1

Introduction

Concurrent programming is the task of writing programs consisting of multiple independent threads of control, called processes. Conceptually, we view these processes as executing in parallel, but in practice their execution may be interleaved on a single processor. For this reason, we distinguish between concurrency in a programming language, and parallelism in hardware. We say that operations in a program are *concurrent* if they can be executed in parallel, and we say that operations in hardware are *parallel* if they overlap in time.

Operating systems, where there is a need to allow useful computation to be done in parallel with relatively slow input/output (I/O) operations, provide one of the earliest examples of concurrency. For example, during its execution, a program P might write a line of text to a printer by calling the operating system. Since this operation takes a relatively long time, the operating system initiates it, suspends P, and starts running another program Q. Eventually, the output operation completes and an *interrupt* is received by the operating system, at which point it can resume executing P. In addition to introducing parallelism and hiding latency, as in the case of slow I/O devices, there are other important uses of concurrency in operating systems. Using interrupts from a hardware interval timer, the operating system can multiplex the processor among a collection of user programs, which is called *time-sharing*. Most time-sharing operating systems allow user programs to interact, which provides a form of user-level concurrency. On multiprocessors, the operating systems are, by necessity, concurrent programs; furthermore, application programs may use concurrent programming to exploit the parallelism provided by the hardware.

One important motivation for concurrent programming is that processes are a useful abstraction mechanism: they provide encapsulation of state and control with well-defined interfaces. Unfortunately, if the mechanisms for concurrent programming are too expensive, then programmers will break the natural abstraction boundaries in order to ensure acceptable performance. The concurrency provided by operating system processes is

the most widespread mechanism for concurrent programming. But, there are several disadvantages with using system-level processes for concurrent programming: they are expensive to create and require substantial memory resources, and the mechanisms for interprocess communication are cumbersome and expensive.[1] As a result, even when faced with a naturally concurrent task, application programmers often choose complex sequential solutions to avoid the high costs of system-level processes.

Concurrent programming languages, on the other hand, provide notational support for concurrent programming, and generally provide lighter-weight concurrency, often inside a single system-level process. Thus, just as efficient subroutine linkages make procedural abstraction more acceptable, efficient implementations of concurrent programming languages make process abstraction more acceptable.

1.1 Concurrency as a structuring tool

This book focuses on the use of concurrent programming for applications with naturally concurrent structure. These applications share the property that flexibility in the scheduling of computation is required. Whereas sequential languages force a total order on computation, concurrent languages permit a partial order, which provides the needed flexibility.

For example, consider the **xrn** program, which is a popular UNIX program that provides a graphical user interface for reading network news. It is both an example of an interactive application and of a distributed-systems application, since it maintains a connection to a remote news server. This program has a rather annoying "feature" that is a result of its being programmed in a sequential language.[2] If **xrn** loses its connection to the remote news server (because the server goes down, or the connection times out), it displays a message window (or "*dialog box*") on the screen to inform the user of the lost connection. Unfortunately, after putting up the window, but before writing the message, **xrn** attempts to reestablish the connection, which causes it to hang until the server comes back on line. Thus, you have the phenomenon of a blank message window appearing on the user's screen, followed by a long pause, followed by the simultaneous display of two messages: the first saying that the connection has been lost, and a second saying that the connection has been restored. Besides being an example of poor interface design, this illustrates the kind of sequential orderings that concurrent programming easily avoids.

[1] It should be noted that most recent operating systems provide support for multiple threads of control inside a single protection domain.

[2] This anecdote refers to version 6.17 of **xrn**.

1.1.1 Interactive systems

Interactive systems, such as graphical user interfaces and window managers, are the primary motivation of much of the work described in this book, and the author believes that they are one of the most important application areas for concurrent programming. Interactive systems are typically programmed in sequential languages, which results in awkward program structures, since these programs are naturally concurrent. They must deal with multiple asynchronous input streams and support multiple contexts, while maintaining responsiveness. In the following discussion, we present a number of scenarios that demonstrate the naturally concurrent structure of interactive software.

