Mordechai Ben-Ari

Ada for Software Engineers

Second Edition with Ada 2005

Springer

Chapter 18
Concurrency

An Ada program contains one or more *tasks* that execute *concurrently*. (Tasks are called *processes* or *threads* in other languages; the distinctions are not important in this context.) We use the term concurrent in preference to "parallel" to emphasize that the parallelism is conceptual, not necessarily physical. A correct *multitasking* Ada program will produce the same result, whether it is run on a multiprocessor system or on a time-shared single processor, though the multiprocessor system will (hopefully) be significantly faster.

For convenience, the material on concurrency is divided into two chapters, with more advanced material in Chapter 19. Multitasking programs are frequently written for embedded computer systems where hardware interfaces and program performance are critical; the constructs in Ada that support embedded systems are discussed in Chapters 20–21.

Chapters 18–21 are not an introduction to concurrent programming, for which the reader is referred to the author's textbook [2]. The software archive accompanying that book contains implementations in Ada of many concurrent and distributed algorithms. More advanced textbooks are [4] on concurrent and real-time programming in Ada 2005, and [5] on algorithms for building real-time systems.

A task is like a subprogram because it has data declarations, a sequence of statements and exception handlers; the difference is that a *thread of control* is associated with each task. The thread is implemented using a data structure containing pointers to the task's current instruction and to local memory such as a stack segment. If each task is assigned a processor, the processors will execute the instructions of the tasks simultaneously. If there are more tasks than processors—at worst, if there is only one processor—a *scheduler* will assign processors to tasks according to some scheduling algorithm.

In Ada, there are three main constructs for writing concurrent programs:

- Load and store of shared variables (Section 20.7); these are usually too low-level.
- Protected objects for asynchronous sharing of resources.
- Rendezvous for direct task-to-task synchronous communication.

(See Sections 21.11–21.12 for other, low-level, constructs defined in the Real-Time Annex.)

The term *asynchronous* means that different tasks need not access the protected object at the same time. In fact, a task can insert data into a protected object and then terminate, while a second task later extracts the data from the object. The rendezvous is *synchronous* because both tasks must participate in the synchronization and communication at the same time. In the next two sections we will solve a simple problem, once using protected objects and once using rendezvous so that we can compare the constructs.

18.1 Tasks and protected objects

The problem that we will solve is called the *producer–consumer problem*. One or more tasks produce data elements which must be transferred to one or more consumer tasks. An example is a network interface that "produces" data downloaded from the net and a web browser that "consumes" the data. A *buffer* is used for data structure transformation (the data may arrive in large blocks, which must be stored in a data structure so that the browser can process one element at a time), and for flow control (the browser must be blocked if no data is currently available, and, similarly, the interface must not download data if the data structure is full).

18.1.1 Case study: producer–consumer (protected object)

The following program solves the producer–consumer problem. For simplicity, the data elements are integer values and the array data structure can hold 8 elements:

protectpc

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 procedure ProtectPC is
3   type Index is mod 8;
4   type Buffer_Array is array(Index) of Integer;
```

`Buffer` is a *protected object*. Syntactically, a protected unit (which can be either a single object, or a type that can be used to declare objects §9.4(1)) is like a package, with a declaration divided into a visible part and a private part, and a body §9.4(2–9).

```
5   protected Buffer is
6     entry Append(I: in  Integer);
7     entry Take  (I: out Integer);
8   private
9     B: Buffer_Array;
10    In_Ptr, Out_Ptr, Count:  Index := 0;
11  end Buffer;
```

A protected unit cannot contain type declarations, so the data types used to implement the buffer have been declared in the enclosing procedure. The visible part of the protected object contains the declaration of two *entries* `Append` and `Take`. The private part contains the declaration of the components belonging to the protected object. Components can only be declared in the private part, while operations such as entries can be declared anywhere in the protected unit declaration §9.4(4–6).

The bodies of the entries are contained within the body of the protected unit:

```
12  protected body Buffer is
13    entry Append(I: in  Integer) when Count < Index'Last is
14    begin
15      B(In_Ptr) := I;
16      Count := Count + 1;
17      In_Ptr := In_Ptr + 1;
18    end Append;
19
20    entry Take(I: out Integer) when Count > 0 is
21    begin
22      I := B(Out_Ptr);
23      Count := Count - 1;
24      Out_Ptr := Out_Ptr + 1;
25    end Take;
26  end Buffer;
```

The statements in the entries are the usual ones used to append data to a buffer and to take data from a buffer. The boolean-valued expressions after the word **when** are called *barriers* and are used to ensure that data will not be appended to a full buffer or taken from an empty buffer. The synchronization of calls to the entries and subprograms of a protected object is described in detail in the next subsection.

Protected objects are passive and just "sit there" waiting for their entries and subprograms to be called. Threads of control are associated with *tasks*. In this case study, there is one producer task which produces 200 integer values and appends them to the buffer:

```
27  task Producer;
28  task body Producer is
29  begin
30    for N in 1..200 loop
31      Put_Line("Producing " & Integer'Image(N));
32      Buffer.Append(N);
33    end loop;
34  end Producer;
```

A task body is syntactically like a procedure body §9.1(6). The task declaration ‡27 must be present even if it is empty §9.1(8).

For the consumers, we declare a task *type* so that more than one consumer can be declared:

```
35   task type Consumer(ID: Integer);
36   task body Consumer is
37     N: Integer;
38   begin
39     loop
40       Buffer.Take(N);
41       Put_Line(Integer'Image(ID) & " consuming " & Integer'Image(N));
42     end loop;
43   end Consumer;
```

The task type ‡35 has a discriminant §9.1(2,16), which is used for configuring the tasks with ID numbers when they are declared. The consumer tasks contain infinite loops: as long as there is data that can be taken from the buffer, they will do so. When the producer task terminates after appending 200 values to the buffer, the consumer tasks will be blocked.

We can now declare the two consumer tasks `C1` and `C2`:

