This chapter begins with introductions to the various kinds of concurrency at the subprogram, or unit level, and at the statement level. Included is a brief description of some common kinds of multiprocessor computer architectures. Next, a lengthy discussion on unit-level concurrency is presented. This begins with a description of the fundamental concepts that must be understood before discussing unit-level concurrency, including competition and cooperation synchronization. Next, the design issues for providing language support for concurrency are described. Following this is a detailed discussion, including code examples, of the three major approaches to language support for concurrency: semaphores, monitors, and message passing. A pseudocode example program is used to demonstrate how semaphores can be used. Ada and Java are used to illustrate monitors; for message passing, Ada is used. The Ada features that support concurrency are described in some detail. These include tasks, protected objects (which are effectively monitors), and asynchronous message passing. Support for unit-level concurrency in Java and C# is then discussed. The last section of the chapter has a discussion of statement-level concurrency, including a short description of part of the language support provided for it in High-Performance Fortran.

13.1 Introduction

Concurrency in software execution can occur at four different levels: instruction level (executing two or more machine instructions simultaneously), statement level (executing two or more source language statements simultaneously), unit level (executing two or more subprogram units simultaneously), and program level (executing two or more programs simultaneously). Because no language design issues are involved with them, we do not discuss instruction-level and program-level concurrency in this chapter. Concurrency at both the subprogram and the statement levels is discussed, with most of the discussion focused on the subprogram level.

Concurrent execution of subprograms can occur either physically, on separate processors, or logically, by sharing a single processor. At first glance, concurrency may appear to be a simple concept, but it presents a significant challenge to the programming language designer.

Concurrent control mechanisms increase programming flexibility. They were originally invented to be used for particular problems faced in operating systems, but they are required for a variety of other programming applications. For example, many software systems are designed to simulate actual physical systems, and many of these physical systems consist of multiple concurrent subsystems. For these applications, the traditional restricted form of subprogram control is inadequate.

Statement-level concurrency is quite different from concurrency at the unit level. From a language designer's point of view, statement-level concurrency is largely a matter of specifying how data should be distributed over multiple memories and which statements can be executed concurrently.

The intention of this chapter is to discuss the aspects of concurrency that are most relevant to language design issues, rather than to present a definitive study of all of the issues of concurrency. That would clearly be inappropriate for a book on programming languages.

13.1.1 Multiprocessor Architectures

A large number of different computer architectures have more than one processor and can support some form of concurrent execution. Before beginning to discuss concurrent execution of programs and statements, we briefly describe some of these architectures.

The first computers that had multiple processors had one general-purpose processor and one or more other processors, often called peripheral processors, that were used only for input and output operations. This architecture allowed these computers, which appeared in the late 1950s, to execute one program while concurrently performing input or output for other programs. Because this kind of concurrency does not require language support, we will not consider it further.

By the early 1960s, there were machines that had multiple complete processors. These processors were used by the job scheduler of the operating system, which distributed separate jobs from a batch-job queue to the separate processors. Systems with this structure supported program-level concurrency.

In the mid-1960s, machines appeared that had several identical partial processors that were fed certain instructions from a single instruction stream. For example, some machines had two or more floating-point multipliers, while others had two or more complete floating-point arithmetic units. The compilers for these machines were required to determine which instructions could be executed concurrently and to schedule these instructions accordingly. Systems with this structure supported instruction-level concurrency.

There are now many different kinds of multiprocessor computers; the most common two categories of these are described in the following two paragraphs.

Computers that have multiple processors that execute the same instruction simultaneously, each on different data, are called Single-Instruction Multiple-Data (SIMD) architecture computers. In an SIMD computer, each processor has its own local memory. One processor controls the operation of the other processors. Because all of the processors, except the controller, execute the same instruction at the same time, no synchronization is required in the software. Perhaps the most widely used SIMD machines are a category of machines called **vector processors.** They have groups of registers that store the operands of a vector operation in which the same instruction is executed on the whole group of operands simultaneously. The kinds of programs that can most benefit from this architecture are common in scientific computation, an area of computing that is often the target of multiprocessor machines.

Computers that have multiple processors that operate independently but whose operations can be synchronized are called Multiple-Instruction

Multiple-Data (MIMD) computers. Each processor in an MIMD computer executes its own instruction stream. MIMD computers can appear in two distinct configurations: distributed and shared memory systems. The distributed MIMD machines, in which each processor has its own memory, can be either built in a single box or distributed, perhaps over a large area. The shared-memory MIMD machines obviously must provide some means of synchronization to prevent memory access clashes. Even distributed MIMD machines require synchronization to operate together on single programs. MIMD computers, which are more expensive and more general than SIMD computers, clearly support unit-level concurrency.

13.1.2 Categories of Concurrency

There are two distinct categories of concurrent unit control. The most general category of concurrency is that in which, assuming that more than one processor is available, several program units from the same program literally execute simultaneously. This is **physical concurrency.** A slight relaxation of this concept of concurrency allows the programmer and the application software to assume that there are multiple processors providing actual concurrency, when in fact, the actual execution of programs is taking place in interleaved fashion on a single processor. This is **logical concurrency.** It is similar to the illusion of simultaneous execution that is provided to different users of a multiprogramming computer system. From the programmer's and language designer's points of view, logical concurrency is the same as physical concurrency. It is the language implementor's task, using the capabilities of the underlying operating system, to map the logical concurrency to the host hardware. Both logical and physical concurrency allow the concept of concurrency to be used as a program design methodology. For the remainder of this chapter, the discussion will apply to both physical and logical concurrency.

One useful technique for visualizing the flow of execution through a program is to imagine a thread laid on the statements of the source text of the program. Every statement reached on a particular execution is covered by the thread representing that execution. Visually following the thread through the source program traces the execution flow through the executable version of the program. Of course, in all but the simplest of programs, the thread follows a highly complex path. A **thread of control** in a program is the sequence of program points reached as control flows through the program.

Programs that have coroutines (see Chapter 9), though they are sometimes called **quasi-concurrent,** have a single thread of control. Programs executed with physical concurrency can have multiple threads of control. Each processor can execute one of the threads. Although logically concurrent program execution may actually have only a single thread of control, such programs can be designed and analyzed only by imagining them as having multiple threads of control. A program designed to have more than one thread of control is said to be **multithreaded.** When a multithreaded program executes on a single-processor machine, its threads are mapped onto a single thread. It becomes, in this case, a virtually multithreaded program.

Statement-level concurrency is a relatively simple concept. Loops that include statements that operate on array elements are unwound so that the processing can be distributed over multiple processors. For example, a loop that executes 500 repetitions and includes a statement that operates on one of 500 array elements may be unwound so that each of 10 different processors can simultaneously process 50 of the array elements.

