Concurrent Programming

Architectural advances of recent years, coupled with the growing availability of networked computers, have led to a new style of computing, called **concurrent programming**, that allows multiple computations to occur simultaneously in cooperation with each other. Many people distinguish two classes of concurrent programming: **Distributed programming** refers to computations that do not share a common memory, and **parallel programming** refers to computations that share a common memory. This distinction is not always helpful, since it is possible to implement a distributed computation on a shared-memory computer, and to implement a parallel computation on a distributed-memory computer. It is up to the compiler and operating system to implement on the underlying architecture whatever concurrency style the programming language promotes. Terminology is less standard in the area of concurrent programming than elsewhere, so I will be somewhat arbitrary, but consistent, in my nomenclature.

A **thread** is a sequential computation that may interact with other simultaneous computations. A program that depends on a particular thread reaching some point in computation before another thread continues must make that dependency explicit; it is erroneous to assume anything about relative speeds of execution. The reason for this rule is that the language does not usually have much control over execution speeds. Individual threads may be implemented by time-sharing a single CPU, and the scheduler may be outside the control of the language (in the operating system). If threads are on different CPUs, the CPUs may have different speeds or may have other work that renders them slow in an unpredictable way. The communication expense for cooperation among threads may be unpredictable. Threads might be dynamically migrated from one CPU to another to improve performance, but with a temporary delay.

The connection between programming languages and operating systems is especially close in the area of concurrent programming. First, threads are sometimes supported by the underlying operating system, so the language implementation needs to make use of those facilities, and the language designer may choose to present or to omit features, depending on the operating system and what it can do. For example, a thread can be modeled by a Unix process. Generally, Unix processes cannot share memory. However, some versions of Unix, such as Solaris, offer threads within a single address space; these threads do share memory. Second, operating systems themselves are often multithreaded; the language design issues in this chapter are often identical with operating-system design issues.

1 ◆ STARTING MULTIPLE THREADS

Syntax for starting multiple computations tends to be straightforward. In Modula (I mean original Modula [Wirth 77], not Modula-2), a thread is started by invoking a procedurelike object; when the "procedure" returns, the thread disappears. Meanwhile, the computation that started the thread continues executing. The new thread may itself create other threads by invoking them. Figure 7.1 shows a program that merge-sorts an array by recursively creating threads.

Figure 7.1

```
type                                                        1
     DataArray = array whatever of integer;                 2

thread MergeSort(                                           3
     reference Tangled : DataArray;                         4
     value LowIndex, HighIndex : integer);                  5
variable                                                    6
     MidPoint : integer := (LowIndex + HighIndex) div 2;    7
begin                                                       8
     if LowIndex + 1 < HighIndex then -- worth sorting      9
          MergeSort(Tangled, LowIndex, MidPoint);           10
          MergeSort(Tangled, MidPoint+1, HighIndex);        11
          Merge(Tangled, 1, MidPoint, MidPoint+1,           12
               HighIndex);                                  13
     end; -- worth sorting                                  14
end; -- MergeSort                                           15
```

MergeSort is declared in line 3 as a **thread**, not a **procedure**. All invocations of MergeSort, including the recursive ones on lines 10 and 11, create new threads running instances of MergeSort that work independently of the main program. Unfortunately, MergeSort fails to wait for its children to finish and rushes ahead to line 12, merging the two halves of the array before they are properly sorted. You will soon see mechanisms for synchronization that will let me fix this bug.

Each thread gets its own stack. Variables declared locally to a thread are like local variables in a procedure; each thread gets its own local variables. Likewise, any procedures called from a thread (such as Merge, called in line 12) get new activation records on the thread's stack. However, variables that are outside the scope of the thread are shared among all threads in which they are visible by normal scope rules. That is, the static chain in a thread's stack eventually points outside of the private stack of the thread into shared stack. (Sometimes this arrangement is called a **cactus stack**, since the stacks resemble the branches on a saguaro or cholla cactus.)

Some languages let threads be started by a **cobegin** statement. All the statements within the **cobegin** are started as separate threads. This construct includes an implicit synchronization step: The **cobegin** does not complete until each of its children has completed. I could fix the MergeSort program by surrounding lines 10 and 11 with **cobegin** and making MergeSort an ordinary procedure.

Some languages, like Modula-3, present a fairly low-level view of threads. A thread is started by a call to a fork procedure, which returns a thread identifier that can be used later for synchronization. Fork takes a procedure parameter that tells it what the thread should do.[1] Usually, programming languages restrict the parameter to fork to be a global procedure, so that cactus stacks are not needed.

Other languages, like Ada, present a much higher-level view of threads. Each thread runs in a module, exporting procedures that may be called by other threads and importing types, procedures, and shared variables. If a block contains a thread declaration, the thread is started when its declaration is elaborated. The block does not complete until all threads started in it have finished.

2 ◆ COOPERATION BY MEANS OF SHARED VARIABLES

The MergeSort example shows that threads sometimes need to wait for each other. We say that a waiting thread is **blocked**. Generally, there are two reasons why threads need to block. First, they may be using variables that are shared with other threads, and they need to take turns. Taking turns is often called **mutual exclusion**, because while one thread is executing instructions that deal with the shared variables, all other threads must be excluded from such instructions. Second, they may need to wait for some operation to complete in some other thread before they may reasonably do their own work. We can explain the MergeSort example by either reason. First, the variables in the Tangled array are shared between parents and children. Second, it makes no sense to merge the two halves of Tangled until they have been sorted.

2.1 Join

The simplest form of synchronization is to block until another thread completes. Such blocking is achieved by the **join** statement, which specifies which thread is to be awaited. The thread that invokes **join** is blocked until that thread has completed. Some languages, such as Modula-3, make **join** an expression that evaluates to the value returned by the thread at the time it terminates. **Cobegin** implicitly invokes **join** at the end of the compound statement for each thread started by that statement.

[1] In Modula-3, the parameter is an object of a particular class that provides a method called apply.

2.2 Semaphores

The heart of most synchronization methods is the **semaphore**. Its implementation (often hidden from the programmer) is shown in Figure 7.2.

Figure 7.2

```
type
     Semaphore =
          record -- fields initialized as shown
               Value : integer := 1;
               Waiters : queue of thread := empty;
          end;
```

Semaphores have two operations, which are invoked by statements. (The operations can be presented as procedure calls instead.) I call the first **down** (sometimes people call it **P**, **wait**, or **acquire**). The second operation is **up** (also called **V**, **signal**, or **release**):

- **down** S decrements S.Value (line 4). It then blocks the caller, saving its identity in S.Waiters, if Value is now negative.
- **up** S increments S.Value. It unblocks the first waiting thread in S.Waiters if Value is now nonpositive.

Both these operations are **indivisible**, that is, they complete in a thread instantaneously so far as other threads are concerned. Therefore, only one thread at a time can either **up** or **down** a particular semaphore at a time.

Semaphores can be used to implement mutual exclusion. All regions that use the same shared variables are associated with a particular semaphore, initialized with Value = 1. A thread that wishes to enter a region **down**s the associated semaphore. It has now achieved mutual exclusion by acquiring an exclusive lock. When the thread exits the region, it **up**s the same semaphore, releasing the lock. The first thread to try to enter its region succeeds. Another thread that tries to enter while the first is still in its region will be blocked. When the first thread leaves the region, the second thread is unblocked. Value is always either 0 or 1 (if only two threads are competing). For this reason, semaphores used for mutual exclusion are often called **binary semaphores**.

Besides mutual exclusion, semaphores can also help achieve more complex synchronization. If thread T needs to wait until thread S accomplishes some goal, they can share a semaphore initialized with Value = 0. When S accomplishes its goal, it **up**s the semaphore. When T reaches the place where it must wait, it **down**s the same semaphore. No matter which one reaches the semaphore call first, T will not proceed until S has accomplished its goal.

2.3 Mutexes

Some languages, such as Modula-3, predeclare a **mutex** type that is implemented by binary semaphores. The **lock** statement surrounds any statements that must exclude other threads, as shown in Figure 7.3.

Figure 7.3

```
variable                                                  1
     A, B : integer;                                      2
     AMutex, BMutex : mutex;                              3

procedure Modify();                                       4
begin                                                     5
     lock AMutex do                                       6
          A := A + 1;                                     7
     end;                                                 8
     lock BMutex do                                       9
          B := B + 1;                                    10
     end;                                                11
     lock AMutex, BMutex do                              12
          A := A + B;                                    13
          B := A;                                        14
     end;                                                15
end; -- Modify                                           16
```

Variable A is protected by mutex AMutex, and B is protected by BMutex. The **lock** statement (as in line 6) is equivalent to a **down** operation at the start and an **up** operation at the end. Several threads may simultaneously execute in Modify. However, a thread executing line 7 prevents any other thread from executing any of lines 7, 13, and 14. It is possible for one thread to be at line 7 and another at line 10. I lock lines 7 and 10 because on many machines, incrementing requires several instructions, and if two threads execute those instructions at about the same time, the variable might get incremented only once instead of twice. I lock lines 13 and 14 together to make sure that no thread can intervene after line 13 and before line 14 to modify A. The multiple lock in line 12 first locks AMutex, then BMutex. The order is important to prevent deadlocks, as I will describe later.

2.4 Conditional Critical Regions

The Edison language has a way to program synchronization that is more expressive than mutexes but less error-prone than bare semaphores [Brinch Hansen 80]. As I mentioned before, synchronization in general is the desire to block an action until a particular condition becomes true.

A standard example that displays the need for synchronization is the **bounded buffer**, which is an array that is filled by producer threads and emptied by consumer threads. All producers and consumers must mutually exclude each other while they are inspecting and modifying the variables that make up the bounded buffer. In addition, when the buffer is full, producers should block instead of **busy waiting**, which is repeatedly testing to see if the buffer has room. Likewise, when the buffer is empty, consumers should block. Figure 7.4 shows how to code this application with conditional critical regions.

Figure 7.4

```
constant                                                1
     Size = 10; -- capacity of the buffer               2
type                                                    3
     Datum = ... -- contents of the buffer              4
variable                                                5
     Buffer : array 0..Size-1 of Datum;                 6
     InCount, OutCount : integer := 0;                  7

procedure PutBuffer(value What : Datum);                8
begin                                                   9
     region Buffer, InCount, OutCount                   10
     await InCount - OutCount < Size do                 11
          Buffer[InCount mod Size] := What;             12
          InCount := InCount + 1;                       13
     end; -- region                                     14
end -- PutBuffer;                                       15

procedure GetBuffer(result Answer : Datum);             16
begin                                                   17
     region Buffer, InCount, OutCount                   18
     await InCount - OutCount > 0 do                    19
          Answer := Buffer[OutCount mod Size];          20
          OutCount := OutCount + 1;                     21
     end; -- region                                     22
end GetBuffer;                                          23
```

The **region** statements starting in lines 10 and 18 are like **lock** statements, except that they name variables to be protected, not mutexes, and they have an **await** component. The compiler can check that shared variables are only accessed within **region** statements, and it can invent appropriate mutexes. The awaited condition is checked while the corresponding mutexes are held. If the condition is false, the mutexes are released and the thread is blocked (on an implicit semaphore). Whenever a thread exits from a region, all threads in conflicting regions (those that use some of the same shared variables) that are blocked for conditions are unblocked, regain their mutexes, and test their conditions again. This repeated rechecking of conditions can be a major performance problem.

2.5 Monitors

One objection to using conditional critical regions is the cost of checking conditions, which must occur whenever a thread leaves a region. A second objection is that code that modifies shared data may be scattered throughout a program. The **monitor** construct, found in Modula and Mesa, was invented to address both issues [Hoare 74; Lampson 80]. It acts both as a data-abstraction device (providing modularity) and a synchronization device.

