PREV
7. Monitors

NEXT
9. Spaces

Chapter 8. Channels

The concurrent programming constructs that we have studied use shared memory, whether directly or through a shared service of an operating system. Now we turn to constructs that are based upon *communications*, in which processes send and receive *messages* to and from each other. We will study synchronization in progressively looser modes of communications: in this chapter, we discuss synchronization mechanisms appropriate for tightly coupled systems (channels, rendezvous and remote procedure call); then we discuss spaces which provide communications with persistence; and finally we will study algorithms that were developed for fully distributed systems. As always, we will be dealing with an abstraction and not with the underlying implementation. The interleaving model will continue to be used; absolute time will not be considered, only the relative order in which messages are sent and received by a process.

The chapter begins with an analysis of alternatives in the design of constructs for synchronization by communications. Sections 8.2–8.4 present algorithms using channels, the main topic of the chapter. The implementation of channels in Promela is discussed in Section 8.5. The rendezvous construct and its implementation in Ada is presented in Section 8.6. The final section gives a brief survey of remote procedure calls.

Models for Communications

Synchronous vs. Asynchronous Communications

Communications by its nature requires two processes—one to send a message and one to receive it—so we must specify the degree of

ready to send but the receiver is not ready to receive, the sender is blocked, and similarly, if the receiver is ready to receive before the sender is ready to send, the receiver is blocked. The act of communications synchronizes the execution sequences of the two processes. Alternatively, the sender can be allowed to send a message and continue without blocking. Communications are then *asynchronous* because there is no temporal dependence between the execution sequences of the two processes. The receiver could be executing any statement when a message is sent, and only later check the communications channel for messages.

Deciding on a scheme is based upon the capacity of the communications channel to buffer messages, and on the need for synchronization. In asynchronous communications, the sender may send many messages without the receiver removing them from the channel, so the channel must be able to buffer a potentially unlimited number of messages. Since any buffer is finite, eventually the sender will be blocked or messages will be discarded. Synchronous and asynchronous communications are familiar from telephone calls and email messages. A telephone call synchronizes the activities of the caller and the person who answers. If the persons cannot synchronize, busy signals and unanswered calls result. On the other hand, any number of messages may be sent by email, and the receiver may choose to check the incoming mail at any time. Of course, the capacity of an electronic mailbox is limited, and email is useless if you need immediate synchronization between the sender and the receiver.

The choice between synchronous and asynchronous communications also depends on the level of the implementation. Asynchronous communications requires a buffer for messages sent but not received, and this memory must be allocated somewhere, as must the code for managing the buffers. Therefore, asynchronous communications is usually implemented in software, rather than in hardware. Synchronous communications requires no support other than send and receive instructions that can be implemented in hardware (see Section 8.3).

Addressing

In order to originate a telephone call, the caller must know the telephone number, the *address*, of the receiver of the call. Addressing is asymmetric, because the receiver does not know the telephone number of the caller. (Caller identification is possible, though the caller may choose to block this option.) Email messages use symmetric addressing, because every message contains the address of the sender.

Symmetric addressing is preferred when the processes are cooperating on the solution of a problem, because the addresses can be fixed at compile time or during a configuration stage, leading to easier programming and more efficient execution. This type of addressing can also be implemented by the use of *channels*: rather than naming the processes, named channels are declared for use by a pair or a group of processes.

Asymmetric addressing is preferred for programming *client–server systems*. The client has to know the name of the service it requests, while the server can be programmed without knowledge of its future clients. If needed, the client identification must be passed dynamically as part of the message.

Finally, it is possible to pass messages without any addressing whatsoever! Matching on the message structure is used in place of addresses. In Chapter 9, we will study spaces in which senders broadcast messages with no address; the messages can be received by any process, even by processes that were not in existence when the message was sent.

Data Flow

A single act of communications can have data flowing in one direction or two. An email message causes data to flow in one direction only, so a reply requires a separate act of communications. A telephone call allows two-way communications.

Asynchronous communications necessarily uses one-way data flow: the sender sends the message and then continues without waiting for the receiver to accept the message. In synchronous communications, either one- or two-way data flow is possible. Passing a message on a one-way channel is extremely efficient, because the sender need not be blocked while the receiver processes the message. However, if the message does in fact need a reply, it may be more efficient to block the sender, rather than release it and later go through the overhead of synchronizing the two processes for the reply message.