```
44   C1: Consumer(1);
45   C2: Consumer(2);
46 begin
47   null;
48 end ProtectPC;
```

The main program has just the **`null`** statement, because all the execution is performed by the threads of control associated with the other three tasks. These tasks are activated just after the **`begin`** of the main subprogram, which must wait until the tasks have terminated. Since the consumer tasks do not terminate, make sure that you know how to break the execution of a program on your computer (usually `ctrl-c`) before running this example. See Section 19.1 for details on task activation and termination.

Task units and protected units are *not* compilation units; they must be declared within a compilation unit such as a subprogram or a package. However, a task or protected *body* can be separately compiled as a subunit §10.1.3(10).

18.1.2 Protected actions

How is a protected unit different from a package? The subprograms of a package can be called concurrently from multiple tasks §6.1(35), possibly leading to race conditions. The operations of a protected unit are *mutually exclusive*, meaning that only one will be executed at a time.

§9.5.1

> 4 A new protected action is not started on a protected object while another protected action on the same protected object is underway, unless both actions are the result of a call on a protected function. This rule is expressible in terms of the execution resource associated with the protected object:
> 5 • *Starting* a protected action on a protected object corresponds to *acquiring* the execution resource associated with the protected object, either for concurrent read-only access if the protected action is for a call on a protected function, or for exclusive read-write access otherwise;
> 6 • *Completing* the protected action corresponds to *releasing* the associated execution resource.

The statements of `Append` in the producer task and those of `Take` in a consumer task should not be executed simultaneously so that they will not try to update `Count` simultaneously. In fact, this will not occur, because one of the tasks will acquire the lock ("execution resource") granting it exclusive read-write access. The other task must wait until the entry body is complete and the lock released.

Functions allow multiple read-only access to the data of a protected object, provided that no subprogram or entry has access to the protected object. See Section 18.6 for more on protected subprograms.

In addition to mutual exclusion, a protected object can also provide flow control.

§9.5.3

> 7 • An entry of a protected object is open if the condition of the entry_barrier of the corresponding entry_body evaluates to True; otherwise it is closed. . . .
> 8 For the execution of an entry_call_statement, evaluation of the name and of the parameter associations is as for a subprogram call (see 6.4). The entry call is then *issued*: For a call on an entry of a protected object, a new protected action is started on the object (see 9.5.1). The named entry is checked to see if it is open; if open, the entry call is said to be *selected immediately*, and the execution of the call proceeds as follows:
> 10 • For a call on an open entry of a protected object, the corresponding entry_body is executed (see 9.5.2) as part of the protected action.
> 12 If the named entry is closed, the entry call is added to an *entry queue* (as part of the protected action, for a call on a protected entry), and the call remains queued until it is selected or cancelled; there is a separate (logical) entry queue for each entry of a given task or protected object (including each entry of an entry family).

The barrier **when** `Count > 0` ‡20 closes the `Take` entry when the buffer is empty; a consumer calling `Take` will be enqueued on the entry queue for `Take`. Similarly, the barrier **when** `Count < Index'Last` ‡13 closes the `Append` entry when the buffer is full. Suppose that the buffer is empty and that one or more calls from consumer tasks are enqueued on the queue

for Take. In this state, only the producer will now succeed in passing its barrier and commencing the execution of its protected action Append.

Since the completion of a protected operation can potentially change the value of a barrier, the barriers are reevaluated, so that if one of them is now open, its entry queue can be serviced:

§9.5.3

> 13 When a queued call is *selected*, it is removed from its entry queue. Selecting a queued call from a particular entry queue is called *servicing* the entry queue. An entry with queued calls can be serviced under the following circumstances:
> 15 • If after performing, as part of a protected action on the associated protected object, an operation on the object, other than a call on a protected function, the entry is checked and found to be open.

To continue with the example, when Append completes, the value of Count is now 1, the barrier **when** Count > 0 evaluates to true, and the entry queue for the Take operation will be serviced as part of the same protected action. Count will become 0 again, closing the barrier, so that additional calls from consumer tasks that are on the queue remain blocked. The protected action is now completed.

18.1.3 Preference for servicing queues

Consider the following scenario, where we assume that there are multiple producer tasks. First, a consumer task attempts to Take an element from an empty buffer and blocks on the entry queue; then, several producers attempt to Append elements to the buffer. One will acquire the lock and be allowed to execute the entry, appending its element to the buffer. When the entry body is completed, there are two ways to continue:

- Another producer can be allowed into the entry body to append its element.
- The queue can be serviced so that a call from the enqueued consumer can take the newly appended element.

§9.5.3

> 18 For a protected object, the above servicing of entry queues continues until there are no open entries with queued calls, at which point the protected action completes.

An entry call (here from a producer) will not begin a new protected action until the ongoing protected action is completed; in other words, there is a *preference* for servicing calls already enqueued on an entry queue.

Figure 18.1 shows how protected objects should be viewed: an outer shell protecting access to the resources, and an inner set of operations and entry queues. Entry calls, represented by

parallelograms with arrows, may be in one of three places: (i) executing an entry body (such as `Append`), (ii) blocked in an entry queue (the one for `Take`), or (iii) attempting to enter the protected object. Calls that have already passed the outer shell are considered part of the "club" and have preference over calls that have not yet been commenced.

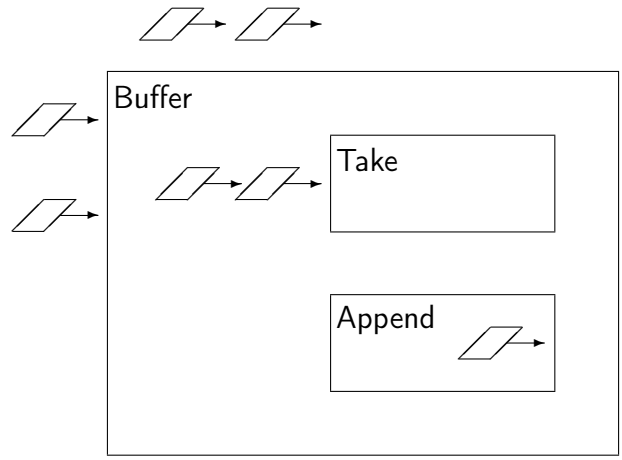


Fig. 18.1 Protected object

The preference for serving entry queues implements *immediate resumption* of blocked tasks. This is done for two reasons:

- If one task modifies a variable that will open the barrier for a blocked task, the awakened task can assume that no third task will intervene and change the state. Therefore, the condition need not be checked again.
- Blocked tasks will not be starved by a stream of new entry calls.

There is no queue associated with the mutual exclusion on the access to the protected object itself. This will not be a problem if protected entries and subprograms are kept very short, so that a call either quickly executes the entry body or is quickly enqueued because its barrier is false; in either case the mutual exclusion is released.

Language Comparison

A protected object is similar to a Java class, all of whose methods are **`synchronized`**. A synchronized method cannot be executed unless the caller has obtained the lock associated with an object, just as the execution of a protected entry or subprogram requires that the calling task obtain the "associated execution resource." In Java, however, there are no queues, no barriers and hence no immediate resumption of an unblocked process. `java.util.concurrent` (added to version 5 of Java) is an extensive library of constructs for implementing synchronization of concurrent programs.

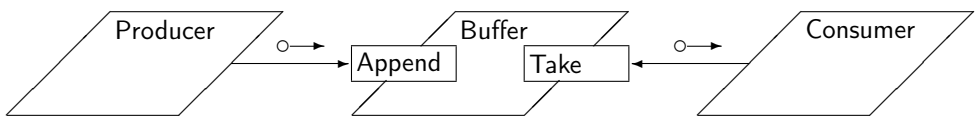
18.2 Rendezvous

Let us examine more closely the body for the entry `Append`:

