1 What is Concurrent Programming?

1.1 Introduction

An "ordinary" program consists of data declarations and assignment and control-flow statements in a programming language. Modern languages include structures such as procedures and modules for organizing large software systems through abstraction and encapsulation, but the statements that are actually executed are still the elementary statements that compute expressions, move data and change the flow of control. In fact, these are precisely the instructions that appear in the machine code that results from compilation. These machine instructions are executed *sequentially* on a computer and access data stored in the main or secondary memories.

A *concurrent program* is a set of sequential programs that can be executed in parallel. We use the word *process* for the sequential programs that comprise a concurrent program and save the term *program* for this set of processes.

Traditionally, the word *parallel* is used for systems in which the executions of several programs overlap in time by running them on separate processors. The word *concurrent* is reserved for potential parallelism, in which the executions may, but need not, overlap; instead, the parallelism may only be apparent since it may be implemented by sharing the resources of a small number of processors, often only one. Concurrency is an extremely useful *abstraction* because we can better understand such a program by pretending that all processes are being executed in parallel. Conversely, even if the processes of a concurrent program are actually executed in parallel on several processors, understanding its behavior is greatly facilitated if we impose an order on the instructions that is compatible with shared execution on a single processor. Like any abstraction, concurrent programming is important because the behavior of a wide range of real systems can be modeled and studied without unnecessary detail.

In this book we will define formal models of concurrent programs and study algorithms written in these formalisms. Because the processes that comprise a concur-

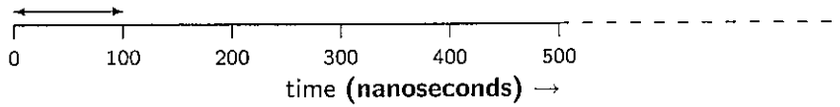
rent program may interact, it is exceedingly difficult to write a correct program for even the simplest problem. New tools are needed to specify, program and verify these programs. Unless these are understood, a programmer used to writing and testing sequential programs will be totally mystified by the bizarre behavior that a concurrent program can exhibit.

Concurrent programming arose from problems encountered in creating real systems. To motivate the concurrency abstraction, we present a series of examples of real-world concurrency.

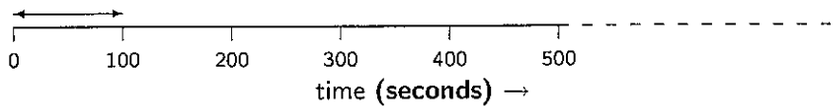
1.2 Concurrency as abstract parallelism

It is difficult to intuitively grasp the speed of electronic devices. The fingers of a fast typist seem to fly across the keyboard, to say nothing of the impression of speed given by a printer that is capable of producing a page with thousands of characters every few seconds. Yet these rates are extremely slow compared to the time required by a computer to process each character.

As I write, the *clock speed* of the central processing unit (CPU) of a personal computer is of the order of magnitude of one gigahertz (one billion times a second). That is, every nanosecond (one-billionth of a second), the hardware clock ticks and the circuitry of the CPU performs some operation. Let us roughly estimate that it takes ten clock ticks to execute one machine language instruction, and ten instructions to process a character, so the computer can process the character you typed in one hundred nanoseconds, that is 0.0000001 of a second:



To get an intuitive idea of how much effort is required on the part of the CPU, let us pretend that we are processing the character by hand. Clearly, we do not consciously perform operations on the scale of nanoseconds, so we will multiply the time scale by one billion so that every clock tick becomes a second:

