크에서 단어를 읽는 데 주 메모리보다 10,000,000배 오랜 시간이 걸릴 수 있습니다. 마찬가지로, 일반 레지스터 파일은 수백 바이트의 정보만을 저장하는 반면에 주 메모리에는 수십억 바이트의 정보가 들어갑니다. 그러나 프로세서는 레지스터 파일에서 데이터를 읽는 것이 주 메모리보다 거의 100배 빠릅니다. 심지어 더 귀찮은 것은 반도체 기술이 연도별로 진전되는 과정에서 이 프로세서-메모리 간격이 계속 커지고 있다는 점입니다. 프로세서를 더 빠르게 실행시키는 것은 주 메모리를 더 빠르게 실행시키는 것보다 쉽고 저렴합니다. 프로세서-메모리 간격을 다루기 위해 시스템 디자이너들은 캐시 메모리(또는 단순히 캐시)라고 하는 빠르고 작은 저장 장치를 포함시킵니다. 이 장치는 프로세서가 가까운 미래에 필요로 할 정보를 임시로 저장하는 역할을 합니다. 그림 1.8은 일반 시스템에 있는 캐시 메모리를 보여줍니다. 프로세서 칩에 있는 L1 캐시는 수만 바이트를 보유하며 레지스터 파일과 거의 같은 속도로 접근할 수 있습니다. 수십만 바이트에서 수백만 바이트의 크기를 가진 더 큰 L2 캐시는 특별한 버스에 의해 프로세서에 연결됩니다. 프로세서가 L1 캐시에 접근하는 데 L2 캐시보다 5배 오래 걸릴 수 있지만, 이것은 여전히 주 메모리에 접근하는 것보다 5배에서 10배 더 빠릅니다. L1과 L2 캐시는 정적 랜덤 접근 메모리(SRAM)라고 하는 하드웨어 기술으로 구현됩니다. 더 최신이고 강력한 시스템에서는 L1, L2, 그리고 L3의 세 수준의 캐시를 가지고 있습니다. 캐싱의 아이디어는 시스템이 프로그램이 지역화된 영역에서 데이터와 코드에 접근하는 경향인 지역성(locality)을 활용하여 아주 큰 메모리와 아주 빠른 메모리를 모두 효과를 볼 수 있는 것입니다. 자주 액세스되는 데이터를 저장해두고 캐시를 설정함으로써 대부분의 메모리 작업을 빠른 캐시를 이용해서 수행할 수 있습니다. 이 책에서 가장 중요한 교훈 중 하나는 캐시 메모리에 대해 알고 있는 응용 프로그램 개발자들이 이를 활용하여 프로그램의 성능을 한 차원 향상시킬 수 있다는 것입니다. 6장에서 이러한 중요한 장치와 그것들을 어떻게 활용할지에 대해 더 자세히 다루게 됩니다.

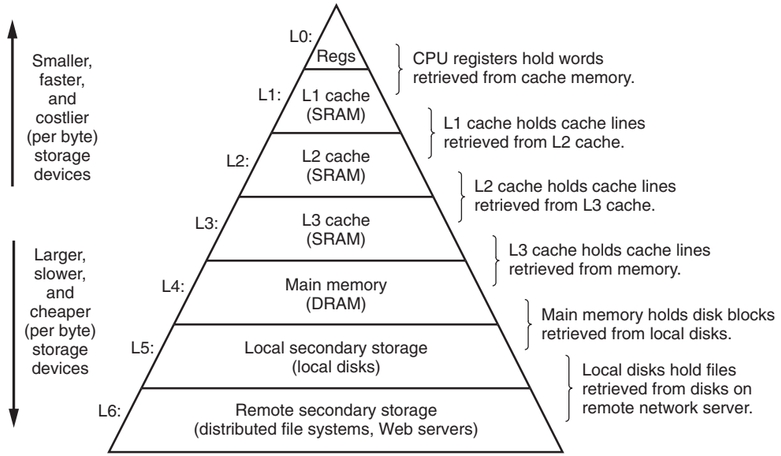


Figure 1.9 **An example of a memory hierarchy.**

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 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. We show three levels of caching L1 to L3, occupying memory hierarchy levels 1 to 3. Main memory occupies level 4, 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. Caches L1 and L2 are caches for L2 and L3, respectively. The L3 cache 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 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.

이 작은, 빠른 저장 장치 (예: 캐시 메모리)를 프로세서와 큰, 느린 장치 (예: 주 메모리) 사이에 삽입하는 개념은 일반적인 아이디어임이 밝혀졌습니다. 사실 모든 컴퓨터 시스템의 저장 장치는 그림 1.9와 유사한 메모리 계층 구조로 구성되어 있습니다. 계층의 맨 위에서부터 아래로 이동할수록 장치는 느리고, 크고, 바이트 당 비용이 적어집니다. 레지스터 파일은 계층 구조의 맨 위에서 차지하며, 레벨 0 또는 L0으로 알려져 있습니다. 우리는 L1에서 L3까지의 세 가지 캐시 레벨을 보여주는데, 이것은 메모리 계층 레벨 1에서 3을 차지합니다. 주 메모리는 레벨 4를 차지하며, 등 이어갑니다. 메모리 계층의 주요 아이디어는 한 수준에서의 저장 장치가 다음 낮은 수준의 저장을 위한 캐시 역할을 한다는 것입니다. 따라서 레지스터 파일은 L1 캐시의 캐시입니다. L1과 L2 캐시는 각각 L2와 L3의 캐시이며, L3 캐시는 주 메모리의 캐시이며, 주 메모리는 디스크의 캐시입니다. 일부 네트워크 시스템에서는 분산된 파일 시스템을 가진 로컬 디스크가 다른 시스템의 디스크에 저장된 데이터를 위한 캐시로써 작동합니다. 프로그래머들은 다른 캐시에 대한 지식을 활용하여 성능을 개선할 수 있듯이, 프로그래머들은 전체 메모리 계층에 대한 이해를 활용할 수 있습니다. 제 6장에서는 이에 대해 더 자세히 다루겠습니다.

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

우리의 hello 예제로 돌아가보겠습니다. 셸이 로드되고 hello 프로그램이 실행되었을 때, 그리고 hello 프로그램이 메시지를 출력했을 때, 어느 프로그램도 접근하지 않았습니다.

Figure 1.10
Layered view of a computer system.

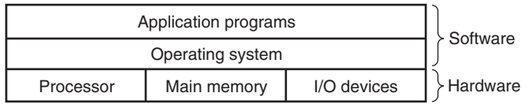
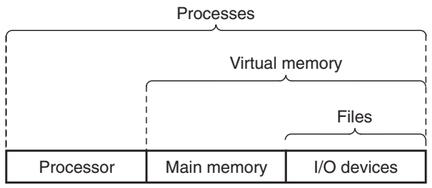


Figure 1.11
Abstractions provided by an operating system.



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 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.

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.

키보드, 디스플레이, 디스크 또는 주 기억장치는 직접적으로 애플리케이션 프로그램에 의해 조작되지 않았고, 대신 운영 체제가 제공하는 서비스에 의존했습니다. 우리는 운영 체제를 애플리케이션 프로그램과 하드웨어 사이에 위치한 소프트웨어 계층으로 생각할 수 있습니다. 그림 1.10에 나타난 대로, 애플리케이션 프로그램이 하드웨어를 조작하려는 모든 시도는 운영 체제를 거쳐야 합니다. 운영 체제에는 두 가지 주요 목적이 있습니다: (1) 과도한 애플리케이션의 오용으로부터 하드웨어를 보호하는 것과 (2) 복잡하고 종종 완전히 다른 저수준 하드웨어 장치를 조작하기 위한 애플리케이션에게 간단하고 일관된 메커니즘을 제공하는 것입니다. 운영 체제는 그림 1.11에 나타난 기본적인 추상화를 통해 두 가지 목표를 모두 달성합니다: 프로세스, 가상 메모리 및 파일. 이 그림이 제시하는 대로, 파일은 입출력 장치의 추상화, 가상 메모리는 주 기억장치와 디스크 입출력 장치의 추상화이며, 프로세스는 프로세서, 주 기억장치 및 입출력 장치를 위한 추상화입니다. 각각에 대해 순차적으로 논의할 것입니다.