CSP and Occam

Channels were introduced by C.A.R. Hoare in the CSP formalism [32], which has been extremely influential in the development of concurrent programming [55, 58]. The programming language occam is directly based upon CSP, as is much of the Promela language that we use in this book. For more information on CSP, occam and their software tools for programming and verification, see the websites listed in Appendix E.

Channels

A *channel* connects a sending process with a receiving process. Channels are *typed*, meaning that you must declare the type of the messages that can be sent on the channel. In this section, we discuss synchronous channels. The following algorithm shows how a producer and a consumer can be synchronized using a channel:

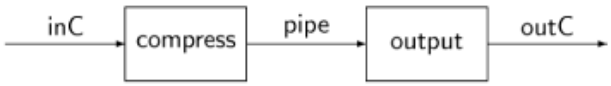
Table 8.1. Producer–consumer (channels)

```
                channel of integer ch

producer                        consumer

      integer x                       integer y
      loop forever                    loop forever
p1:     x ← produce             q1:     ch ⇒ y
p2:     ch ⇐ x                  q2:     consume(y)
```

The notation `ch ⇐ x` means that the value of `x` is sent on the channel `ch`, and similarly, `ch ⇒ y` means that the value of the message received from the channel is assigned to the variable `y`. The producer will attempt to send a message at `p2`, while the consumer will attempt to receive a message at `q1`. The data transfer will take place only after the control pointers of the two processes reach those points. Synchronous execution of the send and receive operations is considered to be a single change in state. So if $x = v$ in process `p`, then executing `ch ⇐ x` in `p` and `ch)y` in process `q` leads to a state in which $y = v$. Channels in operating systems are called *pipes*; they enable programs to be constructed by connecting an existing set of programs.

Pipelined computation using channels can be demonstrated using a variant of *Conway's problem* (Algorithm 8.2). The input to the algorithm is a sequence of characters sent by an environment process to an input channel; the output is the same sequence sent to an environment process on another channel after performing two transformations: (a) runs of $2 \le n \le 9$ occurrences of the same character are replaced by the digit corresponding to n and the character; (b) a newline character is appended following every K th character of the transformed sequence.

The algorithm is quite easy to follow because the two transformations are implemented in separate processes: the `compress` process replaces runs as required and sends the characters one by one on the `pipe` channel, while the `output` process takes care of inserting the newline characters:



Channels are efficient and easy to use but they lack flexibility. In particular, it is difficult to program a *server* using channels, because it is not possible for a server simply to export the interface of the services that it is offering; it must also export a specific set of channels. In other words, programs that use channels must be configured, either at compile time or when the program is initiated. In Section 8.6, we describe the rendezvous which is more flexible and appropriate for writing servers.

Table 8.2. Conway's problem

```
constant integer MAX ← 9
constant integer K ← 4
channel of integer inC, pipe, outC
```

compress	output

`char c, previous ← 0`
`integer n ← 0`
`inC ⇒ previous`
`loop forever`
`p1: inC ⇒ c`
`p2: if (c = previous) and (n < MAX-1)`
`p3: n ← n + 1`
	`char c`
	`integer m ← 0`
	`loop forever`
	`q1: pipe ⇒ c`
	`q2: outC ⇐ c`
	`q3: m ← m + 1`

```
        else
p4:        if n > 0                          q4:    if m >= K
p5:           pipe ⇐ intToChar(n+1)          q5:       outC ⇐ newline
p6:           n ← 0                          q6:       m ← 0
p7:        pipe ⇐ previous                   q7:
p8:        previous ← c                      q8:
```

Parallel Matrix Multiplication

One-way channels are extremely efficient, and can be implemented in hardware. There used to be a computing element called a *transputer*, which contained a CPU, memory and four pairs of one-way channels on a single chip. Synchronous communications was implemented through channel statements which could be directly compiled to machine-language instructions of the transputer. This facilitated the construction of arrays of processors that could be programmed to execute truly parallel programs. While the transputer no longer exists, its architecture has influenced that of modern chips, especially digital signal processors. Here we will demonstrate an algorithm that can be efficiently implemented on such systems.