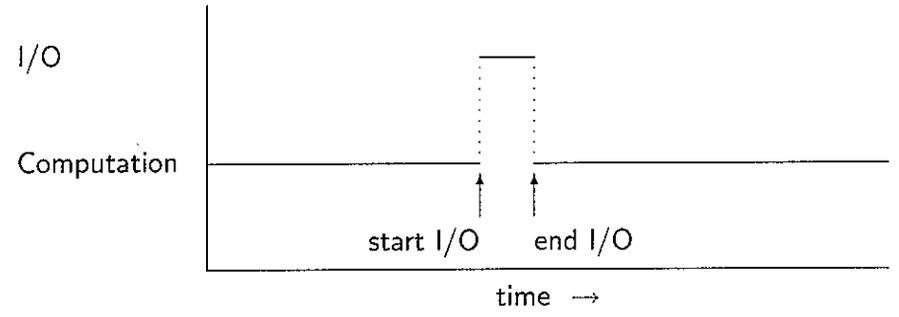


Thus we need to perform 100 seconds of work out of every *billion* seconds. How much is a billion seconds? Since there are $60 \times 60 \times 24 = 86{,}400$ seconds in a

day, a billion seconds is $1{,}000{,}000{,}000/86{,}400 = 11{,}574$ days or about 32 years. You would have to invest 100 seconds every 32 years to process a character, and a 3,000-character page would require only $(3{,}000 \times 100)/(60 \times 60) \approx 83$ hours over half a lifetime. This is hardly a strenuous job!

The tremendous gap between the speeds of human and mechanical processing on the one hand and the speed of electronic devices on the other led to the development of operating systems which allow I/O operations to proceed "in parallel" with computation. On a *single* CPU, like the one in a personal computer, the processing required for each character typed on a keyboard cannot really be done in parallel with another computation, but it is possible to "steal" from the other computation the fraction of a microsecond needed to process the character. As can be seen from the numbers in the previous paragraph, the degradation in performance will not be noticeable, even when the overhead of switching between the two computations is included.

What is the connection between concurrency and operating systems that overlap I/O with other computations? It would theoretically be possible for every program to include code that would periodically sample the keyboard and the printer to see if they need to be serviced, but this would be an intolerable burden on programmers by forcing them to be fully conversant with the details of the operating system. Instead, I/O devices are designed to interrupt the CPU, causing it to jump to the code to process a character. Although the processing is sequential, it is conceptually simpler to work with an abstraction in which the I/O processing performed as the result of the interrupt is a separate process, executed concurrently with a process doing another computation. The following diagram shows the assignment of the CPU to the two processes for computation and I/O.



1.3 Multitasking

Multitasking is a simple generalization from the concept of overlapping I/O with a computation to overlapping the computation of one program with that of another. Multitasking is the central function of the *kernel* of all modern operating systems. A *scheduler* program is run by the operating system to determine which process should be allowed to run for the next interval of time. The scheduler can take into account priority considerations, and usually implements *time-slicing*, where computations are periodically interrupted to allow a fair sharing of the computational resources, in particular, of the CPU. You are intimately familiar with multitasking; it enables you to write a document on a word processor while printing another document and simultaneously downloading a file.

Multitasking has become so useful that modern programming languages support it *within* programs by providing constructs for *multithreading*. Threads enable the programmer to write concurrent (conceptually parallel) computations within a single program. For example, interactive programs contain a separate thread for handling events associated with the user interface that is run concurrently with the main thread of the computation. It is multithreading that enables you to move the mouse cursor while a program is performing a computation.

1.4 The terminology of concurrency

The term *process* is used in the theory of concurrency, while the term *thread* is commonly used in programming languages. A technical distinction is often made between the two terms: a process runs in its own address space managed by the operating system, while a thread runs within the address space of a single process and may be managed by a multithreading kernel within the process. The term *thread* was popularized by *pthreads* (*POSIX threads*), a specification of concurrency constructs that has been widely implemented, especially on UNIX systems. The differences between processes and threads are not relevant for the study of the synchronization constructs and algorithms, so the term *process* will be used throughout, except when discussing threads in the Java language.

The term *task* is used in the Ada language for what we call a process, and we will use that term in discussions of the language. The term is also used to denote small units of work; this usage appears in Chapter 9, as well as in Chapter 13 on real-time systems where *task* is the preferred term to denote units of work that are to be scheduled.