Monitors introduce a new name scope that contains shared data and the procedures that are allowed to access the data. Procedures exported from the monitor are mutually exclusive; that is, only one thread may execute an exported procedure from a particular monitor at a time.

The most straightforward use of monitors is to package all routines that use a set of shared data (represented by a collection of variables) into a single monitor. All accesses to those variables will be forced to use exported procedures, because the variables themselves are hidden from the outside world. For example, I can implement a shared counter that records the number of times some interesting event has happened, as in Figure 7.5.

Figure 7.5

```
monitor Counter;
     export RaiseCount, ReadCount;
     variable Count : integer := 0;

     procedure RaiseCount();
     begin
          Count := Count + 1;
     end; -- RaiseCount;

     procedure ReadCount() : integer;
     begin
          return Count
     end; -- ReadCount;

end; -- Counter
```

One way to picture the monitor is shown in Figure 7.5, which shows the monitor as a floor plan of a building. When a thread tries to invoke an exported procedure, it enters through the entrance queue, where it is blocked until the exported procedure is free of any thread. Door 1 is unlocked only if there is no thread in the main room. Door 2 is always unlocked; when it is opened to let a thread out, door 1 is unlocked.

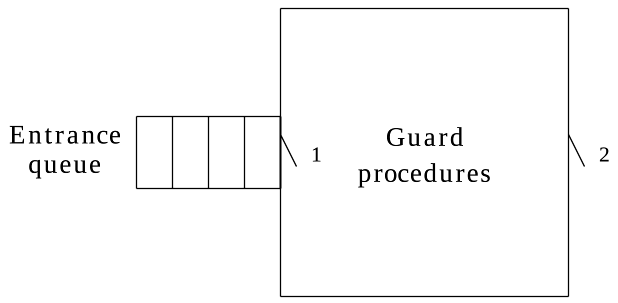

Figure 7.6 A simple monitor

It is not hard to implement this kind of monitor using only binary semaphores. Since the programmer does not need to remember the **up** and **down** operations, monitors are easier and safer to use than bare semaphores. In addition, all the code that can affect shared variables is packaged in one place, so it is easier to check that the variables are properly used.

In order to program a bounded buffer, I also need a way to have threads wait if conditions are not right. Instead of Boolean expressions, monitors introduce the predeclared `condition` data type. The operations on conditions are **wait**, **signal**, and **broadcast**. Using condition variables is more clumsy than programming Boolean expressions in conditional critical regions, because the programmer must remember which variable is associated with each situation and must also remember to signal the conditions when the time is right. A bounded buffer can be programmed as in Figure 7.7.

Figure 7.7

```
monitor BoundedBuffer;                                         1

     export GetBuffer, PutBuffer;                              2

     constant                                                  3
          Size = 10; -- capacity of the buffer                 4
     type                                                      5
          Datum = ... -- contents of the buffer                6
     variable                                                  7
          Buffer : array 0:Size-1 of Datum;                    8
          InCount, OutCount : integer := 0;                    9
          NotEmpty, NotFull : condition;                      10

     procedure PutBuffer(value What : Datum);                 11
     begin                                                    12
          if InCount - OutCount = Size then                   13
               wait NotFull;                                  14
          end;                                                15
          Buffer[InCount mod Size] := What;                   16
          InCount := InCount + 1;                             17
          signal NotEmpty ;                                   18
     end; -- PutBuffer;                                       19

     procedure GetBuffer(result Answer : Datum);              20
     begin                                                    21
          if InCount - OutCount = 0 then                      22
               wait NotEmpty;                                 23
          end;                                                24
          Answer := Buffer[OutCount mod Size];                25
          OutCount := OutCount + 1;                           26
          signal NotFull ;                                    27
     end; -- GetBuffer;                                       28
end; -- BoundedBuffer;                                        29
```

The situations in which the buffer is not full or not empty are indicated by the condition variables `NotFull` and `NotEmpty`. Consumers call `GetBuffer` (line 20), which checks to see if the buffer is empty. If so, it waits on `NotEmpty` (line 23). This operation releases mutual exclusion and blocks the thread. It will remain blocked until some other thread **signal**s `NotEmpty`. Producers do exactly that in line 18. **Signal** has no effect on a condition for which no thread is waiting, unlike **up** on a semaphore. The consumer can be sure when it arrives at line 25 that the buffer is not empty, because either it was not empty when the consumer called `GetBuffer`, and this routine excludes any other threads from the monitor, or it was empty, but some producer has signaled `NotEmpty`, and the consumer has been awakened and regained exclusion.

This discussion raises some troubling questions. Exactly when does the blocked consumer continue? If immediately, then there may be two threads in the monitor at once, and mutual exclusion is ruined. If later, then by the time the consumer continues, some other consumer may already have taken the last datum, and the assumption on line 25 that the buffer is not empty is wrong. If several consumers are waiting at the same time, which one or ones are unblocked by a signal?

The definitions of monitors in the literature disagree on the answers to these questions.

Figure 7.8 Monitors with conditions

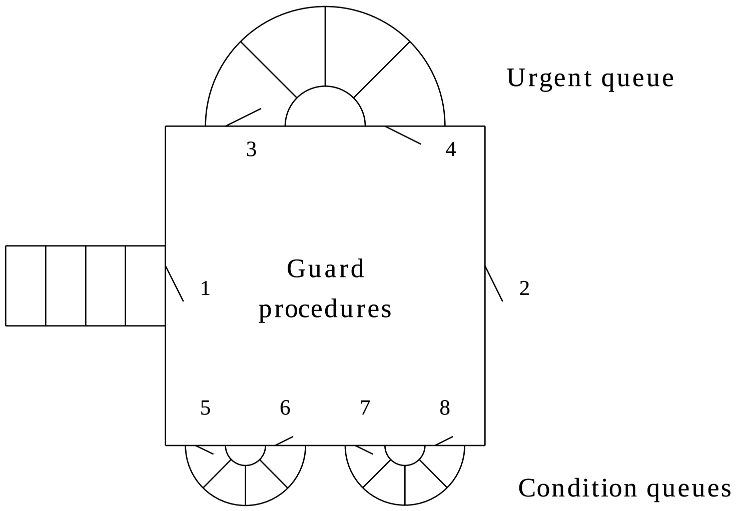

Figure 7.8 expands Figure 7.6 (page 193) to show the effect of conditions. Every condition has a condition queue (shown on the bottom of the monitor), and there is one urgent queue (shown at the top of the monitor). All queues are ordered first in, first out. Threads that are blocked are placed in these queues according to the following rules:

1. New threads wait in the entrance queue. A new thread may enter through door 1 if no thread is currently in the central region.
2. If a thread leaves the central region through door 2 (the exit), one thread is allowed in through door 4 (from the urgent queue) if there is one waiting there. If not, one thread is allowed through door 1 (from the entrance queue) if there is one waiting there.
3. A thread that executes **`wait`** enters the door to the appropriate condition queue (for example, 5 or 7).
4. When a thread executes **`signal`**, the signaled condition queue is inspected. If some thread is waiting in that queue, the signaler enters the urgent queue (door 3), and one waiter is allowed into the central region (door 6 or 8). If no thread is waiting in that queue, the signaler proceeds without leaving the central region. The signal is ignored.

These rules assure that a waiting consumer is unblocked immediately when a producer signals `NotEmpty` and that the producer is blocked in the urgent queue until the consumer has taken the datum.

Programmers have noticed that **`signal`** is almost always the last operation performed in an exported procedure. You can see this behavior in my producer-consumer code (lines 18 and 27). The rules will often make the signaler wait in the urgent queue and then return to the central region (acquiring exclusion) just to get out of the monitor altogether (releasing exclusion). These extra waits ands locks are inefficient. If **signal** is not the last operation, the signaler can't assume that the situation of shared variables is unchanged across **signal**. While it was in the urgent queue, the thread that was unblocked is likely to have modified the variables. The result is that **signal** is error-prone. For these reasons, some languages require that **signal** must be the last operation of an exported procedure and must cause the signaling thread to leave the monitor. Then the implementation doesn't need an urgent queue, and signalers never make invalid assumptions about shared data. However, this restriction makes monitors less expressive. (Such monitors are strictly less powerful in a theoretical sense.)

A related suggestion is to use **broadcast** instead of **signal**. **Broadcast** releases all the members of the given condition queue. Since they can't all be allowed into the central region at once, most are placed in the urgent queue. A released thread can no longer assume that the condition it has awaited is still met by the time it resumes. Programs written with **broadcast** usually replace the **if** statement in lines 13 and 22 of Figure 7.7 with a **while** loop to retest the condition after **wait** returns.

Proper use of monitors follows the guideline that no thread should take too long in the central region. It shouldn't take too long for a thread that is waiting in the entrance queue to get into the monitor and access the shared variables. Any lengthy operation should relax exclusion by entering a condition queue or by doing its work outside the monitor. A fascinating violation of this guideline arises if the thread in an exported procedure makes a call on a procedure exported by another monitor. Under the rules, this thread is still considered to be in the first monitor, preventing any other thread from entering. However, it may take a long time before it returns because it may be forced to wait in the second monitor in a condition queue. (By the guideline, it shouldn't have to wait very long in either the entrance queue or the urgent queue.) This delay in returning violates the guideline with respect to the first monitor. The situation can even lead to deadlock if the condition it awaits in the second monitor can be signaled only by a thread that is currently waiting patiently to enter the first monitor.

Several solutions have been proposed to this **nested-monitor** problem [Haddon 77]:

1. Disallow nested monitor calls.
2. Warn the programmer, but allow the bad situation to develop. That is, nested monitor calls maintain exclusion on the old monitor while in the new one.
3. Release exclusion on the old monitor and enforce it only on the new one. When the thread is ready to return, it must wait in the urgent queue of the first monitor until it can once again achieve exclusive use of the central region.
4. Let the programmer decide whether the nested call should maintain exclusion in the old monitor or not. By default, method 2 is used. The programmer can say **duckout** to release exclusion while still in the monitor and **duckin** to achieve exclusion again. These calls can bracket a nested call to simulate method 3.

Although monitors represent an important advance over raw semaphores, they do have significant problems. Monitors have been criticized for not providing any control over how the queues are ordered. The policy of treating queues in first-in, first-out order is not always appropriate. For example, several threads simultaneously in the urgent queue are like nested interrupts, which are usually released in first-in, last-out (stack) order. Similarly, different waiters in a condition queue may have different priorities, which could be taken into account in selecting an order. Some people prefer a more general mechanism for inspecting and reordering the various queues.

Monitors also display unexpected complexity with respect to nested calls. It is not easy to describe the semantics of **wait** and **signal** without resorting to pictures like Figure 7.8. Complexity is also introduced by the artificial use of condition variables. The programmer is more likely to understand the underlying condition (like InCount - OutCount > 0) than to represent that condition properly by judicious use of NotEmpty, including **signal** at the appropriate places.

Another objection to monitors comes from their data-abstraction ability. If I have several bounded buffers to implement, I would be tempted to build only one monitor and to have the PutBuffer and GetBuffer procedures take a parameter that describes which buffer is to be manipulated. This solution has two drawbacks. One is that the buffer has an existence outside the monitor and so might be inadvertently modified by a nonmonitor procedure. Ada addresses this limitation by providing for variables to be exported from modules in an opaque fashion, so that they cannot be manipulated outside the module. The other drawback is that using only one monitor is too conservative. Every manipulation of one buffer now excludes operations on all other buffers, because mutual exclusion is governed by which monitor is entered, not by which data structure is accessed. What we want is a monitor class that can be instantiated once for each separate buffer.