프로그램인 hello와 같은 것이 현대 시스템에서 실행될 때, 운영 체제는 프로그램이 시스템에서 실행되는 유일한 것으로 보이도록 가장한다. 프로그램은 프로세서, 주 메모리, 그리고 입출력 장치를 배타적으로 사용하는 것처럼 보인다. 프로세서는 중단 없이 프로그램의 명령을 하나씩 실행하는 것으로 보인다. 그리고 프로그램의 코드와 데이터가 시스템 메모리의 유일한 객체로 보인다. 이러한 환상은 프로세스라는 개념에 의해 제공되며, 이는 컴퓨터 과학에서 가장 중요하고 성공적인 아이디어 중 하나이다. 프로세스는 실행 중인 프로그램에 대한 운영 체제의 추상화이다. 여러 프로세스는 동시에 동일한 시스템에서 실행될 수 있으며, 각 프로세스는 하드웨어를 배타적으로 사용하는 것처럼 보인다. 동시에, 다른 프로세스의 명령과 교차되는 것을 의미한다. 대부분의 시스템에서 실행해야 할 프로세스가 실행할 CPU보다 더 많은 것이 있는 것이다.

Aside Unix, Posix, and the Standard Unix Specification

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 Digital Equipment Corporation 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 [93].

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. More recently, a separate standardization effort, known as the "Standard Unix Specification," has joined forces with Posix to create a single, unified standard for Unix systems. As a result of these standardization efforts, the differences between Unix versions have largely disappeared.

1960년대는 IBM의 OS/360 및 Honeywell의 Multics 시스템과 같은 거대하고 복잡한 운영 체제의 시대였습니다. OS/360은 역사상 가장 성공적인 소프트웨어 프로젝트 중 하나였지만, Multics는 수년간 지체되었으며 널리 사용되지는 않았습니다. 벨 랩은 Multics 프로젝트의 원래 파트너였지만 프로젝트의 복잡성과 진전 부족으로 인해 1969년에 탈퇴했습니다. 불쾌한 Multics 경험에 반응으로 벨 랩의 연구원인 켄 톰슨, 데니스 리치, 더글러스 맥일로이, 조 오산나는 1969년 Digital Equipment Corporation의 PDP-7 컴퓨터용으로 머신 랭귀지로 완전히 쓰여진 더 단순한 운영체제에 대해 작업을 시작했습니다. 새로운 시스템에서 계층적 파일 시스템과 쉘을 사용자 수준 프로세스로서의 개념과 같은 많은 아이디어들은 Multics에서 빌려왔지만 더 작고 더 단순한 패키지로 구현되었습니다. 1970년, 브라이언 커니한은 "Multics"의 복잡성을 농담으로 새로운 시스템을 "Unix"라고 명명했습니다. 커널은 1973년에 C로 다시 쓰여졌으며 Unix는 1974년 외부 세계에 공개되었습니다. 벨 랩이 대학교에 관대한 조건으로 소스 코드를 제공했기 때문에 Unix는 대학에서 큰 관심을 끌었습니다. 가장 영향력 있는 작업은 버클리 대학교에서 1970년대 후반과 1980년대 초에 이루어졌으며, 버클리 연구원들은 Unix 4.xBSD(Berkeley Software Distribution)라는 일련의 릴리스에서 가상 메모리와 인터넷 프로토콜을 추가했습니다. 동시에 벨 랩은 자체 버전들을 발표하여 System V Unix로 알려졌습니다. Sun Microsystems의 Solaris 시스템과 같은 다른 벤더의 버전들은 이러한 원래의 BSD와 System V 버전에서 파생되었습니다.

1980년대 중반에 Unix 벤더들이 새로운 기능을 추가하려고 시도하는 가운데 문제가 발생했습니다. 이러한 추세를 극복하기 위해 IEEE(전기 및 전자 기술자 협회)는 Unix를 표준화하기 위한 노력을 발탁했으며, 나중에 리처드 스톨먼에 의해 "Posix"로 명명되었습니다. 이 결과로 "Posix 표준"이라고 알려진 일련의 표준으로, Unix 시스템 콜의 C 언어 인터페이스, 쉘 프로그램 및 유틸리티, 스레드 및 네트워크 프로그래밍과 같은 문제들을 다루고 있습니다. 최근에는 "표준 Unix 사양"이라고 하는 별도의 표준화 노력이 출범하여 Posix와 함께 Unix 시스템의 단일 통합 표준을 만들었습니다. 이러한 표준화 노력으로 인해 Unix 버전 간의 차이가 대부분 사라졌습니다.

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 1.9.2.

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

전통적인 시스템은 한 번에 하나의 프로그램만 실행할 수 있었지만, 최신 멀티코어 프로세서는 여러 프로그램을 동시에 실행할 수 있습니다. 어느 경우에도 단일 CPU는 프로세서가 그 사이를 전환함으로써 여러 프로세스를 동시에 실행하는 것처럼 보일 수 있습니다. 이를 위해 운영 체제는 컨텍스트 스위칭이라는 메커니즘을 사용하여 이러한 교차 작업을 수행합니다. 나머지 토론을 간단히하기 위해 우리는 단일 CPU를 포함하는 단일 처리기 시스템만을 고려합니다. 다중 처리기 시스템에 대해서는 1.9.2절에서 논의하겠습니다. 운영 체제는 프로세스가 실행되기 위해 필요한 모든 상태 정보, 즉 컨텍스트라고 하는 이 상태, 즉 현재 PC 값, 레지스터 파일, 메인 메모리의 내용과 같은 정보를 추적합니다. 언제든지 단일 처리기 시스템은 단일 프로세스의 코드만 실행할 수 있습니다. 운영 체제가 현재 프로세스에서 새로운 프로세스로 제어를 전환하기로 결정하면, 현재 프로세스의 컨텍스트를 저장하고 새로운 프로세스의 컨텍스트를 복원하여 컨텍스트 스위치를 수행합니다.

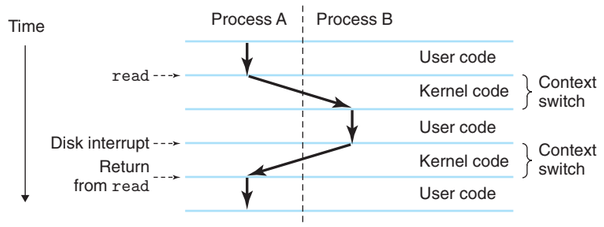


Figure 1.12
Process context switching.

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 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.

As Figure 1.12 indicates, the transition from one process to another is managed by the operating system *kernel*. The kernel is the portion of the operating system code that is always resident in memory. When an application program requires some action by the operating system, such as to read or write a file, it executes a special *system call* instruction, transferring control to the kernel. The kernel then performs the requested operation and returns back to the application program. Note that the kernel is not a separate process. Instead, it is a collection of code and data structures that the system uses to manage all the processes.

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.

