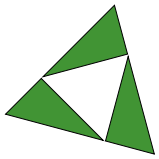
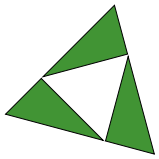


Contents

12 Concurrency **1**

12.1 Background and Motivation 2
12.1.1 A Little History 2
12.1.2 The Case for Multi-Threaded Programs 4
12.1.3 Multiprocessor Architecture 8
12.2 Concurrent Programming Fundamentals 11
12.2.1 Communication and Synchronization 12
12.2.2 Languages and Libraries 13
12.2.3 Thread Creation Syntax 14
12.2.4 Implementation of Threads 22
12.3 Shared Memory 26
12.3.1 Busy-Wait Synchronization 27
12.3.2 Scheduler Implementation 30
12.3.3 Scheduler-Based Synchronization 33
12.3.4 Implicit Synchronization 41
12.4 Message Passing 43
12.4.1 Naming Communication Partners 43
12.4.2 Sending 47
12.4.3 Receiving 50
12.4.4 Remote Procedure Call 56
Summary and Concluding Remarks 58
Review Questions 60
Exercises 61
Bibliographic Notes 66



Chapter 12

Concurrency

The bulk of this text has focused, implicitly, on *sequential* programs—programs with a single active execution context. As we saw in chapter 6, sequentiality is fundamental to imperative programming. It also tends to be implicit in declarative programming, partly because practical functional and logic languages usually include some imperative features, and partly because people tend to develop imperative implementations and mental models of declarative programs (applicative order reduction, backward chaining with backtracking), even when language semantics do not require such a model.

By contrast, a program is said to be *concurrent* if it contains more than one active execution context—more than one "thread of control". Concurrency arises for at least three important reasons:

To capture the logical structure of a problem. Many programs, particularly servers and graphical applications, must keep track of more than one largely independent "task" at the same time. Often the simplest and most logical way to structure such a program is to represent each task with a separate thread of control. We touched on this "multi-threaded" structure when discussing coroutines (section 8.6); we will return to it in section 12.1.2.

To cope with independent physical devices. Some software is by necessity concurrent. An operating system may be interrupted by a device at almost any time. It needs one context to represent what it was doing before the interrupt, and another for the interrupt itself. Likewise a system for real-time control (e.g. of a factory, or even an automobile) is likely to include a large number of processors, each connected to a separate machine or device. Each processor has its own thread(s) of control, which must interact with the threads on other processors to accomplish the overall objectives of the system. Message-routing software for the Internet is in some sense a very large concurrent program, running on thousands of servers around the world.

To increase performance by running on more than one processor at once. Even when concurrency is not dictated by the structure of a program or the hardware on which it has to run, we can often increase performance by choosing to have more than one processor work on the problem simultaneously. On a large multiprocessor, the resulting *parallel speedup* can be very large.

Section 12.1 contains a brief overview of the history of concurrent programming. It highlights major advances in parallel hardware and applications, makes the case for multithreaded programs (even on uniprocessors), and surveys the architectural features of modern multiprocessors. In section 12.2 we survey the many ways in which parallelism may be expressed in an application. We introduce the message-passing and shared-memory approaches to communication and synchronization, and note that they can be implemented either in an explicitly concurrent programming language or in a library package intended for use with a conventional sequential language. Building on coroutines, we explain how a language or library can create and schedule threads. In the two remaining sections (12.3 and 12.4) we look at shared memory and message passing in detail. Most of the shared-memory section is devoted to synchronization.

12.1 Background and Motivation

Concurrency is not a new idea. Much of the theoretical groundwork for concurrent programming was laid in the 1960's, and Algol 68 includes concurrent programming features. Widespread interest in concurrency is a relatively recent phenomenon however; it stems in part from the availability of low-cost multiprocessors and in part from the proliferation of graphical, multimedia, and web-based applications, all of which are naturally represented by concurrent threads of control.

Concurrency is an issue at many levels of a typical computer system. At the digital logic level, almost everything happens in parallel—signals propagate down thousands of connections at once. At the next level up, the pipelining and superscalar features of modern processors are designed to exploit the *instruction-level* parallelism available in well-scheduled programs. In this chapter we will focus on medium to large scale concurrency, represented by constructs that are semantically visible to the programmer, and that can be exploited by machines with many processors. In sections 12.1.3 and 12.3.4 we will also mention an intermediate level of parallelism available on special-purpose *vector* processors.

12.1.1 A Little History

The very first computers were single-user machines, used in *stand-alone* mode: people signed up for blocks of time, during which they enjoyed exclusive use of the hardware. Unfortunately, while single-user machines make good economic sense today, they constituted a terrible waste of resources in the late 1940's, when the cheapest computer cost millions of dollars. Rather than allow a machine to sit idle while the user examined output or pondered the source of a bug, computer centers quickly switched to a mode of operation in which users created *jobs* (sequences of programs and their input) off-line (e.g. on a keypunch machine) and then submitted them to an operator for execution. The operator would keep a *batch* of jobs constantly queued up for input on punch cards or magnetic tape. As its final operation, each program would transfer control back to a *resident monitor* program—a form of primitive operating system—which would immediately read the next program into memory for execution, from the current job or the next one, without operator intervention.

Unfortunately, this simple form of batch processing still left the processor idle much of the time, particularly on commercial applications, which tended to read a large number of data records from cards or tape, with comparatively little computation per record. To

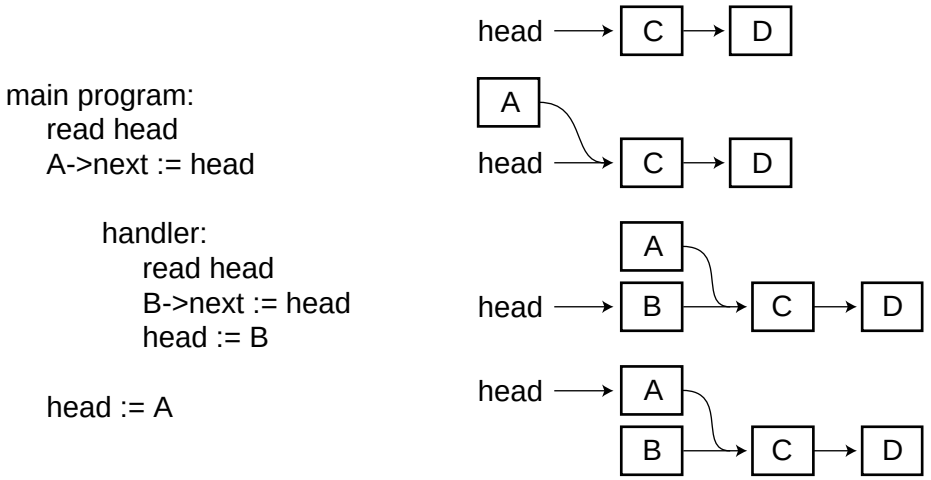


Figure 12.1: Example of a race condition. Here the currently running program attempts to insert a new element into the beginning of a list. In the middle of this operation, an interrupt occurs and the interrupt handler attempts to insert a different element into the list. In the absence of synchronization, one of the elements may become lost (unreachable from the head pointer).

perform an I/O operation (to write results to a printer or magnetic tape, or to read a new program or input data into memory), the processor in a simple batch system would send a command to the I/O device and then *busy-wait* for completion, repeatedly testing a variable that the device would modify when done with its operation. Given a punch card device capable of reading four cards per second, a 40 kHz vacuum-tube computer would waste 10,000 instructions *per card* while waiting for input. If it performed fewer than 10,000 instructions of computation on average before reading another card, the processor would be idle more than half the time! To make use of the cycles lost to busy-waiting, researchers developed techniques to *overlap* I/O and computation. In particular, they developed *interrupt-driven I/O*, which eliminates the need to busy-wait, and *multiprogramming*, which allows more than one application program to reside in memory at once. Both of these innovations required new hardware support: the former to implement interrupts, the latter to implement memory protection, so that errors in one program could not corrupt the memory of another.

On a multiprogrammed batch system, the operating system keeps track of which programs are waiting for I/O to complete and which are currently *runnable*. To read or write a record, the currently running program transfers control to the operating system. The OS sends a command to the device to start the requested operation, and then transfers control immediately to a different program (assuming one is runnable). When the device completes its operation, it generates an interrupt, which causes the processor to transfer back into the operating system. The OS notes that the earlier program is runnable again. It then chooses a program from among those that are runnable and transfers back to it. The only time the processor is idle is when *all* of the programs that have been loaded into memory are waiting for I/O.

Interrupt-driven I/O introduced concurrency within the operating system. Because an interrupt can happen at an arbitrary time, including when control is already in the operating

system, the interrupt handlers and the main bulk of the OS function as concurrent threads of control. If an interrupt occurs while the OS is modifying a data structure (e.g. the list of runnable programs) that may also be used by a handler, then it is possible for the handler to see that data structure in an inconsistent state (see figure 12.1). This problem is an example of a *race condition*: the thread that corresponds to the main body of the OS and the thread that corresponds to the device are "racing" toward points in the code at which they touch some common object, and the behavior of the system depends on which thread gets there first. To ensure correct behavior, we must *synchronize* the actions of the threads—i.e. take explicit steps to control the order in which their actions occur. We discuss synchronization further in section 12.3. It should be noted that not all race conditions are bad: sometimes any of the possible program outcomes is acceptable. The goal of synchronization is to resolve "bad" race conditions—those that might otherwise cause the program to produce incorrect results.

With increases in the size of physical memory, and with the development of virtual memory, it became possible to build systems with an almost arbitrary number of simultaneously loaded programs. Instead of submitting jobs off-line, users could now sit at a terminal and interact with the computer directly. To provide interactive response to keystrokes, however, the OS needed to implement *preemption*. Whereas a batch system switches from one program to another only when the first one blocks for I/O, a preemptive, *timesharing* system switches several times per second as a matter of course. These *context switches* prevent a compute-bound program from hogging the machine for seconds or minutes at a time, denying access to users at keyboards.

By the early 1970's, timesharing systems were relatively common. When augmented with mechanisms to allow data sharing or other forms of communication among currently runnable programs, they introduced concurrency in user-level applications. Shortly thereafter, the emergence of computer networks introduced true parallelism in the form of *distributed* systems—programs running on physically separate machines, and communicating with messages.

Most distributed systems reflect our second rationale for concurrency: they have to be concurrent in order to cope with multiple devices. A few reflect the third rationale: they are distributed in order to exploit the speedup available from multiple processors. Parallel speedup is more commonly pursued on single-chassis multiprocessors, with internal networks designed for very high bandwidth communication. Though multiprocessors have been around since the 1960's, they did not become commonplace until the 1980's.

12.1.2 The Case for Multi-Threaded Programs

Our first rationale for concurrency—to capture the logical structure of certain applications—has arisen several times in earlier chapters. In section 7.9.1 we noted that interactive I/O must often interrupt the execution of the current program. In a video game, for example, we must handle keystrokes and mouse or joystick motions while continually updating the image on the screen. By far the most convenient way to structure such a program is to represent the input handlers as concurrent threads of control, which coexist with one or more threads responsible for updating the screen. In section 8.6, we considered a screen saver program that used coroutines to interleave "sanity checks" on the file system with

updates to a moving picture on the screen. We also considered discrete-event simulation, which uses coroutines to represent the active entities of some real-world system.

The semantics of discrete-event simulation require that events occur atomically at fixed points in time. Coroutines provide a natural implementation, because they execute one at a time. In our other examples, however—and indeed in most "naturally concurrent" programs—there is no need for coroutine semantics. By assigning concurrent tasks to threads instead of to coroutines, we acknowledge that those tasks can proceed in parallel if more than one processor is available. We also move responsibility for figuring out which thread should run when from the programmer to the language implementation.

The need for multi-threaded programs has become particularly apparent in recent years with the development of web-based applications. In a browser such as Netscape Navigator or Internet Explorer (see figure 12.2), there are typically many different threads simultaneously active, each of which is likely to communicate with a remote (and possibly very slow) server several times before completing its task. When the user clicks on a link, the browser creates a thread to request the specified document. For all but the tiniest pages, this thread will then receive a long series of message "packets". As these packets begin to arrive the thread must format them for presentation on the screen. The formatting task is akin to typesetting: the thread must access fonts, assemble words, and break the words into lines. For many special tags within the page, the formatting thread will spawn additional threads: one for each image, one for the background if any, one to format each table, possibly more to handle separate frames. Each spawned thread will communicate with the server to obtain the information it needs (e.g. the contents of an image) for its particular task. The user, meanwhile, can access items in menus to create new browser windows, edit bookmarks, change preferences, etc., all in "parallel" with the rendering of page elements.

The use of many threads ensures that comparatively fast operations (e.g. display of text) do not wait for slow operations (e.g. display of large images). Whenever one thread *blocks* (waits for a message or I/O), the implementation automatically switches to a different thread. In a *preemptive* thread package, the implementation switches among threads at other times as well, to make sure than none of them hogs the CPU. Any reader who remembers the early, more sequential browsers will appreciate the difference that multi-threading makes in perceived performance and responsiveness.

Without language or library support for threads, a browser must either adopt a more sequential structure, or centralize the handling of all delay-inducing events in a single *dispatch loop* (see figure 12.3).[1] Data structures associated with the dispatch loop keep track of all the *tasks* the browser has yet to complete. The state of a task may be quite complicated. For the high-level task of rendering a page, the state must indicate which packets have been received and which are still outstanding. It must also identify the various subtasks of the page (images, tables, frames, etc.) so that we can find them all and reclaim their state if the user clicks on a "stop" button.

To guarantee good interactive response, we must make sure that no subaction of continue_task takes very long to execute. Clearly we must end the current action whenever we wait for a message. We must also end it whenever we read from a file, since disk operations are slow. Finally, if any task needs to compute for longer than about a tenth of a second (the typical human perceptual threshold), then we must divide the task into pieces, between

[1] We saw a simpler example of such a loop in section 8.6 (page ??).

```
procedure parse_page (address : url)
    contact server, request page contents
    parse_html_header
    while current_token in {"<p>", "<h1>", "<ul>", ...,
            "<background", "<image", "<table", "<frameset", ...}
        case current_token of
            "<p>"      : break_paragraph
            "<h1>"     : format_heading; match ("</h1>")
            "<ul>"     : format_list; match ("</ul>")
            ...
            "<background" :
                        a : attributes := parse_attributes
                        fork render_background (a)
            "<image"   : a : attributes := parse_attributes
                        fork render_image (a)
            "<table"   : a : attributes := parse_attributes
                        scan forward for "</table>" token
                        token_stream s :=...    -- table contents
                        fork format_table (s, a)
            "<frameset" :
                        a : attributes := parse_attributes
                        parse_frame_list (a)
                        match ("</frameset>")
        ...
    ...
procedure parse_frame_list (a1 : attributes)
    while current_token in {"<frame", "<frameset", "<noframes>"}
        case current_token of
            "<frame"   : a2 : attributes := parse_attributes
                        fork format_frame (a1, a2)
        ...
```

Figure 12.2: Thread-based code from a hypothetical WWW browser. To first approximation, the parse_page subroutine is the root of a recursive-descent parser for HTML. In several cases, however, the actions associated with recognition of a construct (background, image, table, frameset) proceed concurrently with continued parsing of the page itself. In this example, concurrent threads are created with the `fork` operation. Other threads would be created automatically in response to keyboard and mouse events.

```
type task_descriptor = record
    -- fields in lieu of thread-local variables, plus control-flow information
    ...
ready_tasks : queue of task_descriptor
...
procedure dispatch
    loop
        -- try to do something input-driven
        if a new event E (message, keystroke, etc.) is available
            if an existing task T is waiting for E
                continue_task (T, E)
            else if E can be handled quickly, do so
            else
                allocate and initialize new task T
                continue_task (T, E)
        -- now do something compute bound
            if ready_tasks is non-empty
                continue_task (dequeue (ready_tasks), 'ok')

procedure continue_task (T : task, E : event)
    if T is rendering an image
        and E is a message containing the next block of data
            continue_image_render (T, E)
    else if T is formatting a page
        and E is a message containing the next block of data
            continue_page_parse (T, E)
    else if T is formatting a page
        and E is 'ok'       -- we're compute bound
            continue_page_parse (T, E)
    else if T is reading the bookmarks file
        and E is an I/O completion event
            continue_goto_page (T, E)
    else if T is formatting a frame
        and E is a push of the "stop" button
            deallocate T and all tasks dependent upon it
    else if E is the "edit preferences" menu item
        edit_preferences (T, E)
    else if T is already editing preferences
        and E is a newly typed keystroke
            edit_preferences (T, E)
    ...
```

Figure 12.3: Dispatch loop from a hypothetical non-thread-based WWW browser. The clauses in continue_task must cover all possible combinations of task state and triggering event. The code in each clause performs the next coherent unit of work for its task, returning when (a) it must wait for an event, (b) it has consumed a significant amount of compute time, or (c) the task is complete. Prior to returning, respectively, code (a) places the task in a dictionary (used by dispatch) that maps awaited events to the tasks that are waiting for them, (b) enqueues the task in ready_tasks, or (c) deallocates the task.

which we save state and return to the top of the loop. These considerations imply that the condition at the top of the loop must cover the full range of asynchronous events, and that evaluations of the condition must be interleaved with continued execution of any tasks that were subdivided due to lengthy computation. (In practice we would probably need a more sophisticated mechanism than simple interleaving to ensure that neither input-driven nor compute-bound tasks hog more than their share of resources.)

The principal problem with a dispatch loop—beyond the complexity of subdividing tasks and saving state—is that it hides the algorithmic structure of the program. Every distinct task (retrieving a page, rendering an image, walking through nested menus) could be described elegantly with standard control-flow mechanisms, if not for the fact that we must return to the top of the dispatch loop at every delay-inducing operation. In effect, the dispatch loop turns the program "inside out", making the management of tasks explicit and the control flow within tasks implicit. A thread package turns the program "right-side out", making the management of tasks (threads) implicit and the control flow within threads explicit.

With the development of personal computers, much of the history of operating systems has repeated itself. Early PCs performed busy-wait I/O and ran one application at a time. With the development of Microsoft Windows and the Multifinder version of the MacOS, PC vendors added the ability to hold more than one program in memory at once, and to switch between them on I/O. Because a PC is a single-user machine, however, the need for preemption was not felt as keenly as in multi-user systems. For a long time it was considered acceptable for the currently running program to hog the processor: after all, that program is what the (single) user wants to run. As PCs became more sophisticated, however, users began to demand concurrent execution of threads such as those in a browser, as well as "background" threads that update windows, check for e-mail, babysit slow printers, etc. To some extent background computation can be accommodated by requiring every program to "voluntarily" yield control of the processor at well-defined "clean points" in the computation. This sort of "cooperative multiprogramming" was found in Windows 3.1 and MacOS version 7. Unfortunately, some programs do not yield as often as they should, and the inconsistent response of cooperatively multiprogrammed systems grew increasingly annoying to users. Windows 95 added preemption for 32-bit applications. Windows NT and MacOS X add preemption for all programs, running them in separate address spaces so bugs in one program don't damage another, or cause the machine to crash.

12.1.3 Multiprocessor Architecture

Single-chassis parallel computers can be grouped into two broad categories: those in which processors share access to common memory, and those in which they must communicate with messages. Some authors use the term *multicomputer* for message-passing machines, and reserve the term *multiprocessor* only for machines with shared memory. More commonly, *multiprocessor* is used for both classes, with the distinction being made by context or extra adjectives, e.g. *shared-memory multiprocessor*. The distinction between a multicomputer and a mere collection of computers on a network is that the former is more "tightly coupled", generally occupying a single cabinet or collection of cabinets, with fast, physically short interconnections and a common operating system. The distinction is sometimes a judgment call: one can buy fast, physically short interconnects for a collection of workstations, which

are then administered as a multicomputer. One can also use a commercial multicomputer (with remote terminals) in place of a workstation network.

Small shared-memory multiprocessors are usually *symmetric*, in the sense that all memory is equally distant from all processors. Large shared-memory multiprocessors usually display a *distributed memory* architecture, in which each memory bank is physically adjacent to a particular processor or small group of processors. Any processor can access the memory of any other, but local memory is faster. The small machines are sometimes called SMPs, for "symmetric multiprocessor." Their large cousins are sometimes called NUMA machines, for "non-uniform memory access".

Since the late 1960's, the market for high-end supercomputers has been dominated by so-called *vector processors*, which provide special instructions capable of applying the same operation to every element of an array. Vector instructions are very easy to pipeline. They are useful in many scientific programs, particularly those in which the programmer has explicitly annotated loops whose iterations can execute concurrently (we will discuss such loops in sections 12.2.3 and 12.3.4 below). Given current technological trends, however, it is widely believed that vector processors will be replaced in the next few years by machines built from general-purpose microprocessors. (At the same time, ideas from vector processors have made their way into the microprocessor world, e.g. in the form of the MMX extensions to the Pentium instruction set.)

From the point of view of a language or library implementor, the principal distinction between a message-based multicomputer and a shared-memory multiprocessor is that communication on the former requires the active participation of processors on both ends of the connection: one to send, the other to receive. On a shared-memory machine, a processor can read and write remote memory without the assistance of a remote processor. In most cases remote reads and writes use the same interface (i.e. load and store instructions) as local reads and writes. A few machines (e.g. the Cray T3E) support shared memory but require a processor to use special instruction sequences to access remote locations.