```
entry Append(I: in  Integer) when Count < Index'Last is
begin
  B(In_Ptr) := I;
  Count := Count + 1;
  In_Ptr := In_Ptr + 1;
end Append;
```

The parameter `I` and the assignment of its value to a component of the buffer `B` are used to communicate—to pass data—between the calling task and the protected object. But incrementing the values of `Count` and `In_Ptr` is solely concerned with updating the internal data structure of the protected object. Nevertheless, the calling task is responsible for executing these statements, so there is less potential for concurrent execution. There would be no point in utilizing the extra concurrency for two short instructions, but one can imagine that the buffer is stored in a data structure that requires significant internal processing between insertions and extractions.

Additional concurrency can be implemented by making the buffer itself a task:



Synchronization and communication is done directly with the buffer task. The producer and consumer both initiate calls on entries of the buffer task, but in the case of the consumer, the direction of the data flow (denoted by the small arrow) is opposite the direction of the call.

18.2.1 Case study: producer–consumer (rendezvous)

The following program solves the producer–consumer problem using a buffer task, whose declaration contains the declaration of the two entries Append and Take:

taskpc

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  procedure TaskPC is
3    type Index is mod 8;
4    type Buffer_Array is array(Index) of Integer;
5
6    task Buffer is
7      entry Append(I: in  Integer);
8      entry Take  (I: out Integer);
9    end Buffer;
```

Unlike protected units, the declaration of a task can contain only entries and representation clauses §13.1(2), even in the private part §9.1(4–5).

A task is an active entity and its body contains declarations and a sequence of statements just like a procedure. The body of the task consists of a single nonterminating loop statement:

```
10   task body Buffer is
11     B: Buffer_Array;
12     In_Ptr, Out_Ptr, Count:  Index := 0;
13   begin
14     loop
15       select
16         when Count < Index'Last =>
17           accept Append(I: in Integer) do
18             B(In_Ptr) := I;
19           end Append;
20         Count := Count + 1;
21         In_Ptr := In_Ptr + 1;
22       or
23         when Count > 0 =>
24           accept Take(I: out Integer) do
25             I := B(Out_Ptr);
26           end Take;
27         Count := Count - 1;
28         Out_Ptr := Out_Ptr + 1;
29       end select;
30     end loop;
31   end Buffer;
```

Synchronization and communication with the buffer task are done using a selective accept statement that is described later in this section.

The producer and consumer tasks are unchanged from the previous solution.

18.2.2 Accept statements

The task `Producer` calls the entry `Append` of the task `Buffer`. The calling task and the called task must execute a *rendezvous* at an accept_statement.

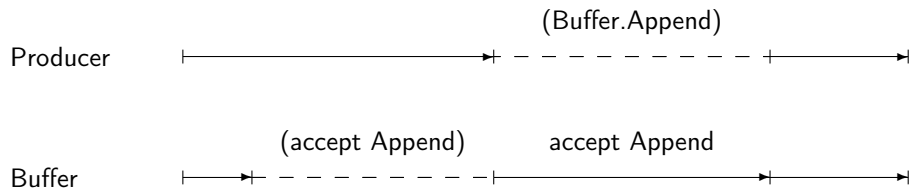
§9.5.2

> 3 accept_statement ::=
> **accept** *entry*_direct_name [(entry_index)] parameter_profile [**do**
> handled_sequence_of_statements
> **end** [*entry*_identifier]];
>
> 24 For the execution of an accept_statement, the entry_index, if any, is first evaluated and converted to the entry index subtype; this index value identifies which entry of the family is to be accepted. Further execution of the accept_statement is then blocked until a caller of the corresponding entry is selected (see 9.5.3), whereupon the handled_sequence_of_statements, if any, of the accept_statement is executed, with the formal parameters associated with the corresponding actual parameters of the selected entry call. Upon completion of the handled_sequence_of_statements, the accept_statement completes and is left. . . .
>
> 25 The above interaction between a calling task and an accepting task is called a *rendezvous*. After a rendezvous, the two tasks continue their execution independently.