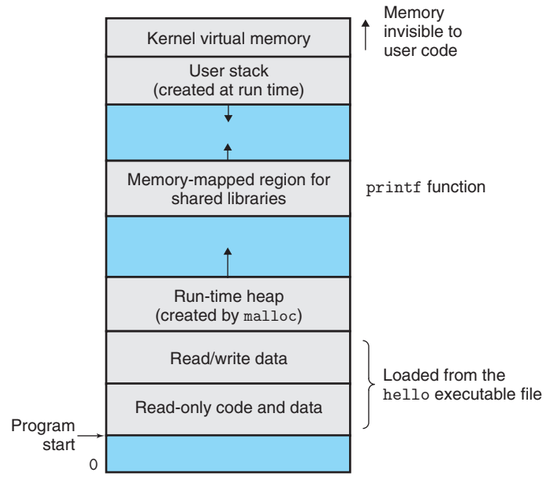
1.7.2 Threads

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

그럼 새로운 프로세스에 제어권을 넘깁니다. 새로운 프로세스는 이전에 중단된 곳에서 진행됩니다. 그림 1.12는 우리의 예시 "hello" 시나리오의 기본 아이디어를 보여줍니다. 우리의 예시 시나리오에는 두 개의 병행 프로세스가 있습니다: 쉘 프로세스와 hello 프로세스입니다. 처음에는 쉘 프로세스가 혼자 실행되어 명령행에서 입력을 기다리고 있습니다. 우리가 "hello" 프로그램을 실행하도록 요청하면, 쉘이 우리의 요청을 수행하기 위해 운영 체제로 제어를 전달하는 시스템 콜이라 불리는 특별한 함수를 호출합니다. 운영 체제는 쉘의 컨텍스트를 저장하고, 새로운 "hello" 프로세스 와 그 컨텍스트를 생성한 후 새로운 "hello" 프로세스로 제어를 넘깁니다. "hello"가 종료되면 운영 체제는 쉘 프로세스의 컨텍스트를 복원하고 제어를 되돌려주어 다음 명령행 입력을 기다리게 합니다. 그림 1.12에서 보여주는 대로, 한 프로세스에서 다른 프로세스로의 전환은 운영 체제 커널에 의해 관리됩니다. 커널은 항상 메모리에 상주하는 운영 체제 코드의 일부입니다. 응용 프로그램이 파일을 읽거나 쓰는 등 운영 체제의 특정 작업이 필요한 경우, 시스템 콜 명령을 실행하여 커널로 제어를 전달합니다. 그 후 커널은 요청된 작업을 수행하고 응용 프로그램으로 제어를 반환합니다. 커널이 별도의 프로세스가 아니라는 점을 주목해 주세요. 대신, 시스템에서 모든 프로세스를 관리하기 위해 사용되는 코드와 데이터 구조 의 모음입니다. 프로세스 추상화를 구현하는데는 저수준 하드웨어와 운영 체제 소프트웨어 간의 밀접한 협력이 필요합니다. 이것이 어떻게 작동하는지, 그리고 응용 프로그램이 자체 프로세스를 생성하고 제어하는 방법에 대해 8장에서 알아보겠습니다.

한국어로 번역: 보통 우리는 프로세스를 단일 통제 흐름으로 생각하지만, 현대 시스템에서 프로세스는 사실 여러 실행 단위, 쓰레드라 불리는 것으로 구성될 수 있습니다. 각각이 프로세스의 문맥에서 실행되며 동일한 코드와 전역 데이터를 공유합니다. 쓰레드는 네트워크 서버에서 동시성이 필요한 요구 때문에 점점 더 중요한 프로그래밍 모델이 됐습니다. 왜냐하면 여러 프로세스보다는 여러 쓰레드 사이에서 데이터를 공유하는 것이 더 쉬우며, 쓰레드는 보통 프로세스보다 효율적입니다. 멀티 쓰레딩은 또 다수의 프로세서가 가능할 때 프로그램을 더 빠르게 실행시키는 한 가지 방법이기도 합니다. 이에 대해 논의할 것입니다.

Figure 1.13
Process virtual address space. (The regions are not drawn to scale.)



Section 1.9.2. 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

섹션 1.9.2. 12장에서 쓰레드 프로그램을 작성하는 방법을 포함하여 동시성의 기본 개념을 배우게 될 것입니다.

가상 메모리는 각 프로세스에게 주 메모리의 배제적 사용을 가정하게 하는 추상화이다. 각 프로세스는 가상 주소 공간이라고 알려진 메모리의 동일한 균일한 뷰를 갖게 된다. Linux 프로세스의 가상 주소 공간은 도표 1.13에 나와 있다. (다른 Unix 시스템은 유사한 레이아웃을 사용한다.) Linux에서 주소 공간의 가장 윗부분은 모든 프로세스에게 공통인 운영 체제의 코드와 데이터를 위해 예약된다. 주소 공간의 하위 영역에는 사용자 프로세스에서 정의된 코드와 데이터가 위치한다. 도표에서 주소는 아래에서 위로 증가함에 유의하라. 각 프로세스가 보는 가상 주소 공간은 특정 목적을 갖는 여러 정의된 영역으로 이루어져 있으며, 책의 뒷부분에서 이러한 영역에 대해 더 자세히 알아볼 것이지만,최하의 주소부터 시작하여 점차 올라가면서 각 영역을 간단히 살펴보는 것이 도움이 될 것이다.

프로그램 코드와 데이터. 코드는 모든 프로세스에 대해 동일한 고정된 주소에서 시작하며, 글로벌 C 변수에 해당하는 데이터 위치가 뒤따릅니다. 코드 및 데이터 영역은 실행 가능한 오브젝트 파일의 내용으로 직접 초기화됩니다. 이 경우에는 "hello" 실행 파일입니다. 제 [illegible] 장에서 링킹과 로딩을 공부할 때 이 주소 공간의 일부에 대해 더 자세히 배우게 될 것입니다.

힙. 코드 영역과 데이터 영역 뒤에는 즉시 실행시간 힙이 따라옵니다. 프로세스가 시작될 때 크기가 고정된 코드 영역과 데이터 영역과는 다르게⊠ 실행시간 힙은 크기가 고정되어 있지 않습니다.

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 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. Instead, they must invoke the kernel to perform these operations.

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 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.

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

실행 중에 힙은 malloc 및 free와 같은 C 표준 라이브러리 루틴의 호출로 인해 동적으로 확장되고 축소됩니다. 9장에서 가상 메모리를 관리하는 법을 배울 때 힙에 대해 자세히 공부할 것입니다.

공유 라이브러리. 주소 공간의 중간 부근에는 C 표준 라이브러리와 수학 라이브러리와 같은 공유 라이브러리의 코드와 데이터가 있는 영역이 있습니다. 공유 라이브러리의 개념은 강력하지만 다소 어려운 개념입니다. 7장에서 동적 링킹을 공부할 때 그들이 어떻게 작동하는지 배우게 될 것입니다.

. Stack. 사용자의 가상 주소 공간의 맨 위에는 컴파일러가 함수 호출을 구현하는 데 사용하는 사용자 스택이 있습니다. 힙과 마찬가지로 사용자 스택은 프로그램 실행 중에 동적으로 확장하고 축소합니다. 특히, 함수를 호출할 때마다 스택이 커집니다. 함수에서 반환할 때마다 스택이 줄어듭니다. 컴파일러가 스택을 사용하는 방법은 3장에서 배우게 될 것입니다.

커널 가상 메모리. 주소 공간의 상위 영역은 커널을 위해 예약되어 있습니다. 응용 프로그램은 이 영역의 내용을 읽거나 쓰거나 커널 코드에 정의된 함수를 직접 호출할 수 없습니다. 대신, 이러한 작업을 수행하기 위해 커널을 호출해야 합니다.

가상 메모리가 작동하려면 하드웨어와 운영 체제 소프트웨어 사이에 정교한 상호 작용이 필요하며, 프로세서에 의해 생성된 모든 주소를 하드웨어로 변환해야 합니다. 기본 아이디어는 프로세스의 가상 메모리 내용을 디스크에 저장하고, 그 후 메인 메모리를 디스크의 캐시로 사용하는 것입니다. 9장에서는 이 동작 방식과 현대 시스템의 작동에 있어 왜 이것이 중요한지에 대해 설명합니다.

파일은 바이트의 연속으로서, 그 이상 그 이하도 없습니다. 디스크, 키보드, 디스플레이, 심지어 네트워크를 포함한 모든 입출력 장치는 파일로 모델링됩니다. 시스템 내 모든 입력과 출력은 유닉스 입출력으로 알려진 일부 작은 집합의 시스템 호출을 사용하여 파일을 읽고 쓰는 것으로 이루어집니다. 파일의 이 간단하고 우아한 개념은 시스템에 포함될 수 있는 다양한 입출력 장치에 대한 응용 프로그램에 통일된 관점을 제공하기 때문에 매우 강력합니다. 예를 들어, 디스크 파일의 내용을 조작하는 응용 프로그램 프로그래머들은 특정 디스크 기술에 대해 전혀 알지 못해도 됩니다. 더 나아가, 동일한 프로그램은 다른 디스크 기술을 사용하는 다른 시스템에서도 실행됩니다. 10장에서 유닉스 입출력에 대해 배우게 됩니다.