No matter what the communication model, every parallel computer requires some sort of interconnection network to tie its processors and memories together. Most small, symmetric machines are connected by a bus. A few are connected by a *crossbar* switch, in which every processor has a direct connection to every memory bank, forming a complete bipartite graph. Larger machines can be grouped into two camps: those with *indirect* and *direct* networks. An indirect network resembles a fishing net stretched around the outside of a cylinder (see figure 12.4). The "knots" in the net are message-routing switches. A direct network has no internal switches: all connections run directly from one node to another. Both indirect and direct networks have many topological variants. Indirect networks are generally designed so that the distance from any node to any other is $O(\log P)$, where P is the total number of nodes. The distance between nodes in a direct network may be as large as $O(\sqrt{P})$. In practice, a hardware technique known as *wormhole routing* makes communication with distant nodes almost as fast as with neighbors.

In any machine built from modern microprocessors, performance depends critically on very fast (low latency) access to memory. To minimize delays, almost all machines depend on caches. On a message-passing machine, each processor caches its own memory. On a shared memory machine, however, caches introduce a serious problem: unless we do something special, a processor that has cached a particular memory location will not see changes that are made to that location by other processors. This problem—how to keep

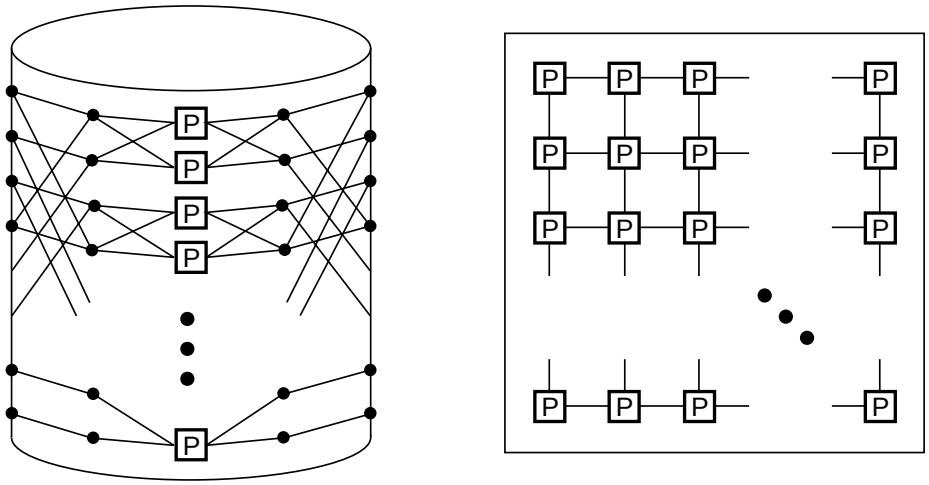


Figure 12.4: Multiprocessor network topology. In an *indirect* network (left), processing nodes are equally distant from one another. They communicate through a log-depth switching network. In a *direct* network (right), there are no switching nodes: each processing node sends messages through a small fixed number of neighbors.

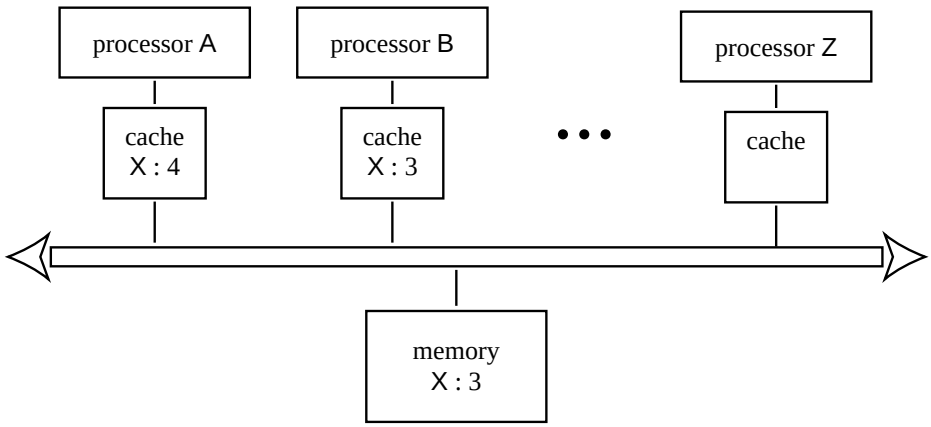


Figure 12.5: The cache coherence problem for shared-memory multiprocessors. Here processors A and B have both read variable X from memory. As a side effect, a copy of X has been created in the cache of each processor. If A now changes X to 4 and B reads X again, how do we ensure that the result is a 4 and not the still-cached 3? Similarly, if Z reads X into its cache, how do we ensure that it obtains the 4 from A's cache instead of the stale 3 from main memory?

cached copies of a memory location consistent with one another—is known as the *coherence* problem (see figure 12.5). On bus-based symmetric machines the problem is relatively easy to solve: the broadcast nature of the communication medium allows cache controllers to eavesdrop (*snoop*) on the memory traffic of other processors. When another processor writes a location that is contained in the local cache, the controller can either grab the new value off the bus or, more commonly, *invalidate* the affected cache line, forcing the processor to go back to memory (or to some other processor's cache) the next time the line is needed. Bus-based cache coherence algorithms are now a standard, built-in part of most commercial microprocessors. On large machines, the lack of a broadcast bus makes cache coherence a significantly more difficult problem; commercial implementations are available, but the subject remains an active topic of research.

As of 1999, small bus-based SMPs are available from dozens of manufacturers, with x86, MIPS, Alpha, PowerPC, Sparc, and PA/RISC processors. Larger, cache-coherent shared-memory multiprocessors are available from several manufacturers, including Convex, Sequent, and SGI. The Cray T3E is a large shared-memory multiprocessor *without* cache coherence; remote locations can be accessed directly, but are never cached. IBM's SP-2 is currently the leader among large, message-based multicomputers. The field is very much in flux: several large parallel machines and manufacturers have disappeared from the market in recent years; several more are scheduled to appear in the near future.

12.2 Concurrent Programming Fundamentals

We will use the word 'concurrency' to characterize any program in which two or more execution contexts may be active at the same time. Under this definition, coroutines are not concurrent, because only one of them can be active at once. We will use the term 'parallelism' to characterize concurrent programs in which execution is actually happening in more than one context at once. True parallelism thus requires parallel hardware. From a semantic point of view, there is no difference between true parallelism and the "quasi-parallelism" of a preemptive concurrent system, which switches between execution contexts at unpredictable times: the same programming techniques apply in both situations.

Within a concurrent program, we will refer to an execution context as a *thread*. The threads of a given program are implemented on top of one or more *processes* provided by the operating system. OS designers often distinguish between a *heavyweight* process, which has its own address space, and a collection of *lightweight* processes, which may share an address space. Lightweight processes were added to most variants of Unix in the late 1980's and early 1990's, to accommodate the proliferation of shared-memory multiprocessors. Without lightweight processes, the threads of a concurrent program must run on top of more than one heavyweight process, and the language implementation must ensure that any data that is to be shared among threads is mapped into the address space of all the processes.

We will sometimes use the word *task* to refer to a well-defined unit of work that must be performed by some thread. In one common programming idiom, a collection of threads shares a common "bag of tasks"—a list of work to be done. Each thread repeatedly removes a task from the bag, performs it, and goes back for another. Sometimes the work of a task entails adding new tasks to the bag.

Unfortunately, the vocabulary of concurrent programming is not consistent across languages or authors. Several languages call their threads processes. Ada calls them tasks.

Several operating systems call lightweight processes threads. The Mach OS, from which OSF Unix is derived, calls the address space shared by lightweight processes a task. A few systems try to avoid ambiguity by coining new words, such as 'actors' or 'filaments'. We will attempt to use the definitions of the preceding two paragraphs consistently, and to identify cases in which the terminology of particular languages or systems differs from this usage.

12.2.1 Communication and Synchronization

In any concurrent programming model, two of the most crucial issues to be addressed are *communication* and *synchronization*. Communication refers to any mechanism that allows one thread to obtain information produced by another. Communication mechanisms for imperative programs are generally based on either *shared memory* or *message passing*. In a shared-memory programming model, some or all of a program's variables are accessible to multiple threads. For a pair of threads to communicate, one of them writes a value to a variable and the other simply reads it. In a message-passing programming model, threads have no common state. For a pair of threads to communicate, one of them must perform an explicit `send` operation to transmit data to another.

Synchronization refers to any mechanism that allows the programmer to control the relative order in which operations occur in different threads. Synchronization is generally implicit in message-passing models: a message must be sent before it can be received. If a thread attempts to receive a message that has not yet been sent, it will wait for the sender to catch up. Synchronization is generally not implicit in shared-memory models: unless we do something special, a "receiving" thread could read the "old" value of a variable, before it has been written by the "sender". In both shared-memory and message-based programs, synchronization can be implemented either by *spinning* (also called *busy-waiting*) or by *blocking*. In busy-wait synchronization, a thread runs a loop in which it keeps reevaluating some condition until that condition becomes true (e.g. until a message queue becomes non-empty or a shared variable attains a particular value)—presumably as a result of action in some other thread, running on some other processor. Note that busy-waiting makes no sense for synchronizing threads on a uniprocessor: we cannot expect a condition to become true while we are monopolizing a resource (the processor) required to make it true. (A thread on a uniprocessor may sometimes busy-wait for the completion of I/O, but that's a different situation: the I/O device runs in parallel with the processor.)

In blocking synchronization (also called *scheduler-based* synchronization), the waiting thread voluntarily relinquishes its processor to some other thread. Before doing so, it leaves a note in some data structure associated with the synchronization condition. A thread that makes the condition true at some point in the future will find the note and take action to make the blocked thread run again. We will consider synchronization again briefly in section 12.2.4, and then more thoroughly in section 12.3.

As noted in section 12.1.3, the distinction between shared memory and message passing applies not only to languages and libraries, but to computer hardware as well. It is important to note that the model of communication and synchronization provided by the language or library need not necessarily agree with that of the underlying hardware. It is easy to implement message passing on top of shared-memory hardware. With a little more effort, one can also implement shared memory on top of message-passing hardware. Systems in this latter camp are sometimes referred to as *software distributed shared memory* (S-DSM).

12.2.2 Languages and Libraries

Concurrency can be provided to the programmer in the form of explicitly concurrent languages, compiler-supported extensions to traditional sequential languages, or library packages outside the language proper. The latter two approaches are by far the most common: the vast majority of parallel programs currently in use are either annotated Fortran for vector machines or C/C++ code with library calls.

Most SMP vendors provide a parallel programming library based on shared memory and threads. Most Unix vendors are converging on the Posix *pthreads* standard [Ope96]. For message-passing hardware, efforts to provide a shared-memory programming model (i.e. via S-DSM) are still considered experimental: most of the library packages for multicomputers and networks provide a message-based programming model. Message-based packages can in turn be grouped into those that are intended primarily for communication among the processes of a single program and those that are intended primarily for communication across program boundaries. Packages in this latter camp usually implement one of the standard Internet protocols [PD96, chap. 6], and bear a strong resemblance to file-based I/O (section 7.9).

The two most popular packages for message passing within a parallel program are PVM [Sun90, GBD+94] and MPI [BDH+95, SOHL+95]. The two packages provide similar functionality in most respects—enough so that their developers are thinking about merging them [GKP96]. PVM is richer in the area of creating and managing processes on a heterogeneous distributed network, in which machines of different types may join and leave the computation during execution. MPI provides more control over how communication is implemented (to map it onto the primitives of particular high-performance multicomputers), and a richer set of communication primitives, especially for so-called *collective communication*—one-to-all, all-to-one, or all-to-all patterns of messages among a set of threads. Implementations of PVM and MPI are available for C, C++, and Fortran.

For communication based on requests from clients to servers, *remote procedure calls* (RPCs) provide an attractive interface to message passing. Rather than talk to a server directly, an RPC client calls a local *stub* procedure, which packages its parameters into a message, sends them to a server, and waits for a response, which it returns to the client in the form of result parameters. Several vendors provide tools that will generate stubs automatically from a formal description of the server interface. In the Unix world, Sun's RPC [Sri95] is the de-facto standard. Several generalizations of RPC, most of them based on binary *components* (page ??), are currently competing for prominence for Internet-based computing [Bro96, Sie96, Sun97].

In comparison to library packages, an explicitly concurrent programming language has the advantage of compiler support. It can make use of syntax other than subroutine calls, and can integrate communication and thread management more tightly with concepts such as type checking, scoping, and exceptions. At the same time, since most programs are sequential, it is difficult for a concurrent language to gain widespread acceptance, particularly if the concurrent features make the sequential case more difficult to understand. As noted in section 12.1, Algol 68 included concurrent features, though they were never widely used. Concurrency also appears in more recent "mainstream" languages, including Ada, Modula-3, and Java. A little farther afield, but still commercially important, the Occam programming language [JG89], based on Hoare's Communicating Sequential Processes

(CSP) notation [Hoa78], has an active user community. Occam was the language of choice for systems built from the INMOS *transputer* processor, widely used in Europe but recently discontinued. Andrews's SR language [AO93] is widely used in teaching.

In the scientific community, expertise with vectorizing compilers has made its way into *parallelizing* compilers for multicomputers and multiprocessors, again exploiting annotations provided by the programmer. Several of the groups involved with this transition came together in the early 1990's to develop High Performance Fortran (HPF) [KLS+94], a *data-parallel* dialect of Fortran 90. (A data-parallel program is one in which the principal source of parallelism is the application of common operations to the members of a very large data set. A *task-parallel* program is one in which much of the parallelism stems from performing *different* operations concurrently. A data-parallel language is one whose features are designed for data-parallel programs.)

12.2.3 Thread Creation Syntax

One could imagine a concurrent programming system in which a fixed collection of threads was created by the language implementation, but such a static form of concurrency is generally too restrictive. Most concurrent systems allow the programmer to create new threads at run time. Syntactic and semantic details vary considerably from one language or library to another. There are at least six common options: (1) `co-begin`, (2) parallel loops, (3) launch-at-elaboration, (4) `fork` (with optional `join`), (5) implicit receipt, and (6) early reply. The first two options delimit threads with special control-flow constructs. The others declare threads with syntax resembling (or identical to) subroutines.

The SR programming language provides all six options. Algol 68 and Occam use `co-begin`. Occam also uses parallel loops, as does HPF. Ada uses both launch-at-elaboration and `fork`. Modula-3 and Java use `fork/join`. Implicit receipt is the usual mechanism in RPC systems. The coroutine `detach` operation of Simula can be considered a form of early reply.

Co-begin

In Algol 68 the behavior of a `begin`... `end` block depends on whether the internal expressions are separated by semicolons or commas. In the former case, we have the usual sequential semantics. In the latter case, we have either non-deterministic or concurrent semantics, depending on whether `begin` is preceded by the keyword `par`. The block

```
begin
    a := 3,
    b := 4
end
```

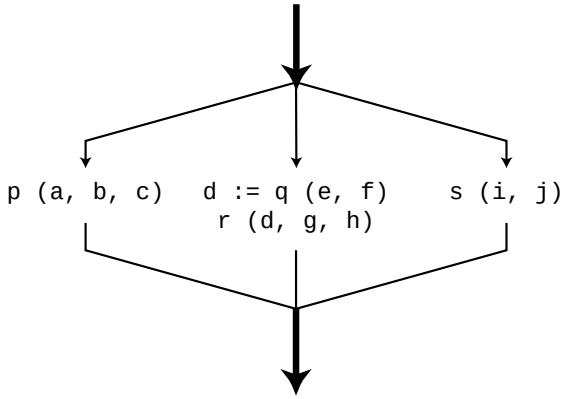
indicates that the assignments to `a` and `b` can occur in either order. The block

```
par begin
    a := 3,
    b := 4
end
```

indicates that they can occur in parallel. Of course, parallel execution makes little sense for such trivial operations as assignments; the `par begin` construct is usually used for more interesting operations:

```
par begin               # concurrent #
    p (a, b, c),
    begin               # sequential #
        d := q (e, f);
        r (d, g, h)
    end,
    s (i, j)
end
```

Here the executions of `p` and `s` can proceed in parallel with the sequential execution of the nested block (with the calls to `q` and `r`):



Several other concurrent languages provide a variant of `par begin`. In Occam, which uses indentation to delimit nested control constructs, one would write

```
par
  p (a, b, c)
  seq
    d := q (e, f)
    r (d, g, h)
  s (i, j)
```

In general, a control construct whose constituent statements are meant to be executed concurrently is known as `co-begin`. In an Algol 68 or Occam implementation, threads created by `co-begin` must share access to a common stack frame. To avoid this implementation complication, SR provides a variant of `co-begin` (delimited by `co...oc`) in which the constituent statements must all be procedure invocations.

Parallel loops

Several concurrent languages, including SR, Occam, and some dialects of Fortran, provide a loop whose iterations are to be executed concurrently. In SR one can say

```
co (i := 5 to 10) ->
    p (a, b, i)         # six instances of p, each with a different i
oc
```

In Occam:

```
par i = 5 for 6
  p (a, b, i)           # six instances of p, each with a different i
```

In SR it is the programmer's responsibility to make sure that concurrent execution is safe, in the sense that correctness will never depend on the outcome of race conditions. In the above example, access to global variables in the various instances of `p` would generally need to be synchronized, to make sure that those instances do not conflict with one another. In Occam, language rules prohibit conflicting accesses. The compiler checks to make sure that a variable that is written by one thread is neither read nor written by any concurrently active thread. In the code above, the Occam compiler would insist that all three parameters to `p` be passed by value (not result). Concurrently active threads in Occam communicate solely by sending messages.

Several parallel dialects of Fortran have provided parallel loops, with varying semantics. The `forall` loop adopted by HPF has since been incorporated into the 1995 revision of Fortran 90. Like the loops above, it indicates that iterations can proceed in parallel. To resolve race conditions, however, it imposes automatic, internal synchronization on the constituent statements of the loop, each of which must be an assignment statement or a nested `forall` loop. Specifically, all reads of variables in a given assignment statement, in all iterations, must occur before any write to the left-hand side, in any iteration. The writes of the left-hand side in turn must occur before any reads in the following assignment statement. In the following example, the first assignment in the loop will read $n-1$ elements of `B` and $n-1$ elements of `C`, and then update $n-1$ elements of `A`. Subsequently, the second assignment statement will read all n elements of `A` and then update $n-1$ of them.

```
forall (i=1:n-1)
    A(i) = B(i) + C(i)
    A(i+1) = A(i) + A(i+1)
end forall
```

Note in particular that all of the updates to `A(i)` in the first assignment statement occur before any of the reads in the second assignment statement. Moreover in the second assignment statement the update to `A(i+1)` is *not* seen by the read of `A(i)` in the "subsequent" iteration: the iterations occur in parallel and each reads the variables on its right-hand side before updating its left-hand side.

For loops that "iterate" over the elements of an array, the `forall` semantics are ideally suited for execution on a vector machine. With a little extra effort, they can also be adapted to a more conventional multiprocessor. In HPF, an extensive set of *data distribution* and *alignment* directives allows the programmer to scatter the elements of an array across the memory associated with a large number of processors. Within a `forall` loop, the computation in a given assignment statement is usually performed by the processor that "owns" the element on the assignment's left-hand side. In many cases an HPF or Fortran 95

compiler can prove that there are no dependences among certain (portions of) constituent statements of a `forall` loop, and can allow them to proceed without actually implementing synchronization.

Launch-at-elaboration

In Ada and SR (and in many other languages), the code for a thread may be declared with syntax resembling that of a subroutine with no parameters. When the declaration is elaborated, a thread is created to execute the code. In Ada (which calls its threads `task`s) we may write:

```
procedure P is
    task T is
        ...
    end T;
begin -- P
    ...
end P;
```

Task `T` has its own `begin`... `end` block, which it begins to execute as soon as control enters procedure `P`. If `P` is recursive, there may be many instances of `T` at the same time, all of which execute concurrently with each other and with whatever task is executing (the current instance of) `P`. The main program behaves like an initial default task.

When control reaches the end of procedure `P`, it will wait for the appropriate instance of `T` (the one that was created at the beginning of this instance of `P`) to complete before returning. This rule ensures that the local variables of `P` (which are visible to `T` under the usual static scope rules) are never deallocated before `T` is done with them.

A launch-at-elaboration thread in SR is called a `process`.

Fork/join

`Co-begin`, parallel loops, and launch-at-elaboration all lead to a concurrent control-flow pattern in which thread executions are properly nested (see figure 12.6(a)). With parallel loops, each thread executes the same code, using different data; with `co-begin` and launch-at-elaboration, the code in different threads can be different. Put another way, parallel loops are generally data-parallel; `co-begin` and launch-at-elaboration are task-parallel.

The `fork` operation is more general: it makes the creation of threads an explicit, executable operation. The companion `join` operation allows a thread to wait for the completion of a previously-`fork`ed thread. Because `fork` and `join` are not tied to nested constructs, they can lead to arbitrary patterns of concurrent control flow (figure 12.6(b)).