(Entry indices are discussed in Section 18.5).

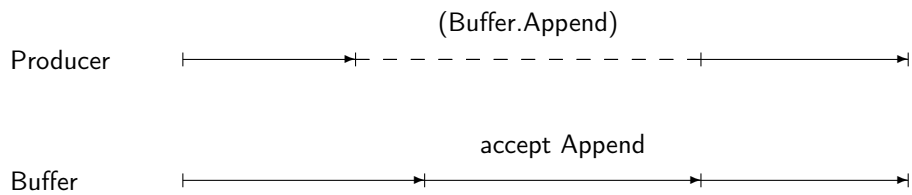
The basic principle of a rendezvous is that the first party to reach the rendezvous point must wait until the second party arrives. The semantics of a rendezvous are illustrated by the time lines below, where solid lines indicate intervals during which the process is ready or executing and dashed lines indicate intervals when the process is blocked on the statement written within parentheses.

Let us ignore the selective_accept statement for now and consider the following diagram, where task `Buffer` executes until it reaches the accept statement for `Append` ‡17, at which point it is blocked:



The task `Producer` executes concurrently until it reaches the entry call `Buffer.Append`. Now the producer is blocked and the buffer executes the sequence of statements within the accept statement. When the rendezvous is completed, both tasks are made ready; in a single-processor system, the scheduler will have to choose one of them.

The following diagram shows another possibility. Here the producer task blocks when it calls the entry, because the buffer task has not yet reached the accept statement. The producer task is made ready again only when the accept statement has been completed.



Before commencing the rendezvous, **`in`** and **`in out`** parameters are transferred to the accepting task; upon completion, **`out`** and **`in out`** parameters are transferred back to the calling task. Data transfer is bidirectional even though the entry call is unidirectional. Furthermore, the call is asymmetrical: the calling task knows the name of the accepting task but not conversely.

When the rendezvous is complete, both tasks can proceed independently, so the producer is not blocked while the internal data structure is updated ‡20–21. The increased concurrency has been obtained at the price of additional overhead associated with the extra task, and additional *context switches* to block the caller and then resume it.

18.2.3 Selective accept

Suppose that the task `Buffer` arrives at the rendezvous, but neither producers nor consumers have yet issued a call. We would like `Buffer` to serve the *first* task that calls one of its entries; however, a task body is just a sequence of statements that have to be executed one after another. If we had written:

```
loop
  accept Append do ...
  accept Take   do ...
end loop;
```

the `Buffer` would block pending a call from a producer, even if there were waiting consumers and data in the buffer.

The selective accept statement enables a task to carry out a rendezvous with any waiting calling task, or, if there are none, to wait simultaneously for calls to multiple entries.

§9.7.1

```
2  selective_accept ::=
       select
          [guard] select_alternative
       { or
          [guard] select_alternative }
       [ else
          sequence_of_statements ]
       end select;
3  guard ::= when condition =>
4  select_alternative  ::= accept_alternative | delay_alternative | terminate_alternative
5  accept_alternative ::= accept_statement [sequence_of_statements]
```

(The delay_alternative, terminate_alternative, and the **else**-part are discussed in Section 19.3.)

The semantics of the selective accept statement are as follows:

§9.7.1

> 14 A select_alternative is said to be *open* if it is not immediately preceded by a guard, or if the condition of its guard evaluates to True. It is said to be *closed* otherwise.
>
> 15 For the execution of a selective_accept, any guard conditions are evaluated; open alternatives are thus determined. . . . Selection and execution of one open alternative, or of the else part, then completes the execution of the selective_accept; the rules for this selection are described below.

The entry `Append` is guarded by `Count < Index'Last` ‡16, which is true only if the buffer is not full; if so, the alternative is open. Similarly, the entry `Take` is guarded by `Count > 0` ‡23, which is true only if the buffer is not empty; if so, the alternative is open. If the buffer is neither full nor empty, both guards are true and both alternatives are open.

§9.7.1

> 21 The exception Program_Error is raised if all alternatives are closed and there is no else part.

Show that it is impossible for both alternatives to be closed; therefore, `Program_Error` will not be raised in this program.

Once the set of open alternatives is determined, the execution of the selective_accept statement proceeds as follows:

§9.7.1

> 16 Open accept_alternatives are first considered. Selection of one such alternative takes place immediately if the corresponding entry already has queued calls. If several alternatives can thus be selected, one of them is selected according to the entry queuing policy in effect (see 9.5.3 and D.4). When such an alternative is selected, the selected call is removed from its entry queue and the handled_sequence_of_statements (if any) of the corresponding accept_statement is executed; after the rendezvous completes any subsequent sequence_of_statements of the alternative is executed. If no selection is immediately possible (in the above sense), and there is no else part, the task blocks until an open alternative can be selected.

The buffer task will select one of the open alternatives with enqueued calls, if any, and perform a rendezvous. If there are no enqueued calls it will block, waiting for either a producer or a consumer to call an entry. If one alternative is closed (say, the buffer is full, so the `Append` alternative is closed), the buffer task will rendezvous with a task blocked on the entry for the open alternative `Take`, if any, or it will block pending an entry call on that open alternative. A new call by a producer to `Append` will be ignored because the alternative is closed.