Mesa addresses the problem of overconservatism in two different ways. A monitor instance can be constructed dynamically for each buffer. However, there is a large space and time penalty for building monitor instances. Instead, the programmer may place the data (the buffer) in a monitored record, which is passed as a parameter to every exported procedure. The monitor declaration indicates that it uses a mutex in that record instead of its own mutex for mutual exclusion among threads executing exported procedures. For example, the bounded buffer in Mesa might be programmed as shown in Figure 7.9.

Figure 7.9

```
constant                                          1
     Size = 10; -- capacity of the buffer         2
type                                              3
     Datum = ... -- contents of a buffer          4
     BufferType :                                 5
          monitored record                        6
               Buffer : array 0:Size-1 of Datum;  7
               InCount, OutCount : integer := 0;  8
               NotEmpty, NotFull : condition;     9
          end;                                    10
     BufferPtrType : pointer to BufferType;       11
```

```
monitor BoundedBuffer;                                          12
     locks BufferPtr^ using BufferPtr : BufferPtrType;          13
     export GetBuffer, PutBuffer;                               14

     procedure PutBuffer(                                       15
          value What : Datum;                                   16
          value BufferPtr : BufferPtrType);                     17
     begin                                                      18
          if BufferPtr^.InCount-BufferPtr^.OutCount = Size      19
          then                                                  20
               wait BufferPtr^.NotFull;                         21
          end;                                                  22
          BufferPtr^.Buffer[BufferPtr^.InCount mod Size]        23
               := What;                                         24
          BufferPtr^.InCount := BufferPtr^.InCount + 1;         25
          signal BufferPtr^.NotEmpty ;                          26
     end; -- PutBuffer;                                         27

     procedure GetBuffer                                        28
          (result Answer : Datum;                               29
          value BufferPtr : BufferPtrType);                     30
     begin                                                      31
          if BufferPtr^.InCount - BufferPtr^.OutCount = 0       32
          then                                                  33
               wait BufferPtr^.NotEmpty;                        34
          end;                                                  35
          Answer := BufferPtr^.Buffer                           36
               [BufferPtr^.OutCount mod Size];                  37
          BufferPtr^.OutCount := BufferPtr^.OutCount + 1;       38
          signal BufferPtr^.NotFull ;                           39
     end; -- GetBuffer;                                         40
end; -- BoundedBuffer;                                          41
```

The buffer type (lines 5–10) implicitly contains a mutex. BoundedBuffer is written as a monitor, which means it implicitly acquires and releases that mutex on entrance and exit from exported procedures and when waiting for conditions. The monitor specifies that it locks BufferPtr^, which must be a parameter to every exported procedure. Unfortunately, if an exported procedure modifies its parameter BufferPtr, chaos can ensue, since the wrong mutex will then be accessed.

Modula-3 goes farther in solving the problem of overconservatism in monitors. It gives up on monitors entirely, providing only the building blocks out of which the programmer can build the necessary structures. That is, Modula-3 has conditions and mutexes as ordinary data types. The **wait** statement specifies both a condition and a mutex. As the thread begins to wait, it releases the mutex. When the thread is awakened by either **signal** or **broadcast**, it regains the mutex. Exported procedures and condition variables may be packaged into modules (to make monitors), data structures (so that each bounded buffer is independently exclusive), or classes (to allow monitors to be instantiated any number of times).

A serious objection to monitors is related to the guideline that exclusion should not be in force for very long. The problem is that shared data might be needed for a very long time. This is exactly the situation in the **readers-**

writers problem, in which some threads (the readers) need to read shared data, and others (the writers) need to write those data. Writers must exclude readers and all other writers. Readers must exclude only writers. Reading and writing are time-consuming operations, but they always finish eventually. If we export Read and Write from the monitor, two readers cannot execute at the same time, which is too restrictive. Therefore, Read must not be a monitor procedure; it must be external to the monitor. Proper use of Read would call the exported procedures StartRead and EndRead around calls to Read, but there is no assurance that a programmer will follow these rules. Monitors can therefore fail to protect shared data adequately.

2.6 Crowd Monitors

Crowd monitors are a nice extension to monitors that address this last problem [Horn 77]. Crowd monitors distinguish exclusive procedures from ordinary procedures within the monitor. Only exclusive procedures are mutually exclusive. Ordinary procedures may be invoked only by activities that have permission to do so; this permission is dynamically granted and revoked by exclusive procedures. A skeleton of the crowd-monitor solution to the readers-writers problem appears in Figure 7.10.

Figure 7.10

```
crowd monitor ReadWrite;                                 1

          export StartRead, EndRead, Read, StartWrite,       2
                    EndWrite, Write;                         3

          variable                                           4
                    Readers : crowd Read;                    5
                    Writers : crowd Read, Write;             6

          exclusive procedure StartRead();                   7
                    ... -- block the caller until reading is safe   8
                    enter Readers;                           9
                    ...                                      10

          exclusive procedure EndRead();                     11
                    ...                                      12
                    leave Readers;                           13
                    ... -- bookkeeping, maybe signal a waiting writer   14

          exclusive procedure StartWrite();                  15
                    ... -- block the caller until writing is safe   16
                    enter Writers;                           17
                    ...                                      18

          exclusive procedure EndWrite();                    19
                    ...                                      20
                    leave Writers;                           21
                    ... -- bookkeeping, maybe signal waiter  22

          procedure Read(...);                               23
                    ... -- actually read from the shared data   24
```

```
    procedure Write(...);                                 25
        ... -- actually modify the shared data            26

end; -- ReadWrite                                         27
```

In lines 5 and 6, I declare two crowds called Readers and Writers. Threads can dynamically enter and leave these crowds. Any member of Readers may access the Read procedure (lines 23–24), and any member of Writers may access both the Read and the Write procedure (lines 25–26). Threads initially belong to no crowds. The exclusive procedures decide when it is appropriate for a thread to enter or leave a crowd. They may use conditions to wait for the right situation. When the exclusive procedure decides to let a reader proceed, it executes **enter** for the Readers crowd (line 9). Similarly, a guard can let a writer enter the Writers crowd (line 17). Although any thread may call Read and Write, because they are exported from the monitor, a runtime check prevents threads from calling them if the threads are not in appropriate crowds. A member only of Readers may not call Write, but, a member of Writers may call either Read or Write, since both are specified in the definition of Writers (line 6).

2.7 Event Counts and Sequencers

Mutual exclusion is not always desirable because it limits concurrency. It is also unnecessary in some cases on physically distributed computers. In fact, if one hasn't yet implemented mutual exclusion, the method discussed here can be used to build semaphores to provide mutual exclusion, too [Reed 79]. Those semaphores will even allow simultaneous **down** operations on several semaphores.

The first type needed is the **event count**. An event count is implemented as a nondecreasing integer variable. It keeps a count of the number of events of interest to the program, such as the number of times a variable has been modified. Event counts have three operations:

1. **advance** E is used to signal the occurrence of events associated with event count E. It has the effect of incrementing E indivisibly.
2. **read** E is an expression that evaluates to the value of the event count E. If **read** returns some number *n*, then at least *n* **advance** operations must have happened. By the time this number is evaluated, the event count may have been advanced again a number of times.
3. **await** E **reaches** v waits for the event count E to have the value v. It blocks the calling thread until at least v **advance** operations have occurred. It is acceptable if more than v **advance** operations have occurred when the thread is finally unblocked. This overshoot could result from very frequent **advance** operations.

These definitions allow both **await** and **read** to be concurrent with **advance**, since the programmer won't care if **read** gives a somewhat stale value or if **await** waits a trifle too long.

Figure 7.11 shows how to encode the bounded buffer using event counts. For the time being, I assume that there is only one producer and one consumer.

Figure 7.11

```
-- other declarations as before                                        1
variable InEvents, OutEvents : eventcount := 0;                        2

procedure PutBuffer(value What : Datum);                               3
begin                                                                  4
     await OutEvents reaches InCount - Size;                           5
     Buffer[InCount mod Size] := What;                                 6
     advance InEvents;                                                 7
     InCount := InCount + 1;                                           8
end ; -- PutBuffer;                                                    9

procedure GetBuffer(result Answer : Datum);                           10
begin                                                                 11
     await InEvents reaches OutCount;                                 12
     Answer := Buffer[OutCount mod Size];                             13
     advance OutEvents;                                               14
     OutCount := OutCount + 1;                                        15
end; -- GetBuffer;                                                    16
```

There is no need to worry that the consumer and producer will simultaneously access the same cell in Buffer. The producer will wait until the consumer has taken the value from any cell before **await** in line 5 will allow it to proceed to refill it. Similarly, the consumer knows that when it accesses a cell of Buffer, the producer must have placed data there, or the **await** in line 12 would not have unblocked. Even if both **advance** operations (lines 7 and 14) happen at the same time, there is no problem, because they deal with different event counts. The bounded buffer may be used simultaneously by both threads because it guarantees that the very same datum will never be touched by both at once. I could have omitted InCount and OutCount, replacing them with **read** InEvents and **read** OutEvents, respectively, but since they are used for indices into Buffer, and **read** can return a stale value, I used separate variables to make sure the right index was always computed.

The second data type for synchronization is the **sequencer**, which assigns an arbitrary order to unordered events. A sequencer is implemented as a nondecreasing integer variable, and has only one operation: **ticket**.

- **ticket** S is an expression that first evaluates to the current value of the sequencer S and then increments S. This operation is indivisible.

Now I can implement a bounded buffer in which there are many producers. For simplicity, I will still have only one consumer. As before, consumption and production need not exclude each other. Multiple producers will take turns to make sure that they don't write into the same cell in Buffer. The new producer program is given in Figure 7.12.

Figure 7.12

```
variable ProducerTurn : sequencer := 0;                          1

procedure PutBuffer(value What : Datum);                         2
variable SequenceNumber : integer;                               3
begin                                                            4
     SequenceNumber := ticket ProducerTurn;                      5
     await InEvents reaches SequenceNumber;                      6
          -- wait for turn                                       7
     await OutEvents reaches SequenceNumber - Size;              8
          -- wait for Buffer                                     9
     Buffer[SequenceNumber mod Size] := What;                   10
     advance InEvents;                                          11
end ; -- PutBuffer;                                             12
```

Each producer must await its turn to produce. The **ticket** operator in line 5 orders active producers. There will be no wait in line 6 unless another producer has just grabbed an earlier ticket and has not yet arrived at line 11. The **await** in line 8 makes sure that the cell in Buffer that is about to be overwritten has been consumed. The **advance** in line 11 tells waiting consumers that this cell in Buffer may be consumed, and it tells waiting producers that this thread has finished its turn.

The **await** in line 6 might seem unnecessary. It's there to make sure that producers write cells of Buffer in order, so that consumers may assume that when InCount is advanced in line 11, the next cell of Buffer has new data. Unfortunately, one effect of this imposed sequential behavior on producers is that separate cells of Buffer cannot be written simultaneously. If the cells are large, producers may exclude each other for a long time.

2.8 Barriers

Some computations occur in phases, and threads that finish one phase must wait until all have finished until any may proceed to the next phase. The **barrier** type provides the necessary synchronization. It has one operation:

- **meet** B causes the thread to block on barrier B until all threads have executed a **meet** statement on B.

An example of barrier synchronization is a bottom-up version of MergeSort, shown in Figure 7.13.