In addition to providing launch-at-elaboration tasks, Ada allows the programmer to define task *types*:

```
task type T is
    ...
begin
    ...
end T;
```

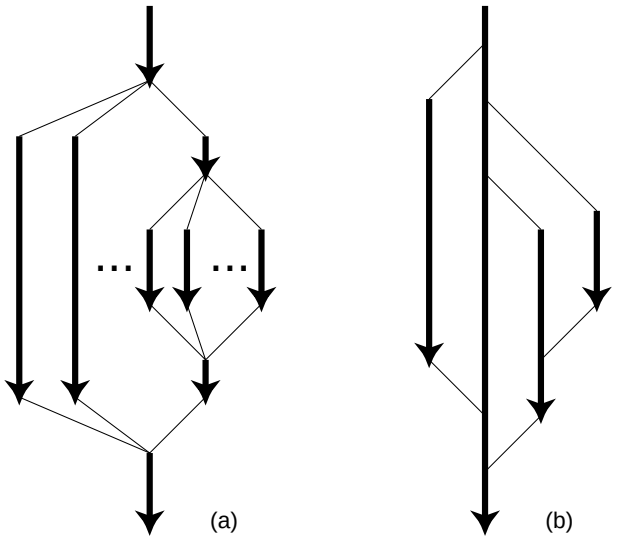


Figure 12.6: Lifetime of concurrent threads. With `co-begin`, parallel loops, or launch-at-elaboration (a), threads are always properly nested. With `fork/join` (b), more general patterns are possible.

The programmer may then declare variables of type `access T` (pointer to `T`), and may create new tasks via dynamic allocation:

```
pt : access T := new T;
```

The `new` operation is a `fork`: it creates a new thread and starts it executing. There is no explicit `join` operation in Ada, though parent and child tasks can always synchronize with one another explicitly if desired, e.g. immediately before the child completes its execution. In any scope in which a task type is declared, control will wait automatically at the end of the scope for all dynamically created tasks of that type to terminate. This convention avoids creating dangling references to local variables (Ada stack frames have limited extent).

Modula-3 provides both `fork` and `join`. The `fork` operation returns a reference of type `thread`; the `join` operation takes this reference as parameter:

```
t := Fork (c);
...
Join (t);
```

Each Modula-3 thread begins execution in a specified subroutine. The language designers could have chosen to make this subroutine the argument to `Fork`, but this choice would have forced all `Fork`ed subroutines to accept the same fixed set of parameters, in accordance with strong typing. To avoid this limitation, Modula-3 defines the parameter to `Fork` to be a "thread closure"[2] object (Modula-3 has object-oriented features). The object contains a reference to the thread's initial subroutine, together with any needed start-up arguments.

[2]Thread closures should not be confused with the closures used for deep binding of subroutine referencing environments, as described in section 3.4. Modula-3 uses closures in the traditional sense of the word when

The `Fork` operation calls the specified subroutine, passing a single argument: a reference to the thread closure object itself. The standard thread library defines a thread closure class with nothing in it except the subroutine reference. Programmers can define derived classes that contain additional fields, which the thread's subroutine can then access. There is no comparable mechanism to pass start-up arguments to a task in Ada; information that would be passed as thread closure fields in Modula-3 must be sent to the already-started task in Ada via messages or shared variables.

Threads may be created in SR by `send`ing a message to a `proc`, which resembles a procedure with a separate forward declaration, called an `op`. One of the most distinctive characteristics of SR is a remarkably elegant integration of sequential and concurrent constructs, and of message passing and subroutine invocation. An SR procedure is actually defined as syntactic sugar for an `op`/`proc` pair that has been limited to `call` style `fork`s, in which the parent thread waits for the child to complete before continuing execution. As in Ada, there is no explicit `join` operation in SR, though a parent and child can always synchronize with one another explicitly if desired.

In Java one obtains a thread by constructing an object of some class derived from a predefined class called `Thread`:

```
class image_renderer extends Thread {
    ...
    public void image_renderer ( args ) {
        ---- constructor
    }
    public void run () {
        ---- code to be run by the thread
    }
}
...
image_renderer rend = new image_renderer ( constructor_args );
```

Superficially, the use of `new` resembles the creation of dynamic tasks in Ada. In Java, however, the new thread does *not* begin execution when first created. To start it, the parent (or some other thread) must call the member function (method) named `start`, which is defined in `Thread`:

```
rend.start ();
```

`Start` makes the thread runnable, arranges for it to execute a member function named `run`, and returns to the caller. The programmer must define an appropriate `run` method in every class derived from `Thread`. The `run` method is meant to be called only by `start`; programmers should not call it directly, nor should they redefine `start`. There is also a `join` method:

---

passing subroutines as parameters, but because its local objects have limited extent (again, see section 3.4), it does not allow nested subroutines to be returned from functions or assigned into subroutine-valued variables. The subroutine reference in a thread "closure" is therefore guaranteed not to require a special referencing environment; it can be implemented as just a code address.

```
rend.join ();      // wait for completion
```

Implicit receipt

The mechanisms described in the last few paragraphs allow a program to create new threads at run time. In each case those threads run in the same address space as the existing threads. In RPC systems it is often desirable to create a new thread automatically in response to an incoming request from some *other* address space. Rather than have an existing thread execute a `receive` operation, a server can *bind* a communication channel (which may be called a link, socket, or connection) to a local thread body or subroutine. When a request comes in, a new thread springs into existence to handle it.

In effect, the `bind` operation grants remote clients the ability to perform a `fork` within the server's address space. In SR the effect of `bind` is achieved by declaring a *capability* variable, initializing it with a reference to a procedure (an `op` for which there is a `proc`), and then sending it in a message to a thread in another address space. The receiving thread can then use that capability in a `send` or `call` operation, just as it would use the name of a local `op`. When it does so, the resulting message has the effect of performing a `fork` in the original address space. In RPC stub systems designed for use with ordinary sequential languages, the creation and management of threads to handle incoming calls is often less than completely automatic; we will consider the alternatives in section 12.4.4 below.

Early reply

The similarity of `fork` and implicit receipt in SR reflects an important duality in the nature of subroutines. We normally think of sequential subroutines in terms of a single thread which saves its current context (its program counter and registers), executes the subroutine, and returns to what it was doing before (figure 12.7 (a)). The effect is the same, however, if we have two threads: one that executes the caller and another that executes the callee (figure 12.7 (b)). The caller waits for the callee to *reply* before continuing execution. The call itself is a `fork`/`join` pair, or a `send` and `receive` on a communication channel that has been set up for implicit receipt on the callee's end.

The two ways of thinking about subroutine calls suggest two different implementations, but either can be used to implement the other. In general, a compiler will want to avoid creating a separate thread whenever possible, in order to save time. As noted in the subsection on `fork`/`join` above, SR uses the two-thread model of subroutine calls. Within a single address space, however, it implements them with the usual subroutine-call mechanism whenever possible. In a similar vein, the Hermes language [SBG+91], which models subroutines in terms of threads and message passing, is able to use the usual subroutine-call implementation in the common case.

If we think of subroutines in terms of separate threads for the caller and callee, there is actually no particular reason why the callee should have to complete execution before it allows the caller to proceed—all it really has to do is complete the portion of its work on which result parameters depend. *Early reply* is a mechanism that allows a callee to return those results to the caller without terminating. After an early reply, the caller and callee continue execution concurrently (figure 12.7 (c)).

If we think of subroutines in terms of a single thread for the caller and callee, then early reply can also be seen as a means of creating new threads. When the thread executing

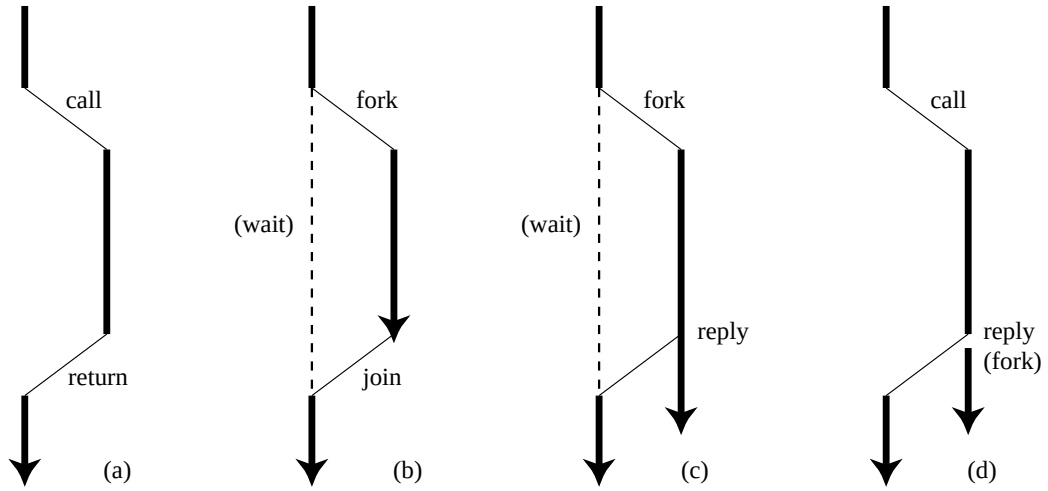


Figure 12.7: Threads, subroutine calls, and early reply. Conventionally, subroutine calls are conceptualized as using a single thread (a). Equivalent functionality can be achieved with separate threads (b). Early reply (c) allows a forked thread to continue execution after "returning" to the caller. To avoid creation of a callee thread in the common case, we can wait until the reply to do the fork (d).

a subroutine performs an early reply, it splits itself into a pair of threads: one of these returns, the other continues execution in the callee (figure 12.7 (d)). In SR, any subroutine can execute an early reply operation:

```
reply
```

For calls within a single address space, the SR compiler waits until the `reply` before creating a new thread; a subroutine that returns without `reply`ing uses a single implementation thread for the caller and callee. Until the time of the `reply`, the stack frame of the subroutine belongs to the calling thread. To allow it to become the initial frame of a newly created thread, an SR implementation can employ a memory management scheme in which stack frames are allocated dynamically from the heap and linked together with pointers. Alternatively, the implementation can copy the current frame into the bottom of a newly allocated stack at the time of the reply. A fully general cactus stack (as described in section 8.6.1) is not required in SR: every thread is created in its own subroutine, and subroutines do not nest. Early reply resembles the coroutine `detach` operation of Simula. It also appears in Lynx [Sco91].

Much of the motivation for early reply comes from applications in which the parent of a newly created thread needs to ensure that the thread has been initialized properly before it (the parent) continues execution. In a web browser, for example, the thread responsible for formatting a page will create a new child for each in-line image. The child will contact the appropriate server and begin to transfer data. The first thing the server will send is an indication of the image's size. The page-formatting thread (the parent of the image-rendering thread) needs to know this size in order to place text and other images properly on the page. Early reply allows the parent to create the child and then wait for it to reply

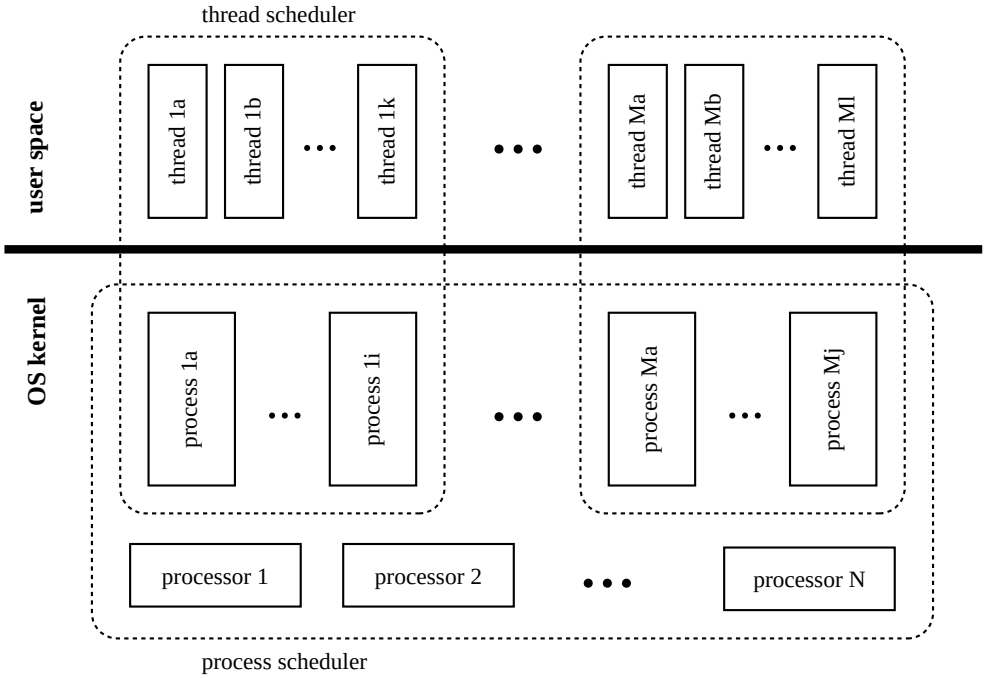


Figure 12.8: Two-level implementation of threads. A thread scheduler, implemented in a library or language run-time package, multiplexes threads on top of one or more kernel-level processes, just as the process scheduler, implemented in the operating system kernel, multiplexes processes on top of one or more physical processors.

with size information, at which point the parent and child can proceed in parallel. (We ignored this issue in figure 12.2.)

In Java, a similar purpose is served by separating thread creation from invocation of the `start` method. In our browser example, a page-formatting thread that creates a child to render an image could call a `get_size` method of the child *before* it calls the child's `start` method. `Get_size` would make the initial contact with the server and return size information to the parent. Because `get_size` is a function member of the child, any data it initializes, including the size and connection-to-server information, will be stored in the thread's data members, where they will be available to the thread's `run` method.

12.2.4 Implementation of Threads

As we noted in section 12.2, the threads of a concurrent program are usually implemented on top of one or more *processes* provided by the operating system. At one extreme, we could use a separate OS process for every thread; at the other extreme we could multiplex all of a program's threads on top of a single process. On a personal computer with a single address space and relatively inexpensive processes, the one-process-per-thread extreme is often acceptable. In a simple language on a uniprocessor, the all-threads-on-one-process extreme may be acceptable. Commonly, language implementations adopt an in-between approach, with a potentially large number of threads running on top of a smaller number of processes (see figure 12.8).

The problem with putting every thread on a separate process is that processes (even "lightweight" ones) are simply too expensive in many operating systems. Because they are implemented in the kernel, performing any operation on them requires a system call. Because they are general-purpose, they provide features that most languages do not need, but have to pay for anyway. (Examples include separate address spaces, priorities, accounting information, and signal and I/O interfaces, all of which are beyond the scope of this book.) At the other extreme, there are two problems with putting all threads on top of a single process: first, it precludes parallel execution on a multiprocessor; second, if the currently running thread makes a system call that blocks (e.g. waiting for I/O), then none of the program's other threads can run, because the single process is suspended by the OS.

In the common two-level organization of concurrency (user-level threads on top of kernel-level processes), similar code appears at both levels of the system: the language run-time system implements threads on top of one or more processes in much the same way that the operating system implements processes on top of one or more physical processors. A multiprocessor operating system may attempt to ensure that processes belonging to the same application run on separate processors simultaneously, in order to minimize synchronization delays (this technique is called *co-scheduling*, or *gang scheduling*). Alternatively, it may give an application exclusive use of some subset of the processors (this technique is called *space sharing*, or *processor partitioning*). Such kernel-level issues are beyond the scope of this book; we concentrate here on user-level threads.

Typically, user-level threads are built on top of coroutines (section 8.6). Recall that coroutines are a sequential control-flow mechanism, designed for implementation on top of a single OS process. The programmer can suspend the current coroutine and resume a specific alternative by calling the `transfer` operation. The argument to `transfer` is typically a pointer to the context block of the coroutine.

To turn coroutines into threads, we can proceed in a series of three steps. First, we hide the argument to `transfer` by implementing a *scheduler* that chooses which thread to run next when the current thread yields the processor. Second, we implement a *preemption* mechanism that suspends the current thread automatically on a regular basis, giving other threads a chance to run. Third, we allow the data structures that describe our collection of threads to be shared by more than one OS process, possibly on separate processors, so that threads can run on any of the processes.

Uniprocessor scheduling

Figure 12.9 illustrates the data structures employed by a simple scheduler. At any particular time, a thread is either *blocked* (i.e. for synchronization) or *runnable*. A runnable thread may actually be running on some processor or it may be awaiting its chance to do so. Context blocks for threads that are runnable but not currently running reside on a queue called the *ready list*. Context blocks for threads that are blocked for scheduler-based synchronization reside in data structures (usually queues) associated with the conditions for which they are waiting. To yield the processor to another thread, a running thread calls the scheduler:

```
procedure reschedule
    t : thread := dequeue (ready_list)
    transfer (t)
```

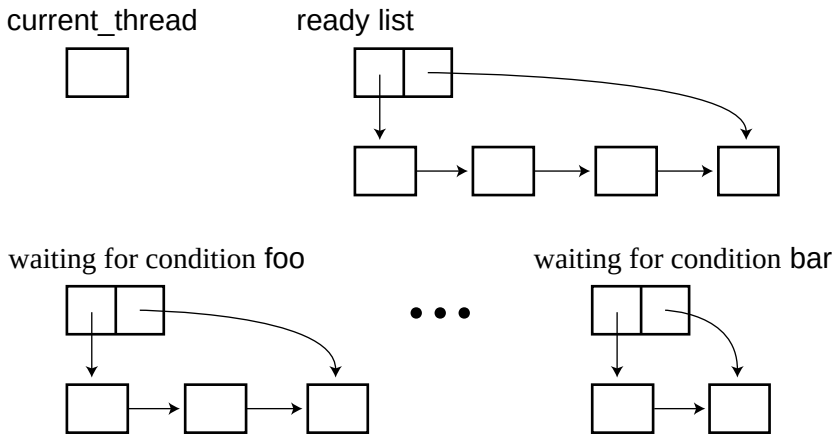


Figure 12.9: Data structures of a simple scheduler. A designated `current_thread` is running. Threads on the ready list are runnable. Other threads are blocked, waiting for various conditions to become true. If threads run on top of more than one OS-level process, each such process will have its own `current_thread` variable. If a thread makes a call into the operating system, its process may block in the kernel.

Before calling into the scheduler, a thread that wants to run again at some point in the future must place its own context block in some appropriate data structure. If it is blocking for the sake of fairness—to give some other thread a chance to run—then it enqueues its context block on the ready list:

```
procedure yield
    enqueue (ready_list, current_thread)
    reschedule
```

To block for synchronization, a thread adds itself to a queue associated with the awaited condition:

```
procedure sleep_on (ref Q : queue of thread)
    enqueue (Q, current_thread)
    reschedule
```

When a running thread performs an operation that makes a condition true, it removes one or more threads from the associated queue and enqueues them on the ready list.

Fairness becomes an issue whenever a thread may run for a significant amount of time while other threads are runnable. To give the illusion of concurrent activity, even on a uniprocessor, we need to make sure that each thread gets a frequent "slice" of the processor. With *cooperative multi-threading*, any long-running thread must yield the processor explicitly from time to time (e.g. at the tops of loops), to allow other threads to run. As noted in section 12.1.2, this approach allows one improperly written thread to monopolize the system. Even with properly written threads, it leads to less than perfect fairness due to non-uniform times between `yield`s in different threads.

Preemption

Ideally, we should like to multiplex the processor fairly and at a relatively fine grain (i.e. many times per second) *without* requiring that threads call `yield` explicitly. On many systems we can do this in the language implementation by using timer signals for *preemptive multithreading*. When switching between threads we ask the operating system (which has access to the hardware clock) to deliver a signal to the currently running process at a specified time in the future. The OS delivers the signal by saving the context (registers and `pc`) of the process at the top of the current stack and transferring control to a previously specified *handler* routine in the language run-time system. When called, the handler modifies the state of the currently running thread to make it appear that the thread had just executed a call to the standard `yield` routine. The handler then "returns" into `yield`, which transfers control to some other thread, as if the one that had been running had relinquished control of the process voluntarily.

Unfortunately, the fact that a signal may arrive at an arbitrary time introduces a race between voluntary calls to the scheduler and the automatic calls triggered by preemption. To illustrate the problem, suppose that a signal arrives when the currently running process has just enqueued the currently running thread onto the ready list in `yield`, and is about to call `reschedule`. When the signal handler "returns" into `yield`, the process will put the current thread into the ready list a second time. If at some point in the future the thread blocks for synchronization, its second entry in the ready list may cause it to run again immediately, when it should be waiting. Even worse problems can arise if a signal occurs in the middle of an `enqueue`, at a moment when the ready list is not even a properly structured queue. To resolve the race and avoid corruption of the ready list, thread packages commonly disable signal delivery during scheduler calls:

```
procedure yield
    disable_signals
    enqueue (ready_list, current_thread)
    reschedule
    reenable_signals
```

For this convention to work, *every* fragment of code that calls `reschedule` must disable signals prior to the call, and must reenable them afterward. Because `reschedule` contains a call to `transfer`, signals may be disabled in one thread, and reenabled in another.