13.1.3 Motivations for Studying Concurrency

The primary reason to study concurrency is that it provides a method of conceptualizing program solutions to problems. Many problem domains lend themselves naturally to concurrency in much the same way that recursion is a natural way to design the solution to some problems. Many programs are written to simulate physical entities and activities. In many cases, the system being simulated includes more than one entity, and the entities do whatever they do simultaneously—for example, aircraft flying in a control area, relay stations in a communications network, and the various machines in a manufacturing facility. To simulate such systems accurately with software, languages that support concurrency are required.

The second reason to discuss concurrency is that multiple-processor computers currently are being used, although that use is not widespread. This creates the need for software to make effective use of that hardware capability. Because of the importance of both statement-level and unit-level concurrency, facilities to provide them must be developed and included in contemporary programming languages.

13.2 Introduction to Subprogram-Level Concurrency

Before we can discuss language support for concurrency, we must introduce the underlying concepts of concurrency and the requirements for it to be useful. Then we can discuss the language design issues for languages that support concurrency.

13.2.1 Fundamental Concepts

A **task** is a unit of a program, similar to a subprogram, that can be in concurrent execution with other units of the same program. Each task in a program can provide one thread of control. Tasks are sometimes called **processes.**

Three characteristics of tasks distinguish them from subprograms. First, a task may be implicitly started, whereas a subprogram must be explicitly called. Second, when a program unit invokes a task, it need not wait for the task to complete its execution before continuing its own. Lastly, when the execution

of a task is completed, control may or may not return to the unit that started that execution.

Tasks fall into two general categories, heavyweight and lightweight. Simply stated, each **heavyweight task** executes in its own address space. **Lightweight tasks** all run in the same address space. It is easier to implement lightweight tasks than heavyweight tasks.

A task can communicate with other tasks through shared nonlocal variables, through message passing, or through parameters. If a task does not communicate with or affect the execution of any other task in the program in any way, it is said to be **disjoint.** Because tasks often work together to create simulations or solve problems and therefore are not disjoint, they must use some form of communication to either synchronize their executions or share data or both.

Synchronization is a mechanism that controls the order in which tasks execute. Two kinds of synchronization are required when tasks share data: cooperation and competition. **Cooperation synchronization** is required between task A and task B when task A must wait for task B to complete some specific activity before task A can continue its execution. **Competition synchronization** is required between two tasks when both require the use of some resource that cannot be simultaneously used. Specifically, if task A needs to access shared data location x while task B is accessing x, task A must wait for task B to complete its processing of x, regardless of what that processing is. So, for cooperation synchronization, tasks may need to wait for the completion of specific processing on which their correct operation depends, whereas for competition synchronization, tasks may need to wait for the completion of any other processing by any task currently occurring on specific shared data.

A simple form of cooperation synchronization can be illustrated by a common problem called the **producer-consumer problem.** This problem originated in the development of operating systems, in which one program unit produces some data value or resource and another uses it. Produced data are usually placed in a storage buffer by the producing unit and removed from that buffer by the consuming unit. The sequence of stores to and removals from the buffer must be synchronized. The consumer unit must not be allowed to take data from the buffer if the buffer is empty. Likewise, the producer unit cannot be allowed to place new data in the buffer if the buffer is full. This is called the **problem of cooperation synchronization** because the users of the shared data structure must cooperate if the buffer is to be used correctly.

Competition synchronization prevents two tasks from accessing a shared data structure at exactly the same time—a situation that could destroy the integrity of that shared data. To provide competition synchronization, mutually exclusive access to the shared data must be guaranteed.

To clarify the competition problem, consider the following scenario: Suppose task A must add 1 to the shared integer variable TOTAL, which has an initial value of 3. Furthermore, suppose task B must multiply the value of

TOTAL by 2. Each task accomplishes its operation on TOTAL with the following three-step process:

1. Fetch the value of TOTAL.
2. Perform the arithmetic operation.
3. Put the new value back in TOTAL.

Without competition synchronization, four different values could result from these operations. If task A completes its operation before task B begins, the value will be 8, which is assumed here to be correct. But if both A and B fetch the value of TOTAL before either task puts its new value back, the result will be incorrect. If A puts its value back first, the value of TOTAL will be 6. This case is shown in Figure 13.1. If B puts its value back first, the value of TOTAL will be 4. Finally, if B completes its operation before task A begins, the value will be 7. A situation that leads to these problems is sometimes called a **race condition,** because two or more tasks are racing to use the shared resource and the behavior of the program depends on which task arrives first. The importance of competition synchronization should now be clear.

One general method for providing mutually exclusive access (to support competition synchronization) to a shared resource is to consider the resource to be something that a task can possess and allow only a single task to possess it at a time. To gain possession of a shared resource, a task must request it. When a task is finished with a shared resource that it possesses, it must relinquish that resource so it can be made available to other tasks.

Three methods of providing for mutually exclusive access to a shared resource are semaphores, which are discussed in Section 13.3; monitors, which are discussed in Section 13.4; and message passing, which is discussed in Section 13.5.

Mechanisms for synchronization must be able to delay task execution. Synchronization imposes an order of execution on tasks that is enforced with these delays. To understand what happens to tasks through their lifetimes, we must consider how task execution is controlled. Regardless of whether a machine has a single processor or more than one, there is always the possibility of there being more tasks than there are processors. A program called a **scheduler** manages the sharing of processors among the tasks. If there were never any interruptions and tasks all had the same priority, the scheduler could simply give each task a time slice, such as 0.1 seconds, and when a task's turn came, the scheduler could let it execute on a processor for that amount of time. Of course, there are several events that complicate this: for example, delays for synchronization and waits for input or output operations.

Tasks can be in several different states, which are:

1. *New:* A task is in the new state when it has been created but has not yet begun its execution.
2. *Ready:* A ready task is ready to run but is not currently running. Either it has not been given processor time by the scheduler, or it had run previously but was blocked in one of the ways described in paragraph 4

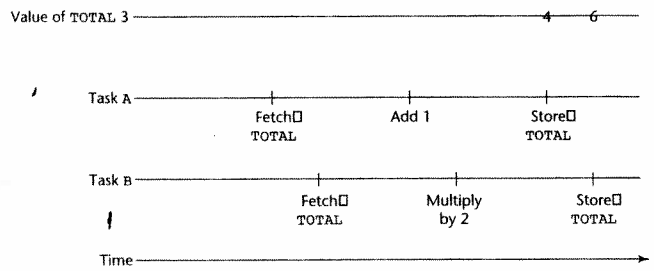


Figure 13.1
The need for competition synchronization

of this subsection. Tasks that are ready to run are stored in a queue that is often called the **task ready queue.**

3. *Running:* A running task is one that is currently executing; that is, it has a processor and its code is being executed.
4. *Blocked:* A task that is blocked has been running, but that execution was interrupted by one of several different events, the most common of which is an input or output operation. Because input and output operations are much slower than program execution, a task that starts an input or output operation is blocked from using the processor while it waits until the input or output operation is completed. In addition to these kinds of blocking, some languages provide operations for the user program to specify that a task be blocked.
5. *Dead:* A dead task is no longer active in any sense. A task dies when its execution is completed or it is explicitly killed by the program.

