Concurrent and Real-Time Programming in Ada

ALAN BURNS AND ANDY WELLINGS

CAMBRIDGE

This page intentionally left blank

CONCURRENT AND REAL-TIME PROGRAMMING IN ADA

Ada is the only ISO standard, object-oriented, concurrent, real-time programming language. It is intended for use in large, long-lived applications where reliability and efficiency are essential, particularly real-time and embedded systems. In this book, Alan Burns and Andy Wellings give a thorough, self-contained account of how the Ada tasking model can be used to construct a wide range of concurrent and real-time systems. This is the only book that focuses on an in-depth discussion of the Ada tasking model. Following on from the authors' earlier title 'Concurrency in Ada', this book brings the discussion up to date to include the new Ada 2005 language and the recent advances in real-time programming techniques. It will be of value to software professionals and advanced students of programming alike; indeed, every Ada programmer will find it essential reading and a primary reference work that will sit alongside the language reference manual.

ALAN BURNS is a Professor in Computer Science at the University of York. His research activities have covered a number of aspects of real-time and safety critical systems, including the assessment of languages for use in the real-time safety critical domain, distributed operating systems, the formal specification of scheduling algorithms and implementation strategies, and the design of dependable user interfaces to safety critical applications. His teaching activities include courses in Operating Systems, Scheduling and Real-time Systems. He has authored over 370 papers and reports and 8 books, including 'Real-time Systems and Programming Languages' (3rd Edition), 'Concurrency in Ada' (2nd Edition) and 'Concurrent and Real-Time Programming in Java'.

ANDY WELLINGS is a Professor of Real-Time Systems in the Computer Science Department at the University of York. He is interested in most aspects of the design and implementation of real-time dependable computer systems and, in particular, in real-time programming languages and operating systems. He is European Editor-in-Chief for the Computer Science journal 'Software-Practice and Experience' and a member of the International Expert Groups currently developing extensions to the Java platform for real-time, safety critical and distributed programming. He has authored over 280 papers and several books, including 'Real-time Systems and Programming Languages' (3rd edition) and 'Concurrency in Ada' (2nd edition).

CONCURRENT AND REAL-TIME PROGRAMMING IN ADA 2005

ALAN BURNS AND ANDY WELLINGS
University of York

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK
Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521866972

First published in print format 2007

ISBN-13 978-0-511-29663-5 eBook (NetLibrary)
ISBN-10 0-511-29663-0 eBook (NetLibrary)

ISBN-13 978-0-521-86697-2 hardback
ISBN-10 0-521-86697-9 hardback

Contents

Preface

The development of the Ada programming language forms a unique and, at times, intriguing contribution to the history of computer languages. As all users of Ada must know, the original language design was a result of competition between a number of organisations, each of which attempted to give a complete language definition in response to a series of documented requirements. This gave rise to Ada 83. Following 10 years of use, Ada was subject to a complete overhaul. The resulting language, Ada 95, had a number of significant changes from its predecessor. A further 10 years of use has produced another version of Ada, known as Ada 2005, this time the changes are less pronounced and yet there are some key extra facilities, especially in the areas of real-time programming.

Closely linked to the development of Ada has been this book on its concurrent features. Starting out as 'Concurrent Programming in Ada', it became 'Concurrency in Ada' when the Ada 95 version of the language was defined. There were two editions of this title. With the new features of Ada 2005, it has been decided to broaden the focus of the book to include real-time issues – hence this first edition of the new title 'Concurrent and Real-Time Programming in Ada 2005'. No prior knowledge of concurrent programming (in general) or of Ada tasking (in particular) is assumed in this book. However, readers should have a good understanding of at least one high-level sequential programming language and some knowledge of operating system principles.

This book is aimed both at professional software engineers and at students of computer science (and other related disciplines). Many millions of lines of Ada 83 and 95 code have been produced world wide, and over the next decade a wide range of new applications will be designed with Ada 2005 as the target language. It is important that Ada programmers do not restrict themselves to a sequential subset of the language on the dubious assumption that tasking is not appropriate to their work, or for fear that the tasking model is too complex and expensive. Tasking is an integral part of the language, and programmers must be familiar with, if not experienced in, its use. Due to space considerations, books that describe the entire language may not deal adequately with the tasking model; this book therefore concentrates exclusively on this model.

Students studying real-time programming, software engineering, concurrent programming or language design should find this book useful in that it gives a comprehensive description of the features that one language provides. Ada is not merely a product of academic research (as are many concurrent programming languages) but is a language intended for actual use in industry. Its model of tasking was therefore integrated into the entire language design, and the interactions between tasking and non-tasking features were carefully defined. Consequently, the study of Ada's model of concurrency should be included in those advanced courses mentioned above. However, this does not imply that the full tasking model is free from controversy, has a proven formal semantic basis or is amenable to efficient implementation. The nature of these areas of 'discussion' are dealt with, as they arise in this book.

Unlike Ada 83, which defined a single language, the Ada 95 and 2005 definitions have a core language design plus a number of domain-specific annexes. A compiler need not support all the annexes but it must support the core language. Most of the tasking features are contained in the core definition. But there are relevant annexes that address systems programming and real-time programming.

The first chapter provides a basic introduction to concurrent and real-time systems and gives an overview of the clock facilities within Ada.

Chapters 2 and 3 look in detail at the uses of concurrent programming and the inherent difficulties of providing inter-process communication. There is, as yet, no agreement on which primitives a concurrent programming language should support and, as a consequence, many different styles and forms exist. In order to understand the Ada tasking model fully, it is necessary to appreciate these different approaches and the problems faced by the user of any language that supports multi-processing.