It turns out that the `sleep_on` routine must also assume that signals are disabled and enabled by the caller. To see why, suppose that a thread checks a condition, finds that it is false, and then calls `sleep_on` to suspend itself on a queue associated with the condition. Suppose further that a timer signal occurs immediately after checking the condition, but before the call to `sleep_on`. Finally, suppose that the thread that is allowed to run after the signal makes the condition true. Since the first thread never got a chance to put itself on the condition queue, the second thread will not find it to make it runnable. When the first thread runs again, it will immediately suspend itself, and may never be awakened. To close this *timing window*—this internal in which a concurrent event may compromise program correctness—the caller must ensure that signals are disabled before checking the condition:

```
disable_signals
if not desired_condition
    sleep_on (condition_queue)
reenable_signals
```

On a uniprocessor, disabling signals allows the check and the sleep to occur as a single, *atomic* operation—they always appear to happen "all at once" from the point of view of other threads.

Multiprocessor scheduling

A few concurrent languages (e.g. Distributed Processes [Bri78] and Lynx [Sco91]) are explicitly non-parallel: language semantics guarantee that only one thread will run in a given address space at a time, with switches among threads occurring only at well-defined points in the code. Most concurrent languages, however, permit threads to run in parallel. As we noted in section 12.2, there is no difference from the programmer's point of view between true parallelism (on multiple processors) and the "quasi-parallelism" of a system that switches between threads on timer interrupts: in both cases, threads must synchronize explicitly to cope with race conditions in the application program.

We can extend our preemptive thread package to run on top of more than one OS-provided process by arranging for the processes to share the ready list and related data structures (condition queues, etc.—note that each process must have a *separate* `current_thread` variable). If the processes run on different processors of a shared-memory multiprocessor, then more than one thread will be able to run at once. If the processes share a single processor, then the program will be able to make forward progress even when all but one of the processes are blocked in the operating system. Any thread that is runnable is placed in the ready list, where it becomes a candidate for execution by any of the application's processes. When a process calls `reschedule`, the queue-based ready list we have been using in our examples will give it the longest-waiting thread. The ready list of a more elaborate scheduler might give priority to interactive or time-critical threads, or to threads that last ran on the current processor, and may therefore still have data in the cache.

Just as preemption introduced a race between voluntary and automatic calls to scheduler operations, true or quasi-parallelism introduces races between calls in separate OS processes. To resolve the races, we must implement additional synchronization to make scheduler operations in separate processes atomic. We will return to this subject in section 12.3.2.

12.3 Shared Memory

As noted in section 12.2.1, synchronization is the principal semantic challenge for shared-memory concurrent programs. One commonly sees two forms of synchronization: *mutual exclusion* and *condition synchronization*. Mutual exclusion ensures that only one thread is executing a *critical section* of code at a given point in time. Condition synchronization ensures that a given thread does not proceed until some specific condition holds (e.g. until a given variable has a given value). It is tempting to think of mutual exclusion as a form of condition synchronization (don't proceed until no other thread is in its critical section), but this sort of condition would require *consensus* among all extant threads, something that condition synchronization doesn't generally provide.

Our implementation of parallel threads, sketched at the end of section 12.2.4, requires that *processes* (provided by the OS) use both mutual exclusion and condition synchronization to protect the ready list and related data structures. Mutual exclusion appears in the requirement that a process must never read or write the ready list while it is being modified by another process; condition synchronization appears in the requirement that a process in need of a thread to run must wait until the ready list is non-empty.

It is worth emphasizing that we do not in general want to overly synchronize programs. To do so would eliminate opportunities for parallelism, which we generally want to maximize in the interest of performance. The goal is to provide only as much synchronization as is necessary in order to eliminate "bad" race conditions—those that might otherwise cause the program to produce incorrect results.

In the first of the three subsections below we consider busy-wait synchronization mechanisms. In the second subsection we use busy-waiting among processes to implement a parallelism-safe thread scheduler. In the final subsection we use the scheduler to implement blocking synchronization for threads.

12.3.1 Busy-Wait Synchronization

Busy-wait condition synchronization is generally easy: if we can cast a condition in the form of "location X contains value Y", then a thread (or process) that needs to wait for the condition can simply read X in a loop, waiting for Y to appear. All that is required from the hardware is that individual load and store instructions be atomic. Providing this atomicity is not a trivial task (memory and/or busses must serialize concurrent accesses by processors and devices), but almost every computer ever made has done it.

Busy-wait mutual exclusion is harder. We consider it under "spin locks" below. We then consider a special form of condition synchronization, namely barriers. A barrier is meant to be executed by all of the threads in a program. It guarantees that no thread will continue past a given point in a program until all threads have reached that point. Like mutual exclusion (and unlike most condition synchronization), barriers require consensus among all extant threads. Barriers are fundamental to data-parallel computing. They can be implemented either with busy-waiting or with blocking; we consider the busy-wait version here.

Spin locks

Dekker is generally credited with finding the first two-thread mutual exclusion algorithm that requires no atomic instructions other than load and store. Dijkstra [Dij65] published a version that works for n threads in 1965. Peterson [Pet81] published a much simpler two-thread algorithm in 1981. Building on Peterson's algorithm, one can construct a hierarchical n-thread lock, but it requires $O(n \log n)$ space and $O(\log n)$ time to get one thread into its critical section [YA93]. Lamport [Lam87] published an n-thread algorithm in 1987 that takes $O(n)$ space and $O(1)$ time in the absence of competition for the lock. Unfortunately, it requires $O(n)$ time when multiple threads attempt to enter their critical section at once.

To achieve mutual exclusion in constant time, one needs a more powerful atomic instruction. Beginning in the 1960's, hardware designers began to equip their processors with instructions that read, modify, and write a memory location as a single atomic operation. The simplest read-modify-write instruction is known as `test_and_set`. It sets a Boolean

```
type lock = Boolean := false;

procedure acquire_lock (ref L : lock)
    while not test_and_set (L)
        while L
            -- nothing -- spin

procedure release_lock (ref L : lock)
    L := false
```

Figure 12.10: A simple test-and-`test_and_set` lock. Waiting processes spin with ordinary read (`load`) instructions until the lock appears to be free, then use `test_and_set` to acquire it. The very first access is a `test_and_set`, for speed in the common (no competition) case.

variable to `true` and returns an indication of whether the variable was `false` previously. Given `test_and_set`, acquiring a spin lock is almost trivial:

```
while not test_and_set (L)
    -- nothing -- spin
```

In practice, embedding `test_and_set` in a loop tends to result in unacceptable amounts of bus traffic on a multiprocessor, as the cache coherence mechanism attempts to reconcile writes by multiple processors attempting to acquire the lock. This overdemand for hardware resources is known as *contention*, and is a major obstacle to good performance on large machines.

To reduce contention, the writers of synchronization libraries often employ a test-and-`test_and_set` lock, which spins with ordinary reads (satisfied by the cache) until it appears that the lock is free (see figure 12.10). When a thread releases a lock there still tends to be a flurry of bus activity as waiting threads perform their `test_and_set`s, but at least this activity happens only at the boundaries of critical sections. On a large machine, bus or interconnect traffic can be further reduced by implementing a *backoff* strategy, in which a thread that is unsuccessful in attempting to acquire a lock waits for a while before trying again.

Many processors provide atomic instructions more powerful than `test_and_set`. Several can swap the contents of a register and a memory location atomically. A few can add a constant to a memory location atomically, returning the previous value. On the x86, most arithmetic instructions can be prefaced with a "lock" byte that causes them to update a memory location atomically. Recent versions of the MIPS, Alpha, and PowerPC architectures have converged on a pair of instructions called `load_linked` and `store_conditional` (`LL/SC`). The first of these instructions loads a memory location into a register and stores certain bookkeeping information into hidden processor registers. The second instruction stores the register back into the memory location, but only if the location has not been modified by any other processor since the `load_linked` was executed. In the time between the two instructions the processor is generally not allowed to touch memory, but it can perform an almost arbitrary computation in registers, allowing the `LL/SC` pair to function

as a *universal* atomic primitive. To add the value in register `r2` to memory location `foo`, atomically, one would execute the following instructions:

```
start:
    r1 := load_linked (foo)
    r1 := r1 + r2
    store_conditional (r1, foo)
    if failed goto start
```

If several processors execute this code simultaneously, one of them is guaranteed to succeed the first time around the loop. The others will fail and try again.

Using instructions such as `atomic_add` or `LL/SC`, one can build spin locks that are *fair*, in the sense that threads are guaranteed to acquire the lock in the order in which they first attempt to do so. One can also build locks that work well—with no contention—on arbitrarily large machines [MCS91]. Finally, one can use universal atomic primitives to build special-purpose concurrent data structures and algorithms that operate *without* locks, by modifying locations atomically in a carefully determined order. Lock-free concurrent algorithms are ideal for environments in which threads may pause (e.g. due to preemption) for arbitrary periods of time. In important special cases (e.g. parallel garbage collection [HM92] or queues [MS96]) lock-free algorithms can also be significantly faster than lock-based algorithms. Herlihy [Her91] and others have developed general-purpose techniques to turn sequential data structures into lock-free concurrent data structures, but these tend to be rather slow.

An important variant on mutual exclusion is the *reader-writer lock* [CHP71]. Reader-writer locks recognize that if several threads wish to *read* the same data structure, they can do so simultaneously without mutual interference. It is only when a thread wants to *write* the data structure that we need to prevent other threads from reading or writing simultaneously. Most busy-wait mutual exclusion locks can be extended to allow concurrent access by readers (see exercise 8).

Barriers

Barriers are common in data-parallel numeric algorithms. In *finite element analysis*, for example, a physical object such as, say, a bridge may be modeled as an enormous collection of tiny metal fragments. Each fragment imparts forces to the fragments adjacent to it. Gravity exerts a downward force on all fragments. Abutments exert an upward force on the fragments that make up base plates. The wind exerts forces on surface fragments. To evaluate stress on the bridge as a whole (e.g. to assess its stability and resistance to failures), a finite element program might divide the metal fragments among a large collection of threads (probably one per physical processor). Beginning with the external forces, the program would then proceed through a sequence of iterations. In each iteration each thread would recompute the forces on its fragments based on the forces found in the previous iteration. Between iterations, the threads would synchronize with a barrier. The program would halt when no thread found a significant change in any forces during the last iteration.

The simplest way to implement a busy-wait barrier is to use a globally shared counter, modified by an atomic `fetch_and_decrement` instruction (or equivalently by `fetch_and_`

```
shared count : integer := n
shared sense : Boolean := true
per-thread private local_sense : Boolean := true

procedure central_barrier
    local_sense := not local_sense
        -- each thread toggles its own sense
    if fetch_and_decrement (count) = 1
        -- last arriving thread
        count := n
            -- reinitialize for next iteration
        sense := local_sense
            -- allow other threads to proceed
    else
        repeat
            -- spin
        until sense = local_sense
```

Figure 12.11: A simple "sense-reversing" barrier. Each thread has its own copy of `local_sense`. Threads share a single copy of `count` and `sense`.

`add`, `LL/SC`, etc.). The counter begins at n, the number of threads in the program. As each thread reaches the barrier it decrements the counter. If it is not the last the arrive, the thread then spins on a Boolean flag. The final thread (the one that changes the counter from 1 to 0) flips the Boolean flag, allowing the other threads to proceed. To make it easy to reuse the barrier data structures in successive iterations (known as barrier *episodes*), threads wait for alternating values of the flag each time through. Code for this simple barrier appears in figure 12.11.

Like a simple spin lock, the "sense-reversing" barrier can lead to unacceptable levels of contention on large machines. Moreover the serialization of access to the counter implies that the time to achieve an n-thread barrier is $O(n)$. It is possible to do better, but even the fastest software barriers require $O(\log n)$ time to synchronize n threads [MCS91]. Several large multiprocessors, including the Thinking Machines CM-5 and the Cray Research T3D and T3E have provided special hardware for near-constant-time busy-wait barriers.

12.3.2 Scheduler Implementation

To implement user-level threads, OS-level processes must synchronize access to the ready list and condition queues, generally by means of spinning. Code for a simple *reentrant* thread scheduler (one that can be "reentered" safely by a second process before the first one has returned) appears in figure 12.12. As in the code in section 12.2.4, we disable timer signals before entering scheduler code, to protect the ready list and condition queues from concurrent access by a process and its own signal handler.

Our code assumes a single "low-level" lock (`scheduler_lock`) that protects the entire scheduler. Before saving its context block on a queue (e.g. in `yield` or `sleep_on`), a thread must acquire the scheduler lock. It must then release the lock after returning from

```
shared scheduler_lock : low_level_lock
shared ready_list : queue of thread
per-process private current_thread : thread

procedure reschedule
    -- assume that scheduler_lock is already held
    -- and that timer signals are disabled
    t : thread
    loop
        t := dequeue (ready_list)
        if t <> nil
            exit
        -- else wait for a thread to become runnable
        release (scheduler_lock)
        -- window allows another thread to access ready_list
        -- (no point in reenabling signals;
        -- we're already trying to switch to a different thread)
        acquire (scheduler_lock)
    transfer (t)
    -- caller must release scheduler_lock
    -- and reenable timer signals after we return

procedure yield
    disable_signals
    acquire (scheduler_lock)
    enqueue (ready_list, current_thread)
    reschedule
    release (scheduler_lock)
    reenable_signals

procedure sleep_on (ref Q : queue of thread)
    -- assume that has caller already disabled timer signals
    -- and acquired scheduler_lock, and will reverse
    -- these actions when we return
    enqueue (Q, current_thread)
    reschedule
```

Figure 12.12: Pseudocode for part of a simple reentrant (parallelism-safe) scheduler. Every process has its own copy of `current_thread`. There is a single shared `scheduler_lock` and a single `ready_list`. If processes have dedicated processors, then the `low_level_lock` can be an ordinary spin lock; otherwise it can be a "spin-then-yield" lock (figure 12.13). The loop inside `reschedule` busy-waits until the ready list is non-empty. The code for `sleep_on` cannot disable timer signals and acquire the scheduler lock itself, because the caller needs to test a condition and then block as a single atomic operation.

```
type lock = Boolean := false;

procedure acquire_lock (ref L : lock)
    while not test_and_set (L)
        count := TIMEOUT
        while L         count −:= 1
            if count = 0
                OS_yield            −− relinquish processor
                count := TIMEOUT

procedure release_lock (ref L : lock)
    L := false
```

Figure 12.13: A simple spin-then-yield lock, designed for execution on a multiprocessor that may be multiprogrammed (i.e. on which OS-level processes may be preempted). If unable to acquire the lock in a fixed, short amount of time, a process calls the OS scheduler to yield its processor. Hopefully the lock will be available the next time the process runs.

`reschedule`. Of course, because `reschedule` calls `transfer`, the lock will usually be acquired by one thread (the same one that disables timer signals) and released by another (the same one that reenables timer signals). The code for `yield` can implement synchronization itself, because its work is self-contained. The code for `sleep_on`, on the other hand, cannot, because a thread must generally check a condition and block if necessary as a single atomic operation:

```
disable_signals
acquire (scheduler_lock)
if not desired_condition
    sleep_on (condition_queue)
release (scheduler_lock)
reenable_signals
```

If the signal and lock operations were moved inside of `sleep_on`, the following race could arise: thread A checks the condition and finds it to be false; thread B makes the condition true, but finds the condition queue to be empty; thread A sleeps on the condition queue forever.

A spin lock will suffice for the "low-level" lock that protects the ready list and condition queues, so long as every process runs on a different processor. As we noted in section 12.2.1, however, it makes little sense to spin for a condition that can only be made true by some other process using the processor on which we are spinning. If we know that we're running on a uniprocessor, then we don't need a lock on the scheduler (just the disabled signals). If we *might* be running on a uniprocessor, however, or on a multiprocessor with fewer processors than processes, then we must be prepared to give up the processor if unable to obtain a lock. The easiest way to do this is with a "spin-then-yield" lock, first suggested by Ousterhout [Ous82]. A simple example of such a lock appears in figure 12.13. On a multiprogrammed machine, it might also be desirable to relinquish the processor inside

`reschedule` when the ready list is empty: though no other process of the current application will be able to do anything, overall system throughput may improve if we allow the operating system to give the processor to a process from another application.

On a large multiprocessor we might increase concurrency by employing a separate lock for each condition queue, and another for the ready list. We would have to be careful, however, to make sure it wasn't possible for one process to put a thread into a condition queue (or the ready list) and for another process to attempt to transfer into that thread before the first process had finished transferring out of it (see exercise 9).

12.3.3 Scheduler-Based Synchronization

The problem with busy-wait synchronization is that it consumes processor cycles, cycles that are therefore unavailable for other computation. Busy-wait synchronization makes sense only if (1) one has nothing better to do with the current processor, or (2) the expected wait time is less than the time that would be required to switch contexts to some other thread and then switch back again. To ensure acceptable performance on a wide variety of systems, most concurrent programming languages employ scheduler-based synchronization mechanisms, which switch to a different thread when the one that was running blocks. In the remainder of this section we consider the three most common forms of scheduler-based synchronization: semaphores, monitors, and conditional critical regions.

In each case, scheduler-based synchronization mechanisms remove the waiting thread from the scheduler's ready list, returning it only when the awaited condition is true (or is likely to be true). By contrast, the spin-then-yield lock described in the previous subsection is still a busy-wait mechanism: the currently running process relinquishes the processor, but remains on the ready list. It will perform a `test_and_set` operation every time it gets a chance to run, until it finally succeeds. It is worth noting that busy-wait synchronization is generally "level-independent"—it can be thought of as synchronizing threads, processes, or processors, as desired. Scheduler-based synchronization is "level-*de*pendent"—it is specific to threads when implemented in the language run-time system, or to processes when implemented in the operating system.

We will use a *bounded buffer* abstraction to illustrate the semantics of various scheduler-based synchronization mechanisms. A bounded buffer is a concurrent queue of limited size into which *producer* threads insert data, and from which *consumer* threads remove data. The buffer serves to even out fluctuations in the relative rates of progress of the two classes of threads, increasing system throughput. A correct implementation of a bounded buffer requires both mutual exclusion and condition synchronization: the former to ensure that no thread sees the buffer in an inconsistent state in the middle of some other thread's operation; the latter to force consumers to wait when the buffer is empty and producers to wait when the buffer is full.

Semaphores

Semaphores are the oldest of the scheduler-based synchronization mechanisms. They were described by Dijkstra in the mid 1960's [Dij68], and appear in Algol 68. They are still heavily used today, both in library packages and in languages such as SR and Modula-3.

```
type semaphore = record
    N : integer -- usually initialized to something non-negative
    Q : queue of threads

procedure P (ref S : semaphore)
    disable_signals
    acquire (scheduler_lock)
    S.N -:= 1
    if S.N < 0
        sleep_on (S.Q)
    release (scheduler_lock)
    reenable_signals

procedure V (ref S : semaphore)
    disable_signals
    acquire (scheduler_lock)
    S.N +:= 1
    if N <= 0
        -- at least one thread is waiting
        enqueue (ready_list, dequeue (S.Q))
    release (scheduler_lock)
    reenable_signals
```

Figure 12.14: Semaphore operations, for use with the scheduler code of figure 12.12.

A semaphore is basically a counter with two associated operations, `P` and `V`.[3] A thread that calls `P` atomically decrements the counter and then waits until it is non-negative. A thread that calls `V` atomically increments the counter and wakes up a waiting thread, if any. It is generally assumed that semaphores are fair, in the sense that threads complete `P` operations in the same order they start them. Implementations of `P` and `V` in terms of our scheduler operations appear in figure 12.14.

A semaphore whose counter is initialized to one and for which `P` and `V` operations always occur in matched pairs is known as a *binary semaphore*. It serves as a scheduler-based mutual exclusion lock: the `P` operation acquires the lock; `V` releases it. More generally, a semaphore whose counter is initialized to k can be used to arbitrate access to k copies of some resource. The value of the counter at any particular time is always k more than the difference between the number of `P` operations ($\#P$) and the number of `V` operations ($\#V$) that have occurred so far in the program. A `P` operation blocks the caller until $\#P \leq \#V + k$. Exercise 16 notes that binary semaphores can be used to implement general semaphores, so the two are of equal expressive power, if not of equal convenience.

Figure 12.15 shows a semaphore-based solution to the bounded buffer problem. It uses a binary semaphore for mutual exclusion, and two general (or *counting*) semaphores for

[3] `P` and `V` stand for the Dutch words *passeren* (to pass) and *vrygeren* (to release). To keep them straight, speakers of English may wish to think of `P` as standing for "pause", since a thread will pause at a `P` operation if the semaphore count is negative. Algol 68 calls the `P` and `V` operations `down` and `up`, respectively.

```
shared buf : array [1..SIZE] of bdata
shared next_full, next_empty : integer := 1, 1
shared mutex : semaphore := 1
shared empty_slots, full_slots : semaphore := SIZE, 0

procedure insert (d : bdata)
    P (empty_slots)
    P (mutex)
    buf[next_empty] := d
    next_empty := next_empty mod SIZE + 1
    V (mutex)
    V (full_slots)

function remove : bdata
    P (full_slots)
    P (mutex)
    d : bdata := buf[next_full]
    next_full := next_full mod SIZE + 1
    V (mutex)
    V (empty_slots)
    return d
```

Figure 12.15: Semaphore-based code for a bounded buffer. The `mutex` binary semaphore protects the data structure proper. The `full_slots` and `empty_slots` general semaphores ensure that no operation starts until it is safe to do so.

condition synchronization. Exercise 12 considers the use of semaphores to construct an n-thread barrier.