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

여기까지 시스템 투어에서 시스템을 독립된 하드웨어 및 소프트웨어의 모음으로 취급했습니다. 실제로 현대 시스템은 종종 네트워크에 의해 다른 시스템과 연결됩니다. 개별 시스템의 관점에서,

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)
```

As Torvalds indicates, his starting point for creating Linux was Minix, an operating system developed by Andrew S. Tanenbaum for educational purposes [113].

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 handheld devices to mainframe computers. A group at IBM has even ported Linux to a wristwatch!

1991년 8월에 핀란드 출신의 대학원생인 리눅스 토르발스가 겸손하게 새로운 유닉스와 비슷한 운영 체제 커널을 발표했습니다:

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 번역: 보낸 사람: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds) 뉴스그룹: comp.os.minix 제목: minix에서 가장 원하는 것은 무엇입니까? 요약: 내 새로운 운영 체제를 위한 소규모 투표 날짜: 91년 8월 25일 20시 57분 08초 GMT

안녕하세요, 미닉스를 사용하는 모든 분들 - 저는 386(486) AT 클론을 위한 (무료) 운영 체제를 만들고 있어요 (즉, 취미로 하고, gnu처럼 크고 전문적일 거라곤 예상하지 마세요). 4월부터 구상하고 있는데, 이제 완성 단계에 들어갔어요. 미닉스와 비슷한 면이 있어서, 미닉스를 사용하는 분들이 좋아하거나 싫어하는 점에 대한 피드백을 받고 싶어요 (실용적인 이유로 파일 시스템의 물리적 레이아웃 등의 이유로).

저는 현재 bash(1.08)와 gcc(1.40)를 이식했고, 모든 것이 작동하는 것으로 보입니다. 이것은 몇 달 안에 실용적인 것을 얻을 것이라는 것을 의미하며, 대부분의 사람들이 원하는 기능은 무엇인지 알고 싶습니다. 모든 제안을 환영하지만, 구현을 약속할 수는 없습니다 :-)

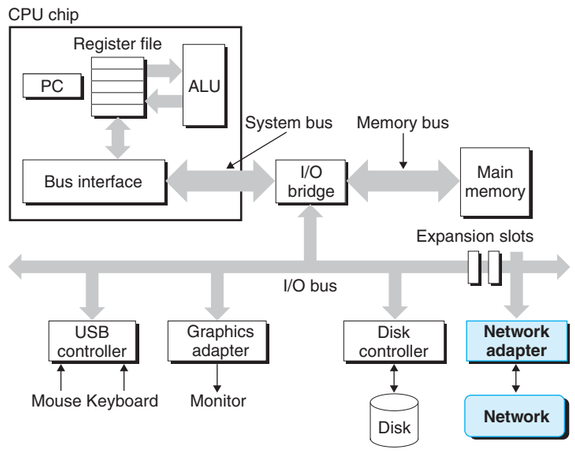
그 이후로야, 역사가 된 말이죠. Linux는 기술적, 문화적 현상으로 진화했습니다. GNU 프로젝트와 협력하여 Linux 프로젝트는 커널과 모든 지원 인프라를 포함한 완전한 Posix 호환 버전의 Unix 운영 체제를 개발했습니다. Linux는 휴대용 장치에서부터 대형 컴퓨터에 이르기까지 다양한 컴퓨터에서 사용할 수 있습니다. IBM의 한 그룹은 심지어 리눅스를 손목 시계에도 이식했습니다!

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 flow across 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 these 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.

네트워크는 그림 1.14에 나와 있는 것처럼 또 다른 입출력 장치로 볼 수 있습니다. 시스템이 주 기억장치에서 바이트 열을 네트워크 어댑터로 복사할 때 데이터는 로컬 디스크 드라이브가 아닌 다른 기계를 통해 네트워크를 통해 흐릅니다. 마찬가지로, 시스템은 다른 기계로부터 전송된 데이터를 읽고 이러한 데이터를 주 기억장치에 복사할 수 있습니다. 인터넷과 같은 글로벌 네트워크의 등장으로 기계 간 정보를 복사하는 것이 컴퓨터 시스템의 가장 중요한 용도 중 하나가 되었습니다. 예를 들어 이메일, 즉시 메시징, 월드 와이드 웹, FTP, 텔넷과 같은 애플리케이션은 모두 네트워크를 통해 정보를 복사할 수 있는 능력에 기반을 두고 있습니다.

Figure 1.14
A network is another I/O device.



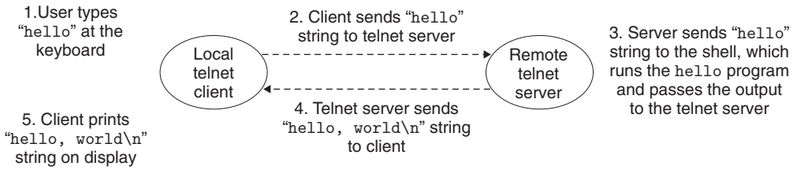


Figure 1.15 Using telnet to run `hello` remotely over a network.

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 in 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 11 you will learn how to build network applications and apply this knowledge to build a simple Web server.

서버는 "hello" 문자열을 셸에 보내고, 셸은 hello 프로그램을 실행하여 출력을 텔넷 서버로 전달합니다.

우리의 hello 예제로 돌아가서, 우리는 익숙한 텔넷 애플리케이션을 사용하여 원격 컴퓨터에서 hello를 실행할 수 있습니다. 우리는 우리의 지역 컴퓨터에서 실행 중인 텔넷 클라이언트를 사용하여 원격 컴퓨터의 텔넷 서버에 연결할 수 있다고 가정해 봅시다. 우리가 원격 컴퓨터에 로그인하고 쉘을 실행한 후, 원격 쉘이 입력 명령을 받을 준비가 되어 있습니다. 이 시점에서, 원격으로 hello 프로그램을 실행하는 것은 Figure 1.15에 나와 있는 다섯 가지의 기본 단계를 따릅니다. 우리가 hello 문자열을 텔넷 클라이언트에 입력하고 Enter 키를 누르면, 클라이언트는 문자열을 텔넷 서버로 전송합니다. 텔넷 서버가 네트워크로부터 문자열을 받은 후, 그것을 원격 쉘 프로그램에게 전달합니다. 그 다음, 원격 쉘은 hello 프로그램을 실행하고 출력 라인을 다시 텔넷 서버로 전달합니다. 마지막으로, 텔넷 서버는 출력 문자열을 네트워크를 통해 텔넷 클라이언트로 전달하여 지역 터미널에 출력 문자열을 출력합니다. 이러한 클라이언트와 서버 간의 교환은 모든 네트워크 애플리케이션의 전형적인 것입니다. 11장에서 네트워크 애플리케이션을 구축하고 이 지식을 적용하여 간단한 웹 서버를 구축하는 방법에 대해 배우게 될 것입니다.

1.9 Important Themes

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 fill in some details about the hardware and the software, and it will show how, by knowing these details, you can write programs that are faster, more reliable, and more secure.

To close out this chapter, we highlight several important concepts that cut across all aspects of computer systems. We will discuss the importance of these concepts at multiple places within the book.

우리의 초기 시스템 휘리윈드 투어가 이것으로 마무리됩니다. 이 토론에서 얻을 수 있는 중요한 아이디어는 시스템이 하드웨어보다 더 많은 것이라는 것입니다. 시스템은 서로 엮인 하드웨어와 시스템 소프트웨어의 집합으로, 응용 프로그램을 실행하는 궁극적인 목표를 달성하기 위해 협력해야 합니다. 이 책의 나머지 부분은 하드웨어와 소프트웨어에 대한 몇 가지 세부 정보를 제공하며, 이러한 세부 정보를 알면 더 빠르고 신뢰할 수 있으며 보안이 더 강화된 프로그램을 작성할 수 있다는 것을 보여줄 것입니다. 이 장을 마무리하기 위해 컴퓨터 시스템의 모든 측면을 관통하는 여러 중요한 개념을 강조하겠습니다. 이러한 개념의 중요성에 대해 이 책 여러 곳에서 논의할 것입니다.

1.9.1 Amdahl's Law

Gene Amdahl, one of the early pioneers in computing, made a simple but insightful observation about the effectiveness of improving the performance of one part of a system. This observation has come to be known as *Amdahl's law*. The main idea is that when we speed up one part of a system, the effect on the overall system performance depends on both how significant this part was and how much it sped up. Consider a system in which executing some application requires time T_{old}. Suppose some part of the system requires a fraction α of this time, and that we improve its performance by a factor of k. That is, the component originally required time αT_{old}, and it now requires time $(\alpha T_{\text{old}})/k$. The overall execution time would thus be

$$\begin{aligned} T_{\text{new}} &= (1-\alpha)T_{\text{old}} + (\alpha T_{\text{old}})/k \\ &= T_{\text{old}}[(1-\alpha) + \alpha/k] \end{aligned}$$

From this, we can compute the speedup $S = T_{\text{old}}/T_{\text{new}}$ as

$$S = \frac{1}{(1-\alpha) + \alpha/k} \tag{1.1}$$

As an example, consider the case where a part of the system that initially consumed 60% of the time ($\alpha = 0.6$) is sped up by a factor of 3 ($k = 3$). Then we get a speedup of $1/[0.4 + 0.6/3] = 1.67\times$. Even though we made a substantial improvement to a major part of the system, our net speedup was significantly less than the speedup for the one part. This is the major insight of Amdahl's law—to significantly speed up the entire system, we must improve the speed of a very large fraction of the overall system.