Consider the problem of matrix multiplication:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 2 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 4 & 2 & 6 \\ 10 & 5 & 18 \\ 16 & 8 & 30 \end{bmatrix}$$

Each element of the resulting matrix can be computed independently of the other elements. For example, the element 30 in the bottom right corner of the matrix is obtained by the following computation, which can be performed before, after or simultaneously with the computation of other elements:

$$[7, 8, 9] \times \begin{bmatrix} 2 \\ 2 \\ 0 \end{bmatrix} = 7 \cdot 2 + 8 \cdot 2 + 9 \cdot 0 = 14 + 16 + 0 = 30.$$

We will implement matrix multiplication for 3 × 3 matrices using small numbers, although this algorithm is practical only for large matrices of large values, because of the overhead involved in the communications.

Figure 8.1 shows 21 processors connected so as to compute the matrix multiplication.[1] Each processor will execute a single process. The one-way channels are denoted by arrows. We will use geographical directions to denote the various neighbors of each processor.

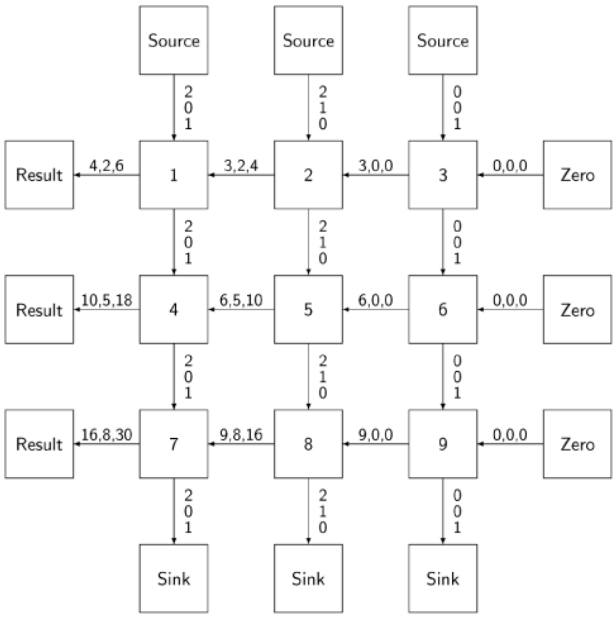
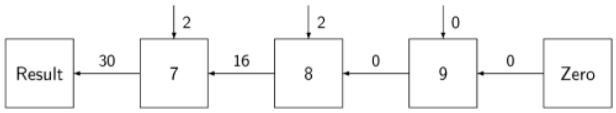


Figure 8.1. Process array for matrix multiplication

The central array of 3 × 3 processes multiplies pairs of elements and computes the partial sums. One process is assigned to each element of the first matrix and initialized with the value of that element. The `Source` processes are initialized with the row vectors of the second matrix and the elements are sent one by one to the `Multiplier` processes to the south; they in turn pass on the elements until they are finally caught in the `Sink` processes. (The `Sink` processes exist so that the programming of all the `Multiplier` processes can be uniform without a special case for the bottom row of the array.) A `Multiplier` process receives a partial sum from the east, and adds to it the result of multiplying its element and the value received from the north. The partial sums are initialized by the `Zero` processes and they are finally sent to the `Result` processes on the west.

Let us study the computation of one element of the result:



This is the computation of the corner element that we showed above as a mathematical equation. Since each process waits for its two input values before computing an output, we can read the diagram from right to left: first, 9 • 0 + 0 = 0, where the partial sum of 0 comes from the `Zero` process; then, 8 • 2 + 0 = 16, but now the the partial sum of 0 comes from the `Multiplier` process labeled `9`; finally, using this partial sum of 16, we get 7 • 2 + 16 = 30 which is passed to the `Result` process.

Except for the `Multiplier` processes, the algorithms for the other processes are trivial. A `Zero` process executes `West ⇐ 0` three times to initialize the `Sum` variables of a `Multiplier` process; a `Source` process executes `South⇐Vector[i]` for each of the three elements of the rows of the second matrix; a `Sink` process executes `North⇒dummy` three times and ignores the values received; a `Result` process executes `East⇒result` three times and prints or otherwise uses the values received. A `Result` process knows which

row and column the result belongs to: the row is determined by its position in the structure and the column by the order in which the values are received.