The default entry queuing policy is `FIFO_Queuing` §D.4(7), so if there are two consumer tasks waiting on the entry `Take`, they will be accepted in the order that they arrived. Other queuing policies are discussed in Section 21.5.

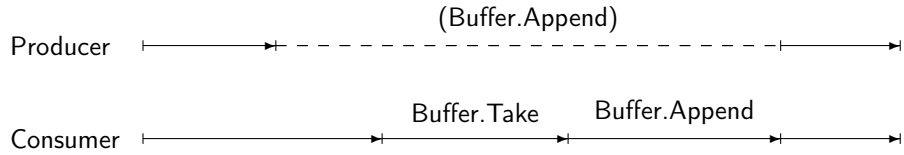
18.3 Implementation of entry calls

Paragraph §9.5.3(13) (see page 350) talks about servicing a queued *entry call*, not a queued *task*.

§9.5.3

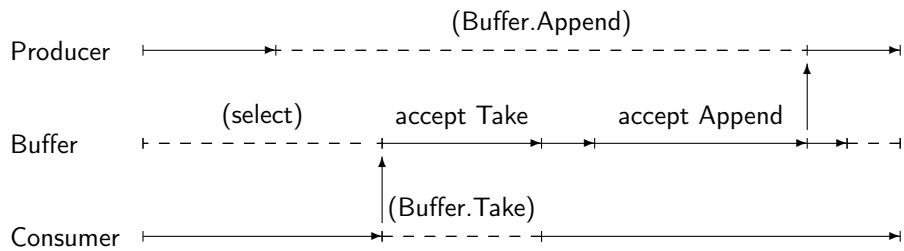
> 22 An implementation may perform the sequence of steps of a protected action using any thread of control; it need not be that of the task that started the protected action. ...

Assume that the producer in the program using protected objects attempts to append an item to a full buffer. The entry call `Buffer.Append` will be enqueued, the producer task will be blocked and a consumer task will be allowed to execute. Eventually, the consumer task will take an item from the buffer. Upon completion of the body of `Take`, servicing of the entry queue can be done by the *consumer task*, as shown in the following diagram:



In effect, appending an item is executed by the consumer on behalf of the producer. Upon completion of the entry call, both tasks become ready.

In contrast, a rendezvous normally requires context switches:



Assume that the buffer task is blocked on the selective accept statement and that the alternative for `Append` is closed because the buffer is full. The producer task blocks on the entry call `Buffer.Append`. When the consumer task executes the call `Buffer.Take`, the rendezvous takes place, removing an item from the buffer. When the selective accept is executed again, the alternative for the entry `Buffer.Append` is now open and the entry call can be accepted. Two extra context switches are needed (vertical arrows): one for the buffer task to execute its accept statement and another to switch either to the producer or to the consumer task.

18.4 Case study: synchronization with rendezvous and protected objects

The concurrent program presented in this section employs both rendezvous and protected objects for synchronization. The case study will be used to present important constructs in Ada tasking: entry families, the requeue statement and the abort statement. The problem is to implement a synchronization scheme described by the following story:

> The CEO (Chief Executive Officer) of a company likes to play golf. He does not allow himself to be interrupted by single employees with problems; instead, they must form themselves into groups before coming to consult him. The size of the group depends on the department to which the employee belongs: engineering, marketing, finance. A waiting group from finance has precedence over a group from marketing, which has precedence over an engineering group.

Let us look first at the structure of the program (Figure 18.2). There will be one task for the CEO and one task for each employee: engineers, salespersons in the marketing department,

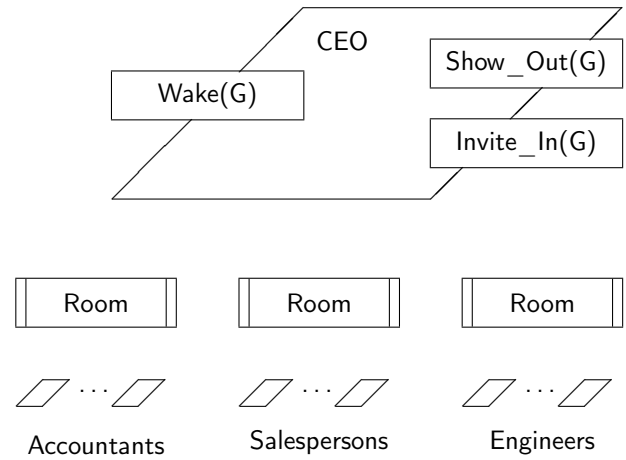


Fig. 18.2 The CEO program

and accountants in the finance department. These are declared as task types, and the tasks themselves are dynamically allocated in the main subprogram. Protected objects of type `Room` are used to synchronize the groups.

The program is contained within a main procedure; we start with the global declarations:

ceot

```
1  pragma Queuing_Policy(Priority_Queuing);
2  with Ada.Text_IO; use Ada.Text_IO;
3  procedure CEOT is
4    type Departments is (Engineering, Finance, Marketing);
5    type ID_Numbers  is range 0..10;
6    Group_Size: constant array(Departments) of
7                  ID_Numbers range 2..ID_Numbers'Last := (
8                    Engineering => 5, Finance => 3, Marketing => 2);
```

The enumeration type `Departments` declares the groups and the integer type `ID_Numbers` is used for the size of the groups. **pragma** `Queuing_Policy` is discussed Section 21.5.