User interaction

Handling user input is the most complex aspect of an interactive program. An application may be sensitive to multiple input devices, such as a mouse and keyboard, and may multiplex these among multiple input contexts (*e.g.*, different windows). Managing this many-to-many mapping is usually the province of *User Interface Management System* (UIMS) toolkits. Since most UIMS toolkits are implemented in sequential languages, they must resort to various techniques to emulate the necessary concurrency. Typically, these toolkits use an *event-loop* that monitors the stream of input events and maps the events to *call-back* functions (or *event handlers*) provided by the application programmer. In effect, this structure is a poor-man's concurrency: the event-handlers are coroutines, and the event-loop is the scheduler.

The call-back approach to managing user input leads to an unnatural program structure, known as the "*inverted program structure*," where the application program hands over control to the library's event-loop. While event-driven code is sometimes appropriate for an application, this choice should be up to the application programmer, and not be dictated by the library.

Multiple services

Interactive applications often provide multiple services; for example, a spreadsheet might provide an editor for composing macros, and a window for viewing graphical displays of the data, in addition to the actual spreadsheet. Each service is largely independent, having its own internal state and control-flow, so it is natural to view them as independent processes.

An additional benefit of using process abstraction to structure such services is that it makes replication of services fairly easy. This is because processes are reentrant by their very nature (*i.e.*, they typically encapsulate their own state). In our spreadsheet example, supporting multiple data sets or graphical views should be as easy as spawning an additional process.

This is a situation where an "object-oriented" language might also claim benefits, since

each service could be encapsulated as an object. In many respects, processes and objects provide the same kind of abstraction: hiding state and providing well-defined interfaces. The obvious difference is that concurrent languages provide the additional benefit of interleaved execution of services, whereas in a sequential object-oriented language the execution of services must be serialized.[3]

Interleaving computation

Even on a uniprocessor, it is possible to gain performance benefits from concurrency. An event driven interactive system has large amounts of "dead-time" between input events. If an application receives a hundred input events per second (which is quite a lot), then it has millions of instructions it can execute between events. Since most events can be handled in a few thousand instructions (*e.g.*, add a character to an input buffer, and echo it), there is an opportunity to execute other useful computation between input events. While multi-tasking operating systems support this kind of interleaving between processes, it is difficult to interleave computation with I/O inside a single sequential program. Applications that are programmed in a concurrent language, however, can exploit this interleaving for "free." Furthermore, as processors become faster, the amount of useful work that can be done between input events increases.

In addition to the performance benefits, interleaving computation is very important to the responsiveness of an application. For example, a user of a document preparation system may want to edit one part of a document while another part is being formatted. Since formatting may take a significant amount of time, providing a responsive interface requires interleaving formatting and editing. A traditional UIMS toolkit makes it difficult to write applications that provide responsive interactive interfaces and do substantial computation, since the application cannot handle events while computing. Typically, the computation must be broken into short-duration chunks that will not affect responsiveness. This structure has the added disadvantage of destroying the separation between the application and the interface. Concurrent programs, in contrast, avoid this problem without any additional complexity.

Output-driven interfaces

The inverted program structure required by traditional UIMS toolkits has the disadvantage of biasing a program's structure towards input. The application is quiescent until the user prods it, at which time it reacts in some way, and then waits for the next event. This is fine for many applications, such as text editors, which do not have much to do when the user's attention is elsewhere, but it is a hindrance for applications with *output-driven* interfaces.

[3] Of course, this restriction does not apply to concurrent object-oriented languages.

Consider, for example, a computationally intensive simulation that maintains a graphical display of its current state. The interface to this application is output-driven, since under normal circumstances the display is updated only when a new state is computed. But this application must also monitor window events, such as refresh and resize notifications, so that it can redraw itself when necessary. In a sequential implementation, the handling of these events must be postponed until the simulation is ready to update the displayed information. Separating the display code and simulation code into separate processes simplifies the handling of asynchronous redrawing.

Discussion

While the use of heavy-weight operating-system processes provides some support for multiple services and interleaved computation, it does not address the other two sources of concurrent program structure. Similarly, although event-loops and call-back functions provide flexibility in reacting to user input, they bias the application towards an *input-driven* model and do not provide much support for interleaved computation. A concurrent language, on the other hand, addresses all of these concerns.

1.1.2 Distributed systems

Distributed programs are, by their very nature, concurrent; each processor, or node, in a distributed system has its own state and control flow. But, in addition to the obvious concurrency that results from executing on multiple processors, there are natural uses of concurrency in the individual node programs.