Figure 7.13

```
constant UpperBound = ... -- size of array                       1
type DataArray = array 0..UpperBound of integer;                 2
variable                                                         3
     Tangled : DataArray;                                        4
     MergeBarrier : barrier UpperBound div 2;                    5
```

```
thread MergeSort(Start : integer);                                 6
variable Width : integer;                                          7
begin                                                              8
     Width := 1;                                                   9
     while Width < UpperBound+1 do -- a phase                      10
          -- Sort Tangled[Start .. Start+2*Width-1]                11
          if Start mod Width = 0 -- participate                    12
               Merge(Tangled, Start, Start+Width-1,                13
                    Start+Width, Start+2*Width-1);                 14
          end;                                                     15
          meet MergeBarrier; -- ends phase                         16
          Width := 2 * Width; -- preparation for next phase        17
     end;                                                          18
end; -- MergeSort                                                  19

begin -- main                                                      20
     for Start := 0 to UpperBound step 2 do                        21
          MergeSort(Start); -- creates a thread                    22
     end;                                                          23
end; -- main                                                       24
```

If UpperBound (line 1) is, say, 9, then line 22 starts five threads, each working on a different two-element section of Tangled. Each thread enters the first phase, sorting its own two-element section. Lines 13 and 14 sort that section, assuming that the two subsections are already sorted. Each thread waits for the phase to complete (line 16) before starting the next. MergeBarrier is declared in line 5 with a capacity equal to the number of threads. Threads that **meet** at the barrier wait until the full capacity of the barrier is reached. Only half the threads active in one phase need to participate in the next phase; they select themselves in line 12. Those that become inactive still participate in the barrier in future phases in order to permit the active ones to make progress.

Two-thread barriers can be implemented by shared variables and busy waiting or by two semaphores. Multithread barriers can be built by various combinations of two-thread barriers; there are also other ways to build them. In most implementations, when the barrier is first initialized, it needs to know exactly which threads will participate.

Some researchers have suggested that **meet** be split into two operations [Gupta 89]. The first, **arrive**, indicates that the thread has finished the previous phase. The second, **depart**, indicates that the thread is about to start the next phase. Between **arrive** and **depart**, the thread need not block if it has useful work to do. Threads are blocked at **depart** until all threads have **arrive**d. This suggestion can increase the effective parallelism of a program if there is significant work that can be done between phases. In the MergeSort example, I could place **arrive** at line 16 and **depart** after line 17. Separating **arrive** from **depart**, however, can lead to programming errors in which the operations fail to balance. I am tempted to place **depart** after line 11, but then threads would **depart** before **arrive**ing.

2.9 Performance Issues

Concurrent programs may fail not only because they contain programming errors that lead to incorrect results, but also because they make no progress due to blocking. They may also run more slowly than necessary because of poor programming.

I have already mentioned that **signal** usually occurs as the last operation in a monitor's exported procedure. In Modula-3, where the exported procedure must explicitly acquire a mutex, it is advisable to release the mutex before **signal**ing. Otherwise, the awakened thread will try to acquire the mutex and immediately block again. The same problem occurs with **broadcast**, but now many threads will try to acquire the mutex, and only one will succeed. It may be preferable (although clumsier) to use **signal** and to have each awakened thread **signal** the next one.

Starvation is a form of unfairness in which a thread fails to make progress, even though other threads are executing, because of scheduling decisions. Although starvation can be the fault of the thread scheduler, it is more often a programming error. For example, a poorly programmed solution to the readers-writers problem will block writers so long as there are any readers. New readers can come and go, but so long as there are any readers, all writers starve. The solution to starvation is to prevent new threads from acquiring mutexes until old threads have completed. In the readers-writers case, new readers can be kept out if any writers are waiting.

Deadlock occurs when a group of threads is blocked waiting for resources (such as mutexes) held by other members of the group. For example, the code of Figure 7.14 will deadlock.

Figure 7.14

```
variable                                  1
    Mutex1, Mutex2 : mutex;               2
    BarrierA : barrier;                   3

procedure ThreadA();                      4
begin                                     5
    lock Mutex1 do                        6
        lock Mutex2 do                    7
            -- anything                   8
        end;                              9
    end;                                  10
end; -- ThreadA                           11

procedure ThreadB();                      12
begin                                     13
    lock Mutex2 do                        14
        lock Mutex1 do                    15
            -- anything                   16
        end;                              17
    end;                                  18
end; -- ThreadB                           19
```

ThreadA might reach line 7 just as ThreadB reaches line 15. Each will then try to lock a mutex held by the other. Neither can make any progress.

The standard and simplest way to avoid deadlock is always to acquire resources in the same order. If ThreadB would first lock Mutex1 and then Mutex2, then there is no schedule that will lead to deadlock between these threads. For this reason, languages that provide conditional critical regions implicitly sort the necessary mutexes and acquire them in a standard order. Of course, nested conditional critical regions can still deadlock.

Another way to deal with deadlock is to provide a way for **wait** statements to be interrupted. Modula-3 provides a version of **wait** that will unblock if an exception is raised in the thread. This exception can be raised by another thread by the **alert** statement. The **alert** statement also sets a flag in the alerted thread that it can inspect in case it is busy with a long computation and is not waiting on a condition.

3 ◆ TRANSACTIONS: ARGUS

The concept of acquiring exclusion over data structures is often extended to deal gracefully with failure. This behavior is especially important for programs that modify large shared databases. A **transaction** is a set of operations undertaken by a thread. Transactions have two important properties. First, these operations are indivisible when taken as a whole. From the point of view of other threads, either they have not started or they have all finished. Second, the transaction is **recoverable**; that is, it can either **commit**, in which case all modifications to shared data take effect, or it can **abort**, in which case none of its modifications takes effect. Because transactions are indivisible, threads cannot see modifications performed by other transactions that are still in progress.

For example, in an airline reservation database, a customer may wish to exchange a seat on a given flight for a seat on another flight. The program might give up the first seat and then reserve the second. If the second plane is full, it is necessary to get back the initial seat, which may already have been allocated to another passenger. If both actions (releasing the first seat and reserving the second) are part of a transaction, then the program can just abort when it fails to reserve the second seat. The first seat will still be reserved by the original customer.

Transactions can be nested. In order to increase concurrency, programs might want to start several threads as children of an initial thread. Each can enter its own subtransaction. If any child thread fails, its own data modifications are recovered, but the parent transaction can still proceed. Unrelated transactions do not see any data modifications until and unless the top-level transaction commits.

Argus provides programming-language support for nested transactions [Liskov 83a]. The statements comprising a transaction are the body of a **transaction** statement. Of course, a procedure may be called from inside a transaction, and the procedure may be recursive, so the lexical nature of transaction entry does not limit the number of transactions. If the transaction statements finish execution, the transaction commits. The statement **abort** causes the current transaction to fail.

Data that are shared among threads must be built out of recoverable types. Argus provides recoverable versions of primitive types, such as integers and arrays. Read and write locks are implicitly acquired when recoverable variables are accessed. (These locks are typically held until the transaction completes.) If a lock cannot be granted immediately because of conflicting locks held by other threads, the accessing thread is blocked. Deadlock is automatically detected and handled by aborting one or more transactions. It is also possible for a program to explicitly acquire a read or write lock and to avoid blocking if the lock is not currently grantable. Structured types can be made recoverable by providing access procedures that use mutual exclusion and ensure that exclusion is only released when the structure's value is internally consistent.

These facilities can be used to build, for example, a bounded buffer of integers for which GetBuffer does not necessarily get the oldest remaining data [Weihl 90], as in Figure 7.15.

Figure 7.15

```
module BoundedBuffer;

    export GetBuffer, PutBuffer;

    type
        Entry = recoverable -- choice type
            Valid : integer;
            Invalid : void;
        end;
    variable
        Buffer : array of Entry -- flexible;

    procedure PutBuffer(value What : integer);
    begin
        region Buffer do -- get exclusion
            Append(Buffer,
                MakeRecoverable(Entry, Valid, What));
        end;
    end; -- PutBuffer;

    procedure GetBuffer(result Answer : integer);
    variable Item : Entry;
    begin
        region Buffer do -- get exclusion
            loop -- iterate until success
                for Item in Buffer do
                    tagcase Item of
                        when writeable Valid(Answer)
                             => ChangeRecoverable
                                 (Item, Invalid);
                            return;
                            -- releases exclusion
                        end; -- writeable
                    end; -- tagcase
                end; -- for Item
                duckout; -- release exclusion             32
                sleep();                                  33
                duckin; -- regain exclusion               34
            end; -- iterate until success                 35
        end; -- mutual exclusion                          36
    end; -- GetBuffer;                                    37

end; -- BoundedBuffer;                                    38
```

Enqueued integers are kept in the flexible array `Buffer` (line 9). Both `PutBuffer` and `GetBuffer` acquire mutual exclusion over the array by using **region** statements. Each item in the array is a recoverable object, which is a choice type (lines 4–7). `PutBuffer` (lines 10–16) puts a new recoverable entry in `Buffer` with the appropriate initial value. I use `Append` to add to the end of a flexible array and `MakeRecoverable` to generate a new recoverable item with an initial value. `GetBuffer` searches the array for an item on which it can acquire a write lock and which is valid. I use a **for** loop (lines 22–31) to scan through the flexible array. The **tagcase** statement (lines 23–30) checks both the variant (I am interested only in `Valid` items) and whether a write lock can be achieved. For those items where the variant is wrong or a write lock cannot be achieved, the single branch of **tagcase** is not selected. For the first item where the variant is correct and the write lock can be achieved, `GetBuffer` stores the value in `Answer` (line 24), changes the value to `Invalid` (Lines 25–26), and returns, releasing exclusion. If it fails to find such an item, it releases exclusion, waits a while, then tries again (lines 32–34). Any value returned by `GetBuffer` is guaranteed to have been placed there by a transaction that is visible to the current one (that is, one that has committed or is an ancestor of the current one) and not to have been removed by any active or committed transaction. Invalid initial elements of the buffer can be removed by a separate thread that repeatedly enters a top-level transaction and removes elements that are writeable and invalid.

This example shows a few drawbacks to the way Argus deals with recoverable types. Given the Argus facilities, it appears that the algorithm shown is the most efficient that can be achieved. However, `GetQueue` is inefficient, because it needs to glance at all initial buffer entries, even if they are in use by other transactions. It uses busy waiting in case it cannot find anything at the moment. Programmers have no control over when commit and abort actually make their changes, so it is possible for a consumer to get several items produced by the same producer out of order. Attempts to enhance the language by adding transaction identifiers and explicit finalization code to be executed upon commit or abort can relieve these shortcomings, but at the expense of far more complicated programs [Weihl 90].

4 ◆ COOPERATION BY PROCEDURE CALL

So far, I have described ways in which threads that cooperate through shared variables can synchronize access to those variables. A different sort of cooperation is achieved by procedure calls. When one thread (the **client**) calls another (the **server**), information can be passed in both directions through parameters. Generally, parameters are restricted to value and result modes. A single thread can act as a client with respect to some calls and a server with respect to others.

4.1 Rendezvous

In Ada, SR, and Concurrent C, procedure calls between threads are handled by a mechanism called a **rendezvous**, which is an explicit way for the server to accept procedure calls from another thread. A thread executes within a module. This module exports **entries**, which are the procedurelike identifiers that may be invoked by other threads. The declaration of an entry includes a declaration of its formal parameters.

A server accepts a call from a client by an **accept** statement, which names the entry and the formal parameters. The **accept** statement blocks until some client invokes this procedure. At that time, the actuals provided by the client are bound to the formals, and the server executes the body of the **accept**. The **accept** statement may be nested in a **select** statement, which may enable several rendezvous, based on values of current variables and even on the values of the actual parameters presented.

A client invokes a rendezvous by a syntax that looks like procedure call. The client blocks until the server executes a matching **accept** statement and either completes the body of that **accept** or explicitly releases the client. Figure 7.16 shows a bounded buffer (in Ada syntax).