Gene Amdahl, 컴퓨팅의 초기 개척자 중 한 명으로, 시스템의 성능을 향상시키는 효과에 관한 단순하지만 통찰력 있는 관찰을 했다. 이 관찰은 Amdahl의 법칙으로 알려져 있다. 주요 아이디어는 시스템 중 한 부분의 성능을 향상시킬 때, 전체 시스템의 성능에 대한 효과는 이 부분이 얼마나 중요했는지와 얼마나 빨라졌는지에 따라 달라진다는 것이다. 어떤 애플리케이션을 실행하는 데 시간 Told가 필요한 시스템을 생각해보자. 시스템의 일부가 이 시간의 분수인 α를 필요로 하고, 이를 개선하여 k배의 성능을 향상시킨다고 가정하자. 즉, 이 부품은 원래 αTold의 시간이 필요했고, 이제 (αTold)/k의 시간이 필요하다. 따라서 전체 실행 시간은

예를 들어, 초기에 시스템의 일부가 60%의 시간(α = 0.6)을 소비했을 때를 고려해 보겠습니다. 그리고 그 일부를 3배로 가속했다고 가정해 봅시다 (k = 3). 그러면 1/[0.4 + 0.6/3] = 1.67×의 속도 향상을 얻게 됩니다. 시스템의 주요한 일부를 상당히 향상시켰음에도 불구하고, 순수한 속도 향상은 그 일부의 속도 향상만큼 많이 나오지 않았습니다. 이것이 Amdahl의 법칙의 주요한 통찰력입니다 - 전체 시스템의 속도를 상당히 향상시키기 위해서는 전체 시스템 중 상당한 부분의 속도를 개선해야 합니다.

Practice Problem 1.1 (solution page 64)

Suppose you work as a truck driver, and you have been hired to carry a load of potatoes from Boise, Idaho, to Minneapolis, Minnesota, a total distance of 2,500 kilometers. You estimate you can average 100 km/hr driving within the speed limits, requiring a total of 25 hours for the trip.

연습 문제 1.1 (해설 페이지 64) 당신이 트럭 운전사로 일하고 있으며 아이다호주 보이시에서 미네소타주 미니애폴리스로 감자를 운반하기 위해 고용되었다고 가정해 봅시다. 이동 거리는 총 2,500킬로미터입니다. 규정 속도 내에서 100km/시간 평균 주행할 수 있다고 추정하면, 이 여행에는 총 25시간이 걸리게 됩니다.

Aside Expressing relative performance

The best way to express a performance improvement is as a ratio of the form T_{old}/T_{new}, where T_{old} is the time required for the original version and T_{new} is the time required by the modified version. This will be a number greater than 1.0 if any real improvement occurred. We use the suffix '×' to indicate such a ratio, where the factor "2.2×" is expressed verbally as "2.2 times."

The more traditional way of expressing relative change as a percentage works well when the change is small, but its definition is ambiguous. Should it be $100 \cdot (T_{old} - T_{new})/T_{new}$, or possibly $100 \cdot (T_{old} - T_{new})/T_{old}$, or something else? In addition, it is less instructive for large changes. Saying that "performance improved by 120%" is more difficult to comprehend than simply saying that the performance improved by 2.2×.

가장 효율적인 방법은 Told가 원래 버전에서 필요한 시간이고 Tnew가 수정된 버전에서 필요한 시간인 형태의 비율로 나타내는 것입니다. 이것은 실제 개선이 발생했다면 1.0보다 큰 숫자가 될 것입니다. 이러한 비율을 나타내기 위해 '×' 접미사를 사용하고, "2.2×" 요소는 말로 "2.2배"로 표현됩니다. 상대적 변화를 백분율로 표현하는 더 전통적인 방법은 작은 변화에는 잘 작동하지만, 그 정의가 모호합니다. 이것은 100 . (Told - Tnew)/Tnew, 또는 가능성있는 경우에는 100 . (Told - Tnew)/Told이어야합니까? 또한, 큰 변화에 대해서는 덜 가르쳐줍니다. "성능이 120% 향상되었다"라고 말하는 것은 간단히 "성능이 2.2배 향상되었다"고 말하는 것보다 이해하기 어렵습니다.

A. You hear on the news that Montana has just abolished its speed limit, which constitutes 1,500 km of the trip. Your truck can travel at 150 km/hr. What will be your speedup for the trip?

B. You can buy a new turbocharger for your truck at www.fasttrucks.com. They stock a variety of models, but the faster you want to go, the more it will cost. How fast must you travel through Montana to get an overall speedup for your trip of 1.67×?

A. 당신은 뉴스에서 몽태나가 속도 제한을 폐지했다는 소식을 들었는데, 이는 여행 중 1,500km를 차지합니다. 당신의 트럭은 시속 150km로 이동할 수 있습니다. 여행 중 당신의 속도는 얼마나 증가하게 될까요?

B. 당신은 트럭용 터보차저를 www.fasttrucks.com에서 새로 살 수 있습니다. 그들은 다양한 모델을 보유하고 있지만, 빨리 가고 싶을수록 비용도 더 많이 들 것입니다. 몬타나를 통해 여행 속도를 1.67배 빠르게 하려면 얼마나 빨리 움직여야 합니까?

Practice Problem 1.2 (solution page 64)

A car manufacturing company has promised their customers that the next release of a new engine will show a 4× performance improvement. You have been assigned the task of delivering on that promise. You have determined that only 90% of the engine can be improved. How much (i.e., what value of k) would you need to improve this part to meet the overall performance target of the engine?

연습 문제 1.2 (해답 페이지 64) 자동차 제조 회사는 고객들에게 새 엔진의 다음 출시 버전에서 4배의 성능 향상이 있을 것이라고 약속했습니다. 당신에게 이 약속을 지키는 임무가 부여되었습니다. 엔진의 90%만 개선될 수 있다고 결정했습니다. 엔진의 전반적인 성능 목표를 달성하기 위해 이 부분을 얼마나 개선해야 하는지 (즉, k의 값) 알아내야 합니까?

One interesting special case of Amdahl's law is to consider the effect of setting k to ∞. That is, we are able to take some part of the system and speed it up to the point at which it takes a negligible amount of time. We then get

$$S_\infty = \frac{1}{(1-\alpha)} \tag{1.2}$$

So, for example, if we can speed up 60% of the system to the point where it requires close to no time, our net speedup will still only be $1/0.4 = 2.5\times$.

Amdahl's law describes a general principle for improving any process. In addition to its application to speeding up computer systems, it can guide a company trying to reduce the cost of manufacturing razor blades, or a student trying to improve his or her grade point average. Perhaps it is most meaningful in the world

Amdahl's law의 흥미로운 특수한 경우 중 하나는 k를 ∞로 설정하는 효과를 고려하는 것입니다. 즉, 시스템의 일부를 속도를 높여서 무시할 만큼의 시간이 걸리게 만들 수 있습니다. 그러면 우리는 ~를 얻게 됩니다.

따라서, 예를 들어, 시스템의 60%를 속도를 높일 수 있다면 요구되는 시간이 거의 없는 지점까지 속도를 올릴 경우, 순수한 가속은 여전히 1/0.4 = 2.5배 만에 남을 것입니다. Amdahl의 법칙은 어떤 과정을 개선하기 위한 일반적인 원리를 설명합니다. 컴퓨터 시스템의 속도를 높이는 데 적용된 것 외에도, 이는 면도날의 제조 비용을 줄이려는 회사나 학생이 학점을 높이려는 경우에도 유용할 수 있습니다. 아마도 이것은 세계에서 가장 의미 있는 것입니다.

of computers, where we routinely improve performance by factors of 2 or more. Such high factors can only be achieved by optimizing large parts of a system.

1.9.2 Concurrency and Parallelism

Throughout the history of digital computers, two demands have been constant forces in driving improvements: we want them to do more, and we want them to run faster. Both of these factors improve when the processor does more things at once. We use the term *concurrency* to refer to the general concept of a system with multiple, simultaneous activities, and the term *parallelism* to refer to the use of concurrency to make a system run faster. Parallelism can be exploited at multiple levels of abstraction in a computer system. We highlight three levels here, working from the highest to the lowest level in the system hierarchy.