Here is the algorithm for a `Multiplier` process:

Table 8.3. Multiplier process with channels

```
integer FirstElement
channel of integer North, East, South, West
integer Sum, integer SecondElement
```

```
      loop forever
p1:       North ⇒ SecondElement
p2:       East ⇒ Sum
p3:       Sum ← Sum + FirstElement · SecondElement
p4:       South ⇐ SecondElement
p5:       West ⇐ Sum
```

Each process must be initialized with five parameters: the element `FirstElement` and channels that are appropriate for its position in the structure. This configuration can be quite language-specific and we do not consider it further here.

Selective Input

In all our examples so far, an input statement `ch⇒var` is for a single channel, and the statement blocks until a process executes an output statement on the corresponding channel. Languages with synchronous communication like CSP, Promela and Ada allow selective input, in which a process can attempt to communicate with more than one channel at the same time:

```
either
    ch1 ⇐ var1
or
    ch2 ⇐ var2
or
    ch3 ⇐ var3
```

Exactly one alternative of a selective input statement can succeed. When the statement is executed, if communications can take place on one or more channels, one of them is selected nondeterministically. Otherwise, the process blocks until the communications can succeed on one of the alternatives.

The matrix multiplication program can use selective input to take advantage of additional parallelism in the algorithm. Instead of waiting for input first from `North` and then from `East`, we can wait for the first available input and then wait for the other one:

Table 8.4. Multiplier with channels and selective input

```
integer FirstElement
channel of integer North, East, South, West
integer Sum, integer SecondElement
```

```
       loop forever
          either
p1:          North ⇒ SecondElement
p2:          East ⇒ Sum
          or
p3:          East ⇒ Sum
p4:          North ⇒ SecondElement
p5:       South ⇐ SecondElement
p6:       Sum ← Sum + FirstElement · SecondElement
p7:       West ⇒ Sum
```

Once one alternative has been selected, the rest of the statements in the alternative are executed normally, in this case, an input statement from the other channel. The use of selective input ensures that the processor to the east of this `Multiplier` is not blocked unnecessarily.

The Dining Philosophers with Channels

A natural solution for the dining philosophers problem is obtained by letting each fork be a process that is connected by a channel to the philosophers on its left and right.[2] There will be five `philosopher` processes and five `fork` processes, shown in the left and right columns of Algorithm 8.5, respectively. The `philosopher` processes start off by waiting for input on the two adjacent `forks` channels. Eventually, the `fork` processes will output values on their channels. The values are of no interest, so we use `boolean` values and just ignore them upon input. Note that once a `fork` process has output a value it is blocked until it receives input from the channel. This will only occur when the `philosopher` process that previously received a value on the channel returns a value after eating.

Table 8.5. Dining philosophers with channels

```
channel of boolean forks[ 5 ]
```

philosopher i

```
       boolean dummy
       loop forever
p1:      think
p2:      forks[i] ⇐ dummy
p3:      forks[i+1] ⇐ dummy
p4:      eat
p5:      forks[i] ⇒ true
p6:      forks[i+1] ⇒ true
```

fork i

```
       boolean dummy
       loop forever
q1:      forks[i] ⇐ true
q2:      forks[i] ⇒ dummy
q3:
q4:
q5:
q6:
```

Channels in Promela[L]

The support for channels in Promela is extensive. In addition to synchronous channels, asynchronous channels can be declared with any fixed capacity, although channels with a large capacity will give rise to an extremely large number of states, making it difficult to verify an algorithm. The message type of the channel can be declared to be of any type or sequence of types, and arrays of channels can be declared. Here is an example of the declaration of a single synchronous channel whose messages are integers, and an array of four asynchronous channels with capacity ten and a message type that is a sequence of an integer and a boolean value:

```
chan ch = [0] of { int }
chan charray[4] = [10] of { int, bool }
```

`ch ? v1, v2, . . .` denotes receiving a value from the channel `ch` and storing it in the variables `v1`, `v2`, . . ., and `ch ! val1, val2, . . .` denotes sending these values on the channel. A receive operation can be performed only if the message matches, that is, if the number and types of parameters match those of the message.