As the story about **xrn** illustrated, synchronous communication over a network can block a program for unacceptable periods of time. This phenomenon is called *remote delay*, where the execution on one machine is delayed while waiting for a response from another. Using multiple threads to manage such communications allows other tasks to be interleaved with remote interactions.

Another performance problem that should be avoided in distributed programs is *local delay*, where execution is blocked because some resource is unavailable. For example, a file server might have to wait for a disk operation to complete before completing a file operation. In a server that allows multiple outstanding requests, local delay can be a serious bottleneck, since the delay associated with one request affects the servicing of other requests. A solution to this is to use a separate local process for each outstanding request; when one process experiences local delay, the others can still proceed. This is essentially the same as using concurrency to hide I/O latency in operating systems.

1.1.3 Other uses of concurrency

Reactive systems are another class of naturally concurrent programs. A reactive system is one that reacts to its environment at a speed determined by its environment. Examples of reactive systems include call processing, process control, and signal processing. User interfaces and, to some extent, the node programs in distributed systems might be considered reactive systems, although they do not have the hard real-time requirements of "pure" reactive systems.

Many other kinds of programs also have a naturally concurrent structure. Discrete event simulations, for example, can be structured as a collection of processes representing the objects being simulated. Concurrency can also be used to provide a different look at a sequential algorithm. For example, power series can be programmed elegantly as dataflow networks. And, of course, concurrency is important in expressing parallel algorithms.

1.2 High-level languages

Writing correct concurrent programs is a difficult task. In addition to the bugs that may arise in sequential programming, concurrent programs suffer from their own particular kinds of problems, such as races, deadlock, and starvation. What makes this even more difficult is that concurrent programs execute nondeterministically, which means that a program may work one time and fail the next. A common suggestion to address this problem is to use formal methods and logical reasoning to verify that one's program satisfies various properties. While this is a useful pedagogical tool, it does not scale well to large programs.

For large systems, one has to mostly rely on good design and careful implementation, which is where the choice of programming language makes an important difference. A high-level language with well-designed primitives is the most important ingredient in helping the programmer write correct and robust software.

The question of which programming notation to use is often the subject of religious debate. The trade-offs are fairly obvious: higher-level languages help programmer productivity and improve the robustness of software, while lower-level languages allow more programmer control over performance details. As implementation technology improves (and computer time becomes cheaper), the trend has been to migrate towards higher-level languages. While most people would agree that the performance lost when switching from assembly language to **C** is well worth the improved programmer productivity, there is no such consensus about the next step.

It is the view of the author that the higher-order language *Standard ML* (**SML**), which provides the sequential substrate for the programming examples in this book, is a good

candidate for the next step. **SML** provides a number of important benefits for programmers:

- A strong type system, which guarantees that programs will not have run-time type errors (or "dump core").
- A garbage collector, which improves modularity and prevents many kinds of memory management errors.
- An expressive module system, which supports large-scale software construction and code reuse.
- Higher-order functions and polymorphism, which further support code reuse.
- A datatype mechanism and pattern matching facility, which provide a concise notation for many data structures and algorithms.
- A well-defined semantics and independence from architectural details, which makes code truly portable.

SML also provides a level of performance that is becoming competitive with lower-level languages, such as **C**. In particular, there is a widely used and freely distributed **SML** system, called *Standard ML of New Jersey* (**SML/NJ**), which provides quite good performance.[4] It is the author's view that the notational advantages of programming in **SML** outweigh the slight loss of performance, when compared to **C**. A thesis of this book is that efficient system software can be written in a language such as **SML**.

1.3 Concurrent ML

This book uses a concurrent extension to **SML**, called *Concurrent ML* (**CML**), as its primary programming notation. **CML** can either be viewed as a library, since it compiles directly on top of **SML/NJ**, or as a language in its own right, since it defines a different programming discipline.