One important issue in task execution is the following: How is a ready task chosen to move to the running state when the task currently running has become blocked or whose time slice has expired? Several different algorithms have been used for this choice, some based on specifiable priority levels. The algorithm that does the choosing is implemented in the scheduler.

Associated with the concurrent execution of tasks and the use of shared resources is the concept of liveness. In the environment of sequential programs, a program has the characteristic of **liveness** if it continues to execute, eventually leading to completion. In more general terms, liveness means that if some event—say, program completion—is supposed to occur, it will occur, eventually. That is, progress is continually made. In the environment of concurrency and the use of shared resources, the liveness of a task can cease to exist, meaning that the program cannot continue and thus will never terminate.

For example, suppose task A and task B both need the shared resources X and Y to complete their work. Further suppose that task A gains possession of X and task B gains possession of Y. After some execution, task A needs resource Y to continue, so it requests Y but must wait until B releases it. Likewise, task B requests X but must wait until A releases it. Neither relinquishes the resource

it possesses, and as a result, both lose their liveness, guaranteeing that execution of the program will never complete normally. This particular kind of loss of liveness is called **deadlock.** Deadlock is a serious threat to the reliability of a program, and therefore its avoidance demands serious consideration in both language and program design.

We are now ready to discuss some of the linguistic mechanisms for providing concurrent unit control.

history note

PL/I was the first programming language to include concurrent tasks. It allowed user programs to execute any subprogram concurrently with the unit that called it. The mechanism for synchronization of these concurrent executions was, however, wholly inadequate. It consisted of only binary semaphores, which were called *events*, and the ability to detect when a task had completed its execution.

ALGOL 68, which allowed compound statement-level concurrency, included a semaphore data type named sema.

13.2.2 Language Design for Concurrency

A number of languages have been designed to support concurrency, beginning with PL/I in the middle 1960s and including the contemporary languages Ada 95, Java, C#, Python, and Ruby.

13.2.3 Design Issues

The most important design issues for language support for concurrency have already been discussed at length: competition and cooperation synchronization. In addition to these, there are several design issues of secondary importance. Prominent among them is how to control task scheduling. Also, there are the issues of how and when tasks start and end executions, and how and when they are created.

Keep in mind that our discussion of concurrency is intentionally incomplete, and only the most important of the language design issues related to support for concurrency are discussed.

The following sections discuss three alternative answers to the design issues for concurrency: semaphores, monitors, and message passing.

13.3 Semaphores

A semaphore is a simple mechanism that can be used to provide synchronization of tasks. In the following paragraphs, we describe semaphores and discuss how they can be used for this purpose.

13.3.1 Introduction

In an effort to provide competition synchronization through mutually exclusive access to shared data structures, Edsger Dijkstra devised semaphores in 1965 (Dijkstra, 1968b). Semaphores can also be used to provide cooperation synchronization.

A **semaphore** is a data structure consisting of an integer and a queue that stores task descriptors. A **task descriptor** is a data structure that stores all of the relevant information about the execution state of a task. The concept of a semaphore is that, to provide limited access to a data structure, guards are placed around the code that accesses the structure. A **guard** is a linguistic device that allows the guarded code to be executed only when a specified condition is true. A guard can be used to allow only one task to access a shared data structure at a time. A semaphore is an implementation of a guard. An integral part of a guard mechanism is a procedure for ensuring that all attempted executions of the guarded code eventually take place. The typical procedure is to have requests for access that occur when access cannot be granted be stored in the task descriptor queue, from which they are later allowed to leave and execute the guarded code. This is the reason a semaphore must have both a counter and a task descriptor queue.

The only two operations provided for semaphores were originally named P and V by Dijkstra, after the two Dutch words *passeren* (to pass) and *vrygeren* (to release) (Andrews and Schneider, 1983). We will refer to these as *wait* and *release* in the remainder of this section.

The process by which semaphores provide guards is described in terms of the most common applications in the following subsection.

13.3.2 Cooperation Synchronization

Through much of this chapter, we use the example of a shared buffer to illustrate the different approaches to providing cooperation and competition synchronization. For cooperation synchronization, such a buffer must have some way of recording both the number of empty positions and the number of filled positions in the buffer (to prevent buffer underflow and overflow). The counter component of a semaphore variable can be used for this purpose. One semaphore variable—for example, `emptyspots`—can be used to store the number of empty locations in a shared buffer, and another—say, `fullspots`—can be used to store the number of filled locations in the buffer. The task queues of these two semaphores store tasks that have been blocked by the delay operation of the semaphore.

Our example buffer is designed as an abstract data type in which all data enters the buffer through the subprogram `DEPOSIT`, and all data leaves the buffer through the subprogram `FETCH`. Then the `DEPOSIT` subprogram needs only to check with the `emptyspots` semaphore to see whether there are any empty positions. If there is at least one, it can go ahead with the `DEPOSIT`, which must include decrementing the counter of `emptyspots`. If the buffer is full, the caller to `DEPOSIT` must be made to wait in the `emptyspots` queue for an empty spot to become available. When the `DEPOSIT` is complete, the `DEPOSIT` subprogram increments the counter of the `fullspots` semaphore to indicate that there is one more filled location in the buffer.

The `FETCH` subprogram has the opposite sequence of `DEPOSIT`. It checks the `fullspots` semaphore to see whether the buffer contains at least one item. If it does, an item is removed and the `emptyspots` semaphore has its counter incremented by 1. If the buffer is empty, the calling process is put in the `fullspots` queue to wait until an item appears. When `FETCH` is finished, it must increment the counter of `emptyspots`.

The operations on semaphore types often are not direct—they are done through the wait and release subprograms. Therefore, the `DEPOSIT` operation just described is actually accomplished in part by calls to wait and release. Note that wait and release must be able to access the task-ready queue.