Figure 7.16

```
task BoundedBuffer is
     entry GetBuffer(Answer : out Datum);
     entry PutBuffer(What : in Datum);
end;

task body BoundedBuffer is
     Size := constant 10; -- capacity of the buffer
     type Datum is ... -- contents of the buffer
     Buffer : array (0..Size-1) of Datum;
     InCount, OutCount : integer := 0;
     entry GetBuffer(Answer : out Datum);
     entry PutBuffer(What : in Datum);
```

```
begin -- body of BoundedBuffer                                12
     loop -- each iteration accepts one call                  13
          select                                              14
               when InCount - OutCount > 0 =>                 15
                    accept GetBuffer(Answer) do               16
                         Answer :=                            17
                              Buffer[OutCount mod Size];      18
                         return;                              19
                         OutCount := OutCount + 1;            20
                    end; -- accept                            21
          or                                                  22
               when InCount - OutCount < Size =>              23
                    accept PutBuffer(What) do                 24
                         return;                              25
                         Buffer[InCount mod Size] := What;    26
                         InCount := InCount + 1;              27
                    end; -- accept                            28
          end; -- select                                      29
     end; -- loop                                             30
end; -- BoundedBuffer                                         31
```

BoundedBuffer is a **task**, that is, a module that contains a thread. Ada separates the specification (lines 1–4) from the implementation (lines 5–31). This module would be declared in the same block as a producer and a consumer module. The **entry** declarations in lines 2–3 (repeated in lines 10–11) provide procedurelike headers that clients of this module may call.

Each of the alternatives in the **select** statement (lines 14–29) is headed by a Boolean guard. When BoundedBuffer executes the **select** command, the guards are evaluated. Those that evaluate to true dictate which branches are open. BoundedBuffer is then blocked until a client invokes a procedure **accept**ed by one of the open branches. If more than one client has already invoked such a procedure, then **select** is nondeterministic; one branch is arbitrarily chosen. It is up to the implementation to attempt to be fair, that is, not to always prefer one branch over another.

The **accept** statements (lines 16–21 and 24–28) introduce new name scopes in which the formal parameters are defined. A client remains blocked until the rendezvous is finished or the server executes **return** (lines 19 and 25). I have placed the **return** statements as early as possible to allow the client to proceed with its own activities.

There is no danger that InCount and OutCount will be simultaneously accessed by several threads, because they are not shared variables. Only BoundedBuffer itself can access them. By the same token, it is not possible for two rendezvous to be active simultaneously. Therefore, rendezvous have less parallelism than can be obtained by, for instance, event counts.

Figure 7.17 Rendezvous

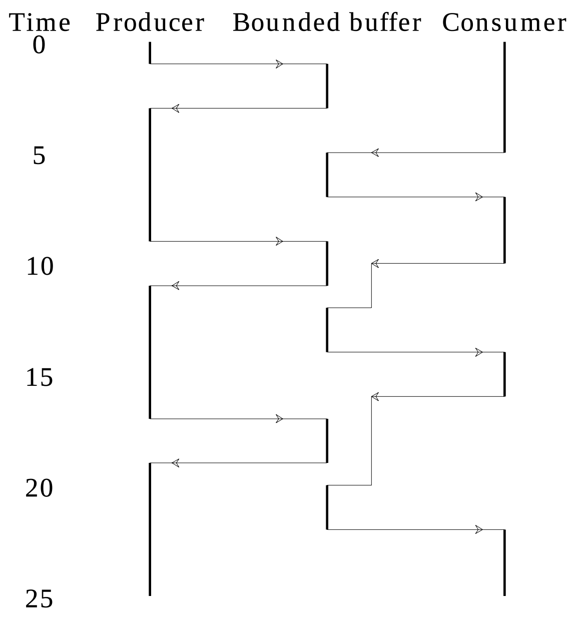

Figure 7.17 shows how the rendezvous code might execute. Time starts at the top and progresses downward. Solid vertical lines indicate execution; spaces indicate waiting. The producer invokes `PutBuffer` at time 1 and gets a response at time 3. Between those times, the producer and the bounded buffer are in rendezvous. The consumer invokes `GetBuffer` at time 5 and gets a response at time 7. The producer makes its second call at time 9. This call is still in progress when the consumer calls `GetBuffer` at time 10. The consumer is blocked until the producer's rendezvous finishes at time 10. The consumer calls the bounded buffer again at time 16. The buffer is empty, so its call is not accepted. The consumer is blocked until the following rendezvous between the producer and bounded buffer finishes.

I have written `BoundedBuffer` as an unterminated loop. Ada terminates all remaining threads in a name scope if all are blocked in calls, **`accept`**, or **`select`** statements. Therefore, when the producer thread finishes, the consumer will be allowed to consume all the remaining entries from the bounded buffer. Then the consumer will block on a call to `GetBuffer`, and `BoundedBuffer` will block in the **`select`** statement. Both will be terminated.

SR and Concurrent C add extra features to Ada's rendezvous to affect the scheduler's decision about which branch of **`select`** to prefer if several are open and have incoming calls. Each branch can be given a numeric priority. (Ada has the concept of static task priority, but not dynamic branch priority.) If there are several waiting calls on a particular **`accept`**, they may be sorted based on the values of the actual parameters. (In Ada, calls are processed strictly in first-come, first-served order.) To show these features, in Figure 7.18 I have rewritten the **`select`** loop from Figure 7.16 to prefer producers to consumers unless the buffer is nearly full, and to prefer low values of data to high values.

Figure 7.18

```
select                                                  1
     priority -- high number is better                  2
          if Size - (InCount - OutCount) < 2            3
               then 1 else 0                            4
     accept GetBuffer(Answer)                           5
     when InCount - OutCount > 0                        6
     do                                                 7
          Answer := Buffer[OutCount mod Size];          8
          return;                                       9
          OutCount := OutCount + 1;                    10
     end; -- accept                                    11
or                                                     12
     priority -- high number is better                 13
          if Size - (InCount - OutCount) < 2           14
               then 0 else 1                           15
     accept PutBuffer(What)                            16
     when InCount - OutCount < Size                    17
     sortedby (-What) -- prefer low values             18
     do                                                19
          return;                                      20
          Buffer[InCount mod Size] := What             21
          InCount := InCount + 1;                      22
     end; -- accept                                    23
end; -- select                                         24
```

The **priority** clauses (lines 2–4, 13–15) decide which branch to prefer if several are open. In this case, the second branch has higher priority unless the buffer is nearly full. I have placed the **when** guard (lines 6, 17) after the **accept** clause so that the guard can take advantage of the formal parameters introduced by **accept**, even though this example doesn't do so. (SR uses this order.) The **sortedby** clause (line 18) reorders multiple calls to PutBuffer based on the formal parameter What.

4.2 Remote Procedure Call (RPC)

If threads do not share variables (for example, if they are running on different machines connected by a network), the only way they can cooperate is by procedure call or messages. Rendezvous is one way of accepting procedure calls. The only calls that are handled are those that match open **accept** statements. **Remote procedure call** (RPC) means an invocation that is handled not by **accept** statements, but by an ordinary exported procedure. Such calls can cross compilation units, processes, computers, and even programs that are written at different times in different languages.

The model of computation for remote procedure calls is somewhat different from what I have been discussing so far. Each address space may have multiple threads, which may share variables in that address space, subject to any scope rules the language imposes. Threads in separate address spaces do not share variables.

A thread may invoke an exported procedure inside another address space by a remote procedure call. There are two equivalent ways to picture the effect of such a call. You can imagine the thread migrating temporarily to the address space of the server, performing the call there, and then returning to the client's address space. Address spaces thus share information by sending their threads to visit other address spaces, bringing and returning data in their parameters. Alternatively, you can imagine the calling thread sending a message to the server and then blocking. A new service thread starts in the server address space for the purpose of handling the call. When it finishes, it sends results back to the blocked client thread, causing the client to awaken. The service thread then terminates. The first view is simpler. The underlying implementation is likely to use something closer to the second view.

In the DP (Distributed Processes) language of Brinch Hansen [Brinch Hansen 78], each address space starts with one thread, which starts running the main program of that address space. That thread cannot create new threads, but it may wait for conditions (using an **await** statement). Remote procedure calls are blocked until no thread is active in the server. A thread is considered inactive if it has terminated or if it is blocked waiting for a condition. It is active, however, if it is in the middle of a remote procedure call to some other address space. Therefore, the programmer does not need to be afraid that variables will suddenly have different values after a remote procedure call returns; such a call is indivisible. An **await** relaxes exclusion, though, allowing a client thread to visit. Therefore, data can change during **await**, but **await** checks a Boolean condition that can prevent it from unblocking until the situation is appropriate.

Figure 7.19 is an implementation of a bounded buffer in DP.

Figure 7.19

```
-- declarations as in Figure 7.4 (page 192).

procedure PutBuffer(value What : Datum);
begin
     await InCount - OutCount < Size do
          Buffer[InCount mod Size] := What;
          InCount := InCount + 1;
     end; -- region
end -- PutBuffer;

procedure GetBuffer(result Answer : Datum);
begin
     await InCount - OutCount < Size do
          Answer := Buffer[OutCount mod Size];
          OutCount := OutCount + 1;
     end; -- region
end GetBuffer;
```

This code is remarkably similar to Figure 7.4 (page 192). The only difference is that there is no need to lock any mutexes, since the thread executing either `PutBuffer` or `GetBuffer` is guaranteed exclusion in any case.

Languages like C that do not have remote procedure call built in can take advantage of a **stub compiler**, which takes a specification of the exported procedures and builds suitable code for both the client and the server [Nelson 81]. One widely available stub compiler is Sun RPC, a remote-procedure-call library designed by Sun Microsystems, Inc.[2] This library includes procedures for both the client (*c*) and server (*s*) for establishing a connection between them (*c* and *s*), sending a remote procedure call (*c*), receiving a remote procedure call (*s*), sending a response (*s*), and receiving the response (*c*). Parameters are transmitted in both directions in a machine-independent data format called “External Data Representation” (XDR); the client and server must call conversion routines to package and unpackage parameters to and from this format.

Many experimental languages have been designed to offer RPC directly, without explicit recourse to library packages. They all offer some mechanism for establishing a connection between a client and server, typically involving search through some name space (so the client can find a server) and connecting to some interface (to make sure the client and the server agree on what calls are valid). They might provide synchronization methods based on any of the methods I have described earlier to control access to variables that are accessed by multiple threads in the same address space. They often include automatic transmission of structured parameters. Argus even allows the parameters to contain pointer types. The runtime routines expand such values for transmission by traversing all the pointers. Argus also supports remote invocation of CLU iterators and lets the invoked procedure raise exceptions.

The compiler sees to it that remote procedure calls are packaged into messages in the client and unpackaged in the server by using a stub compiler. It also tries to ensure that RPC is type-secure, that is, that the procedure header in the server matches the call that the client is making. One mechanism for type security is to represent the type of the procedure (that is, its name and the types and modes of its parameters) as a string and then to derive a hash value from that string [Scott 88]. These hash values can be compiled; they need not be computed at runtime. The hash value is sent in each call message from client to server. The server checks that the hash value is correct; if not, there is a type error. A related idea is to represent each type as a tree, derive a polynomial from the tree, and evaluate the polynomial at a special point to produce a hash value [Katzenelson 92].

4.3 Remote Evaluation (REV)

Remote procedure call only works if the server exports the procedure that the client needs. But clients are often written long after the server, and they may have needs that were not foreseen in the server. Some clients may need the server to run specialized procedures that most clients would not be interested in but which could run far more efficiently on the server than on a client, because the client would need to repeatedly invoke server routines remotely.