The Ada task is introduced in Chapter 4 and the rendezvous and the important select statement are considered in the following two chapters. The rendezvous provides a synchronous communication mechanism. Data-orientated asynchronous communication is considered in Chapter 7, together with the important abstraction of a protected object. This provides a passive means of encapsulating data and providing mutual exclusion. An effective way of increasing the expressive power of the communication primitives is the requeue facility. This is described, with many examples given, in Chapter 8. The relationship between tasks and exceptions is dealt with in Chapter 9. This chapter also covers the means by which one task can affect the behaviour of another task asynchronously.

Chapter 10 considers the interplay between tasking and the object-orientated programming features of the language. This forms the basis from which a collection of concurrency utilities can be defined. A number of these are provided in Chapter 11.

As indicated earlier, a number of the annexes deal with issues of relevance to concurrent programming. Chapter 12 considers systems programming (including support for low level programming). For real-time programmers, perhaps the most important issue is scheduling. Ada provides a comprehensive list of features that are covered in Chapters 13 and 14. In addition to scheduling, real-time programs need to have control over when events are executed and control over the resources that tasks and groups of task require at run-time. These issues are covered in Chapter 15.

Having introduced all of Ada's relevant features, Chapter 16 then provides a collection of real-time utilities that can be used to gain access to the power of the language. This is followed in Chapter 17 by consideration of the usefulness of subsetting Ada and using profiles to gain access to efficient and certifiable implementations. In particular, the Ravenscar profile is described in this chapter. Finally, in Chapter 18 conclusions are provided and a short summary of the differences between Ada 2005 and Ada 95 is given in the context of concurrent and real-time programming, together with a brief look to the future.

The material presented in this book reflects the authors' experiences in both using and teaching Ada tasking. Teaching experience has been obtained by writing and presenting courses at the University of York (UK) and by developing educational material and training.

Further material

Further material on all aspects of real-time and concurrency in Ada 2005 can be found on a www page dedicated to this book:
http://www.cs.york.ac.uk/∼rts/ada/CRTIA.html.

Real-time systems research at York

Alan Burns and Andy Wellings are members of the Real-Time Systems Research Group in the Department of Computer Science at the University of York (UK).

The aim of the group is to undertake fundamental research, and to bring modern techniques, methods and tools into engineering practice. Areas of application of our work include space and avionic systems, engine controllers, automobile control, manufacturing systems, sensor nets and multi-media systems. Specifically, the group is addressing: scheduling theories, language design, kernel design, communication protocols, distributed and parallel architectures and program code analysis.

Further information about the group's activities can be found via our www page: http://www.cs.york.ac.uk/∼rts

Acknowledgements

The authors would like to thank the following individuals who have, directly or indirectly, helped during the preparation of this book. In particular, John Barnes, Andrew Borg, Javier Miranda, Pat Rogers, Ed Schonberg, and Tucker Taft.

The authors also wish to thank members of ISO committee ARG and attendees of the IRTAW series for their input to some of the issues discussed in this book.

Finally, we would like to thank the AdaCore GNAT project team for all their efforts to produce a public domain Ada 2005 compiler. Writing a book for a new language is very difficult when there are no validated compilers to help test the code. Access to the GNAT system has provided us with more confidence that the code given in this book is at least syntactically correct!

1
Introduction

Designing, implementing and maintaining software for large systems is a non-trivial exercise and one which is fraught with difficulties. These difficulties relate to the management of the software production process itself, as well as to the size and complexity of the software components. Ada is a mature general-purpose programming language that has been designed to address the needs of large-scale system development, especially in the embedded systems domain. A major aspect of the language, and the one that is described comprehensively in this book, is its support for concurrent and real-time programming.

Ada has evolved over the last thirty years from an object-based concurrent programming language into a flexible concurrent and distributed object-oriented language that is well suited for high-reliability, long-lived applications. It has been particularly successful in high-integrity areas such as air traffic control, space systems, railway signalling, and both the civil and military avionics domains. Ada success is due to a number of factors including the following.

- Hierarchical libraries and other facilities that support large-scale software development.
- Strong compile-time type checking.
- Safe object-oriented programming facilities.
- Language-level support for concurrent programming.
- A coherent approach to real-time systems development.
- High-performance implementations.
- Well-defined subsetting mechanisms, and in particular the SPARK subset for formal verification.

The development and standardisation of Ada have progressed through a number of definitions, the main ones being Ada 83 and Ada 95. Ada 2005 now builds on this success and introduces a relatively small number of language changes to provide:

- Better support for multiple inheritance through the addition of Java-like interfaces.
- Better support for OO style of programming by use of the `Object.Operator` notation.
- Enhanced structure and visibility control via the introduction of 'limited with clauses' that allow types in two library packages to refer to each other.
- More complete integration between object-oriented and concurrent programming by the introduction of synchronised interfaces.
- Added flexibility to the support for real-time systems development via alternative scheduling mechanisms and more effective resource monitoring.
- A well-defined subset of the tasking model, called the Ravenscar profile, for high-integrity applications.
- New library packages, for example an extensive 'containers' package for lists, maps, vectors etc.

Ada 2005 attempts, successfully, to have the safety and portability of Java and the efficiency and flexibility of C/C++. It also has the advantage of being an international standard with clear well-defined semantics.

The reference manual for Ada (ARM) differentiates between the 'core' language and a number of annexes. Annexes do not add to the syntax of the language but give extra facilities and properties, typically by the introduction of language-defined library packages. For the focus of this book, the key elements of the reference manual are Chapter 9 which deals with tasking and the Real-Time Systems Annex (Annex D). In terms of presentation, this book does not draw attention to core language or annex-defined facilities. All are part of the Ada language.

The remainder of this chapter provides an introduction to concurrency and real-time. The book then considers, in depth, the role that Ada can play in the construction of concurrent and real-time systems. It gives particular emphasis to the new features of Ada 2005. However, prior knowledge of the earlier language definitions is not required as a complete description of these features is provided. But the reader is assumed to be familiar with object-oriented programming in sequential Ada. For a more detailed discussion on the sequential aspects of Ada, the reader should see the further reading section at the end of this chapter.