The wait subprogram is used to test the counter of a given semaphore variable. If the value is greater than zero, the caller can carry out its operation. In this case, the counter value of the semaphore variable is decremented to indicate that there are now one fewer of whatever it counts. If the value of the counter is zero, the caller must be placed on the waiting queue of the semaphore variable, and the processor must be given to some other ready task.

The release operation is used by a task to allow some other task to have one of whatever the counter of the specified semaphore variable counts. If the queue of the specified semaphore variable is empty, which means no task is waiting, release increments its counter (to indicate there is one more of whatever is being controlled that is now available). If one or more tasks are waiting, release moves one of them from the semaphore queue to the ready queue.

The following are concise pseudocode descriptions of wait and release:

```
wait(aSemaphore)
if aSemaphore's counter > 0 then
    decrement aSemaphore's counter
else
    put the caller in aSemaphore's queue
    attempt to transfer control to some ready task
    (if the task ready queue is empty, deadlock occurs)
end if

release(aSemaphore)
if aSemaphore's queue is empty (no task is waiting) then
    increment aSemaphore's counter
else
    put the calling task in the task-ready queue
    transfer control to a task from aSemaphore's queue
end
```

We can now present an example program that implements cooperation synchronization for a shared buffer. In this case, the shared buffer stores integer values and is a logically circular structure. It is designed for use by possibly multiple producer and consumer tasks.

The following pseudocode shows the definition of the producer and consumer tasks. Two semaphores are used to ensure against buffer underflow or overflow, thus providing cooperation synchronization. Assume that the buffer has length BUFLEN, and the routines that actually manipulate it already exist as FETCH and DEPOSIT. Accesses to the counter of a semaphore are specified by dot notation. For example, if `fullspots` is a semaphore, its counter is referenced by `fullspots.count`.

```
semaphore fullspots, emptyspots;
fullspots.count = 0;
emptyspots.count = BUFLEN;
task producer;
  loop
  -- produce VALUE --
  wait(emptyspots);     { wait for a space }
  DEPOSIT(VALUE);
  release(fullspots);   { increase filled spaces }
  end loop;
end producer;

task consumer;
  loop
  wait(fullspots);      { make sure it is not empty }
  FETCH(VALUE);
  release(emptyspots);  { increase empty spaces }
  -- consume VALUE --
  end loop
end consumer;
```

The semaphore `fullspots` causes the `consumer` task to be queued to wait for a buffer entry if it is currently empty. The semaphore `emptyspots` causes the `producer` task to be queued to wait for an empty space in the buffer if it is currently full.

13.3.3 Competition Synchronization

Our buffer example does not provide competition synchronization. Access to the structure can be controlled with an additional semaphore. This semaphore need not count anything, but can simply indicate with its counter whether the buffer is currently being used. The `wait` statement allows the access only if the semaphore's counter has the value 1, which indicates that the shared buffer is not currently being accessed. If the semaphore's counter has a value of 0, there is a current access taking place, and the task is placed on the queue of the semaphore. Notice that the semaphore's counter must be initialized to 1. The queues of semaphores must always be initialized to empty.

A semaphore that requires only a binary-valued counter, like the one used to provide competition synchronization in the following example, is called a **binary semaphore.**

The example pseudocode that follows illustrates the use of semaphores to provide both competition and cooperation synchronization for a concurrently accessed shared buffer. The `access` semaphore is used to ensure mutually exclusive access to the buffer. Note again that there may be more than one producer and more than one consumer.

```
semaphore access, fullspots, emptyspots;
access.count = 1;
fullspots.count = 0;
emptyspots.count = BUFLEN;

task producer;
  loop
  -- produce VALUE --
  wait(emptyspots);     { wait for a space }
  wait(access);         { wait for access }
  DEPOSIT(VALUE);
  release(access);      { relinquish access }
  release(fullspots);   { increase filled spaces }
  end loop;
end producer;

task consumer;
  loop
  wait(fullspots);      { make sure it is not empty }
  wait(access);         { wait for access }
  FETCH(VALUE);
  release(access);      { relinquish access }
  release(emptyspots);  { increase empty spaces }
  -- consume VALUE --
  end loop
end consumer;
```

A brief look at this example may lead one to believe there is a problem with it. Specifically, suppose that while a task is waiting at the `wait(access)` call in `consumer` another task takes the last value from the shared buffer. Fortunately, this cannot happen, because the `wait(fullspots)` reserves a value in the buffer for the task that calls it by decrementing the `fullspots` counter.

There is one crucial aspect of semaphores that thus far has not been discussed. Recall the earlier description of the problem of competition synchronization: Operations on shared data must not be overlapped. If a second operation can be begun while an earlier operation is still in progress, the shared data can become corrupted. A semaphore is itself a shared data object, so the operations on semaphores are also susceptible to the same problem. It is therefore essential that semaphore operations be uninterruptible. Many computers have uninterruptible instructions that were designed specifically for semaphore operations. If such instructions are not available, then using semaphores to provide competition synchronization is a serious problem with no simple solution.

13.3.4 Evaluation

Using semaphores to provide cooperation synchronization creates an unsafe programming environment. There is no way to statically check for the correctness of their use, which depends on the semantics of the program in which they appear. In the buffer example, leaving the `wait(emptyspots)` statement out of the `producer` task would result in buffer overflow. Leaving the `wait(fullspots)` statement out of the `consumer` task would result in buffer underflow. Leaving out either of the releases would result in deadlock. These are cooperation synchronization failures.

The reliability problems that semaphores cause in providing cooperation synchronization also arise when using them for competition synchronization. Leaving out the `wait(access)` statement in either task can cause insecure access to the buffer. Leaving out the `release(access)` statement in either task results in deadlock. These are competition synchronization failures. Noting the danger in using semaphores, Per Brinch Hansen wrote, "The semaphore is an elegant synchronization tool for an ideal programmer who never makes mistakes" (Brinch Hansen, 1973). Unfortunately, programmers of that kind are rare.

13.4 Monitors

One solution to some of the problems of semaphores in a concurrent environment is to encapsulate shared data structures with their operations and hide their representations—that is, to make shared data structures abstract data types. This solution can provide competition synchronization without semaphores by transferring responsibility for synchronization to the run-time system.

13.4.1 Introduction

When the concepts of data abstraction were being formulated, the people involved in that effort applied the same concepts to shared data in concurrent

programming environments to produce monitors. According to Per Brinch Hansen (Brinch Hansen, 1977, p. xvi), Edsger Dijkstra suggested in 1971 that all synchronization operations on shared data be gathered into a single program unit. Brinch Hansen formalized this concept in the environment of operating systems (Brinch Hansen, 1973). The following year, Hoare named these structures *monitors* (Hoare, 1974).