Thread-Level Concurrency

Building on the process abstraction, we are able to devise systems where multiple programs execute at the same time, leading to *concurrency*. With threads, we can even have multiple control flows executing within a single process. Support for concurrent execution has been found in computer systems since the advent of time-sharing in the early 1960s. Traditionally, this concurrent execution was only *simulated*, by having a single computer rapidly switch among its executing processes, much as a juggler keeps multiple balls flying through the air. This form of concurrency allows multiple users to interact with a system at the same time, such as when many people want to get pages from a single Web server. It also allows a single user to engage in multiple tasks concurrently, such as having a Web browser in one window, a word processor in another, and streaming music playing at the same time. Until recently, most actual computing was done by a single processor, even if that processor had to switch among multiple tasks. This configuration is known as a *uniprocessor system.*

When we construct a system consisting of multiple processors all under the control of a single operating system kernel, we have a *multiprocessor system*. Such systems have been available for large-scale computing since the 1980s, but they have more recently become commonplace with the advent of *multi-core* processors and *hyperthreading*. Figure 1.16 shows a taxonomy of these different processor types.

Multi-core processors have several CPUs (referred to as "cores") integrated onto a single integrated-circuit chip. Figure 1.17 illustrates the organization of a

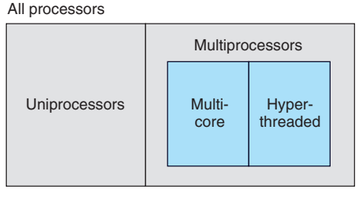


Figure 1.16
Categorizing different processor configurations. Multiprocessors are becoming prevalent with the advent of multi-core processors and hyperthreading.

컴퓨터는 2배 이상의 성능향상을 정기적으로 달성하는데, 이러한 높은 성능향상은 시스템의 클 대부분을 최적화하여야만 가능하다.

컴퓨터 역사를 통틀어서, 디지털 컴퓨터의 두 가지 요구사항이 지속적으로 개선을 동반하여 온 것으로 알려져 있습니다: 더 많은 일을 처리하고, 더 빨리 실행하길 원합니다. 이러한 요소들은 모두 프로세서가 동시에 더 많은 일을 처리할 때 개선됩니다. 우리는 일반적으로 병행성(concurrency)이라는 개념을 사용하여 시스템에 여러 활동이 동시에 수행되는 개념을 나타내며, 동시성을 사용하여 시스템을 빨리 실행하기 위해 병렬성(parallelism)이라는 용어를 사용합니다. 병렬성은 컴퓨터 시스템의 여러 추상화 수준에서 활용될 수 있습니다. 여기서 우리는 시스템 계층구조에서 가장 높은 수준부터 가장 낮은 수준까지 세 가지 수준을 강조합니다.

프로세스 추상화를 기반으로 하여, 여러 프로그램이 동시에 실행되어 병행성을 이끌어내는 시스템을 고안할 수 있습니다. 쓰레드를 사용하면 하나의 프로세스 내에서 여러 개의 제어 흐름을 실행할 수도 있습니다. 병행 실행을 지원하는 기능은 1960년대 초에 타임 쉐어링 시스템이 도입되면서 컴퓨터 시스템에서 찾아볼 수 있습니다. 전통적으로 이러한 병행 실행은 하나의 컴퓨터가 여러 프로세스 사이를 빠르게 전환하면서 시뮬레이션되었습니다. 마치 저글러가 여러 공들을 공중에서 띄우는 것처럼 말이죠. 이러한 병행성 형태는 여러 사용자가 동시에 시스템과 상호 작용할 수 있게 하며, 예를 들어, 많은 사람들이 한 대의 웹 서버에서 페이지를 받아가려 할 때와 같은 상황에 적합합니다. 또한, 단일 사용자가 여러 작업을 동시에 수행할 수도 있습니다. 예를 들어, 하나의 창에서 웹 브라우저를, 다른 창에서 워드프로세서를, 그리고 동시에 스트리밍 음악을 들을 수 있습니다. 최근까지 대부분의 실제 컴퓨팅은 단일 프로세서에 의해 처리되었으나, 그 프로세서가 여러 작업 사이를 전환해야 할 수도 있었습니다. 이러한 구성은 단일 프로세서 시스템이라고 알려져 있습니다. 단일 운영 체제 커널 아래에서 여러 프로세서로 이루어진 시스템을 구성할 때, 우리는 다중 프로세서 시스템을 갖게 됩니다. 이러한 시스템은 1980년대부터 대규모 컴퓨팅에 사용되었으며, 최근에는 멀티 코어 프로세서와 하이퍼스레딩의 도래로 흔히 볼 수 있게 되었습니다. 그림 1.16에는 다양한 프로세서 유형의 분류가 나와 있습니다. 멀티 코어 프로세서는 여러 CPU(코어라고 불리는)가 하나의 집적회로 칩에 통합된 형태입니다. 그림 1.17에서는 멀티 코어 프로세서의 구성을 설명하고 있습니다.

도표 1.16 다양한 프로세서 구성 분류. 멀티프로세서는 멀티코어 프로세서와 하이퍼스레딩의 출현으로 보편화되고 있다.

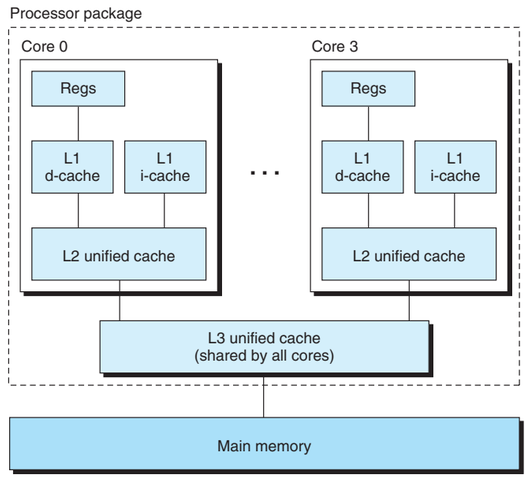


Figure 1.17
Multi-core processor organization. Four processor cores are integrated onto a single chip.

도 1.17 멀티코어 프로세서 구성. 네 개의 프로세서 코어가 단일 칩에 통합되어 있습니다.

typical multi-core processor, where the chip has four CPU cores, each with its own L1 and L2 caches, and with each L1 cache split into two parts—one to hold recently fetched instructions and one to hold data. The cores share higher levels of cache as well as the interface to main memory. Industry experts predict that they will be able to have dozens, and ultimately hundreds, of cores on a single chip.

Hyperthreading, sometimes called *simultaneous multi-threading*, is a technique that allows a single CPU to execute multiple flows of control. It involves having multiple copies of some of the CPU hardware, such as program counters and register files, while having only single copies of other parts of the hardware, such as the units that perform floating-point arithmetic. Whereas a conventional processor requires around 20,000 clock cycles to shift between different threads, a hyperthreaded processor decides which of its threads to execute on a cycle-by-cycle basis. It enables the CPU to take better advantage of its processing resources. For example, if one thread must wait for some data to be loaded into a cache, the CPU can proceed with the execution of a different thread. As an example, the Intel Core i7 processor can have each core executing two threads, and so a four-core system can actually execute eight threads in parallel.

The use of multiprocessing can improve system performance in two ways. First, it reduces the need to simulate concurrency when performing multiple tasks. As mentioned, even a personal computer being used by a single person is expected to perform many activities concurrently. Second, it can run a single application program faster, but only if that program is expressed in terms of multiple threads that can effectively execute in parallel. Thus, although the principles of concurrency have been formulated and studied for over 50 years, the advent of multi-core and hyperthreaded systems has greatly increased the desire to find ways to write application programs that can exploit the thread-level parallelism available with

전형적인 멀티코어 프로세서는 칩에 4개의 CPU 코어가 있으며, 각각이 자체 L1 및 L2 캐시를 가지고 있고, 각 L1 캐시는 최근에 가져온 명령어와 데이터를 보관하는 두 부분으로 나뉩니다. 코어들은 더 높은 수준의 캐시와 메인 메모리와의 인터페이스를 공유합니다. 산업 전문가들은 하나의 칩에 수십 개, 마침내 수백 개의 코어를 가질 수 있을 것으로 예측하고 있습니다.