Listing 8.1 shows a solution to Conway's problem in Promela. Two additional processes are not shown: process `Generator` supplies input on channel `inC`, and process `Printer` displays the output from channel `outC`.

Example 8.1. A solution to Conway's problem in Promela

```
#define N 9
#define K 4
chan inC, pipe, outC = [0] of { byte };
active proctype Compress() {
  byte previous, c, count = 0;
  inC ? previous;
  do
  :: inC ? c ->
     if
    :: (c == previous) && (count < N-1) -> count++
    :: else ->
       if
       :: count > 0 ->
          pipe ! count+1;
          count = 0
       :: else
       fi;
       pipe ! previous;
       previous = c;
    fi
 od
}
active proctype Output() {
  byte c, count = 0;
  do
  :: pipe ? c;
     outC ! c;
     count++;
     if
     :: count >= K ->
        outC ! '\n';
        count = 0
     :: else
     fi
  od
}
```

Here is the `Multiplier` process from Algorithm 8.4 written in Promela:

```
proctype Multiplier (byte Coeff;
       chan North; chan East; chan South; chan West) {
   byte Sum, X;
   do ::
```

```
        if :: North ? X -> East ? Sum;
           :: East ? Sum -> North ? X;
        fi;
        South ! X;
        Sum = Sum + X*Coeff;
        West ! Sum;
    od
}
```

An **init** process is required to invoke **run** for each process with the appropriate channels. The semantics of the **if** statement are that it blocks until one of the guards becomes true, that is, until it is possible to complete a receive operation from either the `North` or `East` channel. Once the **if** statement commits to one of these options, it executes the receive statement for the other channel.

Promela also includes statements for examining if a channel is full or empty, for examining elements in the channel without removing them, and for sending and receiving messages in other than FIFO order.

Rendezvous

The name rendezvous invokes the image of two people who choose a place to meet; the first one to arrive must wait for the arrival of the second. In the metaphor, the two people are symmetric and the rendezvous place is neutral and passive. However, in the synchronization construct, the location of the rendezvous belongs to one of the processes, called the *accepting* process. The other process, the *calling* process, must know the identity of the accepting process and the identity of the rendezvous which is called an *entry* (the same term used in protected objects; see Section 7.10). The accepting process does not know and does not need to know the identity of the calling process, so the rendezvous is appropriate for implementing servers that export their services to all potential clients.

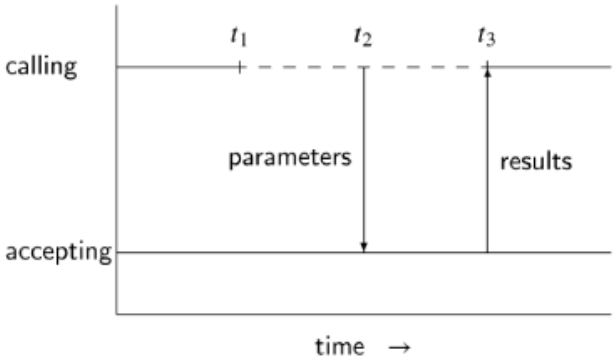
Algorithm 8.6 shows how a rendezvous is used to implement a client–server application. The `client` process calls the `service` entry of the `server` process supplying some parameters; it must now block until the `server` process is ready to `accept` the call. Then the rendezvous is performed using the parameters and possibly producing a result. Upon completion of the rendezvous, the result is returned to the `client` process. At the completion of the rendezvous, the `client` process is unblocked and the `server` process remains, of course, unblocked.

Table 8.6. Rendezvous

client	server

`integer parm, result`
`loop forever`
`p1: parm ← . . .`
`p2: server.service(parm, result)`
`p3: use(result)`
	`integer p, r`
	`loop forever`
	`q1:`
	`q2: accept service(p, r)`
	`q3: r ← do the service(p)`

The semantics of a rendezvous are illustrated in the following diagram:



At time t_1, the calling process is blocked pending acceptance of the call which occurs at t_2. At this time the parameters are transferred to the accepting process. The execution of the statements of the `accept` block by the accepting process (the interval t_2–t_3) is called the execution of the rendezvous. At time t_3, the rendezvous is complete, the results are returned to the calling process and both processes may continue executing. We leave it as an exercise to draw the timing diagram for the case where the accepting task tries to execute the `accept` statement before the calling task has called its entry.