The first programming language to incorporate monitors was Concurrent Pascal (Brinch Hansen, 1975). Modula (Wirth, 1977), CSP/k (Holt et al., 1978), and Mesa (Mitchell et al., 1979) also provide monitors. Among contemporary languages, monitors are supported by Ada, Java, and C#, all of which are discussed in this chapter.

13.4.2 Competition Synchronization

One of the most important features of monitors is that shared data is resident in the monitor rather than in any of the client units. Thus the programmer does not synchronize mutually exclusive access to shared data through the use of semaphores or other mechanisms. Because all accesses are resident in the monitor, the monitor implementation can be made to guarantee synchronized access by allowing only one access at a time. Calls to monitor procedures are implicitly queued if the monitor is busy at the time of the call.

13.4.3 Cooperation Synchronization

Although mutually exclusive access to shared data is intrinsic with a monitor, cooperation between processes is still the task of the programmer. In particular, the programmer must guarantee that a shared buffer does not experience underflow or overflow. Different languages provide different ways of programming cooperation synchronization, all of which are related to semaphores.

A program containing four tasks and a monitor that provides synchronized access to a concurrently shared buffer is shown in Figure 13.2. In this figure, the interface to the monitor is shown as the two boxes labeled `insert` and `remove` (for the insertion and removal of data).

13.4.4 Evaluation

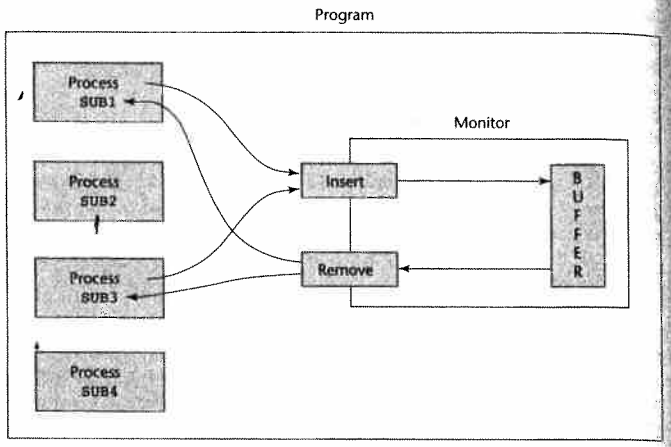
Monitors are a better way to provide competition synchronization than are semaphores, primarily because of the problems of semaphores, as discussed in Section 13.3. The cooperation synchronization is still a problem with monitors, as will be clear when Ada and Java implementations of monitors are discussed in the following sections.

Semaphores and monitors are equally powerful at expressing concurrency control—semaphores can be used to implement monitors and monitors can be used to implement semaphores.

Ada provides two ways to implement monitors. Ada 83 includes a general tasking model that can be used to support monitors. Ada 95 added a cleaner

Figure 13.2

A program using a monitor to control access to a shared buffer



and more efficient way of constructing monitors, called *protected objects*. Both of these approaches use message passing as a basic model for supporting concurrency. The message-passing model allows concurrent units to be distributed, which monitors do not allow. Message passing is described in Section 13.5; Ada support for message passing is discussed in Section 13.6.

13.5 Message Passing

This section introduces the fundamental concept of message passing. Section 13.6 describes the details of Ada support for the message-passing approach.

13.5.1 Introduction

The first efforts to design languages that provide the capability for message passing among concurrent tasks were those of Brinch Hansen (1978) and Hoare (1978). The pioneer developers of message passing also developed a technique for handling the problem of what to do when multiple simultaneous requests were made by other tasks to communicate with a given task. It was decided that some form of nondeterminism was required to provide fairness in choosing which among those requests would be taken first. This fairness can be defined in various ways, but in general it means that all requesters are provided an equal chance of communicating with a given task (assuming that every requester has the same priority). Nondeterministic constructs for statement-level control, called *guarded commands*, were introduced by Dijkstra (1975). (Guarded commands are discussed in Chapter 8.) Guarded commands are the basis of the construct designed for controlling message passing.

13.5.2 The Concept of Synchronous Message Passing

Message passing can be either synchronous or asynchronous. The asynchronous message passing of Ada 95 is described in Section 13.6.8. Here we describe synchronous message passing. The basic concept of synchronous message passing is that tasks are often busy, and when busy, cannot be interrupted by other units. Suppose task A and task B are both in execution, and A wishes to send a message to B. Clearly, if B is busy, it is not desirable to allow another task to interrupt it. That would disrupt B's current processing. Furthermore, messages usually cause associated processing in the receiver, which might not be sensible if other processing is incomplete. The alternative is to provide a linguistic mechanism that allows a task to specify to other tasks when it is ready to receive messages. This approach is somewhat like an executive who instructs his or her secretary to hold all incoming calls until another activity, perhaps an important conversation, is completed. Later, the executive tells the secretary that he or she is now willing to talk to one of the callers who has been placed on hold.

A task can be designed so that it can suspend its execution at some point, either because it is idle or because it needs information from another unit before it can continue. This is like a person who is waiting for an important call. In some cases, there is nothing else to do but sit and wait. In this situation, if task A wants to send a message to B, and B is willing to receive a message, the message can be transmitted. This actual transmission is called a **rendezvous.** Note that a rendezvous can occur only if both the sender and receiver want it to happen. The information of the message can be transmitted in either or both directions.

Both cooperation and competition synchronization of tasks can be conveniently handled with the message-passing model, as described in the following section.

13.6 Ada Support for Concurrency

This section describes the support for concurrency provided by Ada. Ada 83 supports only synchronous message passing; Ada 95 adds support for asynchronous message passing.

13.6.1 Fundamentals

The Ada design for tasks is partially based on the work of Brinch Hansen and Hoare in that message passing is the design basis and nondeterminism is used to choose among competing message-sending tasks.

that are more general than is needed. Protected objects are a better way to provide synchronized shared data.

In the absence of distributed processors with independent memories, the choice between monitors and tasks with message passing as a means of implementing shared data in a concurrent environment is somewhat a matter of taste. However, in the case of Ada, protected objects are clearly better than tasks for supporting concurrent access to shared data. Not only is the code simpler; it is also much more efficient.

For distributed systems, message passing is a better model for concurrency, because it naturally supports the concept of separate processes executing in parallel on separate processors.

13.7 Java Threads

The concurrent units in Java are methods named `run`, whose code can be in concurrent execution with other such methods (of other objects) and with the `main` method. The process in which the `run` methods execute is called a **thread.** Java's threads are lightweight tasks, which means that they all run in the same address space. This is different from Ada tasks, which are heavyweight threads (they run in their own address spaces). One important result of this difference is that threads require far less overhead than Ada's tasks.