1.1 Concurrency

Support for concurrency in a language allows the programmer to express (potential) parallelism in application programs. There are three main motivations for wanting to write concurrent programs.

- To fully utilise the processor – Modern processors run at speeds far in excess of the input and output devices with which they must interact. Concurrency allows the programmer to express other activities to be performed while the processor is waiting for IO to complete.

- To allow more than one processor to solve a problem – A sequential program can only be executed by one processor (unless the compiler has transformed the program into a concurrent one). A concurrent program is able to exploit true parallelism and obtain faster execution.
- To model parallelism in the real world – Real-time and embedded programs have to control and interface with real-world entities (robots, conveyor belts etc.) that are inherently parallel. Reflecting the parallel nature of the system in the structures of the program makes for a more readable, maintainable and reliable application.

One of the major criticisms of concurrent programming is that it introduces overheads and therefore results in slower execution when the program is running on a single-processor system. Nevertheless, the software engineering issues outweigh these concerns, just as the efficiency concerns of programming in a high-level sequential language are outweighed by its advantages over programming with an assembly language. Chapter 2 further explores these concerns, and discusses in detail the nature and use of concurrent programming techniques.

Writing concurrent programs introduces new problems that do not exist in their sequential counterparts. Concurrent activities need to coordinate their actions, if they are to work together to solve a problem. This coordination can involve intricate patterns of communication and synchronisation. If not properly managed, these can add significant complexity to the programs and result in new error conditions arising. These general problems and their solutions are explained in Chapter 3. Chapters 4–11 then discuss in detail the facilities that Ada provides to create concurrent activities (called **tasks** in Ada) and to manage the resulting communication and synchronisation needs.

1.2 Real-time systems

A major application area of concurrent programming is real-time systems. These are systems that have to respond to externally generated input stimuli (including the passage of time) within a finite and specified time interval. They are inherently concurrent because they are often embedded in a larger engineering system, and have to model the parallelism that exists in the real-world objects that they are monitoring and controlling. Process control, manufacturing support, command and control are all example application areas where real-time systems have a major role. As computers become more ubiquitous and pervasive, so they will be embedded in a wide variety of common materials and components throughout the home and workplace – even in the clothes we wear. These computers will need to react with their environment in a timely fashion.

It is common to distinguish between *hard* and *soft* real-time systems. Hard real-time systems are those where it is absolutely imperative that responses occur within the specified deadline. Soft real-time systems are those where response times are important, but the system will still function correctly if deadlines are occasionally missed. Soft systems can be distinguished from interactive ones in which there are no explicit deadlines. For example, the flight control system of a combat aircraft is a hard real-time system because a missed deadline could lead to a catastrophic situation and loss of the aircraft, whereas a data acquisition system for a process control application is soft, as it may be defined to sample an input sensor at regular intervals but to tolerate intermittent delays. A text editor is an example of an interactive system. Here, performance is important (a slow editor will not be used); however, the occasional poor response will not impact on the overall system's performance. Of course, many systems will have both hard and soft real-time subsystems along with some interactive components. What they have in common is that they are all concurrent.

Time is obviously a critical resource for real-time systems and must be managed effectively. Unfortunately, it is very difficult to design and implement systems that will guarantee that the appropriate outputs will be generated at the appropriate times under all possible conditions. To do this and make full use of all computing resources at all times is often impossible. For this reason, real-time systems are usually constructed using processors with considerable spare capacity, thereby ensuring that 'worst-case behavior' does not produce any unwelcome delays during critical periods of the system's operation. Given adequate processing power, language and run-time support are required to enable the programmer to

- specify times at which actions are to be performed,
- specify times at which actions are to be completed,
- respond to situations where all the timing requirements cannot be met,
- respond to situations where the timing requirements are changing dynamically.

These are called real-time control facilities. They enable the program to synchronise with time itself. For example, with digital control algorithms, it is necessary to sample readings from sensors at certain periods of the day, for example, 2pm, 3pm and so on, or at regular intervals, for instance, every 10 milliseconds (with analogue-to-digital converters, sample rates can vary from a few hundred hertz to several hundred megahertz). As a result of these readings, other actions will need to be performed. In order to meet response times, it is necessary for a system's behaviour to be predictable. Providing these real-time facilities is one of the main goals of Ada and its Real-Time Systems Annex.

As well as being concurrent, real-time systems also have the following additional characteristics:

Large and complex. Real-time systems vary from simple single-processor embedded systems (consisting of a few hundred lines of code) to multi-platform, multi-language distributed systems (consisting of millions of lines of code). The issue of engineering large and complex systems is an important topic that Ada and its support environments do address. However, consideration of this area is beyond the scope of this book.

Extremely reliable and safe. Many real-time systems control some of society's critical systems such as air traffic control or chemical/power plants. The software must, therefore, be engineered to the highest integrity, and programs must attempt to tolerate faults and continue to operate (albeit perhaps providing a degraded service). In the worst case, a real-time system should make safe the environment before shutting down in a controlled manner. Unfortunately, some systems do not have easily available safe states when they are operational (for example, an unstable aircraft) consequently, continued operation in the presence of faults or damage is a necessity. Ada's design goals facilitate the design of reliable and robust programs. Its exception handling facility allows error recovery mechanisms to be activated. The Real-Time Systems Annex extends the core language to allow the flexible detection of common timing-related problems (such as missed deadlines).