The Rendezvous in Ada[L]

The rendezvous is one of the main synchronization constructs in the Ada language. Here we describe the construct using the standard example, the bounded buffer:

```
task body Buffer is
  B: Buffer_Array;
  In_Ptr, Out_Ptr, Count: Index := 0;
begin
 loop
   select
      when Count < Index'Last =>
       accept Append(I: in Integer) do
           B(In_Ptr) := I;
       end Append;
      Count := Count + 1; In_Ptr := In_Ptr + 1;
  or
      when Count > 0 =>
       accept Take(I: out Integer) do
          I := B(Out_Ptr);
       end Take;
      Count := Count - 1; Out_Ptr := Out_Ptr + 1;
  or
       terminate;
   end select;
 end loop;
end Buffer;
```

The `select` statement enables the buffer to choose nondeterministically between two *guarded* alternatives. The guards are boolean expressions prefixed the `accept` statements. If the expression evaluates to true, the alternative an *open* alternative and a rendezvous with the `accept` statement is permitted. If the expression evaluates to false, the alternative is *closed* and rendezvous is not permitted.

In the example, if the buffer is empty (*Count* = 0), the only open alternative is the `Append` entry, while if the buffer is full (*Count* = *N*), the only open alternative is `Take`. It is required that there always be

at least one open alternative; in the example this requirement holds because for a buffer of positive length it is impossible that *Count* = 0 and *Count* = *N* simultaneously.

If 0 < *Count* < *N*, both alternatives are open. If there are calling tasks waiting on both entries, the accepting task will choose arbitrarily between the entries and commence a rendezvous with the first task on the queue associated with the chosen entry. If only one entry queue is nonempty, the rendezvous will be with the first calling task on that queue. If both queues are empty, the accepting task will wait for the first task that calls an entry.

An Ada programmer would normally prefer to use a protected object rather than a task with rendezvous to implement a bounded buffer, because protected objects are passive and their statements are executed by the producer and consumer tasks that exist anyway. Here, there is an extra task and thus the extra overhead of context switches. This design would be appropriate if, for example, the task had complex processing to perform on the buffer such as writing parts of it to a disk.

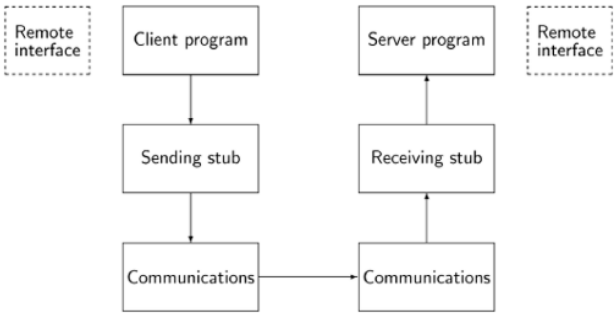
What happens to the buffer task if all producer and consumer tasks that could potentially call it terminate? The answer is that the program would deadlock with the `select` statement indefinitely waiting for an entry call. The `terminate` alternative (line 19) enables graceful shutdown of systems containing server tasks of this form. See [7] or [18] for details of this feature, as well as of other features that enable conditional rendezvous and timeouts.

Remote Procedure Calls[A]

Remote procedure call (RPC) is a construct that enables a client to request a service from a server that may be located on a different processor. The client calls a server in a manner no different from an ordinary procedure call; then, a process is created to handle the invocation. The process may be created on the same processor or on another one, but this is transparent to the client which invokes the procedure and waits for it to return. RPC is different from a rendezvous because the latter involves the active participation of two processes in synchronous communications.

RPC is supported in Ada by the constructs in the *Distributed Systems Annex* and in Java by the *Remote Method Invocation (RMI)* library. Alternatively, a system can implement a language-independent specification called *Common Object Request Broker Architecture (CORBA)*. Writing software for distributed systems using RPC is not too difficult, but it is language-specific and quite delicate. We will just give the underlying concepts here and refer you to language textbooks for the details.