Remote evaluation (REV) is a technique that allows clients to send not only parameters, but also procedures, to the server [Stamos 90]. The procedures may refer to other procedures exported by the server. For example, a mail-delivery server might export a procedure `DeliverMail`. A client that wants to send a hundred identical messages could use RPC, invoking `DeliverMail` a hundred times, each time passing the message. Alternatively, it

[2] 2550 Garcia Avenue, Mountain View, California, 94043

could use REV, sending a small procedure to the server that invokes `DeliverMail` a hundred times. The REV method is likely to be far more efficient. It also frees the mail-server designer of worries that the set of procedures exported by the server is not exactly right for every client.

A programming-language implementation of REV must be able to determine whether the server exports enough operations to support an REV request; if not, it must decide how much code actually needs to be sent. At one extreme, the procedure that is to be evaluated is itself exported by the server. In that case, the client needs to send only the parameters and receive the results; REV becomes RPC. At the other extreme, not only does the server not export the procedure, but several other procedures that it calls in turn are also not exported. The client must bundle and send enough procedures to ensure that the server will be able to complete the REV request. Any nonlocal variables needed by those procedures must also be bundled, and they must be returned by the server to update the values in the client.

The client can bundle the procedures needed by an REV request either at compile time or runtime. Compile-time bundling is more efficient but more restrictive. To make a compile-time bundle, the compiler must know what procedures are exported by the server, and it must traverse the invocation graph of the invoked procedure to discover all the procedures that must be included. To make a runtime bundle, the compiler must prepare the invocation graph and keep it until runtime. When an REV request is encountered, the client must query the server to discover its list of exported procedures and traverse the invocation graph.

REV requests may be nested. A procedure that is sent from a client to a server may contain another REV request to some other server. Compile-time bundling is unlikely to work for nested requests, because the contents of the nested bundle depend on the invocation graph in the server, which is not necessarily available to the compiler of the client.

REV requests that pass procedures as parameters cause a special problem. Compile-time bundling might refuse to deal with such parameters unless their binding is known at compile time.

REV can cause a major security headache. The server must be protected against misbehaving procedures that are sent to it. Authentication protocols can be used to restrict clients to those on an approved list. Running the procedure in a separate thread on the server under some sort of time slicing can protect the server against wasting all its time on a nonterminating computation. Giving that separate thread readonly access to server variables can protect the server against data corruption, but it restricts REV to operations that do not need to modify server data. Interpreting the REV request in the server instead of running it can allow the server to refuse potentially dangerous operations.

REV can be made implicit in every call and divorced from language design. The language runtime support can choose on every RPC whether to implement the request by sending a message for RPC, sending a bundle for REV, or requesting a bundle from the server for local evaluation. This decision can be based on statistics gathered during execution in an attempt to balance communication and computational resources among machines [Herrin 93]. The contents of the bundle need not include more than the procedure mentioned in the RPC; there is no need either at compile time or runtime to deal with invocation graphs. Any procedure that cannot be resolved locally can certainly be resolved remotely.

5 ◆ COOPERATION BY MESSAGES

Although a procedure-call syntax makes concurrent programming look superficially like sequential programming, not all cooperation is easily shoehorned into the procedure-call model. First, a single query might generate multiple results spread over time. If the query is represented as a procedure call, then the results must either be result parameters, which means the client is blocked until the last result is ready, or the results are independent calls in the other direction, which confuses the issue of which thread is client and which is server. Second, some cooperation is unidirectional; there is no need to block the client until the server receives, acts on, and responds to a call. Third, some computations are best viewed as interactions among peers, where no simple client-server hierarchy applies. Fourth, some computations require multicast of the same data to groups of address spaces. It is wasteful to program multicast as multiple procedure calls. Fifth, it might be necessary to reply to requests in a different order from the order in which they arrive.

For these reasons, some experimental languages provide more primitive message-passing notions instead of or in addition to RPC. Often, message passing is provided as a library package to be used within some other language such as C. Operating-system support is needed to make the individual operations efficient. The following table indicates some of the facilities that can be provided as simple language extensions or in library packages.

Operation	**Parameters**	**Results**
connect	partner	connection
group	set of partners	connection
send	connection, data	
receive	connection	data
reply	data	
forward	connection	

This list is neither complete (library packages often provide many more routines) nor required (many library packages have no group or forward operations, for example). Still, it provides a reasonable set of functions for message passing.

The connect operation builds a connection, that is, a channel across which communication takes place; thus, individual send operations need not specify which process is to receive the message. Such a specification is given only once. It might be as simple as a process identifier or as complex as giving a process name or other characteristics to be looked up in a database. The group operation builds a connection that leads to multiple recipients. This facility is helpful for multicast.

The send operation might be designed to block the sender until the message can be copied to a safe place, until the message is sent, until the destination machine(s) receives it, until the destination thread(s) receives it, or until a response arrives back to the sender from the destination thread(s). Semantics that do not wait for the destination machine are usually called **asynchronous**, and those that wait for a response are called **synchronous**. There is a wide spectrum of synchronicity, so these terms are not very precise. The data that are sent can be treated just as an array of characters, or they may have associated type information.

The `receive` operation is used to accept incoming messages. It may be **selective**; that is, it might only accept messages that arrive on a set of connections or messages that match some pattern. It might reorder messages based on their contents. It might block until such a message arrives; it may have a timeout period, after which it fails if no message arrives; or it may just enable a receive but allow the thread to continue executing other statements.

The `reply` operation sends data back to the originator of the most recent message. In some languages, such as SR and Hermes, the program can specify which message is being responded to, so replies need not follow the order of receives. Packages that provide `reply` often have a single operation that combines `send` and `receive`. The client uses `send/receive` and the server uses `receive` followed by `reply`.

The `forward` operation redirects the most recent incoming message (or a specified message) to a different destination. The recipient can then reply directly to the original sender. This facility is called **delegation**.

5.1 CSP

CSP (Communicating Sequential Processes) is a proposal made by C. A. R. Hoare for message passing between threads that do not share variables [Hoare 78]. It is the framework upon which Occam was developed [May 83]. Communication is accomplished by **send** and **receive** statements. Although the **send** statement looks like a procedure invocation, in fact it is a pattern specification, much like Prolog (discussed in Chapter 8). The pattern is built out of an identifier and actual parameters. It is matched against a pattern in a **receive** statement in the destination thread. Variables in the **receive** pattern are like formal parameters; they acquire the values of the actual parameters in the matching **send** pattern. Patterns match if the pattern name and the number of parameters are the same and all formal parameter patterns match the actual parameter patterns. Matching is even used for the assignment statements, as in the examples shown in Figure 7.20.

Figure 7.20

```
left := 3;                                                  1
right := 4;                                                 2
x := cons(left, right); -- assigns pattern "cons(3,4)"      3
form(right) := form(right+1); -- right := right+1           4
factor(cons(left,right)) := factor(cons(5,6));              5
     -- left := 5; right := 6                               6
right = imply(); -- pattern with no parameters              7
muckle(left) := mickle(left+1); -- match error              8
```

Variables can hold pattern values, as in line 3. Here, cons is not a procedure call, just a pattern constructor. Line 4 shows that matching the actual to the formal is like an ordinary assignment. Line 5 shows that matching works recursively. Patterns need not have parameters (line 7). If the pattern name disagrees, match fails (line 8). In each of these cases (except the last), a **receive** in one thread with the pattern on the left-hand side would match a **send** in another thread with the pattern on the right-hand side.

CSP's control structures include Ada's nondeterministic **select** and also a nondeterministic **while**, which iterates open branches until no branch is open. Guards can be Boolean expressions, but they may also have as a final condition a **send** or **receive** statement. If the guard has such a statement, it is called an **output guard** or an **input guard**. For implementation reasons, original CSP did not allow output guards. It is hard, but not impossible, for an implementation to pair communicating threads when several have both **send** and **receive** guards open. Pairing is easier under the restriction that a guarded **send** or **receive** can only be matched with an absolute (unguarded) **receive** or **send**; some implementations of CSP make that restriction and allow output guards.

Figure 7.21 shows how a bounded buffer can be implemented in CSP, using both input and output guards.

Figure 7.21

```
type
    Datum = ... -- contents of the buffer

thread BoundedBuffer;
constant
    Size = 10; -- capacity of the buffer
variable
    Buffer : array 0..Size-1 of Datum;
    InCount, OutCount : integer := 0;

begin
    while -- each iteration handles one interaction
        when InCount - OutCount > 0 and
            receive PutBuffer(Buffer[InCount mod Size])
            from Producer =>
                InCount := InCount + 1;
        when InCount - OutCount < Size and
            send TakeBuffer(Buffer[OutCount mod Size])
            to Consumer =>
                OutCount := OutCount + 1;
    end; -- while
end; -- thread Buffer

thread Producer;
begin
    loop
        Value := ...; -- generate value
        send PutBuffer(Value) to BoundedBuffer;
    end; -- loop
end; -- Producer
```

```
thread Consumer;                                         28
begin                                                    29
    loop                                                 30
        receive TakeBuffer(Value) from BoundedBuffer;    31
        ...; -- use value                                32
    end; -- loop                                         33
end; -- Consumer                                         34
```

The Producer thread (lines 21–27) repeatedly generates a value and **send**s it to the BoundedBuffer thread inside a PutBuffer pattern. This **send** blocks Producer if BoundedBuffer is not able to accept the match immediately, either because it is occupied with something else or because there is no matching **receive** currently open. The Consumer thread (lines 28–34) repeatedly **receive**s a value from BoundedBuffer with a TakeBuffer pattern. This **receive** can block Consumer if BoundedBuffer does not have a matching **send** currently open. BoundedBuffer spends all its time in a nondeterministic **while** loop (lines 10–19) with two branches, one to accept data from Producer (lines 11–14), and the other to feed data to Consumer (lines 15–18). Each branch is guarded to make sure that the buffer situation allows it to be selected. The first guard is an input guard, and the second is an output guard. If the buffer is neither full nor empty, both guards will be open, and whichever of Producer and Consumer is ready first will match its respective **receive** or **send** statement. If both are ready, then the scheduler will select one in an arbitrary, but in the long run fair, way. The **while** will always have at least one branch open, so it will never terminate.

5.2 Lynx

Lynx is an experimental language implemented at the University of Wisconsin and at the University of Rochester [Scott 84, 86]. Address spaces and modules in Lynx reflect the structure of a multicomputer, that is, a distributed-memory machine. Each outermost module represents an address space. As in DP, each address space begins executing a single thread. That thread can create new threads locally and arrange for threads to be created in response to messages from other processes. Threads in the same address space do not execute simultaneously; a thread continues to execute until it blocks, yielding control to some other thread. It is not an error for all threads to be blocked waiting for a message to be sent or received.

Lynx is quite helpful for programming long-running processes (called **server processes**) that provide assistance to ephemeral processes (called **client processes**). Typically, server processes are programmed to build a separate thread for each client process to keep track of the ongoing conversation between server and client processes. That thread may subdivide into new threads if appropriate. Lexical scope rules determine what variables are visible to any thread; the runtime organization uses a cactus stack.

Lynx provides two-way communication **links** as first-class values. A link represents a two-way channel between address spaces. The program dynamically binds links to address spaces and entries. Links can be used for reconfigurable, type-checked connections between very loosely coupled processes that are designed in isolation and compiled and loaded at disparate times.

A link variable accesses one end of a link, much as a pointer accesses an object in Pascal. The only link constant is `nolink`. Built-in functions allow new links to be created (both ends start by being bound to the creator's address space) and old ones to be destroyed. Neither end of a destroyed link is usable.

Objects of any data type can be sent in messages. If a message includes link variables or structures containing link variables, then the link ends referenced by those variables are moved to the receiving address space. This method could be called "destructive value" mode, since the value is transmitted, but becomes inaccessible at the sender. Link variables in the sender that refer to those ends become dangling references; a runtime error results from any attempt to use them.