Interaction with hardware interfaces. The nature of embedded systems requires the computer components to interact with the external world. They need to monitor sensors and control actuators for a wide variety of real-world devices. These devices interface to the computer via input and output registers, and their operational requirements are device and computer dependent. Devices may also generate interrupts to signal to the processor that certain operations have been performed or that error conditions have arisen. In the past, the interfacing to devices either has been left under the control of the operating system or has required the application programmer to resort to assembly language inserts to control and manipulate the registers and interrupts. Nowadays, because of the variety of devices and the time-critical nature of the associated interactions, their control must often be direct, and not through a layer of operating system functions. Furthermore, reliability requirements argue against the use of low-level programming techniques. Ada's representation items allow memory-mapped device registers to be accessed and interrupts to be handled by protected procedures.

Efficient implementation and a predictable execution environment. Since real-time systems are time-critical, efficiency of implementation will be more important than in other systems. It is interesting that one of the main benefits of using a high-level language is that it enables the programmer to abstract away from implementation details and to concentrate on solving the problem at hand. Unfortunately, embedded computer systems programmers cannot afford this luxury. They must be constantly concerned with the cost of using particular language features. For example, if a response to some input is required within a microsecond, there is no point in using a language feature whose execution takes a millisecond! Ada makes predictability a primary concern in all its design trade-offs.

Chapters 12–17 discuss Ada's support for real-time systems in general, and also, where appropriate, how it facilitates the programming of efficient and reliable embedded systems.

1.3 Ada's time and clock facilities

Time values and clocks are used throughout this book to help manage interactions between concurrent activities and communication with the external environment. Consequently, this chapter concludes with a detailed discussion of the Ada facilities in this area.

To coordinate a program's execution with the natural time of the environment requires access to a hardware clock that approximates the passage of real time. For long-running programs (that is, years of non-stop execution), this clock may need to be resynchronised to some external standard (such as International Atomic Time) but from the program's point of view, the clock is the source of *real* time.

Ada provides access to this clock by providing several packages. The main section of the ARM (Ada Reference Manual) defines a compulsory library package called `Ada.Calendar` that provides an abstraction for 'wall clock' time that recognises leap years, leap seconds and other adjustments. Child packages support the notion of time zones, and provide arithmetic and formatting functions. In the Real-Time Systems Annex, a second representation is given that defines a monotonic (that is, non-decreasing) regular clock (package `Ada.Real_Time`). Both these representations should map down to the same hardware clock but cater for different application needs.

First consider package `Ada.Calendar`:

```ada
package Ada.Calendar is

  type Time is private;

  subtype Year_Number is Integer range 1901..2399;
  subtype Month_Number is Integer range 1..12;
  subtype Day_Number is Integer range 1..31;
  subtype Day_Duration is Duration range 0.0..86_400.0;
  function Clock return Time;

  function Year(Date:Time) return Year_Number;
  function Month(Date:Time) return Month_Number;
  function Day(Date:Time) return Day_Number;
  function Seconds(Date:Time) return Day_Duration;

  procedure Split(Date:in Time; Year:out Year_Number;
             Month:out Month_Number; Day:out Day_Number;
             Seconds:out Day_Duration);

  function Time_Of(Year:Year_Number; Month:Month_Number;
           Day:Day_Number;
           Seconds:Day_Duration := 0.0) return Time;

  function "+"(Left:Time;Right:Duration) return Time;
  function "+"(Left:Duration;Right:Time) return Time;
  function "-"(Left:Time;Right:Duration) return Time;
  function "-"(Left:Time;Right:Time) return Duration;

  function "<"(Left,Right:Time) return Boolean;
  function "<="(Left,Right:Time) return Boolean;
  function ">"(Left,Right:Time) return Boolean;
  function ">="(Left,Right:Time) return Boolean;

  Time_Error:exception;
  -- Time_Error may be raised by
  -- Time_Of, Split, Year, "+" and "-"

private
  ... -- not specified by the language
end Ada.Calendar;
```

A value of the private type `Time` is a combination of the date and the time of day, where the time of day is given in seconds from midnight. Seconds are described in terms of a subtype `Day_Duration` which is, in turn, defined by means of `Duration`. The fixed point type `Duration` is one of the predefined `Scalar` types and has a range which, although implementation dependent, must be at least –86_400.0 .. +86_400.0. The value 86_400 is the number of seconds in a day. The accuracy of `Duration` is also implementation dependent but the smallest representable value (`Duration'Small`) must not be greater than 20 milliseconds (it is recommended in the ARM that it is no greater than 100 microseconds).

The current time is returned by the function `Clock`. Conversion between `Time` and program accessible types, such as `Year_Number`, is provided by subprograms `Split` and `Time_Of`. In addition, some arithmetic and boolean operations are specified. Package `Calendar`, therefore, defines an appropriate structure for an abstract data type for time.

New to Ada 2005 is a child package of `Calendar` that provides further support for arithmetic on time values. It is now possible to add and subtract a number of days to and from a time value (rather than express the interval as a duration).

```
package Ada.Calendar.Arithmetic is
   -- Arithmetic on days:
   type Day_Count is range
     -366*(1+Year_Number'Last - Year_Number'First) ..
     366*(1+Year_Number'Last - Year_Number'First);

   subtype Leap_Seconds_Count is Integer range -999..999;
   procedure Difference (Left, Right : in Time;
             Days : out Day_Count;
             Seconds : out Duration;
             Leap_Seconds : out Leap_Seconds_Count);

   function "+" (Left : Time; Right : Day_Count)
                 return Time;
   function "+" (Left : Day_Count; Right : Time)
                return Time;
-- similarly for "-"
end Ada.Calendar.Arithmetic;
```