There are two ways to define a class with a `run` method. One of these is to define a subclass of the predefined class `Thread` and override its `run` method. However, if the new subclass has a necessary natural parent, then defining it as a subclass of `Thread` obviously will not work. In these situations, a subclass that inherits from its natural parent and implements the `Runnable` interface is defined. `Runnable` provides the `run` method protocol. This approach still requires a `Thread` object, as will be seen in Section 13.7.4.

Java's threads can be used to implement monitors, as discussed in Section 13.7.3.

13.7.1 The `Thread` Class

The `Thread` class is not the natural parent of any other classes. It provides some services for its subclasses, but it is not related in any natural way to their computational purposes. Nevertheless, `Thread` is the only *class* available to the programmer for creating concurrent Java programs. As previously stated, Section 13.7.4 will briefly discuss the use of the `Runnable` interface.

The bare essentials of `Thread` are two methods named `run` and `start`. The `run` method is always overridden by subclasses of `Thread`. The code of the `run` method describes the actions of the thread. The `start` method of `Thread` starts its thread as a concurrent unit by calling its `run` method.[2] The call to

2. Calling the `run` method directly does not always work, because initialization that is sometimes required is included in the `start` method.

start is unusual in that control returns immediately to the caller, which then continues its execution, in parallel with the newly started run method.

Following is a skeletal subclass of Thread and a code fragment that creates an object of the subclass and starts the run method's execution in the new thread:

```
class MyThread extends Thread {
  public void run() { ... }
}
...
Thread myTh = new MyThread();
myTh.start();
```

When a Java application program (as opposed to an applet) begins execution, a new thread is created (in which the **main** method will run) and **main** is called. Applets also run in their own threads. Therefore, all Java programs run in threads.

When a program has multiple threads, a scheduler must determine which thread or threads will run at any given time. In most cases, there is only a single processor, so only one thread actually runs at a time. It is difficult to give a precise description of how the Java scheduler works, because the different implementations (Solaris, Windows, and so on) do not necessarily schedule threads in exactly the same way. Typically, however, the scheduler gives equal-size time slices to each ready thread in round-robin fashion, assuming all of these threads have the same priority. Section 13.7.2 describes how different priorities can be given to different threads.

The Thread class provides several methods for controlling the execution of threads. The yield method, which takes no parameters, is a request from the running thread to voluntarily surrender the processor. The thread is immediately put in the task-ready queue, making it ready to run. The scheduler then chooses the highest-priority thread from the task-ready queue. If there are no other ready threads with priority higher than the one that just yielded the processor, it may also be the next thread to get the processor.

The sleep method has a single parameter, which is the integer number of milliseconds that the caller of sleep wants the thread to be blocked. After the specified number of milliseconds has passed, the thread will be put in the task-ready queue. Because there is no way to know how long a thread will be in the task-ready queue before it runs, the parameter to sleep is the minimum amount of time the thread will *not* be in execution. The sleep method can throw an InterruptedException, which must be handled in the method that calls sleep. Exceptions are described in detail in Chapter 14.

The join method is used to force a method to delay its execution until the run method of another thread has completed its execution. join is used when the processing of a method cannot continue until the work of the other thread is complete. For example, we might have the following run method:


```
public void run() {
  ...
  Thread myTh = new Thread();
  myTh.start();
  // do part of the computation of this thread
  myTh.join();  // Wait for myTh to complete
  // do the rest of the computation of this thread
}
```

The join method puts the thread that calls it in the blocked state, which can be ended only by the completion of the thread on which join was called. If that thread happens to be blocked, there is the possibility of deadlock. To prevent this, join can be called with a parameter, which is the time limit in milliseconds of how long the calling thread will wait for the called thread to complete. For example,

```
myTh.join(2000);
```

will cause the calling thread to wait two seconds for myTh to complete. If it has not completed its execution after two seconds have passed, the calling thread is put back in the ready queue, which means that it will continue its execution as soon as it is scheduled.

Early versions of Java included three more Thread methods: stop, suspend, and resume. All three of these have been deprecated because of safety problems. The stop method is sometimes overridden with a simple method that destroys the thread by setting its reference variable to **null**.

The normal way a run method ends its execution is by reaching the end of its code. However, in many cases threads run until told to terminate. Regarding this, there is the question of how a thread can determine whether it should continue or end. The interrupt method is one way to communicate to a thread that it should stop. This method does not stop the thread; rather, it sends the thread a message that actually just sets a bit in the thread, which can be checked by the thread. The bit is checked with the predicate method, isInterrupted. This is not a complete solution, because the thread one is attempting to interrupt may be sleeping or waiting at the time the interrupt method is called, which means that it will not be checking to see if it has been interrupted. For these situations, the interrupt method also throws an exception, InterruptedException, which also causes the thread to awaken (from sleeping or waiting). So, a thread can periodically check to see whether it has been interrupted and if so, whether it can terminate. The thread cannot miss the interrupt, because if it was asleep or waiting when the interrupt occurred, it will be awakened by the interrupt. Actually, there are more details to the actions and uses of interrupt, but they are not covered here (Arnold et al., 2006).

13.7.2 Priorities

The priorities of threads need not all be the same. A thread's default priority is the same as the thread that created it. If `main` creates a thread, its default priority is the constant `NORM_PRIORITY`, which is usually 5. `Thread` defines two other priority constants, `MAX_PRIORITY` and `MIN_PRIORITY`, whose values are usually 10 and 1, respectively.[3] The priority of a thread can be changed with the method `setPriority`. The new priority can be any of the predefined constants or any other number between `MIN_PRIORITY` and `MAX_PRIORITY`. The `getPriority` method returns the current priority of a thread.

When there are threads with different priorities, the scheduler's behavior is controlled by those priorities. When the executing thread is blocked or killed or the time slice for it expires, the scheduler chooses the thread from the task-ready queue that has the highest priority. A thread with lower priority will run only if one of higher priority is not in the task-ready queue when the opportunity arises.

13.7.3 Competition Synchronization

In Java, competition synchronization is implemented by specifying that the methods that access shared data are run completely before another method is executed on the same object. In other words, we can specify that once a particular method begins its execution, that execution will be completed before any other method begins its execution on the same object. Such methods place a lock on the object, which prevents other synchronized methods from executing on the object. This is specified on a method by adding the **`synchronized`** modifier to the method's definition, as in the following skeletal class definition:

```
class ManageBuf {
  private int [100] buf;
  ...
  public synchronized void deposit(int item) { ... }
  public synchronized int fetch() { ... }
  ...
}
```

The two methods defined in `ManageBuf` are both defined to be **`synchronized`**, which prevents them from interfering with each other while executing on the same object, even if they are called by separate threads.