Message transmission looks like RPC from the client's point of view. The client dispatches a request and waits for a reply from the server. From the server's point of view, messages may be received by rendezvous, using an **accept** statement, or by thread creation, in which a new service thread is built to execute a procedure when a message arrives.

Servers decide dynamically which approach to use for each link. They arrange to receive requests by thread creation through the **bind** statement, which binds a link to an exported procedure (I will call it an "entry"). This arrangement is cancelled by **unbind**. A link may be simultaneously bound to more than one entry and may even be used in **accept** statements. These provisions make it possible for threads to multiplex independent conversations on the same link. If a client invokes an entry via a link that is not currently bound to that entry, the invocation blocks until the server either binds the link to that entry, enters a rendezvous for that entry, or destroys the link.

When all threads in an address space are blocked, the runtime support package attempts to receive a message on any link that is bound or is the subject of an outstanding **accept**. Since messages are like RPC, they specify the exported procedure that they are attempting to invoke. The name of the procedure is matched against those of the active **accept**s and the bound links to decide whether to resume a blocked thread or create a new one. Bindings or **accept**s that cause ambiguity are runtime errors.

Lynx provides type-secure RPC in the fashion described earlier on page 213. Its exception-handling mechanism permits recovery from errors that arise in the course of message passing, and allows one thread to interrupt another.

Figure 7.22 shows how a bounded buffer can be programmed in Lynx.

Figure 7.22
```
constant
     Size = 10; -- capacity of the buffer
type
     Datum = ... -- contents of the buffer
variable
     Buffer : array 0..Size-1 of Datum;
     InCount, OutCount : integer := 0;
     ParentLink, ProducerLink, ConsumerLink : link;
entry
     Initialize(value link, link); -- for rendezvous
     PutBuffer, GetBuffer; -- full header and bodies below
```

```
procedure PutBuffer(value What : Datum);                           12
begin                                                              13
     Buffer[InCount mod Size] := What;                             14
     InCount := InCount + 1;                                       15
     if InCount - OutCount = 1 then -- no longer empty             16
          bind ConsumerLink to GetBuffer;                          17
     end;                                                          18
     if InCount - OutCount = Size then -- now full                 19
          unbind ProducerLink from PutBuffer;                      20
     end;                                                          21
end -- PutBuffer;                                                  22

procedure GetBuffer(result Answer : Datum);                        23
begin                                                              24
     Answer := Buffer[OutCount mod Size];                          25
     OutCount := OutCount + 1;                                     26
     if InCount - OutCount = 0 then -- now empty                   27
          unbind ConsumerLink from GetBuffer;                      28
     end;                                                          29
     if InCount - OutCount = Size-1 then -- no longer full         30
          bind ProducerLink to PutBuffer;                          31
     end;                                                          32
end; -- GetBuffer                                                  33

begin -- main                                                      34
     accept Initialize(ProducerLink, ConsumerLink)                 35
          on ParentLink;                                           36
     bind ProducerLink to PutBuffer;                               37
end; -- main                                                       38
```

The program defines three entries (lines 9–11); one is for rendezvous, and the others are handled by thread creation. This program begins with one thread that executes **accept** (lines 35–36) to get values for the links to the producer and consumer. It gets these values in a startup message from its parent, to which it is connected by ParentLink. I ignore how ParentLink gets initialized. Then the program **bind**s ProducerLink (line 37) to its entry PutBuffer. It makes no sense to bind ConsumerLink yet, because there is nothing yet to consume. Then the main thread terminates. Incoming RPC will create new threads as needed. Both PutBuffer and GetBuffer arrange for **bind**ing and **unbind**ing entries when the buffer gets full, empty, or no longer full or empty (lines 16–21 and 27–32). PutBuffer and GetBuffer themselves do not need to block if the buffer is not ready for them, because the pattern of bindings and the nonpreemptive scheduler assure that they cannot be called unless the state of the buffer permits them to proceed.

5.3 Linda

Like CSP, Linda also uses patterns instead of procedure calls in its messages [Gelernter 85]. Unlike CSP, the **send** statement does not indicate the thread to which data are to be sent, nor does **receive** indicate from which thread the data are coming. Instead, **send** places the data in a global data pool that can be accessed by any thread, and **receive** takes data from that pool. It is up to the implementation to organize data so that threads running on multiple machines can find data in the global pool. Typically, implementations will hash on the pattern name[3] and store each bucket redundantly on $n^{1/2}$ out of n machines. The **receive** pattern can include parameters that are variables (like actual parameters in result mode, to be bound to values during matching), constants (to selectively receive by restricting what data match this pattern), and don't-cares. **Receive** blocks the caller until matching data appears in the pool, and then it indivisibly removes the matching data from the pool. There is also a **read** statement with the same semantics as **receive** except that the data are not removed from the pool.

A Linda implementation of the bounded buffer would be identical to the CSP one in Figure 7.21 (page 217), except that the **send** and **receive** statements would not indicate which thread was the intended partner. Multiple producers and consumers could use the same code. However, such a bounded buffer thread would be illogical in Linda, since the data pool itself is an unbounded buffer. Even if the bounded buffer is full, the producer would still be able to repeatedly **send** the `PutBuffer` pattern. It would be more straightforward for the producer to just **send** a `BufferData` pattern and for the consumer to **receive** that pattern. A truly bounded buffer can actually be implemented in Linda; see the exercises for details.

The advantage of the Linda approach is that programs need not consider the destination and synchronization aspects of each message that is passed. If a particular destination thread is important, that can be coded into the pattern, of course, but many applications will not need such explicit control.

One set of applications to which Linda is well suited involves problems whose solutions create subproblems. All problems are placed in a "problem heap" as they are generated. The heap is stored in the global pool. Each thread repeatedly extracts a problem (using **receive**) and solves it, putting any new subproblems back on the heap (using **send**). This situation is much like a bounded buffer, but there is no concept of order connecting the elements of the buffer.

Linda is generally implemented as a library package added to some other language, such as C. A more type-safe design called Lucinda, which combines Linda with Russell, has also been devised [Butcher 91].

5.4 SR

The SR language was developed over a period of ten years by Gregory Andrews at the University of Arizona [Andrews 88]. It contains features for both distributed- and shared-memory concurrency.

SR modules are separately compiled. A module specification and its body may be compiled separately. At runtime, modules are dynamically instantiated and given initial actual parameters. By default, a new module shares the address space of its creator, but it can be placed on any machine (physical or virtual) instead.

Ordinary modules may import declaration modules. Each declaration module may import other declaration modules and introduce constants,

[3] What I call the pattern name would actually be the first element of a tuple in Linda, but I find CSP nomenclature a bit easier to understand.

types, variables, entries, and procedures. These may appear in any order, so dynamic-sized arrays are easy to build. Declaration modules also have initialization code. The declaration modules are instantiated (at most once each) and initialized at runtime in whatever order is dictated by the partial order of imports. One copy is created dynamically per address space the first time it is needed, so that threads in each space have access to the declarations. Global variables imported from declaration modules should be treated as readonly; modifying a global variable only affects the copy in the current address space.

SR contains a wide variety of synchronization and communication methods. It provides synchronization by semaphores (implemented as a module type with entries for **up** and **down**), and communication by rendezvous, RPC, and messages. The client may choose whether to use synchronous or asynchronous calls, that is, RPC or messages. The server may choose to receive messages by thread creation or by rendezvous. It may inspect how many calls are outstanding on any entry. The rendezvous **accept** statement[4] includes both a synchronization (**when**) clause and a scheduling (**sortedby**) clause, both of which may depend on the formal parameters of the call. Both a **reply** and a **forward** statement are included.

Destinations for calls can be represented by pointers to modules, which can even reference modules across machine boundaries. The declaration for module pointers includes which module type they may reference. Every module instance has a pseudovariable `self` that points to itself. Calls and replies may pass module pointers, so communication paths may vary dynamically. In addition, threads may invoke an entry imported from a declaration module. Any module instance that imports that declaration module may receive such an invocation.

In addition to initialization code, a module can be contain a **thread** declaration, much as in Modula. The compiler converts that declaration to an anonymous entry with no parameters; the act of instantiating the module implicitly sends a message to that entry, which creates the new thread. A module may also contain finalization code, which is invoked in any instance when the instance is terminated. All instances are terminated when deadlock occurs, as in Ada.

5.5 Object-Oriented Programming

The object-oriented paradigm (see Chapter 5) lends itself nicely to distributed-memory machines, because each object may reside entirely within a single memory, and interaction between that object and the rest of the computation is entirely mediated by messages. There are several object-oriented languages for concurrent programming. For example, DC++ is a version of C++ with threads [Carr 93], Distributed Eiffel [Gunaseelan 92] and Eiffel Linda [Jellinghaus 90] extend the object-oriented Eiffel language, and CST (Concurrent Smalltalk) extends Smalltalk [Dally 89]. You can read a survey of these languages and others in [M. Nelson 91].

[4] I am changing the keywords, as usual, for the sake of consistency.

To show you a concrete example, I will focus on the ALBA language [Hernández 93], another example of an object-oriented concurrent programming language. ALBA is strongly typed and is in many ways a typical object-oriented language; that is, it provides classes and inheritance. What sets it apart is its recognition that it executes in a distributed environment. Unfortunately, the ALBA document is incomplete, so I have added further specifications of my own that the authors may not agree with.

There are no class variables, because different instances of a class are likely to be in different memories. Instance variables, of course, exist. Any number of threads may simultaneously execute methods in an object unless the object is an instance of a **serialized class**, which allows only one thread at a time. It is unclear whether serialized ALBA objects accept new threads when the existing thread is blocked waiting for a call or when the existing thread is in the middle of invoking a method in some other object.

Objects may be created at any time; their identity is stored in an instance variable of their creator, so that the creator can send them messages. This identity can be passed to other objects in a parameter in order to allow them to invoke methods in the newly created object. Each object has a pseudovariable `creator` that points to the creator and `self` that points to itself.

When an object is created, the programmer has some control over where it will be placed initially. The ALBA implementation does not dynamically move objects once they are created, but techniques for such migration are well understood [Artsy 89]. The class declaration may restrict its instances to a subset of the machines, and the instance-creation request may further restrict the positioning. For this purpose, ALBA has a data type for sets of machine identifiers.

As in Ada, Lynx, and SR, ALBA objects can accept incoming messages by rendezvous. Alternatively, an invocation of a method may be handled by thread creation.

During execution of a method, two more pseudovariables are defined: `sender` and `reply`. Typically, they are identical, pointing to the object that invoked the method. However, ALBA provides for delegation. A method may be invoked with an explicit "reply-to" specification, which will be copied to the recipient's `reply` pseudovariable.

Figure 7.23 shows an ALBA implementation of merge sort.