Other new clock-related packages in Ada 2005 include a package to support the formatting of time values for input and output, and rudimentary support for time zones.

```
with Ada.Calendar.Time_Zones;
package Ada.Calendar.Formatting is
   type Day_Name is (Monday, Tuesday, Wednesday, Thursday,
       Friday, Saturday, Sunday);

   function Day_of_Week (Date : Time) return Day_Name;

   subtype Hour_Number is Natural range 0 .. 23;
   subtype Minute_Number is Natural range 0 .. 59;
   subtype Second_Number is Natural range 0 .. 59;
   subtype Second_Duration is Day_Duration range 0.0 .. 1.0;

   function Hour(Date : Time;
                 Time_Zone : Time_Zones.Time_Offset := 0)
                 return Hour_Number;
   ... -- similarly for Minute, Second, Sub_Second

   function Seconds_Of(Hour : Hour_Number; Minute : Minute_Number;
            Second : Second_Number := 0;
            Sub_Second : Second_Duration := 0.0)
                return Day_Duration;

   procedure Split(Seconds : in Day_Duration;
             Hour : out Hour_Number;
             Minute : out Minute_Number;
             Second : out Second_Number;
             Sub_Second : out Second_Duration);
   ... --  other variations

   function Image(Date : Time;
            Include_Time_Fraction : Boolean := False;
            Time_Zone  : Time_Zones.Time_Offset := 0)
                return String;

   function Value(Date : String;
            Time_Zone  : Time_Zones.Time_Offset := 0)
                return Time;
   function Image (Elapsed_Time : Duration;
            Include_Time_Fraction : Boolean := False)
                return String;
   function Value (Elapsed_Time : String) return Duration;
end Ada.Calendar.Formatting;
```

```
package Ada.Calendar.Time_Zones is

   -- Time zone manipulation:

   type Time_Offset is range -1440 .. 1440;

   Unknown_Zone_Error : exception;

   function UTC_Time_Offset (Date : Time := Clock)
           return Time_Offset;

end Ada.Calendar.Time_Zones;
```

The Ada.Real_Time package has a similar form to the Ada.Calendar package:

```
package Ada.Real_Time is
  type Time is private;
  Time_First: constant Time;
  Time_Last: constant Time;
  Time_Unit: constant := implementation-defined-real-number;
  Time_Unit : constant := 1.0;

  type Time_Span is private;
  Time_Span_First: constant Time_Span;
  Time_Span_Last: constant Time_Span;
  Time_Span_Zero: constant Time_Span;
  Time_Span_Unit: constant Time_Span;

  Tick: constant Time_Span;
  function Clock return Time;

  function "+" (Left: Time; Right: Time_Span) return Time;
  function "+" (Left: Time_Span; Right: Time) return Time;
  function "-" (Left: Time; Right: Time_Span) return Time;
  function "-" (Left: Time; Right: Time) return Time_Span;
  function "<" (Left, Right: Time) return Boolean;
  function "<="(Left, Right: Time) return Boolean;
  function ">" (Left, Right: Time) return Boolean;
  function ">="(Left, Right: Time) return Boolean;

  function "+" (Left, Right: Time_Span) return Time_Span;
  function "-" (Left, Right: Time_Span) return Time_Span;
  function "-" (Right: Time_Span) return Time_Span;
  function "/" (Left,Right : Time_Span) return Integer;
  function "/" (Left : Time_Span; Right : Integer)
           return Time_Span;
  function "*" (Left : Time_Span; Right : Integer)
           return Time_Span;
  function "*" (Left : Integer; Right : Time_Span)
           return Time_Span;

  function "<" (Left, Right: Time_Span) return Boolean;
  function "<="(Left, Right: Time_Span) return Boolean;
  function ">" (Left, Right: Time_Span) return Boolean;
  function ">="(Left, Right: Time_Span) return Boolean;

  function "abs"(Right : Time_Span) return Time_Span;

  function To_Duration (Ts : Time_Span) return Duration;
  function To_Time_Span (D : Duration) return Time_Span;

  function Nanoseconds  (NS: Integer) return Time_Span;
  function Microseconds (US: Integer) return Time_Span;
  function Milliseconds (MS: Integer) return Time_Span;

  type Seconds_Count is range implementation-defined;
  procedure Split(T : in Time; SC: out Seconds_Count;
                  TS : out Time_Span);
  function Time_Of(SC: Seconds_Count; TS: Time_Span)
           return Time;
private
  ... -- not specified by the language
end Ada.Real_Time;
```

The `Real_Time.Time` type represents time values as they are returned by `Real_Time.Clock`. The constant `Time_Unit` is the smallest amount of time representable by the `Time` type. The value of `Tick` must be no greater than one millisecond; the range of `Time` (from the epoch that represents the program's start-up) must be at least fifty years. Other important features of this time abstraction are described in the Real-Time Systems Annex; it is not necessary, for our purposes, to consider them in detail here.

To illustrate how the above packages could be used, consider the following code which tests to see if some sequence of statements executes within 1.7 seconds:

```
declare
  Start, Finish : Time;   -- Ada.Calendar.Time
  Interval : Duration := 1.7;
begin
  Start := Clock;
  -- sequence of statements.
  Finish := Clock;
  if Finish - Start > Interval then
    raise Overrun_Error;  -- a user-defined exception.
  end if;
end;
```

The above code fragment would also execute correctly with the real-time clock if `Interval` were declared as follows:

```
Interval : Time_Span := To_Time_Span(1.7);
           -- for use with Ada.Real_Time.Time
```

or

```
Interval : Time_Span := Milliseconds(1700);
```

Delay primitives

As well as having access to a real-time clock, a concurrent activity must also be able to delay itself for a period of time. This enables it to be queued on some future event rather than busy-wait on calls to a `Clock` function:

```
-- busy-wait to delay ten seconds.
Start := Clock;
loop
  exit when Clock - Start > 10.0;
end loop;
```

Ada provides an explicit *delay statement* for use with the `Calendar` clock:

```
delay 10.0;
```

The expression following 'delay' must yield a value of the predefined type `Duration`.

Important note: It is important to appreciate that 'delay' is an approximate time construct, the inclusion of which in a task indicates that the task will be delayed by at least the amount specified. There is no explicit upper bound given on the actual delay, although it may be possible to calculate one from an understanding of the implementation of delay and the clock. The significant point is that the delay cannot be less than that given in the delay statement.