Monitors

Though widely used, semaphores are also widely considered to be too "low-level" for well-structured, maintainable code. They suffer from two principal problems. First, because they are simply subroutine calls, it is easy to leave one out, e.g. on a control path with several nested `if` statements. Second, unless they are hidden inside an abstraction, uses of a given semaphore tend to get scattered throughout a program, making it difficult to track them down for purposes of software maintenance.

Monitors were suggested by Dijkstra [Dij72] as a solution to these problems. They were developed more thoroughly by Brinch Hansen [Bri73], and formalized by Hoare [Hoa74] in the early 1970's. They have been incorporated into at least a score of languages, of which Concurrent Pascal [Bri75], Modula (1) [Wir77b], and Mesa [LR80] have probably been the most influential.

A monitor is a module or object with operations, internal state, and a number of *condition variables*. Only one operation of a given monitor is allowed to be active at a given point in time. A thread that calls a busy monitor is automatically delayed until the monitor is free. On behalf of its calling thread, any operation may suspend itself by *wait*ing on a

condition variable. An operation may also *signal* a condition variable, in which case one of the waiting threads is resumed, usually the one that waited first.

Because the operations (*entries*) of a monitor automatically exclude one another in time, the programmer is relieved of the responsibility of using `P` and `V` operations correctly. Moreover because the monitor is an abstraction, all operations on the encapsulated data, including synchronization, are collected together in one place. Hoare defined his monitors in terms of semaphores. Conversely, it is easy to define semaphores in terms of monitors (exercise 15). Together, the two definitions prove that semaphores and monitors are equally powerful: each can express all forms of synchronization expressible with the other.

Hoare's definition of monitors employs one thread queue for every condition variable, plus two bookkeeping queues: the *entry queue* and the *urgent queue*. A thread that attempts to enter a busy monitor waits in the entry queue. When a thread executes a `signal` operation from within a monitor, and some other thread is waiting on the specified condition, then the `signal`ing thread waits on the monitor's urgent queue and the first thread on the appropriate condition queue obtains control of the monitor. If no thread is waiting on the `signal`ed condition, then the `signal` operation is a no-op. When a thread leaves a monitor, either by completing its operation or by `wait`ing on a condition, it unblocks the first thread on the urgent queue or, if the urgent queue is empty, the first thread on the entry queue, if any.

Figure 12.16 shows a monitor-based solution to the bounded buffer problem. It is worth emphasizing that monitor condition variables are not the same as semaphores. Specifically, they have no "memory": if no thread is waiting on a condition at the time that a `signal` occurs, then the `signal` has no effect. Whereas a `V` operation on a semaphore increments the semaphore's counter, allowing some future `P` operation to succeed, an un-awaited `signal` on a condition variable is lost.

Correctness for monitors depends on the notion of a *monitor invariant*. The invariant is a predicate that captures the notion that "the state of the monitor is consistent". The invariant needs to be true initially, and at monitor exit. It also needs to be true at every `wait` statement and, in a Hoare monitor, at `signal` operations as well. For our bounded buffer example, a suitable invariant would assert that `full_slots` correctly indicates the number of items in the buffer, and that those items lie in slots numbered `next_full` through `next_empty - 1` (mod `SIZE`). Careful inspection of the code in figure 12.16 reveals that the invariant does indeed hold initially, and that anytime we modify one of the variables mentioned in the invariant, we always modify the others accordingly before `wait`ing, `signal`ing, or returning from an entry.

The semantic details of monitors vary significantly from one language to the next. The two principal areas of variation are the semantics of the `signal` operation and the management of mutual exclusion when a thread `wait`s inside a nested sequence of two or more monitor calls.

In general, one `signal`s a condition variable when some condition on which a thread may be waiting has become true. If we want to guarantee that the condition is still true when the thread wakes up, then we need to switch to the thread as soon as the signal occurs—hence the need for the urgent queue, and the need to ensure the monitor invariant at `signal` operations. In practice, switching contexts on a `signal` tends to induce unnecessary scheduling overhead: a `signal`ing thread seldom changes the condition associated with the `signal` during the remainder of its operation. To reduce the overhead, and to eliminate

```
monitor bounded_buf
imports bdata, SIZE
exports insert, remove

    buf : array [1..SIZE] of data
    next_full, next_empty : integer := 1, 1
    full_slots : integer := 0
    full_slot, empty_slot : condition

    entry insert (d : bdata)
        if full_slots = SIZE
            wait (empty_slot)
        buf[next_empty] := d
        next_empty := next_empty mod SIZE + 1
        full_slots −:= 1
        signal (full_slot)

    entry remove : bdata
        if full_slots = 0
            wait (full_slot)
        d : bdata := buf[next_full]
        next_full := next_full mod SIZE + 1
        full_slots +:= 1
        signal (empty_slot)
        return d
```

Figure 12.16: Monitor-based code for a bounded buffer. `Insert` and `remove` are *entry* subroutines: they require exclusive access to the monitor's data. Because conditions are memory-less, both `insert` and `remove` can safely end their operation by generating a `signal`.

the need to ensure the monitor invariant, Mesa specifies that `signal`s are only *hints*: the language run-time system moves some waiting thread to the ready list, but the `signal`er retains control of the monitor, and the `wait`er must recheck the condition when it awakes. In effect, the standard idiom

```
if not desired_condition
    wait (condition_variable)
```

in a Hoare monitor becomes

```
while not desired_condition
    wait (condition_variable)
```

in a Mesa monitor. Modula-3 takes a similar approach. An alternative appears in Concurrent Pascal, which specifies that a `signal` operation causes an immediate return from the monitor operation in which it appears. This rule keeps overhead low, and also preserves

invariants, but precludes algorithms in which a thread does useful work in a monitor after `signal`ing a condition.

In most monitor languages, a `wait` in a nested sequence of monitor operations will release mutual exclusion on the innermost monitor, but will leave the outer monitors locked. This situation can lead to *deadlock* if the only way for another thread to reach a corresponding `signal` operation is through the same outer monitor(s). In general, we use the term "deadlock" to describe any situation in which a collection of threads are all waiting for each other, and none of them can proceed. In this specific case, the thread that entered the outer monitor first is waiting for the second thread to execute a `signal` operation; the second thread, however, is waiting for the first to leave the monitor. Several monitor implementations for uniprocessors (including the original Modula implementation [Wir77a]) avoid the nested monitor problem by providing mutual exclusion across *all* operations of *all* monitors, releasing exclusion on all of them when a `wait` occurs.

Conditional Critical Regions

Conditional critical regions are another alternative to semaphores, proposed by Brinch Hansen at about the same time as monitors [Bri73]. A critical region is a syntactically delimited critical section in which code is permitted to access a *protected* variable. A *conditional* critical region also specifies a Boolean condition, which must be true before control will enter the region:

```
region protected_variable when Boolean_condition do
    ...
end region
```

No thread can access a protected variable except within a `region` statement for that variable, and any thread that reaches a `region` statement waits until the condition is true and no other thread is currently in a region for the same variable. Regions can nest, though as with nested monitor calls, the programmer needs to worry about deadlock. Figure 12.17 uses conditional critical regions to implement a bounded buffer.

Conditional critical regions avoid the question of `signal` semantics, because they use explicit Boolean conditions instead of condition variables, and because conditions can only be awaited at the beginning of critical regions. At the same time, they introduce potentially significant inefficiency. In the general case, the code used to exit a conditional critical region must tentatively resume each waiting thread, allowing that thread to recheck its condition in its own referencing environment. Optimizations are possible in certain special cases (e.g. for conditions that depend only on global variables, or that consist of only a single Boolean variable), but in the worst case it may be necessary to perform context switches in and out of every waiting thread on every exit from a region.

Conditional critical regions appear in the concurrent language Edison [Bri81], and also seem to have influenced the synchronization mechanisms of Ada 95 and Java. Both of these latter languages might be said to blend the features of monitors and conditional critical regions, albeit in different ways.

The principal mechanism for synchronization in Ada, introduced in Ada 83, is based on message passing; we will describe it in section 12.4 below. Ada 95 augments this mechanism

```
buffer : record
    buf : array [1..SIZE] of data
    next_full, next_empty : integer := 1, 1
    full_slots : integer := 0

procedure insert (d : bdata)
    region buffer when full_slots < SIZE
        buf[next_empty] := d
        next_empty := next_empty mod SIZE + 1
        full_slots −:= 1

function remove : bdata
    region buffer when full_slots > 0
        d : bdata := buf[next_full]
        next_full := next_full mod SIZE + 1
        full_slots +:= 1
    return d
```

Figure 12.17: Conditional critical regions for a bounded buffer. Boolean conditions on the `region` statements eliminate the need for explicit condition variables.

with a notion of *protected object*. A protected object can have three types of member subroutines: functions, procedures, and *entries*. Functions can only read the data members of the object; procedures and entries can read and write them. An implicit reader-writer lock on the protected object ensures that potentially conflicting operations exclude one another in time: a procedure or entry obtains exclusive access to the object; a function can operate concurrently with other functions, but not with a procedure or entry.

Procedures and entries differ from one another in two important ways. First, an entry can have a Boolean expression *guard*, for which the calling task (thread) will wait before beginning execution (much as it would for the condition of a conditional critical region). Second, an entry supports three special forms of call: *timed* calls, which abort after waiting for a specified amount of time, *conditional* calls, which execute alternative code if the call cannot proceed immediately, and *asynchronous* calls, which begin executing alternative code immediately, but abort it if the call is able to proceed before the alternative completes.

In comparison to the conditions of conditional critical regions, the guards on entries of protected objects in Ada 95 admit a more efficient implementation, because they do not have to be evaluated in the context of the calling thread. Moreover, because all guards are gathered together in the definition of the protected object, the compiler can generate code to test them as a group as efficiently as possible, in a manner suggested by Kessels [Kes77]. Though an Ada task cannot wait on a condition in the middle of an entry (only at the beginning), it can *requeue* itself on another entry, achieving much the same effect.

In Java, every object accessible to more than one thread has an implicit mutual exclusion lock, acquired and released by means of `synchronized` statements:

```
synchronized (my_shared_obj) {
    ...     // critical section
}
```

All executions of `synchronized` statements that refer to the same shared object exclude one another in time. `Synchronized` statements that refer to different objects may proceed concurrently. As a form of syntactic sugar, a member function of a class may be prefixed with the `synchronized` keyword, in which case the body of the method is considered to have been surrounded by an implicit `synchronized (this)` statement. Invocations of non-synchronized methods of a shared object—and direct accesses to public data members—can proceed concurrently with each other, or with `synchronized` statements or methods.

Within a `synchronized` statement or method, a thread can suspend itself by calling the predefined method `wait`. `Wait` has no arguments in Java: the language does not distinguish among the different reasons why threads may be suspended on a given object. As in Mesa, Java programs typically embed the use of `wait` within a condition-testing loop:

```
while (!condition) {
    wait ();
}
```

A thread that calls the `wait` method of an object releases the object's lock. With nested `synchronized` statements, however, or with nested calls to `synchronized` methods, the thread does *not* release locks on any other objects.

To resume a thread that is suspended on a given object, some other thread must execute the predefined methomethod `notify` from within a `synchronized` statement or method that refers to the same object. Like `wait`, `notify` has no arguments. In response to a `notify` call, the language run-time system picks an arbitrary thread suspended on the object and makes it runnable. If there are no such threads then the `notify` is a no-op. In some situations, it may be appropriate to awaken *all* threads waiting in a given object. Java provides a built-in `notifyAll` function for this purpose.

It is important to realize when a `notify` occurs that the choice among waiting threads is arbitrary. If threads are waiting for more than one condition (i.e. if their `wait`s are embedded in dissimilar loops), there is no guarantee that the "right" thread will awaken. To ensure that an appropriate thread does wake up, the programmer may choose to use `notifyAll` instead of `notify`. To ensure that only *one* thread continues, the first thread to discover that its condition has been satisfied must modify the state of the object in such a way that other awakened threads, when they get to run, will simply go back to sleep. Unfortunately, since all waiting threads will end up reevaluating their conditions every time one of them can run, this "solution" to the multiple-condition problem can be prohibitively expensive. In general, Java programmers tend to look for algorithms in which there are never threads waiting for more than one condition within a given object.

Java objects that use only `synchronized` methods (no other `synchronized` statements) closely resemble Mesa monitors in which there is a limit of one condition variable per monitor. By the same token, a `synchronized` statement in Java that begins with a `wait` in a loop resembles a conditional critical region in which the retesting of conditions has been made explicit. Because `notify` also is explicit, a Java program need not reevaluate

conditions on every exit from a critical section—only those in which a `notify` occurs. It turns out to be possible (see exercise 19) to solve completely general synchronization problems with conditional critical regions in which all threads wait for the same condition. If the programmer chooses, however—either with conditional critical regions or in Java—to have threads wait for more than one condition of the same object at the same time, then execution may cycle through an arbitrary number of threads before one of them finds that it is able to continue. The optimizations possible in Ada 95 do not generally apply: conditions must be evaluated in the context of the waiting thread.

Ada 95 code for a bounded buffer would closely resemble the pseudocode of figure 12.17. Java code would use `wait`s within `while` loops in place of syntactically distinguished Boolean guards. Java code would also end each `insert` or `remove` operation with an explicit `notify`. We leave the details as an exercise (20).

12.3.4 Implicit Synchronization

In several shared-memory languages, the operations that threads can perform on shared data are restricted in such a way that synchronization can be implicit in the operations themselves, rather than appearing as separate, explicit operations. We have seen one example of implicit synchronization already: the `forall` loop of HPF and Fortran 95 (section 12.2.3, page 16). Separate iterations of a `forall` loop proceed concurrently, semantically in lockstep with each other: each iteration reads all data used in its instance of the first assignment statement before any iteration updates its instance of the left-hand side. The left-hand side updates in turn occur before any iteration reads the data used in its instance of the second assignment statement, and so on. Compilation of `forall` loops for vector machines, while far from trivial, is more-or-less straightforward. On a more conventional multiprocessor, however, good performance usually depends on high-quality *dependence analysis*, which allows the compiler to identify situations in which statements within a loop do not in fact depend on one another, and can proceed without synchronization.

Dependence analysis plays a crucial role in other languages as well. In section 6.6.1 we mentioned Sisal, a purely functional language with Pascal-like syntax (recall that iterative constructs in Sisal are syntactic sugar for tail recursion). Because Sisal is side-effect free, its constructs can be evaluated in any order—or concurrently—so long as no construct attempts to use a value that has yet to be computed. The Sisal implementation developed at Lawrence Livermore National Lab uses extensive compiler analysis to identify promising constructs for parallel execution. It also employs tags on data objects that indicate whether the object's value has been computed yet. When the compiler is unable to guarantee that a value will have been computed by the time it is needed at run time, the generated code uses tag bits for synchronization, spinning or blocking until they are properly set. Sisal's developers claim [Can92] that their language and compiler rival parallel Fortran in performance.

In a less ambitious vein, the Multilisp [Hal85, MKH91] dialect of Scheme allows the programmer to enclose any function evaluation in a special `future` construct:

```
(future (my-function my-args))
```

In a purely functional program, `future` is semantically neutral: program behavior will be exactly the same as if `(my-function my-args)` had appeared without the surrounding call. In the implementation, however, `future` arranges for the embedded function to be

evaluated by a separate thread of control. The parent thread continues to execute until it actually tries to use the return value of `my-function`, at which point it waits for execution of the `future` to complete. If two or more arguments to a function are enclosed in `future`s, then evaluation of the arguments can proceed in parallel:

```
(parent-func (future (child-1 args-1)) (future (child-2 args-2)))
```

In a program that uses the imperative features of Scheme, the programmer must take care to make sure that concurrent execution of `future`s will not compromise program correctness. There are no additional synchronization mechanisms: `future` itself is Multilisp's only addition to Scheme.

Both Multilisp and Sisal employ the same basic idea: concurrent evaluation of functions in a side-effect-free language. Where the Sisal compiler attempts to find code fragments that can profitably be executed in parallel, the Multilisp programmer must identify them explicitly. In both languages, the synchronization required to delay a thread that attempts to use a yet-to-be-computed value is implicit. In some ways the `future` construct resembles the built-in `delay` and `force` of Scheme (section 6.6.2). Where `future` supports concurrency, `delay` supports lazy evaluation: it defers evaluation of its embedded function until the return value is known to be needed. Any use of a `delay`ed expression in Scheme must be surrounded by `force`. By contrast, synchronization on a `future` is implicit: there is no analog of `force`.

Several researchers have noted that the backtracking search of logic languages such as Prolog is also amenable to parallelization. Two strategies are possible. The first is to pursue in parallel the subgoals found in the right-hand side of a rule. This strategy is known as *AND parallelism*. The fact that variables in logic, once initialized, are never subsequently modified ensures that parallel branches of an AND cannot interfere with one another. The second strategy is known as *OR parallelism*; it pursues alternative resolutions in parallel. Because they will generally employ different unifications, branches of an OR must use separate copies of their variables. In a search tree such as that of figure 11.4 (page ??), AND parallelism and OR parallelism create new threads at alternating levels.

OR parallelism is *speculative*: since success is required on only one branch, work performed on other branches is in some sense wasted. OR parallelism works well, however, when in a goal cannot be satisfied (in which case the entire tree must be searched), or when there is high variance in the amount of execution time required to satisfy a goal in different ways (in which case exploring several branches at once reduces the expected time to find the first solution). Both AND and OR parallelism are problematic in Prolog, because they fail to adhere to the deterministic search order required by language semantics.

Some of the ideas embodied in concurrent functional languages can be adapted to imperative languages as well. CC++ [Fos95], for example, is a concurrent extension to C++ in which synchronization is implicit in the use of *single-assignment* variables. To declare a single-assignment variable, the CC++ programmer prepends the keyword `synch` to an ordinary variable declaration. The value of a `synch` variable is initially undefined. A thread that attempts to read the variable will wait until it is assigned a value by some other thread. It is a run-time error for any thread to attempt to assign to a `synch` variable that already has a value.

In a similar vein, Linda [ACG86] is a set of concurrent programming mechanisms that can be embedded into almost any imperative language. It consists of a set of subroutines

that manipulate a shared abstraction called the *tuple space*. The elements of tuple space resemble the tuples of ML (section 7.2.5), except that they have single assignment semantics, and are accessed associatively by content, rather than by name. The `in` procedure adds a tuple to the tuple space. The `out` procedure extracts a tuple that matches a specified *pattern*, waiting if no such tuple currently exists. The `read` procedure is a non-destructive `out`. A special form of `in` forks a concurrent thread to calculate the value to be inserted, much like a `future` in Multilisp. All three subroutines can be supported as ordinary library calls, but performance is substantially better when using a specially designed compiler that generates optimized code for commonly occurring patterns of tuple space operations.

A few multiprocessors, including the Denelcor HEP [Jor85] and the forthcoming Tera machine [ACC+90], provide special hardware support for single-assignment variables in the form of so-called *full-empty* bits. Each memory location contains a bit that indicates whether the variable in that location has been initialized. Any attempt to access an uninitialized variable stalls the current processor, causing it to switch contexts (in hardware) to another thread of control.

12.4 Message Passing

While shared-memory concurrent programming is common on small-scale multiprocessors, most concurrent programming on large multicomputers and networks is currently based on messages. In sections 12.4.1 through 12.4.3 we consider three principal issues in message-based computing: naming, sending, and receiving. In section 12.4.4 we look more closely at one particular combination of send and receive semantics, namely remote procedure call. Most of our examples will be drawn from the Ada, Occam, and SR programming languages, the Java network library, and the PVM and MPI library packages.

12.4.1 Naming Communication Partners

To send or receive a message, one must generally specify where to send it to, or where to receive it from—communication partners need names for (or references to) one another. Names may refer directly to a thread or process. Alternatively, they may refer to an *entry* or *port* of a module, or to some sort of *socket* or *channel* abstraction. We illustrate these options in figure 12.18.

The first naming option—addressing messages to processes—appears in Hoare's original CSP proposal, and in PVM and MPI. Each PVM or MPI process has a unique `id` (an integer), and each `send` or `receive` operation specifies the `id` of the communication partner. MPI implementations are required to be reentrant; a process can safely be divided into multiple threads, each of which can send or receive messages on the process's behalf. PVM has hidden state variables that are not automatically synchronized, making threaded PVM programs problematic.

The second naming option—addressing messages to ports—appears in Ada. An Ada *entry call* of the form `t.foo (args)` sends a message to the `entry` named `foo` in task (thread) `t` (`t` may be either a task name or the name of a variable whose value is a pointer to a task). As we saw in section 12.2.3, an Ada task resembles a module; its entries resemble subroutine headers nested directly inside the task. A task receives a message that has been sent to one of its entries by executing an `accept` statement (to be discussed in section 12.4.3

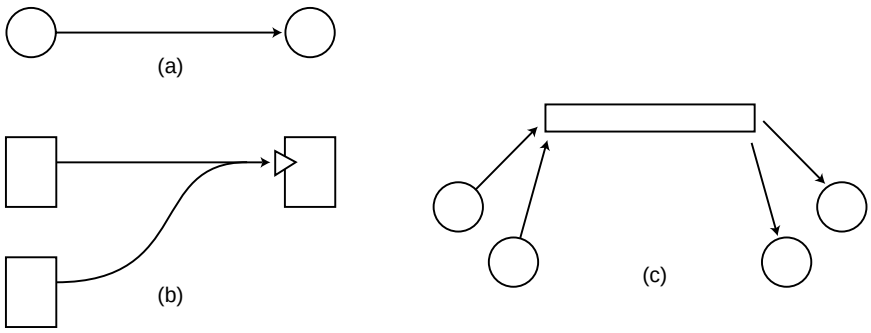


Figure 12.18: Three common schemes to name communication partners. In (a), processes name each other explicitly. In (b), senders name an *input port* of a receiver. The port may be called an *entry* or an *operation*. The receiver is typically a module with one or more threads inside. In (c), senders and receivers both name an independent *channel* abstraction, which may be called a *connection* or a *mailbox*.

below). Every `entry` belongs to exactly one task; all messages sent to the same `entry` must be received by that one task.