3. The number of priorities is implementation-dependent, so there may be fewer or more than 10 levels in some implementations.

An object whose methods are all synchronized is effectively a monitor. Note that an object may have one or more synchronized methods, as well as one or more unsynchronized methods.

In some cases, the number of statements that deal with the shared data structure is significantly less than the number of other statements in the method in which it resides. In these cases, it is better to synchronize the code segment that accesses or changes the shared data structure rather than the whole method. This can be done with a so-called *synchronized statement*, whose general form is

```
synchronized (expression)
  statement
```

where the expression must evaluate to an object and the statement can be a single statement or a compound statement. The object is locked during execution of the statement or compound statement, so the statement or compound statement is executed exactly as if it were the body of a synchronized method.

An object that has synchronized methods defined for it must have a queue associated with it that stores the synchronized methods that have attempted to execute on it while it was being operated upon by another synchronized method. When a synchronized method completes its execution on an object, a method that is waiting in the object's waiting queue, if there is such a method, is put in the task-ready queue.

13.7.4 Cooperation Synchronization

Cooperation synchronization in Java is accomplished by using the `wait`, `notify`, and `notifyAll` methods that are defined in `Object`, the root class of all Java classes. All classes except `Object` inherit these methods. Every object has a wait list of all of the threads that have called `wait` on the object. The `notify` method is called to tell one waiting thread that the event it was waiting for has happened. The specific thread that is awakened by `notify` cannot be determined, because the Java Virtual Machine (JVM) chooses one from the wait list of the thread object at random. Because of this, along with the fact that the waiting threads may all be waiting for different conditions, the `notifyAll` method is often used, rather than `notify`. The `notifyAll` method awakens all of the threads on the object's wait list, starting their execution just after their call to `wait`.

The methods `wait`, `notify`, and `notifyAll` can be called only from within a synchronized method, because they use the lock placed on an object by such a method. The call to `wait` is usually put in a **`while`** loop that is controlled by the condition for which the method is waiting. Because of the use of `notifyAll`, some other thread may have changed the condition to false since it was last tested.

The wait method can throw InterruptedException, which is a descendant of Exception (Java's exception handling is discussed in Chapter 14). Therefore, any code that calls wait must also catch InterruptedException. Assuming the condition for which we wait is called theCondition, the conventional way to use wait is as follows:

```
try {
  while (!theCondition)
    wait();
  -- Do whatever is needed after theCondition comes true
}
catch(InterruptedException myProblem) { ... }
```

The following program implements a circular queue for storing **int** values. It illustrates both cooperation and competition synchronization.

```
// Queue
// This class implements a circular queue for storing int
// values. It includes a constructor for allocating and
// initializing the queue to a specified size. It has
// synchronized methods for inserting values into and
// removing values from the queue.

class Queue {
  private int [] que;
  private int nextIn,
              nextOut,
              filled,
              queSize;

  public Queue(int size) {
    que = new int [size];
    filled = 0;
    nextIn = 1;
    nextOut = 1;
    queSize = size;
  } //** end of Queue constructor

  public synchronized void deposit (int item) {
    try {
      while (filled == queSize)
        wait();
      que [nextIn] = item;
      nextIn = (nextIn % queSize) + 1;
      filled++;
      notifyAll();
    } //** end of try clause
    catch(InterruptedException e) {}
  } //** end of deposit method

  public synchronized int fetch() {
    int item = 0;
    try {
      while (filled == 0)
        wait();
      item = que [nextOut];
      nextOut = (nextOut % queSize) + 1;
      filled--;
      notifyAll();
    } //** end of try clause
    catch(InterruptedException e) {}
    return item;
  } //** end of fetch method
} //** end of Queue class
```


Notice that the exception handler (**catch**) does nothing here.

Classes to define producer and consumer objects that could use the Queue class can be defined as follows:

```
class Producer extends Thread {
  private Queue buffer;
  public Producer(Queue que) {
    buffer = que;
  }
  public void run() {
    int new_item;
    while (true) {
      //-- Create a new_item
      buffer.deposit(new_item);
    }
  }
}

class Consumer extends Thread {
  private Queue buffer;
  public Consumer(Queue que) {
    buffer = que;
  }
  public void run() {
    int stored_item;
    while (true) {
      buffer.fetch(stored_item);
```

```
        //-- Consume the stored_item
      }
    }
  },
```

The following code creates a Queue object, and a Producer and a Consumer object, both attached to the Queue object, and starts their execution:

```
Queue buff1 = new Queue(100);
Producer producer1 = new Producer(buff1);
Consumer consumer1 = new Consumer(buff1);
producer1.start();
consumer1.start();
```

We could define one or both of the Producer and the Consumer as implementations of the Runnable interface rather than as subclasses of Thread. The only difference is in the first line, which would now appear as

```
class Producer implements Runnable {
```

To create and run an object of such a class, it is still necessary to create a Thread object that is connected to the object. This is illustrated in the following code:

```
Producer producer1 = new Producer(buff1);
Thread producerThread = new Thread(producer1);
producerThread.start();
```

13.7.5 Evaluation

Java's support for concurrency is relatively simple but effective. Because they are heavyweight threads, Ada's tasks easily can be distributed to different processors, in particular different processors with different memories, which could be on different computers in different places. These kinds of systems are not possible with Java's lightweight threads.

13.8 C# Threads

Although C#'s threads are loosely based on those of Java, there are significant differences. Following is a brief overview of C#'s threads.

13.8.1 Basic Thread Operations

Rather than just methods named run, as in Java, any C# method can run in its own thread. A C# thread is created by creating a Thread object. The Thread

constructor must be sent an instantiation of a predefined delegate class, ThreadStart,[4] to which must be sent the method that implements the actions of the thread. For example, we might have

```
public void MyRun1() { ... }
...
Thread myThread = new Thread(new ThreadStart(MyRun1));
```

As with Java, creating a thread does not start its concurrent execution. Once again, execution must be requested through a method, in this case named Start, as in

```
myThread.Start();
```

As in Java, a thread can be made to wait for another thread to finish its execution before continuing, using the similarly named method Join.

A thread can be suspended for a specified amount of time with Sleep, which is a public static method of Thread. The parameter to Sleep is an integer number of milliseconds. Unlike its Java relative, C#'s Sleep does not raise any exceptions, so it need not be called in a **try** block.

A thread can be terminated with the Abort method, although it does not literally kill the thread. Instead, it throws the ThreadAbortException, which the thread can catch. When the thread catches this exception, it usually deallocates any resources it allocated, and then ends (by getting to the end of its code).