Figure 7.23
```
type                                                      1
     DataArray = array whatever of integer;               2

class MergeSort;                                          3

method Done -- for rendezvous                             4
     (Sorted : DataArray; LowIndex, HighIndex : Integer); 5

method Sort -- thread-creating                            6
     (Tangled : DataArray; LowIndex, HighIndex : integer); 7
variable                                                  8
     MidPoint : integer := (LowIndex + HighIndex) div 2;  9
     LeftChild, RightChild : MergeSort;                   10
     Responses : integer := 0;                            11
```

```
begin -- method Sort                                                12
     if LowIndex + 1 < HighIndex then -- worth sorting              13
     MidPoint : integer := (LowIndex + HighIndex) div 2;            14
          create LeftChild;                                         15
          create RightChild;                                        16
          send LeftChild.Sort(Tangled, 1, MidPoint);                17
          send RightChild.Sort                                      18
               (Tangled, MidPoint+1, HighIndex);                    19
          while Responses < 2 do                                    20
               accept Done(Sorted, LowIndex, HighIndex)             21
               from {LeftChild, RightChild}                         22
               do                                                   23
                    Tangled[LowIndex .. HighIndex] :=               24
                         Sorted[LowIndex ..  HighIndex];            25
                    Responses := Responses + 1;                     26
               end; -- accept                                       27
          end -- while                                              28
          Merge(Tangled, 1, MidPoint, MidPoint+1,                   29
               HighIndex);                                          30
     end; -- worth sorting                                          31
     send creator.Done(Tangled, 1, HighIndex);                      32
     destroy(self);                                                 33
end; -- method Sort                                                 34

end; -- class MergeSort                                             35
```

A client that wishes to sort an array creates an instance of MergeSort (I will call it the "worker") and invokes the thread-creating Sort method (lines 6–34). Because objects do not share memory, all parameters to Sort are passed in value mode. The worker creates left and right child instances (lines 15–16); they are declared in line 10. The worker then invokes the Sort method in the children on the appropriate regions of the array (lines 17–19). These calls are marked **send** to indicate that the call is asynchronous; that is, the caller need not wait for a response. Asynchronous calls are only allowed on methods that do not return values. When the children are finished, they will invoke the Done method in their creator, the worker. The worker accepts these invocations in a rendezvous (lines 21–27), placing the result that comes with the invocation back into a slice of the local array (lines 24–25). When it has received both responses, the worker merges the two halves of the array (lines 29–30). It then tells its own creator that it is done (line 32), providing the sorted array as a parameter. This invocation of Done is asynchronous, but it does not create a new thread, because it is accepted in a rendezvous by the creator. The worker then destroys its instance (line 33), including all threads that may currently be active. Its purpose has been accomplished.

ALBA supports multicast by letting a program asynchronously invoke a non-value-returning method on any subset of the existing instances of a class. The destination of an invocation can be an instance (the usual case), a set of instances, or a class (all existing instances are sent the message). Rendezvous can be selective by restricting attention to messages from a given instance, a set of instances, or a class. In line 22, I have restricted attention to the two children, although such a restriction is not necessary.

5.6 Data-Parallel Programming

Scientific applications often require similar computations across very large data sets, which may represent a connected physical entity. For example, a weather simulator might advance time in small increments while keeping track of wind patterns, cloud cover, precipitation, sunlight-induced wind currents, and so forth over a large geographical area represented as interconnected records, each covering a few square miles. Such applications often strain the computational ability of any single computer, so they are programmed on shared-memory or distributed-memory computers. Each computer is given a region of the data and computes as independently as possible of the other computers. When necessary, computers exchange information with others that deal with neighboring data. This style of computing is called **data-parallel** computing with **coarse-grain parallelism**. That is, the machines work in parallel on different parts of the data, and they only coordinate their activities on occasion.

Several languages have been implemented specifically to deal with coarse-grain parallelism. Some, like PVM [Sunderam 89], are implemented as library packages to be invoked from any conventional language for passing messages. Charm is a more complex language that extends C with dynamically creatable threads that inhabit modules [Kalé 90]. Global variables can be accessed only by a runtime procedure because they may be stored anywhere. The threads communicate both by messages (much like method invocations in object-oriented programming) and through serialized modules that accumulate data, creating such results as sums and averages.

The Canopy language is more complex yet. It is implemented as a library package to be used by ordinary C programs. Unlike PVM and Charm, it imposes a distinctively data-parallel view on the programmer.

Data in Canopy are represented as records stored on a grid of sites. Grids are dynamically constructed by calls to a runtime routine. Definition routines are provided for many standard topologies, such as three-dimensional meshes, and the programmer may define any desired topology by using a more primitive routine. A computation may use several different grids, although using more than one is unusual. Each site in a grid has coordinates and is connected to its neighbors by links. Data records are associated with each site and each link.

The runtime support software arranges for sites to be located on physical machines. Typically, there are far more sites than machines; a typical problem may have a million sites running on a hundred machines. The programmer has no control over the mapping of sites to machines, and there is no way for a program to discover that mapping. Each machine has a complete copy of all code and global data and has space to allocate site-local data.

Computation proceeds in phases. Each phase is initiated by a distinguished site called the controller, which executes the control program. Before the first phase, the controller establishes the grids, site sets, mappings between grids, and record fields (local variables) that will be used. It then calls `CompleteDefinitions` to activate these definitions. For each phase, the control program may initialize global variables by broadcasting a copy to all sites (actually, to all machines). Individual sites should treat such global data as readonly. The controller then invokes a procedure on each site in a grid or subset of a grid by calling `DoTask`. Each such site gets its own thread to execute that procedure simultaneously with all other sites. (Actually, the implementation has each machine cycle through all the sites that reside on that machine, but the programmer doesn't need to know that.) When all sites have finished, the control program resumes to begin the next phase.

The controller passes information to the sites and receives information back from them via parameters of DoTask. These parameters are arranged in triples, which represent the parameter-passing mode, the address of the parameter, and its length. (A true compiler, instead of a library package for C, would not use addresses and would not need to be told the lengths.) The modes available are value, procedure (that is, passing a procedure), and accumulate, which combines results from all the sites and presents them to the controller. The accumulation techniques include summation, maximum, and minimum, and the programmer may provide other accumulation techniques. The parameter-passing mode is presented as a pointer to a record that includes a routine that combines two values. This routine, which should be commutative and associative, is repeatedly invoked as sites terminate. (The implementation invokes it on the site's own machine until it has exhausted all its sites, and then repeatedly on the controller's machine until all machines have reported values.)

During a phase, each thread has access not only to its own local variables (those in the records associated with its site) and the local variables of its adjacent links, but all local variables in every site and link. Canopy provides library routines that fetch and store the values of these variables. Fetches return a pointer, which either points to the data itself, if it is on the same machine, or to a temporary copy, if it is not. Therefore, threads should treat fetched data as readonly.

Sites can be described in various ways for the purpose of fetching and storing. The pseudovariable home is the thread's own site. Site variables point to sites. Their values can be computed based on a path from home or any other site, or based on absolute site coordinates.

Synchronization is sometimes needed among threads to prevent conflicts over local variables. For example, the sites may be arranged in a two-dimensional grid, and each site may need read access to local variables owned by adjacent sites. Canopy provides several alternative synchronization methods. First, the controller can choose to start only a subset of the sites during each phase. For example, the sites in the two-dimensional grid may be "colored" red or black as on a checkerboard. The controller can start only black sites during one phase, and then red sites in the next. Then each thread is assured that its neighbors are not active when it is. Second, the sites in a grid can be given priorities. A thread may call a synchronize routine specifying any site. This routine will block until that site has finished the current phase if it is of higher priority than the thread's own site. So the controller can start all the sites in the two-dimensional grid, but assign black sites higher priority than red sites. Each site will synchronize with its neighbors before fetching their local variables. The effect is that black sites will execute first, then red sites, but if the amount of computation varies across sites, some black sites may still be executing when red sites elsewhere are already in progress or even finished. Thus this technique allows greater parallelism than the first one. Since it is so useful, Canopy provides a synchronized version of the fetch routine that combines it with synchronize.

Good programming practice in Canopy suggests that a thread should only update local variables on its home site, and that if it updates a local variable, it should never read the same variable from another site that is currently active. This second rule is achieved by using synchronized fetch for such variables, but the faster ordinary fetch for variables that are not modified locally.

Canopy programmers must be careful with global variables of the underlying language, C. They can be used for readonly initialized data, but only if the value is broadcast by the controller before the phase starts. If a site writes into a global variable, the change is observable only by those sites that happen to be on the same machine. A thread that uses a global variable for communication between procedures runs the risk of having the variable overwritten by another site when the local machine chooses to suspend that thread to achieve synchronization or to batch cross-machine communication.

6 ◆ FINAL COMMENTS

Concurrent programming has been studied for at least twenty years, but it has been steadily gaining popularity. One reason is that high-performance computers have turned increasingly to parallelism as a way of achieving a high rate of computation. Another is that workstation clusters are increasingly common in research environments. The former trend has led to increased interest in threads that cooperate by shared variables; the latter makes message passing attractive. Operating systems are being designed that make shared variables meaningful across memories and that make message passing fast within a single memory, so the correspondence between physical architecture and programming language approach is not straightforward.

Languages that provide some modest extensions to successful sequential languages, such as ML, C++, or even FORTRAN, might be more successful in the long run than specialty languages, because they already have widespread use and are perhaps easier to learn than completely new languages. Concurrent C, Concurrent Pascal, and HPF (High Performance FORTRAN) extend standard imperative languages; CST (Concurrent Smalltalk), DC++, and Distributed Eiffel extend object-oriented languages.

High-level operations can go a long way toward efficient use of the underlying architecture without introducing concurrency explicitly into the language. For example, FORTRAN 90 specifies vector and matrix operations that a subroutine library may implement quite efficiently in a concurrent fashion. As another example, speculative evaluation in functional programming languages, as discussed in Chapter 4, can take advantage of implicit concurrency.

EXERCISES

Review Exercises

7.1 What is the usual initial value for the `Value` field in a semaphore?

7.2 Show a code fragment that, if executed by two threads, can leave the value of `x` either 0 or 14, depending on the order in which the two threads interleave their execution. Don't use any synchronization.

7.3 Show how to use each of the following methods to restrict a code fragment `C` so that it can only be executed by one thread at a time: semaphores, mutexes, conditional critical regions.

7.4 Make a deadlock situation with only one thread, using each of the following methods: semaphores, mutexes, conditional critical regions.

7.5 What will be the effect in a CSP program if I misspell a pattern in an input guard?

Challenge Exercises

7.6 On page 189, I say that arguments to `fork` are usually restricted to global procedures so that cactus stacks do not need to be built. What is the connection between using global procedures and cactus stacks?

7.7 Does Ada require cactus stacks?

7.8 What will be the effect of a semaphore whose `Value` field is initialized to 2 if it is used for mutual exclusion?

7.9 What would be the use of a semaphore whose `Value` field is initialized to -2 with two dummy threads initially enqueued on its `Waiters` field?

7.10 Show how to implement conditional critical regions using semaphores. You will need an indivisible **updown** statement that **up**s one semaphore and **down**s another, and **upall** which performs **up** until there are no more threads blocked on the semaphore.

7.11 Show how to implement a capacity-2 barrier using two semaphores. You may use different code for the two threads involved. Implement not only **meet**, but also **arrive** and **depart**.

7.12 Show how to build a multiple-producer, multiple-consumer bounded buffer using event counts and sequencers.

7.13 Figure 7.9 (page 197) shows a Mesa solution to the bounded buffers problem. It assumes that **signal** only awakens one waiter. Actually, Mesa provides only **broadcast**, not **signal**. Fix the code.

7.14 Show how to build semaphores with event counts and sequencers. The **up** and **down** operations should not require mutual exclusion.

7.15 I suggest on page 213 representing the type of a procedure as a string in order to implement type-secure RPC. What sort of type equivalence does this method represent?

7.16 What are the ramifications of using REV in an environment where each address space has several threads?

7.17 The CSP implementation of a bounded buffer Figure 7.21 (page 217) uses both input and output guards. Can bounded buffers be implemented without output guards? Without input guards? Without either?

7.18 On page 221, I suggest that a proper Linda implementation of the bounded buffer (one that does not use an intermediate thread to hold the data and is truly bounded) is possible. Show how. Hint: Use anti-data to indicate an available slot in the buffer.

7.19 Languages like Lynx, ALBA, and SR allow servers to handle messages either by rendezvous or by thread creation. Would it make sense to allow a single entry to be handled both ways?