The declaration of the CEO task contains three entry families: `Wake` to awaken the CEO task, `Invite_In`, which each employee task calls to enter the CEO's office, and `Show_Out`, upon which employee tasks block until the consultation is finished:

```
9    task CEO is
10     entry Wake(Departments);
11     entry Invite_In(Departments)(ID: ID_Numbers);
12     entry Show_Out (Departments)(ID: ID_Numbers);
13   end CEO;
```

Groups are synchronized in a waiting room that is implemented by a protected type. Its body is placed in a subunit (Section 15.3) which will be explained later:

```
14  type Door_State is (Open, Closed);
15  protected type Room(Department: Departments; Size: ID_Numbers) is
16    entry Register;
17    procedure Open_Door;
18  private
19    entry Wait_for_Last_Member;
20    Waiting:   ID_Numbers := 0;
21    Entrance:  Door_State := Open;
22    Exit_Door: Door_State := Closed;
23  end Room;
24  protected body Room is separate;
```

The protected type has two discriminants that are used to specify the department and the size of the group. Entry `Register` is called by an employee task wishing to join a group. Once the correct number of employees for a group of this department is in the room, the `Entrance` is closed and the group waits for the CEO to receive them. Procedure `Open_Door` is called by the CEO to reset `Entrance` to True so that a new group can register.

The protected type has a private entry `Wait_for_Last_Member` that is used to block registering tasks until the last member of the group has arrived, at which point `Exit_Door` will be opened. The component `Waiting` counts the number of waiting employee tasks.

The room for each department is an object of the protected type `Room`:

```
25  Engineering_Room: Room(Engineering, Group_Size(Engineering));
26  Finance_Room:     Room(Finance,     Group_Size(Finance));
27  Marketing_Room:   Room(Marketing,   Group_Size(Marketing));
```

The task body for the CEO is straightforward, consisting of a selective accept with an alternative for each of the three departments. The body of the accept statement for `Wake` is empty because it serves simply as a synchronization point; no processing is done and no data is exchanged with the caller:

```
28  task body CEO is
29    I: ID_Numbers;
30  begin
31    loop
32      select
33        accept Wake(Finance);
```

The sequence of statements following the accept statement implements the CEO's algorithm: the employees are invited in, consult with the CEO and are then shown out. The syntax of the selective accept statement is such that a (possibly guarded) accept statement must *immediately*

follow **select** and each **or**, but an arbitrary sequence of statements may be included in the alternative following the **accept** §9.7.1(5–6):

```
34          for N in 1..Group_Size(Finance) loop
35            accept Invite_In(Finance)(ID: ID_Numbers) do
36              I := ID;
37            end Invite_In;
38          end loop
39          -- Consult
40          for N in 1..Group_Size(Finance) loop
41            accept Show_Out(Finance)(ID: ID_Numbers) do
42              I := ID;
43            end Show_Out;
44          end loop;
45          Finance_Room.Open_Door;
```

For the other groups, the guards ‡47,51 in the selective accept statement use the `Count` attribute: `E'Count` gives the number of tasks currently waiting in the queue for entry `E` §9.9(4–5):

```
46        or
47          when Wake(Finance)'Count = 0 =>
48            accept Wake(Marketing);
49            -- As above with Marketing replacing Finance
50        or
51          when Wake(Finance)'Count = 0 and Wake(Marketing)'Count = 0 =>
52            accept Wake(Engineering);
53            -- As above with Engineering replacing Finance
54        or
55          terminate;
56        end select;
57      end loop;
58    end CEO;
```

This is used to implement the precedence requirement: if the CEO finds that more than one group is attempting to wake him, he will rendezvous with `Wake(Marketing)` only if there are no tasks waiting in the queue for `Wake(Finance)`. The guard for `Wake(Engineering)` is correspondingly more complex. (There is a subtle race condition that prevents the precedence requirement from being fulfilled without the use of **pragma** `Queuing_Policy` as explained in Section 21.5.)

The employee tasks are very simple: an employee who needs to consult with the CEO registers at the department waiting room, waits until invited in and then consults with the CEO until shown out. Each task declaration has a discriminant, which is used to give the task its ID number. Delay statements of different durations are used to introduce some asymmetry into the execution of the program:

```
59   task type Finance_Task(ID: ID_Numbers);
60   task body Finance_Task is
61   begin
62     loop
63       delay 4.0;
64       Finance_Room.Register;
65       CEO.Invite_In(Finance)(ID);
66       CEO.Show_Out(Finance)(ID);
67     end loop;
68   end Finance_Task;
69
70   task type Marketing_Task(ID: ID_Numbers);
71   task body Marketing_Task is
72     -- As above with Marketing replacing Finance
73   task type Engineer_Task(ID: ID_Numbers);
74   task body Engineer_Task is
75     -- As above with Engineering replacing Finance
```

The tasks are allocated dynamically in loops so that each employee receives a distinct ID as a discriminant constraint in the allocator.

```
76   type Finance_Ptr   is access Finance_Task;
77   type Marketing_Ptr is access Marketing_Task;
78   type Engineer_Ptr  is access Engineer_Task;
79   Accountants:       array(1..5) of Finance_Ptr;
80   Salespersons:      array(1..8) of Marketing_Ptr;
81   Engineers:         array(1..7) of Engineer_Ptr;
82 begin
83   for I in Accountants'Range loop
84     Accountants(I) := new Finance_Task(ID_Numbers(I));
85   end loop;
86   -- Similarly for Salespersons and Engineers
```

Accesses to the tasks are stored in arrays so that the employees can be fired by using the **abort** statement. The CEO task will terminate because it has a **terminate** alternative ‡55. These constructs are explained in Section 19.1.

```
87   delay 15.0;
88   for I in Engineers'Range loop
89     abort Engineers(I).all;
90   end loop;
91   -- Similarly for Salepersons and Accountants
92 end CEOT;
```

18.5 Entry families

Entry families are used to declare sets of related entries:

```
task CEO is
  entry Wake(Departments);
  entry Invite_In(Departments)(ID: ID_Numbers);
  entry Show_Out(Departments)(ID:  ID_Numbers);
end CEO;
```

§9.5.2

2 entry_declaration ::=
    [overriding_indicator]
    entry defining_identifier [(discrete_subtype_definition)] parameter_profile;

(On overriding indicators see Section 8.6; overriding of entries is discussed in Section 17.4.)