하이퍼스레딩은 가끔 동시 멀티 스레딩이라고도 불리며, 단일 CPU가 여러 제어 흐름을 실행할 수 있는 기술입니다. 이것은 프로그램 카운터와 레지스터 파일과 같은 일부 CPU 하드웨어의 다중 복사본을 가지면서, 부동 소수점 연산을 수행하는 유닛과 같은 하드웨어의 다중 복사본은 하나씩 가지고 있습니다. 일반 프로세서는 다른 스레드 간 전환하는 데 약 20,000개의 클럭 주기가 필요하지만, 하이퍼스레드 프로세서는 주기별로 실행할 스레드를 결정하여 CPU가 처리 리소스를 더 잘 이용할 수 있게 합니다. 예를 들어, 인텔 코어 i7 프로세서는 각 코어가 두 스레드를 실행할 수 있으므로 4코어 시스템은 실제로 8개의 스레드를 병렬로 실행할 수 있습니다. 여러 처리를 사용하면 시스템 성능을 두 가지 방법으로 향상시킬 수 있습니다. 먼저, 여러 작업을 수행할 때 동시성을 모사하는 필요성이 줄어듭니다. 언급했듯이, 단일 사용자가 사용하는 개인용 컴퓨터조차도 동시에 많은 활동을 수행해야 할 것으로 예상됩니다. 둘째, 단일 응용 프로그램을 더 빨리 실행할 수 있지만, 그 프로그램이 병렬로 효과적으로 실행될 수 있는 여러 스레드로 표현되어야합니다. 따라서, 동시성의 원리는 50년 이상 동안 공식화되고 연구되어 왔지만, 멀티코어 및 하이퍼스레드 시스템의 출현으로 스레드 수준 병렬성을 활용할 수 있는 응용 프로그램을 작성하는 방법을 찾는 욕망이 크게 증가했습니다.

the hardware. Chapter 12 will look much more deeply into concurrency and its use to provide a sharing of processing resources and to enable more parallelism in program execution.

Instruction-Level Parallelism

At a much lower level of abstraction, modern processors can execute multiple instructions at one time, a property known as *instruction-level parallelism*. For example, early microprocessors, such as the 1978-vintage Intel 8086, required multiple (typically 3–10) clock cycles to execute a single instruction. More recent processors can sustain execution rates of 2–4 instructions per clock cycle. Any given instruction requires much longer from start to finish, perhaps 20 cycles or more, but the processor uses a number of clever tricks to process as many as 100 instructions at a time. In Chapter 4, we will explore the use of *pipelining*, where the actions required to execute an instruction are partitioned into different steps and the processor hardware is organized as a series of stages, each performing one of these steps. The stages can operate in parallel, working on different parts of different instructions. We will see that a fairly simple hardware design can sustain an execution rate close to 1 instruction per clock cycle.

Processors that can sustain execution rates faster than 1 instruction per cycle are known as *superscalar* processors. Most modern processors support superscalar operation. In Chapter 5, we will describe a high-level model of such processors. We will see that application programmers can use this model to understand the performance of their programs. They can then write programs such that the generated code achieves higher degrees of instruction-level parallelism and therefore runs faster.

Single-Instruction, Multiple-Data (SIMD) Parallelism

At the lowest level, many modern processors have special hardware that allows a single instruction to cause multiple operations to be performed in parallel, a mode known as *single-instruction, multiple-data* (SIMD) parallelism. For example, recent generations of Intel and AMD processors have instructions that can add 8 pairs of single-precision floating-point numbers (C data type `float`) in parallel.

These SIMD instructions are provided mostly to speed up applications that process image, sound, and video data. Although some compilers attempt to automatically extract SIMD parallelism from C programs, a more reliable method is to write programs using special *vector* data types supported in compilers such as GCC. We describe this style of programming in Web Aside OPT:SIMD, as a supplement to the more general presentation on program optimization found in Chapter 5.

1.9.3 The Importance of Abstractions in Computer Systems

The use of *abstractions* is one of the most important concepts in computer science. For example, one aspect of good programming practice is to formulate a simple application program interface (API) for a set of functions that allow programmers to use the code without having to delve into its inner workings. Different program-

하드웨어. 제12장에서는 병행성 및 그 사용에 대해 더 깊게 다루며, 처리 자원의 공유를 제공하고 프로그램 실행에서 더 많은 병렬성을 가능하게 합니다.

현대 프로세서는 추상화 수준이 훨씬 낮은 많은 명령을 한 번에 실행할 수 있도록 설계되어 있으며 이를 명령 수준 병렬성이라고 합니다. 예를 들어 1978년에 출시된 Intel 8086과 같은 초기 마이크로프로세서는 일반적으로 3~10개의 클럭 주기를 필요로 했지만, 최근의 프로세서는 클럭 주기당 2~4개의 명령을 실행할 수 있습니다. 특정 명령이 시작해서 끝까지 필요한 시간은 20개 이상의 주기가 소요될 수도 있지만, 프로세서는 다양한 속임수를 사용하여 한 번에 100개의 명령을 처리할 수 있습니다. 4장에서는 파이프라이닝의 사용을 탐구할 것인데, 이는 명령을 실행하는 데 필요한 작업을 여러 단계로 나누고, 프로세서 하드웨어를 각 단계를 수행하는 일련의 단계로 구성하는 것입니다. 이러한 단계는 병렬로 작동하여 다른 명령어의 다른 부분을 처리합니다. 우리는 꽤 단순한 하드웨어 디자인이 클럭 주기당 거의 1개의 명령을 지속적으로 실행할 수 있는 성능을 유지할 수 있음을 알게 될 것입니다. 1 클럭 주기당 1개 이상의 명령을 실행할 수 있는 프로세서를 슈퍼스칼라 프로세서라고 합니다. 대부분의 현대 프로세서는 슈퍼스칼라 운영을 지원합니다. 5장에서는 이러한 프로세서의 고수준 모델을 설명할 것인데, 응용 프로그래머는 이 모델을 사용하여 프로그램의 성능을 이해할 수 있습니다. 그들은 그 후에 프로그램을 작성하여 생성된 코드가 명령 수준의 병렬성을 높이고, 따라서 더 빨리 실행될 수 있도록 할 수 있습니다.

가장 낮은 수준에서 많은 현대 프로세서는 특별한 하드웨어를 가지고 있어 단일 명령어가 병렬로 여러 작업을 수행하게 할 수 있으며, 이를 단일 명령어 다중 데이터(SIMD) 병렬 처리 모드라고 합니다. 예를 들어, 최근의 Intel 및 AMD 프로세서 세대에는 단일 정밀도 부동 소수점 숫자(C 데이터 유형 float) 8쌍을 병렬로 더할 수 있는 명령어가 있습니다. 이러한 SIMD 명령어는 주로 이미지, 음향 및 비디오 데이터를 처리하는 애플리케이션의 속도를 높이기 위해 제공됩니다. 일부 컴파일러는 C 프로그램에서 SIMD 병렬 처리를 자동으로 추출하려고 시도하지만, 보다 신뢰할 수 있는 방법은 gcc와 같은 컴파일러에서 지원하는 특수 벡터 데이터 유형을 사용하여 프로그램을 작성하는 것입니다. 저희는 이러한 프로그래밍 스타일을 옵트:SIMD 웹 곁다리에서 설명하고 있으며, 이는 제5장에서 찾을 수 있는 프로그램 최적화에 대한 일반적인 소개의 보충 자료입니다.

추상화의 사용은 컴퓨터 과학에서 가장 중요한 개념 중 하나입니다. 예를 들어, 좋은 프로그래밍 실천의 한 측면은 프로그래머가 코드의 내부 구조에 심층적으로 파고들지 않고도 사용할 수 있는 함수 집합을 위한 간단한 응용 프로그램 인터페이스(API)를 작성하는 것입니다. 다른 프로그램-

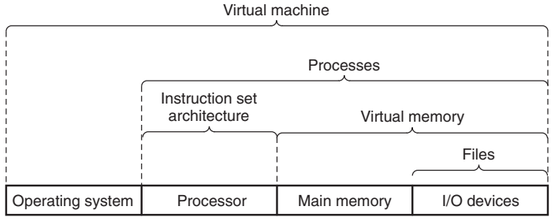


Figure 1.18
Some abstractions provided by a computer system. A major theme in computer systems is to provide abstract representations at different levels to hide the complexity of the actual implementations.