The third naming option—addressing messages to channels—appears in Occam (though not in CSP). Channel declarations are supported with the built-in `CHAN` and `CALL` types:

```
CHAN OF BYTE stream :
CALL lookup (RESULT [36]BYTE name, VAL INT ssn) :
```

These declarations specify a one-directional channel named `stream` that carries messages of type `BYTE` and a two-directional channel named `lookup` that carries requests containing an integer named `ssn` and replies containing a 36-byte string named `name`. `CALL` channels are syntactic sugar for a pair of `CHAN` channels, one in each direction. To send a message on a `CHAN` channel, an Occam thread uses a special "exclamation point" operator:

```
stream ! 'x'
```

To send a message (and receive a reply) on a `CALL` channel, a thread uses syntax that resembles a subroutine call:

```
lookup (name, 123456789)
```

We noted in section 12.2.3 ("parallel loops") that language rules in Occam prohibit concurrent threads from making conflicting accesses to the same variable. For channels, the basic rule is that exactly one thread may send to a channel, and exactly one may receive from it. (For `CALL` channels, exactly one thread may send requests, and exactly one may accept them and send replies). These rules are relaxed in Occam 3 to permit `SHARED` channels, which provide a mutual exclusion mechanism. Only one thread may accept requests over a `SHARED CALL` channel, but multiple threads may send them. In a similar vein, multiple threads may `CLAIM` a set of `CHAN` channels for exclusive use in a critical section, but only one thread may `GRANT` those channels; it serves as the other party for every message sent or received.

In SR and the Internet libraries of Java we see combinations of our naming options. An SR program executes on a collection of one or more *virtual machines*, each of which has a separate address space, and may be implemented on a separate node of a network. Within a virtual machine, messages are sent to (and received from) a channel-like abstraction called an `op`. Unlike an Occam channel, an SR `op` has no restrictions on the number or identity of sending and receiving threads: any thread that can see an `op` under the usual lexical scoping rules can send to it or receive from it. A `receive` operation must name its `op` explicitly; a `send` operation may do so also, or it may use a *capability* variable. A capability is like a pointer to an `op`, except that pointers work only within a given virtual machine, while capabilities work across the boundaries between them. Aside from start-up parameters and possibly I/O, capabilities provide the *only* means of communicating among separate virtual machines. At the outermost level, then, an SR program can be seen as having a port-like naming scheme: messages are sent (via capabilities) to `op`s of virtual machines, within which they may potentially be received by any local thread.

Java's standard `java.net` library provides two styles of message passing, corresponding to the UDP and TCP Internet protocols. UDP is the simpler of the two. It is a *datagram* protocol, meaning that each message is sent to its destination independently and unreliably. The network software will attempt to deliver it, but makes no guarantees. Moreover two messages sent to the same destination (assuming they both arrive) may arrive in either order. UDP messages use port-based naming (option (b) in figure 12.18): each message is sent to a specific *Internet address* and *port number*.[4] The TCP protocol also uses port-based naming, but only for the purpose of establishing *connections* (option (c) in figure 12.18), which it then uses for all subsequent communication. Connections deliver messages reliably and in order.

To send or receive UDP messages, a Java thread must create a *datagram socket*:

```
DatagramSocket my_socket = new DatagramSocket(port_id);
```

The parameter of the `DatagramSocket` constructor is optional; if it is not specified, the operating system will choose an available port. Typically servers specify a port and clients allow the OS to choose. To send a UDP message, a thread says

```
DatagramPacket my_msg = new DatagramPacket (buf, len, addr, port);
...  // initialize message
my_socket.send (my_msg);
```

The parameters to the `DatagramPacket` constructor specify an array of bytes `buf`, its length `len`, and the Internet address and port of the receiver.

For TCP communication, a server typically "listens" on a port to which clients send requests to establish a connection:

[4]Every machine on the Internet has its own unique address. As of 1999, addresses are 32-bit integers, usually printed as four period-separated fields (e.g. 192.5.54.209). Internet name servers translate symbolic names (e.g. `gate.cs.rochester.edu`) into numeric addresses. Port numbers are also integers, but are local to a given Internet address. Ports 1024 through 4999 are generally available for application programs; larger and smaller numbers are reserved for servers.

```
ServerSocket my_server_socket = new ServerSocket(port_id);
Socket client_connection = my_server_socket.accept();
```

The `accept` operation blocks until the server receives a connection request from a client. Typically a server will immediately fork a new thread to communicate with the client; the parent thread loops back to wait for another connection with `accept`.

A client sends a connection request by passing the server's symbolic name and port number to the `Socket` constructor:

```
Socket server_connection = new Socket (host_name, port_id);
```

Once a connection has been created, a client and server in Java typically call member functions of the `Socket` class to create input and output `streams`, which support all of the standard Java mechanisms for text I/O (section 7.9.3):

```
DataInputStream in =
    new DataInputStream(client_connection.getInputStream());
PrintStream out =
    new PrintStream(client_connection.getOutputStream());
// This is in the server; the client would make streams out
// of server_connection.
...
String s = in.readLine();
out.println ("Hi, Mom\n");
```

Among all the message-passing mechanisms we have considered, datagrams are the only one that does not provide some sort of *ordering* constraint. In general, most message-passing systems guarantee that messages sent over the same "communication path" arrive in order. When naming processes explicitly, a path links a single sender to a single receiver. All messages from that sender to that receiver arrive in the order sent. When naming ports, a path links an arbitrary number of senders to a single receiver (though as we saw in SR, if a receiver is a complex entity like a virtual machine, it may have many threads inside). Messages that arrive at a port in a given order will be seen by receivers in that order. Note, however, that while messages from the same sender will arrive at a port in order, messages from *different* senders may arrive in different orders.[5] When naming channels, a path links all the senders that can use the channel to all the receivers that can use it. A Java TCP connection has a single OS process at each end, but there may be many threads inside, each of which can use its process's end of the connection. An SR `op` can be used by any thread to which it is visible. In both cases, the channel functions as a queue: send (enqueue) and receive (dequeue) operations are ordered, so that everything is received in the order it was sent.

[5]Suppose, for example, that process A sends a message to port p of process B, and then sends a message to process C, while process C first receives the message from A and then sends its own message to port p of B. If messages are sent over a network with internal delays, and if A is allowed to send its message to C before its first message has reached port p, then it is possible for B to hear from C before it hears from A. This apparent reversal of ordering could easily happen on the Internet, for example, if the message from A to B traverses a satellite link, while the messages from A to C and from C to B use ocean-floor cables.

12.4.2 Sending

One of the most important issues to be addressed when designing a `send` operation is the extent to which it may block the caller: once a thread has initiated a `send` operation, when is it allowed to continue execution? Blocking can serve at least three purposes:

resource management: A sending thread should not modify outgoing data until the underlying system has copied the old values to a safe location. Most systems block the sender until a point at which it can safely modify its data, without danger of corrupting the outgoing message.

failure semantics: Particularly when communicating over a long-distance network, message passing is more error-prone than most other aspects of computing. Many systems block a sender until they are able to guarantee that the message will be delivered without error.

return parameters: In many cases a message constitutes a *request*, for which a *reply* is expected. Many systems block a sender until a reply has been received.

When deciding how long to block, we must consider synchronization semantics, buffering requirements, and the reporting of run-time errors.

Synchronization semantics

On its way from a sender to a receiver, a message may pass through many intermediate steps, particularly if traversing the Internet. It first descends through several layers of software on the sender's machine, then through a potentially large number of intermediate machines, and finally up through several layers of software on the receiver's machine. We could imagine unblocking the sender after any of these steps, but most of the options would be indistinguishable in terms of user-level program behavior. If we assume for the moment that a message-passing system can always find buffer space to hold an outgoing message, then our three rationales for delay suggest three principal semantic options:

no-wait send: The sender does not block for more than a small, bounded period of time. The message-passing implementation copies the message to a safe location and takes responsibility for its delivery.

synchronization send: The sender waits until its message has been received.

remote-invocation send: The sender waits until it receives a reply.

These three alternatives are illustrated in figure 12.19. No-wait `send` appears in SR and in the Java Internet library. Synchronization `send` appears in Occam. Remote-invocation `send` appears in SR, Occam, and Ada. PVM and MPI provide an implementation-oriented hybrid of no-wait `send` and synchronization `send`: a `send` operation blocks until the data in the outgoing message can safely be modified. In implementations that do their own internal buffering, this rule amounts to no-wait `send`. In other implementations, it amounts to synchronization `send`. PVM programs must be written to cope with the latter, more restrictive option. In MPI, the programmer has the option, if desired, to insist on no-wait `send` or synchronization `send`; performance may suffer on some systems if the request is different from the default.

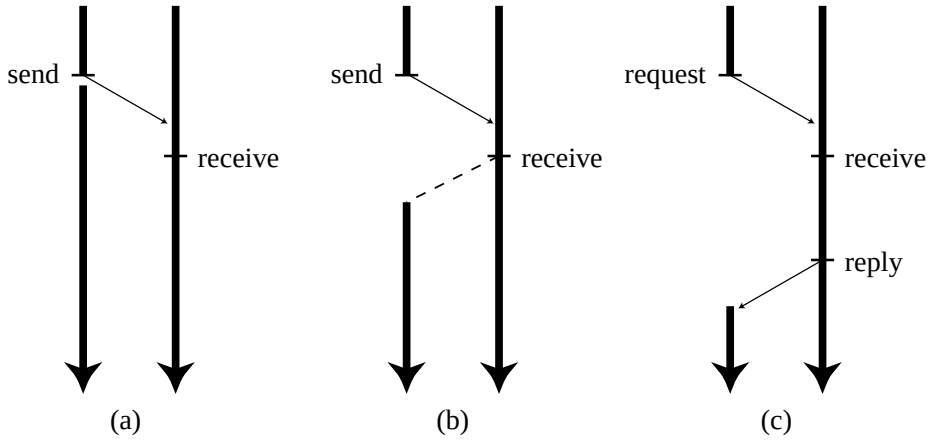


Figure 12.19: Synchronization semantics for the `send` operation: (a) no-wait `send`; (b) synchronization `send`; (c) remote-invocation `send`. In each diagram we have assumed that the original message arrives before the receiver executes its `receive` operation; this need not in general be the case.

Buffering

In practice, unfortunately, no message-passing system can provide a version of `send` that never waits (unless of course it simply throws some messages away). If we imagine a thread that sits in a loop sending messages to a thread that never receives them, we quickly see that unlimited amounts of buffer space would be required. At some point, any implementation must be prepared to block an overactive sender, to keep it from overwhelming the system. For any fixed amount of buffer space, it is possible to design a program that requires a larger amount of space to run correctly. Imagine, for example, that the message-passing system is able to buffer n messages on a given communication path. Now imagine a program in which A sends $n + 1$ messages to B, followed by one message to C. C then sends one message to B, on a different communication path. For its part, B insists on receiving the message from C before receiving the messages from A. If A blocks after message n, implementation-dependent deadlock will result. The best that an implementation can do is to provide a sufficiently large amount of space that realistic applications are unlikely to find the limit to be a problem.

For synchronization `send` and remote-invocation `send`, buffer space is not generally a problem: the total amount of space required for messages is bounded by the number of threads, and there are already likely to be limits on how many threads a program can create. A thread that sends a reply message can always be permitted to proceed: we know that we shall be able to reuse the buffer space quickly, because the thread that sent the request is already waiting for the reply.

Error reporting

In addition to limits on buffering, no-wait `send` suffers from the problem of error reporting. As long as the sender is blocked, errors that occur in attempting to deliver a message can be reflected back as exceptions, or as status information in result parameters or global variables. Once a sender has continued, there is no obvious way in which to report any problems that

arise. For UDP, the solution is to state that messages are unreliable: if something goes wrong, the message is simply lost, silently. For TCP, the "solution" is to state that only "catastrophic" errors will cause a message to be lost, in which case the connection will become unusable and future calls will fail immediately. An even more drastic approach is taken in MPI: certain implementation-specific errors may be detected and handled at run time, but in general if a message cannot be delivered then the program as a whole is considered to have failed. PVM provides a *notification* mechanism that will send a message to a previously designated process in the event of a node or process failure. The designated process can then abort any related, dependent processes, start new processes to pick up the work of those that failed, etc.

Emulation of alternatives

All three varieties of `send` can be emulated by the others. To obtain the effect of remote-invocation `send`, a thread can follow a no-wait `send` of a request with a `receive` of the reply. Similar code will allow us to emulate remote-invocation `send` using synchronization `send`. To obtain the effect of synchronization `send`, a thread can follow a no-wait `send` with a `receive` of an *acknowledgment* message, which the receiver will send immediately upon receipt of the original message. To obtain the effect of synchronization `send` using remote-invocation `send`, a thread that receives a request can simply reply immediately, with no return parameters.

To obtain the effect of no-wait `send` using synchronization `send` or remote-invocation `send`, we must interpose a buffer process (the message-passing analogue of our shared-memory bounded buffer) that replies immediately to "senders" or "receivers" whenever possible. The space available in the buffer process makes explicit the resource limitations that are always present below the surface in implementations of no-wait `send`.

Unfortunately, user-level emulations of alternative `send` semantics are seldom as efficient as optimized implementations using the underlying primitives. Suppose for example that we wish to use remote-invocation `send` to emulate synchronization `send`. Suppose further that our implementation of remote-invocation `send` is built on top of network software that does not guarantee message delivery (we might perhaps have an implementation of Ada on top of UDP). In this situation the language run-time system is likely to employ hidden acknowledgment messages: after sending an (unreliable) request, the client's run-time system will wait for an acknowledgment from the server (figure 12.20). If the acknowledgment does not appear within a bounded interval of time, then the run-time system will retransmit the request. After sending a reply, the server's run-time system will wait for an acknowledgment from the client. If a server thread can work for an arbitrary amount of time before sending a reply, then the run-time system will need to send separate acknowledgments for the request and the reply. If a programmer uses this implementation of remote-invocation `send` to emulate synchronization `send`, then the underlying network may end up transmitting a total of four messages (more if there are any transmission errors). By contrast, a "native" implementation of synchronization `send` would require only two underlying messages. In some cases the run-time system for remote-invocation `send` may be able to delay transmission of the first acknowledgment long enough to "piggy-back" it on the subsequent reply if there is one; in this case an emulation of synchronization `send` may transmit three

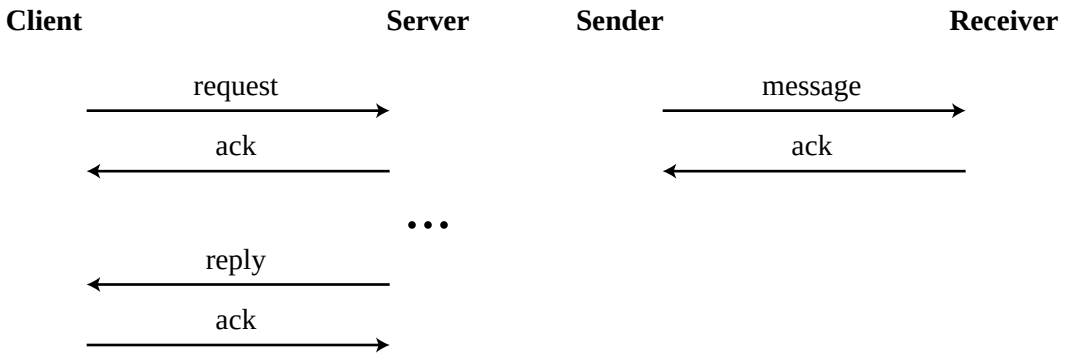


Figure 12.20: Acknowledgment messages. If the underlying message-passing system is unreliable, a language or library can provide reliability by waiting for *acknowledgment* messages, and resending if they don't appear within a reasonable amount of time. In the absence of piggy-backing, remote-invocation `send` (left) may require four underlying messages; synchronization `send` (right) may require two.

underlying messages instead of only two. We consider the efficiency of emulations further in exercises 26 and 29.

Syntax and language integration

In the emulation examples above, our hypothetical syntax assumed a library-based implementation of message passing. Because `send`, `receive`, `accept`, etc. are ordinary subroutines in such an implementation, they take a fixed, static number of parameters, two of which typically specify the location and size of the message to be sent. To send a message containing values held in more than one program variable, the programmer must explicitly *gather*, or *marshal*, those values into the fields of a record. On the receiving end, the programmer must *scatter* (*un-marshal*) the values back into program variables. By contrast, a concurrent programming language can provide message-passing operations whose "argument" lists can include an arbitrary number of values to be sent. Moreover, the compiler can arrange to perform type checking on those values, using techniques similar to those employed for subroutine linkage across compilation units (as described in section 9.6.2). Finally, as we will see in section 12.4.3, an explicitly concurrent language can employ non-procedure-call syntax, e.g. to couple a remote-invocation `accept` and `reply` in such a way that the `reply` doesn't need to explicitly identify the `accept` to which it corresponds.

12.4.3 Receiving

Probably the most important dimension on which to categorize mechanisms for receiving messages is the distinction between explicit `receive` operations and the *implicit* receipt described in section 12.2.3 (page 20). Among the languages and systems we have been using as examples, only SR provides implicit receipt (some RPC systems also provide it, as we shall see in section 12.4.4 below).

With implicit receipt, every message that arrives at a given port (or over a given channel) will create a new thread of control, subject to resource limitations (any implementation will

```
task buffer is
    entry insert (d : in bdata);
    entry remove (d : out bdata);
end buffer;

task body buffer is
    SIZE : constant integer := 10;
    subtype index is integer range 1..SIZE;
    buf : array (index) of bdata;
    next_empty, next_full : index := 1;
    full_slots : integer range 0..SIZE := 0;
begin
    loop
        select
          when full_slots < SIZE =>
            accept insert (d : in bdata) do
                buf(next_empty) := d;
            end;
            next_empty := next_empty mod SIZE + 1;
            full_slots := full_slots + 1;
        or
          when full_slots > 0 =>
            accept remove (d : out bdata) do
                d := buf(next_full);
            end;
            next_full := next_full mod SIZE + 1;
            full_slots := full_slots - 1;
        end select;
    end loop;
end buffer;
```

Figure 12.21: Bounded buffer in Ada, with an explicit manager task.

have to stall incoming requests when the number of threads grows too large). With explicit receipt, a message must be queued until some already-existing thread indicates a willingness to receive it. At any given point in time there may be a potentially large number of messages waiting to be received. Most languages and libraries with explicit receipt allow a thread to exercise some sort of *selectivity* with respect to which messages it wants to consider.

In PVM and MPI, every message includes the `id` of the process that sent it, together with an integer *tag* specified by the sender. A `receive` operation specifies a desired sender `id` and message tag. Only matching messages will be received. In many cases receivers specify "wild cards" for the sender `id` and/or message tag, allowing any of a variety of messages to be received. Special versions of `receive` also allow a process to test (without blocking) to see if a message of a particular type is currently available (this operation is known as *polling*), or to "time out" and continue if a matching message cannot be received within a specified interval of time.

Because they are languages instead of library packages, Ada, Occam, and SR are able to use special, non-procedure-call syntax for selective message receipt. Moreover because

messages are built into the naming and typing system, these languages are able to receive selectively on the basis of port/channel names and parameters, rather than the more primitive notion of tags. In all three languages, the selective `receive` construct is a special form of *guarded command*, as described in section 6.7.

Figure 12.21 contains code for a bounded buffer in Ada 83. Here an active "manager" thread executes a `select` statement inside a loop. (Recall that it is also possible to write a bounded buffer in Ada using *protected objects*, without a manager thread, as described in section 12.3.3.) The Ada `accept` statement receives the `in` and `in out` parameters (section 8.3.1) of a remote invocation request. At the matching `end`, `accept` returns the `in out` and `out` parameters as a reply message. A client task would communicate with the bounded buffer using an *entry call*:

```
-- producer:                -- consumer:
buffer.insert (3);          buffer.remove (x);
```

The `select` statement in our buffer example has two arms. The first arm may be selected when the buffer is not full and there is an available `insert` request; the second arm may be selected when the buffer is not empty and there is an available `remove` request. Selection among arms is a two-step process: first the guards (`when` expressions) are evaluated, then for any that are true the subsequent `accept` statements are considered to see if a message is available. (The guard in front of an `accept` is optional; if missing it behaves like `when true =>`.) If both of the guards in our example are true (the buffer is partly full) and both kinds of messages are available, then either arm of the statement may be executed, at the discretion of the implementation. (For a discussion of issues of *fairness* in the choice among true guards, refer back to section 6.7.)