Following the tradition of **SML**, **CML** is designed to provide high-level, but efficient, mechanisms for concurrent programming. Its basic model of communication and synchronization is synchronous message-passing (or *rendezvous*) on typed channels. Creation of both threads and channels is dynamic. Most importantly, **CML** provides a mechanism for users to define new communication and synchronization abstractions. This allows proper isolation of the implementation details, improving the robustness of

[4] Of course, this is application specific. **SML** beats **C** on many symbolic applications, but does not compete as well on array-based numerical applications.

the system. **CML** also addresses performance issues by providing efficient concurrency primitives. **CML** has a number of features that make it a good programming notation for building large-scale systems software:

- **CML** is a *higher-order* concurrent language. Just as **SML** supports functions as first-class values, **CML** supports synchronous operations as first-class values. These values, called *events*, provide the tools for building new synchronization abstractions.
- It provides integrated support for I/O operations. Potentially blocking I/O operations, such as reading from an input stream, are full-fledged synchronous operations. Low-level support is also provided, from which distributed communication abstractions can be constructed.
- It provides automatic reclamation of threads and channels, once they become inaccessible. This permits a technique of speculative communication, which is not possible in other concurrent languages and libraries.
- Scheduling is preemptive, which ensures that a single thread cannot monopolize the processor. This allows "off-the-shelf" code to be incorporated in a concurrent thread without destroying interactive responsiveness.
- It has an efficient implementation. Concurrency operations, such as thread creation, context switching, and message passing are very fast, often faster than lower-level **C**-based libraries.
- **CML** is implemented on top of **SML/NJ** and is written completely in **SML**. It runs on every hardware and operating system combination supported by **SML/NJ** (although we have not had a chance to test all combinations).

CML, and the multithreaded user-interface toolkit **eXene**, which is built on top of **CML**, have been used for a number of substantial applications (*i.e.*, applications consisting of at least 10,000 lines of **SML**/**CML** code).

Notes

The importance of using concurrency in the construction of user-interface software has been argued by a number of people. Most notably by Pike [Pik89a, Pik94], and by Gansner and the author [RG86, GR92]. A recent paper by Hauser et al. describes the various paradigms of thread use in two large interactive software systems [HJT+93].

Liskov, Herlihy, and Gilbert give arguments as to why the node programs in distributed systems should either have dynamic thread creation or use one-way message

passing (which they call asynchronous communication) for communicating with other nodes [LHG86]. The basic argument is that without one of these features, avoiding performance problems with local and remote delays requires too much complexity. These arguments are also motivation for using a concurrent language, instead of a sequential one, to program the nodes. Many languages and toolkits for distributed programming provide some form of light-weight concurrency for use in the node programs. Examples include the language **Argus** [LS83], the language **SR** [AOCE88, AO93], and the **Isis** toolkit [BCJ+90].

In addition to the examples found in later chapters of this book, there are a number of examples of the application of concurrency in the literature. McIlroy describes the implementation of various power series computations using the language **newsqueak** [McI90]. Andrews and Olsson illustrate the use of **SR** with a number of small application examples, including parallel numerical computation and simulation [AO93]. A book by Armstrong, et al. gives examples of the use of the concurrent language **Erlang** for a number of applications, including database servers, real-time control, and telephony [AVWW96]. **CML** has also been used for experimental telephony applications [FO93].

It is expected that the reader is familiar with the 1997 revision of **SML**, since it is the primary notation used in this book. A number of books give good introductions to **SML**. The second edition of Paulson's book [Pau96] gives a comprehensive description of the language, concentrating mostly on "functional programming" using **SML**, but it also discusses the module system and imperative features. A book by Ullman provides an introduction to **SML** aimed at people with experience programming in traditional imperative languages, and has been recently revised for **SML**'97 [Ull98]. Other books include Reade's book on functional programming using **SML** [Rea89], Stansifer's primer on **ML** [Sta92], and a book on abstract data types in **SML** by Harrison [Har93]. A recent book by Okasaki describes many purely functional data structures [Oka98]; these can be particularly useful for concurrent programming since there is no danger of shared state. In addition, there are tutorials written by Harper [Har86] and Tofte [Tof89]. The formal definition of **SML** and commentary on the definition were published as a pair of books [MTH90, MT91]; a revised definition, which covers **SML**'97, has also been published [MTHM97]. In addition, the **SML** Basis Library Manual is available electronically from

```
http://standardml.org/Basis/index.html
```

and also as a book [GR04].

This book also uses libraries and features that are specific to the **SML/NJ** system; these are described as they are encountered. More detailed documentation is included in the **SML/NJ** distribution, and is also available from the **SML/NJ** home page. An abridged

version of the *CML Reference Manual* is provided in Appendix A. As described in the preface, both **SML/NJ** and **CML** are distributed electronically free of charge.