The difference between the actual delay and the desired value is called the *local drift*. If a repeating loop contains a delay statement (as in, for example, a periodic activity – see Chapter 13), the local drift values will accumulate to give increasing *cumulative* drift. This can be eliminated by use of the following alternative delay primitive.

The use of delay supports a relative time period (that is, ten seconds from now). If a delay to an absolute time is required, then the *delay until* statement should be used. For example, if an action should take place ten seconds after the start of some other action, then the following code should be used (with `Ada.Calendar`):

```
Start := Clock;
First_Action;
delay until Start + 10.0;
Second_Action;
```

Note that this is *not* the same as

```
Start := Clock;
First_Action;
delay (Start + 10.0) - Clock;
Second_Action;
```

In order for this second formulation to have the same behaviour as the first, then

```
delay (Start + 10.0) - Clock;
```

would have to be an uninterruptible action, which it is not. For example, if `First_Action` took two seconds to complete, then

```
(Start + 10.0) - Clock;
```

would equate to eight seconds. But after having calculated this value, if the task involved is preempted by some other task then it could be three seconds (say) before it next executes. At that time, it will delay for eight seconds rather than five. The *delay until* formulation does not suffer from this problem, and it can be used with time values from either clock package.

As with delay, *delay until* is accurate only in its lower bound. The task involved will not be released before the current time has reached that specified in the statement, but it may be released later.

Warning: The ‘delay’ statement only takes an expression that evaluates to an object of type `Duration`. In contrast, the expression in a ‘delay until’ statement can evaluate to any `Time` type object (e.g., `Calendar.Time` or `Real_Time.Time`) including an implementation-defined one.

1.4 Summary

Support for concurrency is essential in a modern programming language in order to support both the construction of parallel algorithms and the development of complex real-time embedded systems. This book initially focuses on the topic of concurrent programming and explores, in detail, the Ada model. Both strengths and weaknesses are discussed. Following this, attention is turned to one of the most important application areas for concurrent programming, that of real-time systems. The Ada approach is examined in depth. Throughout, examples are given of good programming practice to exploit the powerful Ada abstractions.

The chapter also includes a discussion of clocks and time. Ada provides a standard package for accessing a clock function and an abstract data type for time. An implementation may also support a real-time package that supports a monotonic, fine grain clock. Ada also defines delay primitives that allow a task to delay itself for an absolute or relative period of time.

1.5 Further reading

J. Barnes, *Programming in Ada 2005*, Addison-Wesley, 2006.

A. Burns and A.J. Wellings, *Real-Time Systems and Programming Languages*, 3rd Edition. Addison-Wesley, 2001.

2
The nature and uses of concurrent programming

Any language, natural or computer, has the dual property of enabling expression whilst at the same time limiting the framework within which that expressive power may be applied. If a language does not support a particular notion or concept, then those that use the language cannot apply that notion and may even be totally unaware of its existence. Pascal, C, C++, Eiffel, Fortran and COBOL share the common property of being sequential languages. Programs written in these languages have a single thread of control. They start executing in some state and then proceed, by executing one statement at a time, until the program terminates. The path through the program may differ due to variations in input data, but for any particular execution of the program there is only one path.

A modern computer system will, by comparison, consist of one or more central processors and many I/O devices, all of which are operating in parallel. Moreover, an operating system that provides for interactive use will always support many executing programs (called processes†), in the sense that they are being time-sliced onto the available processors. The effect of fast process switching is to give behaviour that is almost indistinguishable from true parallelism. In the programming of embedded systems, one must deal with the inherent parallelism of the larger system. A real-time language must therefore provide some facility for multi-programming. This can be achieved by specifying a standard interface to a multiprocessing operating system‡ or by allowing multiple process systems to be expressed in the language itself.

Ada provides for the direct programming of parallel activities. Within an Ada program there may be a number of *tasks*, each of which has its own thread of control. It is thus possible to match the parallel nature of the application area with

† Each process typically executes in its own address space. Early operating systems only supported a single thread of control within a process. Nowadays, most operating systems support multi-threaded processes.

‡ However, care must be taken with this approach. If the compiler is unaware that a concurrent program is being written, it might perform some inappropriate code optimisations.

syntactical forms that reflect these structures. This has proved to be an invaluable aid in the production of clear and correct software. Languages whose conceptual framework includes parallelism are known as *concurrent programming languages*. Ada is such a language but it is by no means the first (or last); for example C#, Java, SCOOP, CHILL, CSP, PEARL, occam and LINDA all deal with concurrency, although in radically different ways. In addition to these procedural languages, there are also functional, logic-based and data-flow languages available for the specification and implementation of concurrent systems. Even C++ is now considering directly supporting a standard library for concurrent programming.

In general, each individual thread of control within a concurrent program is known as a *process*, although Ada uses the term *task*. No distinction is made, in this book, between tasks and processes. The execution of a process usually takes one of three forms:

(1) All processes may share a single processor.
(2) Each process may have its own processor, and the processors share common memory.
(3) Each process may have its own processor, and the processors are distributed (that is, they are connected by a communications network).

Hybrids of these three methods are also evident. Ada is designed for all the above situations. Because of the different implementation methods, the term *concurrent*, rather than *parallel*, is of more use in this area of computing. Two processes are said to be executing in *parallel* if at any instant they are both executing. Therefore, in the above classifications only cases (2) and (3) are truly parallel.

By comparison, two processes are said to be *concurrent* if they have the potential for executing in parallel. A concurrent program is thus one that has more than one thread of control. Execution of this program will, if processors are available, run each of these threads of control in parallel. Otherwise, the threads will be interleaved. The important concept is therefore concurrency (as it encompasses all three of the above cases) irrespective of whether or not the implementation of concurrency involves parallelism or pseudo-parallelism. A correct concurrent program will execute in accordance with its functional specification on all three platforms. Only its performance should change, perhaps dramatically, as the nature of the hardware is altered.