13.8.2 Synchronizing Threads

There are three different ways that C# threads can be synchronized: the Interlock class, the lock statement, and the Monitor class. Each of these mechanisms is designed for a specific need. The Interlock class is used when the only operations that need to be synchronized are the incrementing and decrementing of an integer. These operations are done atomically with the two methods of Interlock, Increment and Decrement, which take a reference to an integer as the parameter. For example, to increment a shared integer named counter in a thread, we could use

```
Interlocked.Increment(ref counter);
```

The lock statement is used to mark a critical section of code in a thread. The syntax of this is as follows:

```
lock(expression) {
```

4. A C# delegate is an object-oriented version of a function pointer. In this case, it literally points to the method we want to run in the new thread.

```
    // The critical section
}
```

The expression, which looks like a parameter to `lock`, is usually a reference to the object on which the thread is running, **`this`**.

The `Monitor` class has four methods, `Enter`, `Wait`, `Pulse`, and `Exit`, that can be used to provide more sophisticated synchronization of threads. The `Enter` method, which takes an object reference as its parameter, marks the beginning of synchronization of the thread on that object. The `Wait` method suspends execution of the thread and instructs the Common Language Runtime (CLR) of .NET that this thread wants to resume its execution the next time there is an opportunity. The `Pulse` method, which also takes an object reference as its parameter, notifies waiting threads they now have a chance to run again. `Pulse` is similar to Java's `notifyAll`. Threads that have been waiting are run in the order in which they called the `Wait` method. The `Exit` method ends the critical section of the thread.

13.8.3 Evaluation

C#'s threads are a slight improvement over those of its predecessor, Java. For one thing, any method can be run in its own thread. Recall that in Java, only methods named run can run in their own threads. Thread termination is also cleaner with C# (calling a method (`Abort`) is more elegant than setting the thread's pointer to null). Synchronization of thread execution is more sophisticated in C#, because C# has several different mechanisms, each for a specific application. C# threads, like those of Java, are lightweight, so they cannot be as versatile as Ada's tasks.

13.9 Statement-Level Concurrency

In this section, we take a brief look at language design for statement-level concurrency. From the language design point of view, the objective of such designs is to provide a mechanism that the programmer can use to inform the compiler of ways it can map the program onto a multiprocessor architecture.[5]

In this section, we discuss only one collection of linguistic constructs from one language for statement-level concurrency. Furthermore, we will describe the constructs and their objectives in terms of SIMD architecture machines (see Section 13.1.1), although they were designed to be useful for a variety of architectural configurations.

The problem addressed by the language constructs we discuss is that of minimizing the communication required among processors and the memories

5. Although ALGOL 68 included a semaphore type that was meant to deal with statement-level concurrency, we do not discuss that application of semaphores here.

of other processors. The assumption is that it is faster for a processor to access data in its own memory than that of some other processor. Well-designed compilers can do a great deal in this process, but much more can be done if the programmer is able to provide information to the compiler about the possible concurrency that could be employed.

13.9.1 High-Performance Fortran

High-Performance Fortran (HPF; ACM, 1993b) is a collection of extensions to Fortran 90 that are meant to allow programmers to specify information to the compiler to help it optimize the execution of programs on multiprocessor computers. HPF includes both new specification statements and intrinsic, or built-in, subprograms. This section discusses only some of the new statements.

The primary specification statements of HPF are for specifying the number of processors, the distribution of data over the memories of those processors, and the alignment of data with other data in terms of memory placement. The HPF specification statements appear as special comments in a Fortran program. Each of them is introduced by the prefix `!HPF$`, where `!` is the character used to begin lines of comments in Fortran 90. This prefix makes them invisible to Fortran 90 compilers but easy for HPF compilers to recognize.

The `PROCESSORS` specification has the form

```
!HPF$ PROCESSORS procs (n)
```

This statement is used to specify to the compiler the number of processors that can be used by the code generated for this program. This information is used in conjunction with other specifications to tell the compiler how data is to be distributed to the memories associated with the processors.

The `DISTRIBUTE` statement specifies what data is to be distributed and the kind of distribution that is to be used. Its form is

```
!HPF$ DISTRIBUTE (kind) ONTO procs :: identifier_list
```

In this statement, kind can be either `BLOCK` or `CYCLIC`. The identifier list is the names of the array variables that are to be distributed. A variable that is specified to be `BLOCK` distributed is divided into n equal groups, where each group consists of contiguous collections of array elements evenly distributed over the memories of all the processors. For example, if an array with 500 elements named `LIST` is `BLOCK` distributed over five processors, the first 100 elements of `LIST` will be stored in the memory of the first processor, the second 100 in the memory of the second processor, and so forth. A `CYCLIC` distribution specifies that individual elements of the array are cyclically stored in the memories of the processors. For example, if `LIST` is `CYCLIC` distributed, again over five processors, the first element of `LIST` will be stored in the memory of the first processor, the second element in the memory of the second processor, and so forth.

The form of the ALIGN statement is

```
ALIGN array1_element WITH array2_element
```

ALIGN is used to relate the distribution of one array with that of another. For example,

```
ALIGN list1(index) WITH list2(index+1)
```

specifies that the index element of list1 is to be stored in the memory of the same processor as the index+1 element of list2, for all values of index. The two array references in an ALIGN appear together in some statement of the program. Putting them in the same memory (which means the same processor) ensures that the references to them will be as close as possible.

Consider the following example code segment:

```
      REAL list_1 (1000), list_2 (1000)
      INTEGER list_3 (500), list_4 (501)
!HPF$ PROCESSORS proc (10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: list_1, list_2
!HPF$ ALIGN list_3 (index) WITH list_4 (index+1)
      ...
      list_1 (index) = list_2 (index)
      list_3 (index) = list_4 (index+1)
```

In each execution of these assignment statements, the two referenced array elements will be stored in the memory of the same processor.

The HPF specification statements actually only provide information for the compiler that it may or may not use to optimize the code it produces. What the compiler actually does depends on its level of sophistication and the particular architecture of the target machine.

The FORALL statement specifies a collection of statements that may be executed concurrently. For example,

```
FORALL (index = 1:1000) list_1 (index) = list_2 (index)
```

specifies the assignment of the elements of list_2 to the corresponding elements of list_1. Conceptually, it specifies that the right side of all 1000 assignments can be evaluated first, before any assignments take place. This permits concurrent execution of all of the assignment statements. The HPF FORALL statement is included in Fortran 95 and Fortran 2003.

We have briefly discussed only a part of the capabilities of HPF. However, it should be enough to provide the reader with an idea of the kinds of language extensions that are useful for programming computers with possibly large numbers of processors.