ming languages provide different forms and levels of support for abstraction, such as Java class declarations and C function prototypes.

We have already been introduced to several of the abstractions seen in computer systems, as indicated in Figure 1.18. On the processor side, the *instruction set architecture* provides an abstraction of the actual processor hardware. With this abstraction, a machine-code program behaves as if it were executed on a processor that performs just one instruction at a time. The underlying hardware is far more elaborate, executing multiple instructions in parallel, but always in a way that is consistent with the simple, sequential model. By keeping the same execution model, different processor implementations can execute the same machine code while offering a range of cost and performance.

On the operating system side, we have introduced three abstractions: *files* as an abstraction of I/O devices, *virtual memory* as an abstraction of program memory, and *processes* as an abstraction of a running program. To these abstractions we add a new one: the *virtual machine*, providing an abstraction of the entire computer, including the operating system, the processor, and the programs. The idea of a virtual machine was introduced by IBM in the 1960s, but it has become more prominent recently as a way to manage computers that must be able to run programs designed for multiple operating systems (such as Microsoft Windows, Mac OS X, and Linux) or different versions of the same operating system.

We will return to these abstractions in subsequent sections of the book.

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

Figure 1.18 컴퓨터 시스템이 제공하는 몇 가지 추상화. 컴퓨터 시스템의 주요 주제는 실제 구현의 복잡성을 숨기기 위해 다양한 수준에서 추상적인 표현을 제공하는 것입니다.

프로그래밍 언어는 Java 클래스 선언 및 C 함수 프로토타입과 같은 추상화의 다양한 형태와 수준을 제공합니다. 우리는 이미 Figure 1.18에서 보았던 컴퓨터 시스템의 여러 가지 추상화를 소개받았습니다. 프로세서 쪽에서는 명령어 집합 아키텍처가 실제 프로세서 하드웨어의 추상화를 제공합니다. 이 추상화를 통해 머신 코드 프로그램은 한 번에 하나의 명령어를 수행하는 프로세서에서 실행되는 것처럼 동작합니다. 기본 하드웨어는 병렬로 여러 명령어를 실행하지만 항상 간단한 순차 모델과 일관된 방식으로 실행됩니다. 같은 실행 모델을 유지하면서 다양한 프로세서 구현은 다양한 비용과 성능을 제공할 수 있습니다. 운영 체제 측면에서는 세 가지 추상화를 소개했습니다: I/O 디바이스의 추상화로서의 파일, 프로그램 메모리의 추상화로서의 가상 메모리, 실행 중인 프로그램의 추상화로서의 프로세스. 이러한 추상화에 새로운 것을 추가합니다: 전체 컴퓨터의 추상화를 제공하는 가상 머신. 가상 머신의 아이디어는 1960년대 IBM에 의해 소개되었지만, 최근에는 여러 운영 체제에 대한 프로그램을 실행할 수 있는 컴퓨터를 관리하기 위한 방법으로 더욱 중요해졌습니다 (예: Microsoft Windows, Mac OS X, Linux 등의 다양한 운영 체제 또는 같은 운영 체제의 다른 버전). 책의 후속 섹션에서 이러한 추상화에 다시 집중하겠습니다.

컴퓨터 시스템은 하드웨어와 시스템 소프트웨어로 이루어져 있으며 어플리케이션 프로그램을 실행하기 위해 협력합니다. 컴퓨터 내의 정보는 비트들의 그룹으로 표현되며 맥락에 따라 다르게 해석됩니다. 프로그램은 ASCII 텍스트로 시작해 컴파일러와 링커에 의해 이진 실행 파일로 번역됩니다. 프로세서는 메인 메모리에 저장된 이진 명령을 읽고 해석합니다. 컴퓨터는 대부분의 시간을 메모리, 입출력 장치 및 CPU 레지스터 간에 데이터를 복사하는 데 사용하므로 시스템의 저장 장치는 CPU 레지스터가 상단에 위치하고 하드웨어 캐시 메모리, DRAM 메인 메모리 및 디스크 저장과 같은 여러 수준을 거쳐 계층적으로 배열됩니다. 계층 구조에서 상위에 위치한 저장 장치일수록 하위 저장 장치보다 비트 당 속도가 빠르고 원가가 비쌀 수 있습니다.

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 as 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.

계층구조. 계층구조에서 상위에 있는 저장 장치는 하위에 있는 장치들의 캐시 역할을 합니다. 프로그래머들은 메모리 계층구조를 이해하고 활용함으로써 C 프로그램의 성능을 최적화할 수 있습니다. 운영 체제 커널은 응용프로그램과 하드웨어 사이의 중개자 역할을 합니다. 이는 3가지 기본적인 추상화를 제공합니다: (1) 파일은 I/O 장치의 추상화입니다. (2) 가상 메모리는 주 메모리와 디스크의 추상화입니다. (3) 프로세스는 프로세서, 주 메모리, 그리고 I/O 장치의 추상화입니다. 마지막으로, 네트워크는 컴퓨터 시스템들이 서로 통신할 수 있는 방법을 제공합니다. 특정 시스템의 관점에서 네트워크는 단순히 다른 I/O 장치일 뿐입니다.

Bibliographic Notes

Ritchie has written interesting firsthand accounts of the early days of C and Unix [91, 92]. Ritchie and Thompson presented the first published account of Unix [93]. Silberschatz, Galvin, and Gagne [102] provide a comprehensive 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. The Posix standards are available online at (www.unix.org).

리치가 C와 Unix 초기의 흥미로운 일대기를 직접 썼다. [91, 92]. 리치와 톰슨은 Unix의 최초 발표된 이력을 제시했다 [93]. 실버슈츠, 갈빈, 그리고 갠의 [102]은 Unix의 다양한 플레이버에 관한 포괄적인 역사를 제공한다. GNU (www.gnu.org)와 Linux (www.linux.org) 웹페이지에는 현재와 과거 정보가 풍부하다. Posix 표준은 (www.unix.org)에서 온라인으로 제공된다.

Solutions to Practice Problems

Solution to Problem 1.1 (page 58)

This problem illustrates that Amdahl's law applies to more than just computer systems.

A. In terms of Equation 1.1, we have $\alpha = 0.6$ and $k = 1.5$. More directly, traveling the 1,500 kilometers through Montana will require 10 hours, and the rest of the trip also requires 10 hours. This will give a speedup of $25/(10 + 10) = 1.25\times$.

B. In terms of Equation 1.1, we have $\alpha = 0.6$, and we require $S = 1.67$, from which we can solve for k. More directly, to speed up the trip by $1.67\times$, we must decrease the overall time to 15 hours. The parts outside of Montana will still require 10 hours, so we must drive through Montana in 5 hours. This requires traveling at 300 km/hr, which is pretty fast for a truck!

문제 1.1 해결책 (58페이지) 이 문제는 암달의 법칙이 컴퓨터 시스템뿐만 아니라 다른 영역에도 적용된다는 것을 보여줍니다.

방정식 1.1에 따르면, α = 0.6이고 k = 1.5이다. 보다 구체적으로 몬타나를 통해 1,500 킬로미터를 여행하는 데 10시간이 소요되고, 나머지 여행도 또한 10시간이 소요된다. 이는 25/(10 + 10) = 1.25× 의 가속화를 제공할 것이다.

식 1.1에 따르면, α = 0.6이고, S = 1.67이 필요하며, 이를 통해 k를 풀 수 있습니다. 더 직접적으로, 여행을 1.67배 속도를 내기 위해 총 시간을 15시간으로 줄여야 합니다. 몬타나 주 바깥 부분은 여전히 10시간이 걸릴 것이므로, 몬타나를 5시간 안에 통과해야 합니다. 이를 위해 300km/시속으로 운전해야 하며, 트럭으로서는 상당히 빠른 속도입니다!

Solution to Problem 1.2 (page 59)

Amdahl's law is best understood by working through some examples. This one requires you to look at Equation 1.1 from an unusual perspective. This problem is a simple application of the equation. You are given $S = 4$ and $\alpha = 0.9$, and you must then solve for k:

$$4 = 1/(1 - 0.9) + 0.9/k$$
$$0.4 + 3.6/k = 1.0$$
$$k = 6.0$$

문제 1.2의 해결책 (59페이지) Amdahl의 법칙은 몇 가지 예제를 통해 가장 잘 이해됩니다. 이것은 당신이 일반적이지 않은 관점에서 방정식 1.1을 살펴보도록 요구합니다. 이 문제는 방정식의 간단한 응용입니다. S = 4 및 α = 0.9가 주어지고, 그런 다음 k를 해결해야 합니다.