Concurrent programming is the name given to programming notations and techniques for expressing potential parallelism and for solving the resulting synchronisation and communication problems. Implementation of parallelism is a topic in computer systems (hardware and software) that is essentially independent of concurrent programming. Concurrent programming is important because it provides an abstract setting in which to study parallelism without getting bogged down in the implementation details (Ben-Ari, 1982).

The problems of synchronisation and communication are considered in the next chapter.

2.1 Uses of concurrent programming

Concurrency is used extensively in the programming of embedded systems. A system is also *real-time* if its specification has time-dependent features. Virtually all embedded systems are inherently parallel; the software must, therefore, control the simultaneous operations of the coexisting hardware components. Typically, this is achieved by associating with each external device a process that controls the input and output operations of that device. These processes, together with the necessary internal data management processes, constitute the software model. Embedded systems themselves are to be found in a wide variety of applications, for example:

- process control;
- air traffic control;
- avionics systems;
- industrial robots;
- engine controllers;
- domestic appliances;
- environmental monitors;
- command and control systems.

The implementation of these multi-task systems can be achieved by integrating several sequential programs, but this necessitates the support of an underlying operating system that will map programs onto the processes and allow data communication. The use of a concurrent programming language may, however, preclude operating system support, in which case the run-time system of the language implementation must control scheduling; data communication (plus synchronisation) is programmed directly in the language.

Concurrency is also of value in mapping software efficiently onto multiprocessor hardware to exploit the properties of concurrent algorithms. For instance, the need to sort 10,000 objects (a standard sequential problem) may be more effectively undertaken as ten parallel processes, each sorting 1,000 objects, followed by a merge operation. Here, the distinction between pseudo- and true parallelism is important: the above algorithm will almost certainly be less efficient than a standard approach if the processes are time-sliced onto a single processor. The hardware architecture can, therefore, have a serious effect upon certain non-functional aspects of the portability of concurrent programs.

It should be noted that concurrent programming is not the only way of exploiting parallel hardware. Array and vector processors are better utilised by having access to concurrent operators such as vector addition. Also, certain data-flow machines require a completely different computational model. None the less, concurrent programming does represent one of the main ways of gaining access to standard multi-processor hardware.

Software engineering principles indicate that the implementation languages should, wherever possible, mimic the structure of the application domain. If the application contains inherent parallelism, then the design and construction of the software product will be less error-prone, easier to prove correct and easier to adapt if concurrency is available in the design and implementation languages. Two examples of this type of use of Ada are in the development of information processing system prototypes and HCI components. In the first of these, a data-flow description of an information processing system consists, primarily, of tasks (where information is processed) and data-flows that link these tasks. The transformation of this description into a series of sequential programs is time consuming and error-prone. With a concurrent language such as Ada, the implementation is straightforward and can be undertaken almost automatically. An HCI subsystem enables the application software (which may be multi-tasking) and the user interface implementation to be designed separately and programmed as concurrent objects.

These two quite different examples indicate that concurrency is not just concerned with coding embedded systems but also a fundamental language issue. The wide availability and use of Ada allow programmers from many differing application domains to have available, if necessary, concurrent programming facilities. It is therefore important that all users of Ada understand the tasking model.

2.2 Program entities

The object-oriented programming paradigm encourages system (and program) builders to consider the artifact under construction as a collection of co-operating objects (or, to use a more neutral term, *entities*). Within this paradigm, it is constructive to consider two kinds of entities – active and reactive.

Definition: *Active* entities undertake spontaneous actions (with the help of a processor): they enable the computation to proceed. *Reactive* entities, by comparison, only perform actions when 'invoked' by an active entity. Both active and reactive entities can be accessed via *interfaces*.

Other programming paradigms, such as data-flow or real-time networks, identify active agents and passive data.

The Ada language itself does not prescribe a specific programming paradigm, and it is not the intention of this book to present just a single model. Rather, we take a more abstract view: an Ada program is deemed to consist of (be populated by) active entities, reactive entities and passive entities. Only active entities give rise to spontaneous actions.

Definition: Resources are reactive but can control access to their internal states (and any real resources they control).

Some resources can only be used by one agent at a time; in other cases, the operations that can be carried out at a given time depend on the resources' current states. A common example of the latter is a data buffer whose elements cannot be extracted if it is empty.

Definition: The term *passive* will be used to indicate a reactive entity that can allow open access.

The implementation of resource entities requires some form of control agent. If the control agent is itself passive (such as a semaphore), then the resource is said to be *protected*. Alternatively, if an active agent is required to program the correct level of control, then the resource is in some sense active.

Definition: The term *server* will be used to identify an active resource entity, and the term *protected resource* to indicate the passive kind. *Synchronized interfaces* allow clients to access resources without being concerned with whether the resource controller is active or protected.

The four abstract program entities used in this book, therefore, are *active*, *passive*, *server* and *protected resource*.

In a concurrent programming language, active entities are represented by processes. Passive entities either can be represented directly as data variables or can be encapsulated by some module/package construct that provides a procedural interface. Protected resources may also be encapsulated in a module-like construct and require the availability of a low-level synchronisation facility. Servers, because they need to program the control agent, require a process.

A key question for language designers is whether to support primitives for both protected resources and servers. Resources, because they typically use a low-level control agent, are normally efficient (at least on single-processor systems). But they can be inflexible and lead to poor program structures for some classes of problem (this is discussed further in the next chapter). Servers, because the control agent is programmed using a process, are eminently flexible. The drawback of this approach is that it can lead to proliferation of processes, with a resulting high number of context switches during execution. This is particularly problematic if the language does not support protected resources and hence servers must be used for all such entities.