§9.5.2

20 An entry_declaration with a discrete_subtype_definition (see 3.6) declares a *family* of distinct entries having the same profile, with one such entry for each value of the *entry index subtype* defined by the discrete_subtype_definition. A name for an entry of a family takes the form of an indexed_component, where the prefix denotes the entry_declaration for the family, and the index value identifies the entry within the family. The term *single entry* is used to refer to any entry other than an entry of an entry family.

Each entry of the family is a distinct entry with its own queue. Entry families are somewhat like arrays of entries: calls and accept statements for an entry of a family use indexed notation as shown, for example, in ‡33, 35, 41, 65, 66. The index need not be constant, but it is not possible to write an accept statement that will wait simultaneously on all the entries of a family. If `D` is a variable, **accept** `Wake(D)` waits for an entry call on the queue corresponding to the current value of `D`. This construct is used to perform a rendezvous sequentially with each member of the family ‡34–44. Of course you would only use this construct if you knew that each entry would actually be called; otherwise, the program could deadlock. See Section 19.3 for other polling techniques, and Section 18.8 for the rules for entry families of protected objects.

18.6 Protected subprograms

Let us now study the body of the protected object `Room`. The CEO task calls the *protected procedure* `Open_Door` to indicate that the waiting room for this group can be re-opened:

```
98  separate(CEOT)
99  protected body Room is
100   procedure Open_Door is
101   begin
102     Entrance := Open;
103   end Open_Door;
```

§9.5.1

> 2 Within the body of a protected function (or a function declared immediately within a protected_body), the current instance of the enclosing protected unit is defined to be a constant (that is, its subcomponents may be read but not updated). Within the body of a protected procedure (or a procedure declared immediately within a protected_body), and within an entry_body, the current instance is defined to be a variable (updating is permitted).

Unlike entries, protected procedures have no queues associated with them and a call is not blocked once it passes the outer exclusion "shell" of the protected object.

Several tasks can execute protected functions concurrently provided that no other task is executing a protected procedure or entry §9.5.1(4–5). The current instance of a protected unit is a constant §9.5.1(2), so a function cannot modifying its components and no race conditions can result. Protected functions are used to return a value that depends on the private components:

```
function Crowded return Boolean is
begin
  return Waiting >= Group_Size / 2;
end Crowded;
```

An access to a protected subprogram §3.10(11) can be declared.

18.7 The requeue statement

As employee tasks call the entry `Register` to enter the `Room`, the calls will be enqueued upon the private entry `Wait_for_Last_Member` until the last member of the group has entered. At that point, it closes the `Entrance` door and opens the `Exit_Door`:

```
108     entry Register when Entrance = Open is
109     begin
110       Waiting := Waiting + 1;
111       if Waiting < Size then
112         requeue Wait_for_Last_Member with abort;
113       else
114         Waiting := Waiting - 1;
115         Entrance  := Closed;
116         Exit_Door := Open;
117       end if;
118     end Register;
```

The barrier of `Wait_for_Last_Member` is initially closed so that the tasks will be blocked until the last task of the group has registered and opened the barrier:

```
119     entry Wait_for_Last_Member when Exit_Door = Open is
120     begin
121       Waiting := Waiting - 1;
122       if Waiting = 0 then
123         Exit_Door := Closed;
124         requeue CEO.Wake(Department) with abort;
125       end if;
126     end Wait_for_Last_Member;
127   end Room;
```

The implementation of the protected object `Room` is based on the concept of *cascaded wakeup* and uses the **requeue** statement to move calling tasks from one entry to another. Initially, tasks calling `Register` are requeued on the entry `Wait_for_Last_Member` ‡112. The barrier of this entry `Exit_Door=Open` is false, so the calls will be enqueued. When the last member of the group executes `Register`, it will set `Exit_Door` to `Open` ‡116, making the barrier of `Wait_for_Last_Member` true. As each task completes its execution of this entry, the barrier is re-evaluated and since it remains true, another waiting task will be unblocked. This cascade of awakened tasks continues until all tasks enqueued on `Wait_for_Last_Member` are released and their protected actions completed. The last released task will close `Exit_Door` ‡123 and awaken the CEO by requeuing itself on the entry `CEO.Wake` for its department ‡124.

§9.5.4

2 requeue_statement ::= **requeue** *entry*_name [**with abort**];

8 For the execution of a requeue on an entry of a target task, after leaving the enclosing callable construct, the named entry is checked to see if it is open and the requeued call is either selected immediately or queued, as for a normal entry call (see 9.5.3).
9 For the execution of a requeue on an entry of a target protected object, after leaving the enclosing callable construct:
10 • if the requeue is an internal requeue (that is, the requeue is back on an entry of the same protected object—see 9.5), the call is added to the queue of the named entry and the ongoing protected action continues (see 9.5.1);

(**with abort** is discussed in Section 19.1.3.)

`Room` contains examples of requeue on both task entries ‡124 and protected entries ‡112.

Closing the entrance door ‡115 closes the barrier `Entrance=Open` of entry `Register` and prevents other employees of the same type from overtaking the group that has been formed.