To implement RPC, both the client and the server processes must be compiled with a *remote interface* containing common type and procedure declarations; the client process will invoke the procedures that are implemented in the server process. In Java, a remote interface is created by extending the library interface `java.rmi.Remote`. In Ada, packages are declared with **pragma** `Remote_Types` and **pragma** `Remote_Call_Interface`. The following diagram shows what happens when the client calls a procedure that is implemented in the server:



Since the procedure does not actually exist in the client process, the call is processed by a *stub*. The stub *marshals* the parameters, that is, it transforms the parameters from their internal format into a sequence of data elements that can be sent through the communications channel. When the remote call is received by the server, it is sent to another stub which *unmarshals* the parameters, transforming them back into the internal format expected by the language. Then it performs the call to the server program on behalf of the client. If there are return values, similar processing is performed: the values are marshaled, sent back to the client and unmarshaled.

To execute a program using RPC, processes must be assigned to processors, and the clients must be able to locate the services offered by the servers. In Java this is performed by a system called a *registry*; servers call the registry to bind the services they offer and the clients call the registry to lookup the services they need. In Ada, a configuration tool is used to allocate programs to *partitions* on the available processors. Again, the details are somewhat complex and we refer you to the language documentation.

Transition

Channels enable us to construct decentralized concurrent programs that do not necessarily share the same address space. Synchronous communication, where the sender and receiver wait for each other, is the basic form of synchronization, as it does not require design decisions about buffering. More complex forms of communication, the rendezvous in Ada and the remote procedure call implemented in many systems, are used for higher-level synchronization, and are especially suited to client–server architectures.

What is common to all the synchronization constructs studied so far is that they envision a set of processes executing more or less simultaneously, so it makes sense to talk about one process blocking while waiting for the execution of a statement in another process. The Linda model for concurrency discussed in the next chapter enables highly flexible programs to be written by replacing process-based synchronization with data-based synchronization.

Exercises

1. Develop an algorithm for pipeline sort. There are n processes and n numbers are fed into the input channel of the first process. When the program terminates, the numbers are stored in ascending order in the processes:

2. Develop a solution for the dining philosophers problem under the restriction that a channel must be connected to exactly one sender and one receiver.

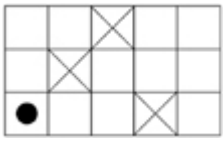
3. Develop an algorithm to merge two sequences of data. A process receives data on two input channels and interleaves the data on one output channel. Try to implement a *fair merge* that is free from starvation of both input channels.

4. Develop an algorithm to simulate a digital logic circuit. Each gate in the circuit will be represented by a process and wires between the gates will be represented by channels. You need to decide how to handle fan-out, when a single wire leading from one gate is connected to several other gates.

5. (Hamming's problem) Develop an algorithm whose output is the sequence of all multiples of 2, 3 and 5 in ascending order. The first elements of the sequence are: 0, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15. There will be four processes: one to multiply by each of the three factors and a fourth process to merge the results.

6. (Hoare [32]) Let $x_1, x_2, \ldots$ and $y_1, y_2, \ldots$ be two sequences of numbers. Develop an algorithm to compute the sequence $2x_1 + 3y_1, 2x_2 + 3y_2, \ldots$. The multiplications must be performed in parallel. Modify the program to compute $2x_1 + 3x_1, 2x_2 + 3x_2, \ldots$ by splitting the input sequence $x_1, x_2, \ldots$ into two identical sequences.

7. (Hoare [32]) Develop an algorithm to simulate the following game. A counter is placed on the lower lefthand square of a rectangular board, some of whose squares are blocked. The counter is required to move to the upper righthand square of the board by a sequence of moves either upward or to the right:



8. Draw a timing diagram similar to the one in Section 8.6 for the case where the accepting task in a rendezvous tries to execute the `accept` statement before the calling task has called its entry.

9. Suppose that an exception (runtime error) occurs during the execution of the server (accepting) process in a rendezvous. What is a reasonable action to take and why?

10. Compare the monitor solution of the producer–consumer problem in Java given in Section 7.11, with the rendezvous solution in Ada given in Section 8.6. In which solution is there a greater potential for parallel execution?

[1] In the general case, $n^2 + 4n$ processors are needed for multiplication of matrices of size $n \times n$.

[2] This works in Promela because a channel can be used by more than one pair of processes.