1.5 Multiple computers

The days of one large computer serving an entire organization are long gone. Today, computers hide in unforeseen places like automobiles and cameras. In fact, your personal "computer" (in the singular) contains more than one processor: the graphics processor is a computer specialized for the task of taking information from the computer's memory and rendering it on the display screen. I/O and communications interfaces are also likely to have their own specialized processors. Thus, in addition to the multitasking performed by the operating systems kernel, parallel processing is being carried out by these specialized processors.

The use of multiple computers is also essential when the computational task requires more processing than is possible on one computer. Perhaps you have seen pictures of the "server farms" containing tens or hundreds of computers that are used by Internet companies to provide service to millions of customers. In fact, the entire Internet can be considered to be one *distributed system* working to disseminate information in the form of email and web pages.

Somewhat less familiar than distributed systems are *multiprocessors*, which are systems designed to bring the computing power of several processors to work in concert on a single computationally-intensive problem. Multiprocessors are extensively used in scientific and engineering simulation, for example, in simulating the atmosphere for weather forecasting and studying climate.

1.6 The challenge of concurrent programming

The challenge in concurrent programming comes from the need to *synchronize* the execution of different processes and to enable them to *communicate*. If the processes were totally independent, the implementation of concurrency would only require a simple scheduler to allocate resources among them. But if an I/O process accepts a character typed on a keyboard, it must somehow communicate it to the process running the word processor, and if there are multiple windows on a display, processes must somehow synchronize access to the display so that images are sent to the window with the current focus.

It turns out to be extremely difficult to implement safe and efficient synchronization and communication. When your personal computer "freezes up" or when using one application causes another application to "crash," the cause is generally an error in synchronization or communication. Since such problems are time- and situation-dependent, they are difficult to reproduce, diagnose and correct.

The aim of this book is to introduce you to the constructs, algorithms and systems that are used to obtain correct behavior of concurrent and distributed programs. The choice of construct, algorithm or system depends critically on assumptions concerning the requirements of the software being developed and the architecture of the system that will be used to execute it. This book presents a survey of the main ideas that have been proposed over the years; we hope that it will enable you to analyze, evaluate and employ specific tools that you will encounter in the future.

Transition

We have defined concurrent programming informally, based upon your experience with computer systems. Our goal is to study concurrency abstractly, rather than a particular implementation in a specific programming language or operating system. We have to carefully specify the abstraction that describe the allowable data structures and operations. In the next chapter, we will define the concurrent programming abstraction and justify its relevance. We will also survey languages and systems that can be used to write concurrent programs.

2 The Concurrent Programming Abstraction

2.1 The role of abstraction

Scientific descriptions of the world are based on abstractions. A living animal is a system constructed of organs, bones and so on. These organs are composed of cells, which in turn are composed of molecules, which in turn are composed of atoms, which in turn are composed of elementary particles. Scientists find it convenient (and in fact necessary) to limit their investigations to one level, or maybe two levels, and to "abstract away" from lower levels. Thus your physician will listen to your heart or look into your eyes, but he will not generally think about the molecules from which they are composed. There are other specialists, pharmacologists and biochemists, who study that level of abstraction, in turn abstracting away from the quantum theory that describes the structure and behavior of the molecules.

In computer science, abstractions are just as important. Software engineers generally deal with at most three levels of abstraction:

Systems and libraries Operating systems and libraries—often called Application Program Interfaces (API)—define computational resources that are available to the programmer. You can open a file or send a message by invoking the proper procedure or function call, without knowing how the resource is implemented.

Programming languages A programming language enables you to employ the computational power of a computer, while abstracting away from the details of specific architectures.

Instruction sets Most computer manufacturers design and build families of CPUs which execute the same instruction set as seen by the assembly language programmer or compiler writer. The members of a family may be implemented in totally different ways—emulating some instructions in software or using memory for registers—but a programmer can write a compiler for that instruction set without knowing the details of the implementation.