Requeue is also essential to avoid overtaking when awakening the CEO. Suppose that a group of accountants has been formed, that is, the last task of the group has completed `Wait_for_Last_Member`. Without requeue, the last accountant task would have to complete the protected action and then call the entry `CEO.Wake(Finance)` from within the sequence of statements of its task body:

```
Finance_Room.Register;
if I_Am_Last_Member then
  CEO.Wake(Finance);
end if;
```

This task could be preempted—and a group of engineers could be formed and enqueued on `CEO.Wake(Engineering)`—*after* the protected action `Finance_Room.Register` completes, but *before* the call to `CEO.Wake(Finance)` is issued. With requeue, the protected action of the last accountant task is not *completed* until the task entry call has been made and immediately selected or enqueued. If it is enqueued, the guards on the accept statements of the selective accept prevent overtaking.

18.8 Additional rules for protected types*

The following subsections present addition rules for protected types.

18.8.1 Formal parameters in barriers

§9.5.2

> 18 A name that denotes a formal parameter of an entry_body is not allowed within the entry_barrier of the entry_body.

In the protected object `Buffer`, we cannot refuse to append negative numbers by writing:

```
entry Append(I: in  Integer)
  when Count < Index'Last and I >= 0 is
    -- Error, the barrier cannot use the formal parameter I
```

The reason is that all calls on the queue for an entry are considered to be waiting for the *same* event to occur. If you want calls to wait for distinct events, you should use different entries or an entry family. Furthermore, allowing formal parameters in barrier would make it inefficient to evaluate barriers, because the run-time system would have to scan the entry queue and re-evaluate the barrier *for each call*. With this rule, the code executed for each barrier is fixed regardless of how many calls are enqueued.

If blocking of a call really does depend on the formal parameters, call an entry with the barrier `True`, examine the formal parameters within the body and requeue on other private entries. For example, in the CEO case study, we might have considered passing a parameter to the `Register` entry so that last caller need not register:

```
entry Register(Is_Last: Boolean) when not Is_Last is  -- Error
```

Instead, all members of a group call this entry and a decision to requeue or not is made by performing the computation `Waiting < Size` on a component of the protected object and a discriminant.

18.8.2 Potentially blocking operations

A protected action is not allowed to invoke an operation that could result in blocking the calling task *within* the protected action.

§9.5.1

> 8 During a protected action, it is a bounded error to invoke an operation that is *potentially blocking*. . . .

Potentially blocking operations are listed in §9.5.1(9–16); in particular, calling an entry is potentially blocking. The bounded error *need not* be detected by the implementation §9.5.1(17), though Annex §H High Integrity Systems contains **pragma** `Detect_Blocking` §H.5, which requires an implementation to to detect this error.

Ada 95

pragma `Detect_Blocking` is not in Ada 95.

18.8.3 Parameters of the requeue target

The requeue statement is restricted as follows:

§9.5.4

> 3 The *entry_*name of a requeue_statement shall resolve to denote an entry (the *target entry*) that either has no parameters, or that has a profile that is type conformant (see 6.3.1) with the profile of the innermost enclosing entry_body or accept_statement.
>
> 12 If the new entry named in the requeue_statement has formal parameters, then during the execution of the accept_statement or entry_body corresponding to the new entry, the formal parameters denote the same objects as did the corresponding formal parameters of the callable construct completed by the requeue. In any case, no parameters are specified in a requeue_statement; any parameter passing is implicit.

In the case study, both requeues are to entries without parameters. The entry family index in **requeue** `CEO.Wake(Group)` is not a parameter.

18.8.4 Internal and external requeues

A call or requeue can be to the same protected object—an *internal call or requeue*—or it can be to a subprogram or entry of a different object—an *external call or requeue* §9.5(2–7). (A call must be to a subprogram, because a call to an entry is potentially blocking, as noted previously). An external call or requeue uses the syntax of a selected component to distinguish it from an internal call or requeue: **requeue** `Wait_for_Last_Member` is internal and correct, but

requeue `Room.Wait_for_Last_Member` is an external call to the same protected object and thus incorrect because it is potentially blocking §9.5.1(15).

There is an important difference between the semantics of internal and external requeues:

§9.5.4

10 • if the requeue is an internal requeue (that is, the requeue is back on an entry of the same protected object—see 9.5), the call is added to the queue of the named entry and the ongoing protected action continues (see 9.5.1);

11 • if the requeue is an external requeue (that is, the target protected object is not implicitly the same as the current object—see 9.5), a protected action is started on the target object and proceeds as for a normal entry call (see 9.5.3).

An external call or requeue will initiate a new protected action so it may have to wait to obtain exclusive access to the protected object. An internal requeue is immediately enqueued on the existing queue; then, as part of the completion of the protected action, the barriers will be re-evaluated. The requeued task will receive no precedence over tasks already enqueued on entries, but we are assured that if some entry is open, a protected action will be immediately executed for some task.

18.8.5 Families of protected entries

An entry family can be declared in a protected unit; like an entry family for a task, it can be considered as if it were an array of entries, but the syntax of the entry body is more flexible. One entry body with an entry_index_specification defines the common code for processing all entries in the family.

§9.5.2

```
6 entry_body_formal_part ::= [(entry_index_specification)] parameter_profile
7 entry_barrier            ::= when condition
8 entry_index_specification::= for defining_identifier in discrete_subtype_definition
```

In the following declarations, `E1` is a family of entries each with one formal parameter, while `E2` a single entry with two formal parameters:

```
entry E1(Departments)(I: in Integer);
entry E2(D: in Departments; I: in Integer);
```

There is a *separate* queue for each member of the family `E1`, but only one queue for `E2`. The barrier of `E1` can depend on its entry index `D`, but, as we have seen, the barrier of `E2` cannot depend on its formal parameter `D`.