Every `select` statement must have at least one arm beginning with `accept` (and optionally `when`). In addition, it may have three other types of arms:

```
when condition => delay how_long
    other_statements
...
or when condition => terminate
...
else ...
```

A `delay` arm may be selected if no other arm becomes selectable within *how_long* seconds. (Ada implementations are required to support delays as long as one day or as short as 20 milliseconds.) A `terminate` arm may be selected only if all potential communication partners have already terminated or are likewise stuck in `select` statements with `terminate` arms. Selection of the arm causes the task that was executing the `select` statement to terminate. An `else` arm, if present, will be selected when none of the guards are true or when no `accept` statement can be executed immediately. A `select` statement with an `else` arm is not permitted to have any `delay` arms. In practice, one would probably want to include a `terminate` arm in the `select` statement of a manager-style bounded buffer.

Occam's equivalent of `select` is known as `ALT`. As in Ada, the choice among arms can be based both on Boolean conditions and on the availability of messages. (One minor

difference: Occam semantics specify a one-step evaluation process; message availability is considered part of the guard.) The body of our bounded buffer example is shown below. Recall that Occam uses indentation to delimit control-flow constructs. Also note that Occam has no `mod` operator.

```
-- channel declarations:
CHAN OF BDATA producer, consumer :
CHAN OF BOOL request :


-- buffer manager:
...     -- (data declarations omitted)
WHILE TRUE
  ALT
    full_slots < SIZE & producer ? d
      SEQ
        buf[next_empty] := d
        IF
          next_empty = SIZE
            next_empty := 1
          next_empty < SIZE
            next_empty := next_empty + 1
        full_slots := full_slots + 1
    full_slots > 0 & request ? t
      SEQ
        consumer ! buf[next_full]
        IF
          next_full = SIZE
            next_full := 1
          next_full < SIZE
            next_full := next_full + 1
        full_slots := full_slots - 1
```

The question-mark operator (`?`) is Occam's `receive`; the exclamation-mark operator (`!`) is its `send`. As in Ada, an active manager thread must embed the `ALT` statement in a loop. As written here, the `ALT` statement has two guards. The first guard is true when `full_slots < SIZE` and a message is available on the channel named `producer`; the second guard is true when `full_slots > 0` and a message is available on the channel named `request`. Because we are using synchronization `send` in this example, there is an asymmetry between the treatment of producers and consumers: the former need only send the manager data; the latter must send it a dummy argument and then wait for the manager to send the data back:

```
BDATA x :

-- producer:                -- consumer:
producer ! x                request ! TRUE
                            consumer ? x
```

The asymmetry could be removed by using remote invocation on `CALL` channels:

```
-- channel declarations:
CALL insert (VAL BDATA d) :
CALL remove (RESULT BDATA d) :



-- buffer manager:
WHILE TRUE
  ALT
    full_slots < SIZE & ACCEPT insert (VAL BDATA d)
      buf[next_empty] := d
      IF  -- increment next_empty, etc.
      ...
    full_slots > 0 & ACCEPT remove (RESULT BDATA d)
      d := buf[next_full]
      IF  -- increment next_full, etc.
      ...
```

Client code now looks like this:

```
-- producer:                -- consumer:
insert(x)                   remove(x)
```

In the code of the buffer manager, the body of the `ACCEPT` is the single subsequent statement (the one that accesses `buf`). Updates to `next_empty`, `next_full`, and `full_slots` occur after replying to the client.

The effect of an Ada `delay` can be achieved in Occam by an `ALT` arm that "receives" from a *timer* pseudo-process:

```
clock ? AFTER quit_time
```

An arm can also be selected on the basis of a Boolean condition alone, without attempting to receive:

```
a > b & SKIP        -- do nothing
```

Occam's `ALT` has no equivalent of the Ada `terminate`, nor is there an `else` (a similar effect can be achieved with a very short delay).

In SR, selective receipt is again based on guarded commands:

```
resource buffer
  op insert (d : bdata)
  op remove () returns d : bdata
body buffer
  const SIZE := 10;
  var buf[0:SIZE-1] : bdata
  var full_slots := 0, next_empty := 0, next_full := 0
  process manager
    do true ->
      in insert (d) st full_slots < SIZE ->
          buf[next_empty] := d
          next_empty := next_empty % SIZE + 1
          full_slots++
      [] remove () returns d st full_slots > 0 ->
          d := buf[next_full]
          next_full := next_full % SIZE + 1
          full_slots--
      ni
    od
  end  # manager
end  # buffer
```

The `st` stands for "such that"; it introduces the Boolean half of a guard. Client code looks like this:

```
# producer:                 # consumer:
call insert(x)              call remove(x)
```

If desired, an explicit `reply` to the client could be inserted between the access to `buf` and the updates of `next_empty`, `next_full`, and `full_slots` in each arm of the `in`.

In a significant departure from Ada and Occam, SR arranges for the parameters of a potential message to be in the scope of the `st` condition, allowing a receiver to "peek inside" a message before deciding whether to receive it:

```
in insert (d) st d % 2 = 1 ->       # only accept odd numbers
```

A receiver can also accept messages on a given port (i.e. of a given `op`) out-of-order, by specifying a *scheduling expression*:

```
in insert (d) st d % 2 = 1 by -d ->
    # only accept odd numbers, and pick the largest one first
```

Like an Ada `select`, an SR `in` statement can end with an `else` guard; this guard will be selected if no message is immediately available. There is no equivalent of `delay` or `terminate`.

12.4.4 Remote Procedure Call

Any of the three principal forms of `send` (no-wait, synchronization, remote-invocation) can be paired with either of the principal forms of `receive` (explicit or implicit). The combination of remote-invocation `send` with explicit receipt (e.g. as in Ada) is sometimes known as *rendezvous*. The combination of remote-invocation `send` with implicit receipt is usually known as *remote procedure call*. RPC is available in several concurrent languages (SR obviously among them), and is also supported on many systems by augmenting a sequential language with a *stub compiler*. The stub compiler is independent of the language's regular compiler. It accepts as input a formal description of the subroutines that are to be called remotely. The description is roughly equivalent to the subroutine headers and declarations of the types of all parameters. Based on this input the stub compiler generates source code for *client* and *server stubs*. A client stub for a given subroutine marshals request parameters and an indication of the desired operation into a message buffer, sends the message to the server, waits for a reply message, and un-marshals that message into result parameters. A server stub takes a message buffer as parameter, un-marshals request parameters, calls the appropriate local subroutine, marshals return parameters into a reply message, and sends that message back to the appropriate client. Invocation of a client stub is relatively straightforward. Invocation of server stubs is discussed in the subsection on "implementation" below.

Semantics

A principal goal of most RPC systems is to make the remote nature of calls as *transparent* as possible—that is, to make remote calls look as much like local calls as possible [BN84]. In a stub compiler system, a client stub should have the same interface as the remote procedure for which it acts as proxy; the programmer should usually be able to call the routine without knowing or caring whether it is local or remote.

Several issues make it difficult to achieve transparency in practice:

parameter modes: It is difficult to implement call-by-reference parameters across a network, since actual parameters will not be in the address space of the called routine. (Access to global variables is similarly difficult.)

performance: There is no escaping the fact that remote procedures may take a long time to return. In the face of network delays, one cannot use them casually.

failure semantics: Remote procedures are much more likely to fail than are local procedures. It is generally acceptable in the local case to assume that a called procedure will either run exactly once or else the entire program will fail. Such an assumption is overly restrictive in the remote case.

We can use value/result parameters in place of reference parameters so long as program correctness does not rely on the aliasing created by reference parameters. As noted in section 8.3.1, Ada declares that a program is *erroneous* if it can tell the difference between pass-by-reference and pass-by-value/result implementations of `in out` parameters. If absolutely necessary, reference parameters and global variables can be implemented with message-passing thunks in a manner reminiscent of call-by-name parameters (section 8.3.1),

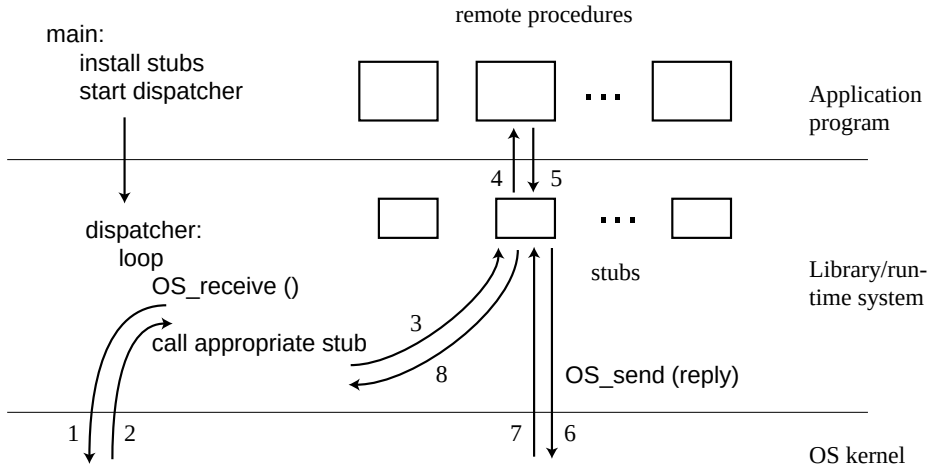


Figure 12.22: Implementation of a Remote Procedure Call server. Application code initializes the RPC system by installing stubs generated by the stub compiler (not shown). It then calls into the run-time system to enable incoming calls. Depending on details of the particular system in use, the dispatcher may use the main program's single process (in which case the call to start the dispatcher never returns), or it may create a pool of processes that handle incoming requests.

but only at very high cost. As noted in section 7.10, a few languages and systems perform deep copies of linked data structures passed to remote routines.

Performance differences between local and remote calls can only be hidden by artificially slowing down the local case. Such an option is clearly unacceptable.

Exactly-once failure semantics can be provided by aborting the caller in the event of failure or, in highly reliable systems, by delaying the caller until the operating system or language run-time system is able to rebuild the failed computation using information previously dumped to disk. (Failure recovery techniques are beyond the scope of this text.) An attractive alternative is to accept "at-most-once" semantics with notification of failure. The implementation retransmits requests for remote invocations as necessary in an attempt to recover from lost messages. It guarantees that retransmissions will never cause an invocation to happen more than once, but it admits that in the presence of communication failures the invocation may not happen at all. If the programming language provides exceptions then the implementation can use them to make communication failures look just like any other kind of run-time error.

Implementation

At the level of the kernel interface, `receive` is an explicit operation on almost all operating systems. To make `receive` appear implicit to the application programmer, the code produced by an RPC stub compiler (or the run-time system of a language such as SR) must bridge this explicit-to-implicit gap. We describe the implementation here in terms of stub compilers; in a concurrent language with implicit receipt the regular compiler does essentially the same work.

Figure 12.22 illustrates the layers of a typical RPC system. Code above the upper horizontal line is written by the application programmer. Code in the middle is a combination of library routines and code produced by the RPC stub generator. To initialize the RPC system, the application makes a pair of calls into the run-time system. The first provides the system with pointers to the stub routines produced by the stub compiler; the second starts a *message dispatcher*. What happens after this second call depends on whether the server is concurrent and, if so, whether its threads are implemented on top of one OS process or several.

In the simplest case—a single-threaded server on a single OS process—the dispatcher runs a loop that calls into the kernel to receive a message. When a message arrives, the dispatcher calls the appropriate RPC stub, which un-marshals request parameters and calls the appropriate application-level procedure. When that procedure returns, the stub marshals return parameters into a reply message, calls into the kernel to send the message back to the caller, and then returns to the dispatcher.

This simple organization works well so long as each remote request can be handled quickly, without ever needing to block. If remote requests must sometimes wait for user-level synchronization, then the server's process must manage a ready list of threads, as described in section 12.2.4, but with the dispatcher integrated into the usual thread scheduler. When the current thread blocks (in application code), the scheduler/dispatcher will grab a new thread from the ready list. If the ready list is empty, the scheduler/dispatcher will call into the kernel to receive a message, fork a new thread to handle it, and then continue to execute runnable threads until the list is empty again.

In a multi-process server, the call to start the dispatcher will generally ask the kernel to fork a "pool" of processes to service remote requests. Each of these processes will then perform the operations described in the previous paragraphs. In a language or library with a one-one correspondence between threads and processes, each process will repeatedly receive a message from the kernel and then call the appropriate stub. With a more general thread package, each process will run threads from the ready list until the list is empty, at which point it (the process) will call into the kernel for another message. So long as the number of runnable threads is greater than or equal to the number of processes, no new messages will be received. When the number of runnable threads drops below the number of processes, then the extra processes will call into the kernel, where they will block until requests arrive.

Summary and Concluding Remarks

Concurrency and parallelism have become ubiquitous in modern computer systems. It is probably safe to say that most computer research and development today involves concurrency in one form or another. High-end computer systems are almost always parallel, and multiprocessor PCs are becoming increasingly common. With the explosion in the mid 1990's of multimedia and Internet-based applications, multi-threaded and message-passing programs have become central to day-to-day computing even on uniprocessors.

In this chapter we have provided an introduction to concurrent programming with an emphasis on programming language issues. We began with a quick synopsis of the history of concurrency, the motivation for multi-threaded programs, and the architecture of modern multiprocessors. We then surveyed the fundamentals of concurrent software, including communication, synchronization, and the creation and management of threads. We

distinguished between shared memory and message-passing models of communication and synchronization, and between language and library-based implementations of concurrency.

Our survey of thread creation and management described some six different constructs for creating threads: `co-begin`, parallel loops, launch-at-elaboration, `fork/join`, implicit receipt, and early reply. Of these `fork/join` is the most common; it is found in Ada, Java, Modula-3, SR, and library-based packages such as PVM and MPI. RPC systems usually use `fork/join` internally to implement implicit receipt. Regardless of thread creation mechanism, most concurrent programming systems implement their language or library-level threads on top of a collection of OS-level processes, which the operating system implements in a similar manner on top of a collection of hardware processors. We built our sample implementation in stages, beginning with coroutines on a uniprocessor, then adding a ready list and scheduler, then timers for preemption, and finally parallel scheduling on multiple processors.

Our section on shared memory focused primarily on synchronization. We distinguished between mutual exclusion and condition synchronization, and between busy-wait and scheduler-based implementations. Among busy-wait mechanisms we looked in particular at spin locks and barriers. Among scheduler-based mechanisms we looked at semaphores, monitors, and conditional critical regions. Of the three, semaphores are the simplest and most common. Monitors and conditional critical regions provide a better degree of encapsulation and abstraction, but are not amenable to implementation in a library. Conditional critical regions might be argued to provide the most pleasant programming model, but cannot in general be implemented as efficiently as monitors. We also considered the implicit synchronization found in the loops of High Performance Fortran, the functional constructs of Sisal, and the `future`-like constructs of Multilisp, Linda, and CC++.

Our section on message-passing examined four principal issues: how to name communication partners, how long to block when sending a message, whether to receive explicitly or implicitly, and how to select among messages that may be available for receipt simultaneously. We noted that any of the three principal `send` mechanisms (no-wait, synchronization, remote-invocation) can be paired with either of the principal `receive` mechanisms (explicit, implicit). Remote-invocation `send` with explicit receipt is sometimes known as *rendezvous*. Remote-invocation `send` with implicit receipt is generally known as *remote procedure call*.

As in previous chapters, we saw many cases in which language design and language implementation influence one another. Some mechanisms (cactus stacks, conditional critical regions, content-based message screening) are sufficiently complex that many language designers have choosen not to provide them. Other mechanisms (Ada-style parameter modes) have been developed specifically to facilitate an efficient implementation technique. And in still other cases (the semantics of no-wait send, blocking inside a monitor) implementation issues play a major role in some larger set of tradeoffs.

Despite the very large number of concurrent languages that have been designed to date, most concurrent programming continues to employ conventional sequential languages augmented with library packages. As of 1999, HPF and other concurrent languages for large-scale multicomputers have yet to seriously undermine the dominance of PVM and MPI. For smaller-scale shared-memory computing, programmers continue to rely on library packages in C and C++. At the very least, it would appear that a necessary (but not sufficient) condition for widespread acceptance of any concurrent language is that it be seen as an extension to some successful and popular sequential language, in which programmers have

already made a substantial intellectual investment, and for which outstanding compilers are available. Among languages currently on the horizon, Java seems to be the most likely exception to this rule. Its suitability for network-based computing, its extreme portability across platforms, and the enthusiasm with which it has been embraced by the popular press appear to be establishing an enormous base of support in spite of initially poor compilers. For the relatively safe, on-demand construction of programs that span the Internet, Java currently has no serious competitor.

Review Questions

1. Explain the rationale for concurrency: why do people write concurrent programs?
2. Describe the evolution of computer operation from *stand-alone* mode to *batch processing*, to *multiprogramming* and *timesharing*.
3. Describe six different syntactic constructs commonly used to create new threads of control in a concurrent program.
4. Explain the difference between a thread and a coroutine.
5. What are the tradeoffs between language-based and library-based implementations of concurrency?
6. Name four explicitly concurrent programming languages.
7. What is *busy-waiting*? What is its principal alternative?
8. What is a *race condition*?
9. What is a *context switch*? What is *preemption*?
10. Explain the *coherence problem*, e.g. in the context of multiprocessor caches.
11. Describe the *bag of tasks* programming model.
12. Explain the difference between *data parallelism* and *task parallelism*.
13. What is *co-scheduling*? What is its purpose?
14. What is a *critical section*?
15. What does it mean for an operation to be *atomic*?
16. Explain the difference between *mutual exclusion* and *condition synchronization*.
17. Describe the behavior of a `test_and_set` instruction. Show how to use it to build a *spin lock*.
18. Describe the behavior of the `load_linked` and `store_conditional` instructions. What advantages do they offer in comparison to `test_and_set`?
19. Explain how a *reader-writer lock* differs from an "ordinary" lock.

20. What is a *barrier*? In what types of programs are barriers common?
21. What does it mean for code to be *reentrant*?
22. What is a *semaphore*? What operations does it support? How do *binary* and *general* semaphores differ?
23. What is a *monitor*? How do monitor *condition variables* differ from semaphores?
24. What is a *conditional critical region*? How does it differ from a monitor?
25. What is *deadlock*?
26. Describe the semantics of the HPF/Fortran 95 `forall` loop.
27. Explain the difference between *AND parallelism* and *OR parallelism* in Prolog.
28. What are *single-assignment variables*? In what languages do they appear?
29. What are *gather* and *scatter* operations in a message-passing program?
30. Describe three ways in which processes commonly name their communication partners.
31. What are the three principal synchronization options for the sender of a message? What are the tradeoffs among them?
32. Describe the tradeoffs between *explicit* and *implicit* message receipt.
33. What is a *remote procedure call* (RPC)? What is a *stub compiler*?
34. What are the obstacles to *transparency* in an RPC system?
35. What is a *rendezvous*? How does it differ from a remote procedure call?
36. What is an *early reply*?

Exercises

1. Give an example of a "benign" race condition—one whose outcome affects program behavior, but not correctness.
2. We have defined the *ready list* of a thread package to contain all threads that are runnable but not running, with a separate variable to identify the currently running thread. Could we just as easily have defined the ready list to contain *all* runnable threads, with the understanding that the one at the head of the list is running? (Hint: think about multiprocessors.)
3. Imagine you are writing the code to manage a hash table that will be shared among several concurrent threads. Assume that operations on the table need to be atomic. You could use a single mutual exclusion lock to protect the entire table, or you could devise a scheme with one lock per hash-table bucket. Which approach is likely to work better, under what circumstances? Why?

4. The typical spin lock holds only one bit of data, but requires a full word of storage, because only full words can be read, modified, and written atomically in hardware. Consider, however, the hash table of the previous exercise. If we choose to employ a separate lock for each bucket of the table, explain how to implement a "two-level" locking scheme that couples a conventional spin lock for the table as a whole with a *single bit* of locking information for each bucket. Explain why such a scheme might be desirable, particularly in a table with external chaining. (Hint: see the paper by Stumm et al. [UKGS94].)

5. Many of the most compute-intensive scientific applications are "dusty-deck" Fortran programs, generally very old and very complex. Years of effort may sometimes be required to rewrite a dusty-deck program to run on a parallel machine. An attractive alternative would be to develop a compiler that could "parallelize" old programs automatically. Explain why this is not an easy task.

6. The `load_linked` and `store_conditional` (`LL/SC`) instructions described in section 12.3.1 resemble an earlier universal atomic operation known as `compare-and-swap` (`CAS`). `CAS` was introduced by the IBM 370 architecture, and also appears in the 680x0 and SPARC V9 instruction sets. It takes three operands: the location to be modified, a value that the location is expected to contain, and a new value to be placed there if (and only if) the expected value is found. Like `store_conditional`, `CAS` returns an indication of whether it succeeded. The atomic add instruction sequence shown for `load_linked/store_conditional` on page 29 would be written as follows with `CAS`:

```
start:
    r1 := foo
    r3 := r1 + r2
    CAS (foo, r1, r3)
    if failed goto start
```

   Discuss the relative advantages of `LL/SC` and `CAS`. Consider how they might be implemented on a cache-coherent multiprocessor. Are there situations in which one would work but the other would not? (Hints: consider algorithms in which a thread may need to touch more than one memory location. Also consider algorithms in which the contents of a memory location might be changed and then restored.)

7. On most machines, a `SC` instruction can fail for any of several reasons, including the occurrence of an interrupt in the time since the matching `LL`. What steps must a programmer take to make sure that algorithms work correctly in the face of such "spurious" `SC` failures?

8. Starting with the test-and-`test_and_set` lock of figure 12.10, implement busy-wait code that will allow readers to access a data structure concurrently. Writers will still need to lock out both readers and other writers. You may use any reasonable atomic instruction(s), e.g. `LL/SC`. Consider the issue of fairness. In particular, if there are *always* readers interested in accessing the data structure, your algorithm should ensure that writers are not locked out forever.

9. The mechanism used in figure 12.12 (page 31) to make scheduler code reentrant employs a single OS-provided lock for all the scheduling data structures of the application. Among other things, this mechanism prevents threads on separate processors from performing `P` or `V` operations on unrelated semaphores, even when none of the operations needs to block. Can you devise another synchronization mechanism for scheduler-related operations that admits a higher degree of concurrency but that is still correct?

10. We have seen how the scheduler for a thread package that runs on top of more than one OS-provided process must both disable timer signals *and* acquire a spin lock to safeguard the integrity of the ready list and condition queues. To implement processes within the operating system, the kernel still uses spin locks, but with processors instead of processes, and hardware interrupts instead of signals. Unfortunately, the kernel cannot afford to disable interrupts for more than a small, bounded period of time, or devices may not work correctly. A straightforward adaptation of the code in figure 12.12 will not suffice because it would attempt to acquire a spin lock (an unbounded operation) while interrupts were disabled. Similarly, the kernel cannot afford to acquire a spin lock and then disable interrupts because, if an interrupt occurs in-between these two operations, other processors may be forced to spin for a very long time. How would you solve this problem? (Hint: look carefully at the loop in the middle of `reschedule`, and consider a hybrid technique that disables interrupts and acquires a spin lock as a single operation.)

11. To make spin locks useful on a multiprogrammed multiprocessor, one might want to ensure that no process is ever preempted in the middle of a critical section. That way it would always be safe to spin in user space, because the process holding the lock would be guaranteed to be running on some other processor, rather than preempted and possibly in need of the current processor. Explain why an operating system designer might not want to give user processes the ability to disable preemption arbitrarily. (Hint: think about fairness and multiple users.) Can you suggest a way to get around the problem? (References to several possible solutions can be found in the paper by Kontothanassis, Wisniewski, and Scott [KWS97].)

12. Show how to use semaphores to construct an n-thread barrier.

13. Would it ever make sense to declare a semaphore with an initially negative count? Why or why not?

14. Without looking at Hoare's definition, show how to implement monitors with semaphores.

15. Using monitors, show how to implement semaphores. What is your monitor invariant?

16. Show how to implement general semaphores, given only binary semaphores.

17. Suppose that every monitor has a separate mutual exclusion lock, and that we want to release all locks when a thread `wait`s in the innermost of a sequence of nested monitor calls. When the thread awakens it will need to reacquire the outer locks. In what order should it do so? (Hint: think about deadlock.) Can we guarantee that

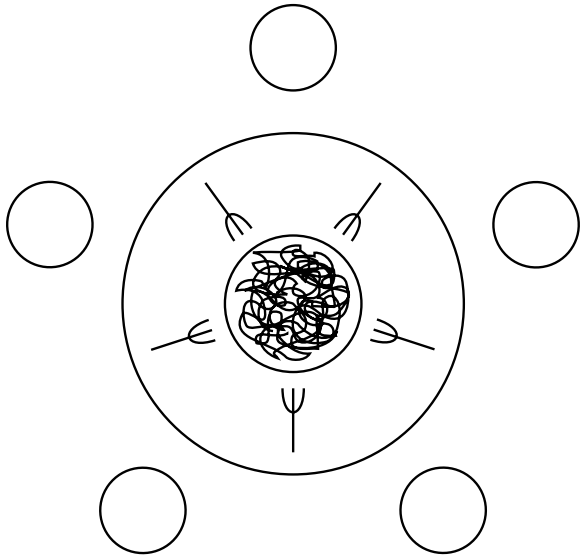


Figure 12.23: The Dining Philosophers. Hungry philosophers must contend for the forks to their left and right in order to eat.

the awakened thread will be the next to run in the innermost monitor? (For further hints, see Wettstein [Wet78].)

18. In addition to the usual `signal` operation, Mesa and Modula-3 provide a `broadcast` operation that awakens *all* threads waiting on a given monitor condition variable. Show how the programmer can achieve a similar effect (awakening all threads) in a Hoare monitor. What is the advantage of the built-in `broadcast` operation?

19. Show how general semaphores can be implemented with conditional critical regions in which all threads wait for the same condition, thereby avoiding the overhead of unproductive wake-ups.

20. Write code for a bounded buffer using Java and/or the protected object mechanism of Ada 95.

21. The *dining philosophers problem* [Dij72] is a classic exercise in synchronization (figure 12.23). Five philosophers sit around a circular table. In the center is a large communal plate of spaghetti. Each philosopher repeatedly thinks for a while and then eats for a while, at intervals of his or her own choosing. On the table between each pair of adjacent philosophers is a single fork. To eat, a philosopher requires both adjacent forks: the one on the left and the one on the right. Because they share a fork, adjacent philosophers cannot eat simultaneously.

    Write a solution to the dining philosophers problem in which each philosopher is represented by a process and the forks are represented by shared data. Synchronize access to the forks using semaphores, monitors, or conditional critical regions. Try to maximize concurrency.

22. In the previous exercise you may have noticed that the dining philosophers are prone to deadlock. One has to worry about the possibility that all five of them will pick up their right-hand forks simultaneously, and then wait forever for their left-hand neighbors to finish eating.

    Discuss as many strategies as you can think of to address the deadlock problem. Can you describe a solution in which it is provably impossible for any philosopher to go hungry forever? Can you describe a solution that is fair in a strong sense of the word (i.e. in which no one philosopher gets more chance to eat than some other over the long term)? For a particularly elegant solution, see the paper by Chandy and Misra [CM84].

23. In some concurrent programming systems, global variables are shared by all threads. In others, each newly created thread has a separate copy of the global variables, commonly initialized to the values of the globals of the creating thread. Under this private globals approach, shared data must be allocated from a special heap. In still other programming systems, the programmer can specify which global variables are to be private and which are to be shared.

    Discuss the tradeoffs between private and shared global variables. Which would you prefer to have available, for which sorts of programs? How would you implement each? Are some options harder to implement than others? To what extent do your answers depend on the nature of processes provided by the operating system?

24. AND parallelism in logic languages is analogous to the parallel evaluation of arguments in a functional language (e.g. Multilisp). Does OR parallelism have a similar analog? (Hint: think about special forms (section 11.2.2).) Can you suggest a way to obtain the effect of OR parallelism in Multilisp?

25. In section 12.3.4 we claimed that both AND parallelism and OR parallelism were problematic in Prolog, because they failed to adhere to the deterministic search order required by language semantics. Elaborate on this claim. What specifically can go wrong?

26. Find out how message-passing is implemented in some locally available concurrent language or library. Does this system provide no-wait `send`, synchronization `send`, remote-invocation `send`, or some related hybrid? If you wanted to emulate the other options using the one available, how expensive would be emulation be, in terms of low-level operations performed by the underlying system? How would this overhead compare to what could be achieved on the same underlying system by a language or library that provided an optimized implementation of the other varieties of `send`?

27. In section 12.3.3 we cast monitors as a mechanism for synchronizing access to shared memory, and we described their implementation in terms of semaphores. It is also possible to think of a monitor as a module inhabited by a single process, which accepts request messages from other processes, performs appropriate operations, and replies. Give the details of a monitor implementation consistent with this conceptual model. Be sure to include condition variables. (Hint: see the discussion of early reply in section 12.2.3, page 20.)

28. Show how shared memory can be used to implement message passing. Specifically, choose a set of message-passing operations (e.g. no-wait `send` and explicit message receipt) and show how to implement them in your favorite shared-memory notation.

29. When implementing reliable messages on top of unreliable messages, a sender can wait for an acknowledgment message, and retransmit if it doesn't receive it within a bounded period of time. But how does the receiver know that its acknowledgment has been received? Why doesn't the sender have to acknowledge the acknowledgment (and the receiver acknowledge the acknowledgment of the acknowledgment . . .)? (For more information on the design of fast, reliable protocols, you might want to consult a text on computer networks [Tan96, PD96].)

30. An arm of an Occam `ALT` statement may include an *input guard*—a receive (`?`) operation—in which case the arm can be chosen only if a potential partner is trying to send a matching message. One could imagine allowing *output guards* as well—send (`!`) operations that would allow their arm to be chosen only if a potential partner were trying to receive a matching message. Neither Occam nor CSP (as originally defined) permits output guards. Can you guess why? Suppose you wished to provide them. How would the implementation work? (Hint: for ideas, see the articles of Bernstein [Ber80], Buckley and Silbershatz [BS83], Bagrodia [Bag86], or Ramesh [Ram87].)

31. In section 12.4.3 we described the semantics of a `terminate` arm on an Ada `select` statement: this arm may be selected if and only if all potential communication partners have terminated, or are likewise stuck in `select` statements with `terminate` arms. Occam and SR have no similar facility, though the original CSP proposal does. How would you implement `terminate` arms in Ada? Why do you suppose they were left out of Occam and SR? (Hint: for ideas, see the work of Apt and Francez [Fra80, AF84].)

Bibliographic Notes

Much of the early study of concurrency stems from a pair of articles by Dijkstra [Dij68, Dij72]. Andrews and Schneider [AS83] provide an excellent survey of concurrent programming notations. The more recent book by Andrews [And91] extends this survey with extensive discussion of axiomatic semantics for concurrent programs and algorithmic paradigms for distributed computing. Holt et al. [HGLS78] is a useful reference for many of the classic problems in concurrency and synchronization. Anderson [ALL89] discusses thread package implementation details and their implications for performance. The July 1989 issue of *IEEE Software* and the September 1989 issue of *ACM Computing Surveys* contain survey articles and descriptions of many concurrent languages. References for monitors appear in section 12.3.3.

Peterson's two-process synchronization algorithm appears in a remarkably elegant and readable two-page paper [Pet81]. Lamport's 1978 article on "Time, Clocks, and the Ordering of Events in a Distributed System" [Lam78] argued convincingly that the notion of global time cannot be well defined, and that distributed algorithms must therefore be based on causal *happens before* relationships among individual processes. Reader-writer locks are due to Courtois, Heymans, and Parnas [CHP71]. Mellor-Crummey and Scott [MCS91]

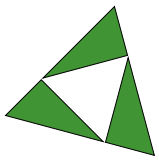
survey the principal busy-wait synchronization algorithms and introduce locks and barriers that scale without contention to very large machines. The seminal paper on lock-free synchronization is that of Herlihy [Her91].

Concurrent logic languages are surveyed by Shapiro [Sha89], Tick [Tic91], and Ciancarini [Cia92]. Parallel Lisp dialects include Multilisp [Hal85, MKH91] (section 12.3.4), Qlisp [GG89], and Spur Lisp [ZHL+89].

Remote Procedure Call received increasing attention in the wake of Nelson's Ph.D. research [Nel81, BN84]. Schroeder and Burrows [SB90] discuss the efficient implementation of RPC on a network of workstations. Bershad [BALL90] discusses its implementation across address spaces within a single machine.

Almasi and Gottlieb [AG94] describe the principal classes of parallel computers and the styles of algorithms and languages that work well on each. The leading texts on computer networks are by Tanenbaum [Tan96] and Peterson and Davie [PD96]. The recent text of Culler, Singh, and Gupta [CS98] contains a wealth of information on parallel programming and multiprocessor architecture. PVM [Sun90, GBD+94] and MPI [BDH+95, SOHL+95] are documented in a variety of articles and books. Sun RPC is documented in Internet RFC number 1831 [Sri95].

Software distributed shared memory (S-DSM) was originally proposed by Li as part of his Ph.D. research [LH89]. Stumm and Zhou [SZ90] and Nitzberg and Lo [NL91] provide early surveys of the field. The TreadMarks system [ACD+96] from Rice University is widely considered the best of the more recent implementations.



Bibliography

[ACC+90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, Amsterdam, The Netherlands, June 1990. ACM Press, New York, NY. In *ACM Computer Architecture News*, 18(3), September 1990.

[ACD+96] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[ACG86] Shakil Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[AF84] Krzysztof R. Apt and Nissim Francez. Modeling the distributed termination convention of CSP. *ACM Transactions on Programming Languages and Systems*, 6(3):370–379, July 1984.

[AG94] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, Redwood City, CA, second edition, 1994. ISBN 0-8053-0443-6.

[ALL89] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[And91] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, Redwood City, CA, 1991. ISBN 0-8053-0086-4.

[AO93] Gregory R. Andrews and Ronald A. Olsson. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings, Redwood City, CA, 1993. ISBN 0-8053-0088-0.

[AS83] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[Bag86] Rajive L. Bagrodia. A distributed algorithm to implement the generalized alternative command of CSP. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, pages 422–427, Cambridge, MA, May 1986. IEEE Computer Society Press, Washington, DC.

[BALL90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[BDH+95] Jehoshua Bruck, Danny Dolev, Ching-Tien Ho, Marcel-Catalin Rosu, and Ray Strong. Efficient message passing interface (MPI) for parallel computing on clusters of workstations. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 64–73, Santa Barbara, CA, July 1995.

[Ber80] Arthur J. Bernstein. Output guards and nondeterminism in 'Communicating Sequential Processes'. *ACM Transactions on Programming Languages and Systems*, 2(2):234–238, April 1980.

[BN84] Andrew D. Birrell and Bruce J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Bri73] Per Brinch Hansen. *Operating System Principles*. Prentice-Hall, Englewood Cliffs, NJ, 1973. ISBN 0-13-637843-9.

[Bri75] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, SE–1(2):199–207, June 1975.

[Bri78] Per Brinch Hansen. Distributed processes: A concurrent programming concept. *Communications of the ACM*, 21(11):934–941, November 1978.

[Bri81] Per Brinch Hansen. The design of Edison. *Software—Practice and Experience*, 11(4):363–396, April 1981.

[Bro96] Kraig Brockschmidt. How OLE and COM solve the problems of component software design. *Microsoft Systems Journal*, 11(5):63–82, May 1996.

[BS83] G. N. Buckley and A. Silbershatz. An effective implementation for the generalized input-output construct of CSP. *ACM Transactions on Programming Languages and Systems*, 5(2):223–235, April 1983.

[Can92] David Cann. Retire Fortran? A debate rekindled. *Communications of the ACM*, 35(8):81–89, August 1992.

[CHP71] P. J. Courtois, F. Heymans, and David L. Parnas. Concurrent control with 'readers' and 'writers'. *Communications of the ACM*, 14(10):667–668, October 1971.

[Cia92] Paolo Ciancarini. Parallel programming with logic languages: A survey. *Computer Languages*, 17(4):213–239, October 1992.

[CM84] K. Mani Chandy and Jayadev Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[CS98] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, San Francisco, CA, 1998. With Anoop Gupta. ISBN 1-55860-343-3.

[Dij65] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[Dij68] Edsger W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, London, England, 1968.

[Dij72] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. In Charles Antony Richard Hoare and Ronald H. Perrott, editors, *Operating Systems Techniques*, A.P.I.C. Studies in Data Processing #9, pages 72–93. Academic Press, London, England, 1972. ISBN 0-123-50650-6. Also *Acta Informatica*, 1(8):115–138, 1971.

[Fos95] Ian Foster. Compositional C++. In *Debugging and Building Parallel Programs*, chapter 5, pages 167–204. Addison-Wesley, Reading, MA, 1995. ISBN 0-201-57594-9. Available in hypertext at `http://www.mcs.anl.gov/dbpp/text/node51.html`.

[Fra80] Nissim Francez. Distributed termination. *ACM Transactions on Programming Languages and Systems*, 2(1):42–55, January 1980.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation series. MIT Press, Cambridge, MA, 1994. ISBN 0-262-57108-0. Available in hypertext at `http://www.netlib.org/pvm3/book/pvm-book.html`.

[GG89] Ron Goldman and Richard P. Gabriel. Qlisp: Parallel processing in Lisp. *IEEE Software*, 6(4):51–59, July 1989.

[GKP96] George Al Geist, J. A. Kohl, and P. M. Papadopoulos. PVM and MPI: a comparison of features. *Calculateurs Paralleles*, 8(2), 1996.

[Hal85] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[Her91] Maurice P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[HGLS78] Richard C. Holt, G. Scott Graham, Edward D. Lazowska, and Mark A. Scott. *Structured Concurrent Programming with Operating Systems Applications*. Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, MA, 1978. ISBN 0-201-02937-5.

[HM92] Maurice P. Herlihy and J. Elliot B. Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):304–311, May 1992.

[Hoa74] Charles Antony Richard Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hoa78] Charles Antony Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[JG89] Geraint Jones and Michael Goldsmith. *Programming in occam2*. Prentice-Hall International Series in Computer Science. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1989. ISBN 0-13-730334-3.

[Jor85] Harry F. Jordan. HEP architecture, programming and performance. In Janusz S. Kowalik, editor, *Parallel MIMD Computation: The HEP Supercomputer and its Applications*, pages 1–40. MIT Press, Cambridge, MA, 1985. ISBN 0-262-11101-2.

[Kes77] J. L. W. Kessels. An alternative to event queues for synchronization in monitors. *Communications of the ACM*, 20(7):500–503, July 1977.

[KLS+94] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation Series. MIT Press, Cambridge, MA, 1994. ISBN 0-262-61094-9.

[KWS97] Leonidas I. Kontothanassis, Robert Wisniewski, and Michael L. Scott. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, February 1997.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam87] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LR80] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[MKH91] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[MS96] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, Philadelphia, PA, May 1996.

[Nel81] Bruce J. Nelson. *Remote Procedure Call*. Ph. D. dissertation, Carnegie-Mellon University, 1981. School of Computer Science Technical Report CMU-CS-81-119.

[NL91] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[Ope96] Open Software Foundation. *OSF DCE Application Development Reference, Release 1.1*. OSF DCE Series. Prentice-Hall, Upper Saddle River, NJ, 1996. ISBN 0-13-185869-6.

[Ous82] John K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the Third International Conference on Distributed Computing Systems*, pages 22–30, Miami/Ft. Lauderdale, FL, October 1982. IEEE Computer Society Press, Silver Spring, MD.

[PD96] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann, San Francisco, CA, 1996. ISBN 1-55860-368-9.

[Pet81] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[Ram87] S. Ramesh. A new efficient implementation of CSP with output guards. In *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 266–273, Berlin, West Germany, September 1987. IEEE Computer Society Press, Washington, DC.

[SB90] Michael Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[SBG+91] Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, Daniel M. Yellin, and Shaula Alexander Yemini. *Hermes: A Language for Distributed Computing*. Prentice-Hall Series in Innovative Technology. Prentice-Hall, Englewood Cliffs, NJ, 1991. ISBN 0-13-389537-8.

[Sco91] Michael L. Scott. The Lynx distributed programming language: Motivation, design, and experience. *Computer Languages*, 16(3/4):209–233, 1991.

[Sha89] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, September 1989. Correction appears in Volume 21, Number 4.

[Sie96] Jon Siegel. *CORBA Fundamentals and Programming*. John Wiley and Sons, New York, NY, 1996. ISBN 0-471-12148-7.

[SOHL+95] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. Scientific and Engineering Computation series. MIT Press, Cambridge, MA, 1995. ISBN 0-262-69184-1. Available in hypertext at `http://www.netlib.org/utk/papers/mpi-book/mpi-book.html`.

[Sri95] Raj Srinivasan. RPC: Remote procedure call protocol specification version 2. Internet Request for Comments #1831, August 1995. Available as `http://www.cis.ohio-state.edu/htbin/rfc/rfc1831.html`.

[Sun90] Vaidyalingam S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency—Practice and Experience*, 2(4):315–339, December 1990.

[Sun97] Sun Microsystems, Mountain View, CA. *JavaBeans*, July 1997. Available at `http://www.java.sun.com/beans`.

[SZ90] Michael Stumm and Songnian Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 23(5):54–64, May 1990.

[Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, Upper Saddle River, NJ, third edition, 1996. ISBN 0-13-349945-6.

[Tic91] Evan Tick. *Parallel Logic Programming*. Logic Programming Series. MIT Press, Cambridge, MA, 1991. ISBN 0-262-20087-2.

[UKGS94] Ronald C. Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Experiences with locking in a NUMA multiprocessor operating system kernel. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 139–152, Monterey, CA, November 1994.

[Wet78] Horst Wettstein. The problem of nested monitor calls revisited. *ACM Operating Systems Review*, 12(1):19–23, January 1978.

[Wir77a] Niklaus Wirth. Design and implementation of Modula. *Software—Practice and Experience*, 7(1):67–84, January–February 1977.

[Wir77b] Niklaus Wirth. Modula: A language for modular multiprogramming. *Software—Practice and Experience*, 7(1):3–35, January–February 1977.

[YA93] Jae-Heon Yang and James H. Anderson. Fast, scalable synchronization with minimal hardware support (extended abstract). In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, pages 171–182, Ithaca, NY, August 1993.

[ZHL+89] Benjamin G. Zorn, Kimson Ho, James Larus, Luigi Semenzato, and Paul Hilfinger. Multiprocessing extensions in Spur Lisp. *IEEE Software*, 6(4):41–49, July 1989.