Important note: Ada 83 only supported the single notion of a **task**; thus active and all control entities were encoded in tasks. Ada 95 introduced a new abstraction for resource entities – **the protected type**. An object of such a type can control access to the data it protects but does not have a thread of control. Thus resources should be coded as protected objects, with tasks being used for active objects and servers.

The design of a concurrent Ada program must therefore incorporate early recognition of the key active, passive, resource (protected) and server entities, and use the appropriate Ada language features for representing their required behaviour directly. Ada 2005 provides extra support by allowing better integration between its concurrency mechanisms and the object-oriented programming paradigm.

2.3 Process representation

Various notations are used to specify the concurrent components of a program, and different methods are also employed to indicate when a process should start executing and when it should terminate. Coroutines were one of the first methods of expressing concurrency, although the coroutines themselves cannot actually execute in parallel. A coroutine is structured like a procedure; at any one time a single coroutine is executing, with control being passed to another coroutine by means of the **resume** statement. The scheduling of the coroutines is therefore explicitly expressed in the program. A resumed coroutine continues executing, from its last executing state, until it again encounters a resume statement. The resume statement itself names the coroutine to be resumed.

Coroutines have been used, primarily, in discrete event simulation languages such as SIMULA, although they are also supported in Modula-2. Because coroutines are not adequate for true parallel processing, they are not available in Ada. Instead, Ada, like many other concurrent programming languages, uses a direct representation of process which, as has been noted, is called a task. Moreover, the execution of a task is started, in effect, by the scope rules of the language. This is in contrast to languages such as Algol68, CSP and occam where execution is started by a *cobegin ... coend* or *Par* structure; for example

```
cobegin  -- not Ada
  P1; P2; P3;
coend;
```

will cause the concurrent execution of processes `P1`, `P2` and `P3`.

A detailed examination of task declaration is given in Chapter 4. However, a more informal description will be of use at this stage. Consider the following program skeleton:

```
procedure Main is
  task A;
  task B; -- two tasks have been declared and named.
  ...

  task body A is separate; -- implementation of task A.
  task body B is separate; -- implementation of task B.
begin
  -- A and B are now both executing concurrently.
  ...
end Main;
```

A task has a similar syntactical structure to that of a package (in that it has a specification and a body). In the sequence of statements of procedure `Main`, three concurrent objects are executing: the two tasks (`A` and `B`) and the statements of the procedure. The procedure will itself only terminate when all these statements have been executed and the two tasks have terminated. Execution of tasks `A` and `B` is deemed to commence immediately after **begin**, that is, before any of the statements of `Main`.

2.4 A simple embedded system

In order to illustrate some of the advantages and disadvantages of concurrent programming, a simple embedded system will now be considered. Figure 2.1 outlines this simple system: a process T takes readings from a set of thermocouples (via an analogue-to-digital converter, ADC) and makes appropriate changes to a heater (via a digitally controlled switch). Process P has a similar function, but for pressure (it uses a digital to analogue converter, DAC). Both T and P must communicate data to S, which presents measurements to an operator via a screen. Note that P and T are active entities; S is a resource (it just responds to requests from T and P): it may be implemented as a protected resource or a server if it interacts more extensively with the user.

The overall objective of this embedded system is to keep the temperature and pressure of some chemical process within defined limits. A real system of this type would clearly be more complex – allowing, for example, the operator to change the limits. However, even for this simple system, implementation could take one of three forms:

(1) A single program is used which ignores the logical concurrency of T, P and S. No operating system support is required.

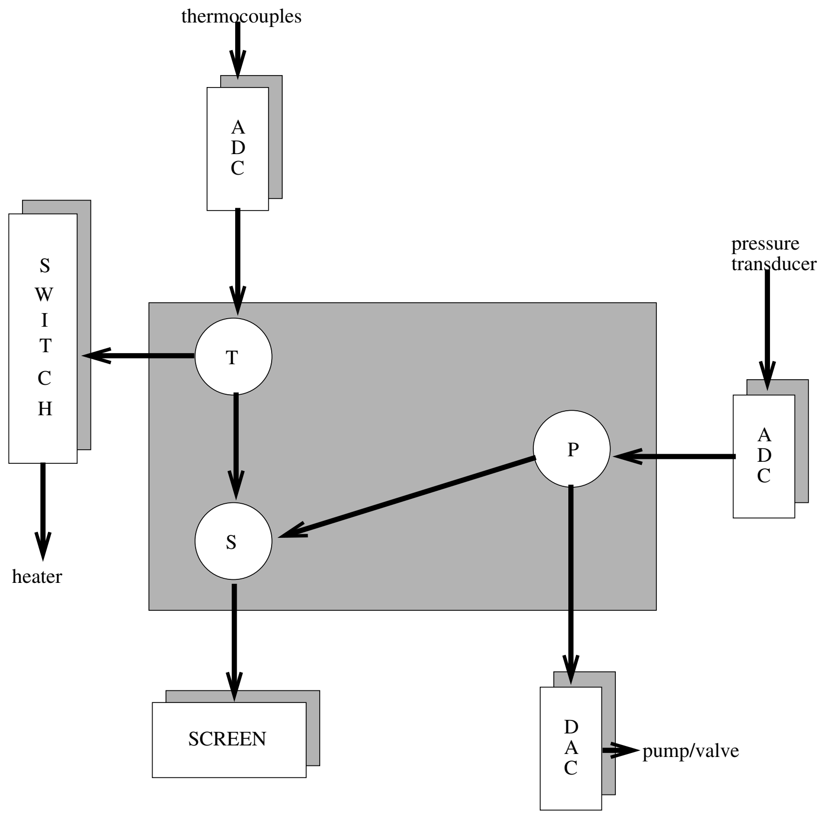


Fig. 2.1: A simple embedded system

(2) T, P and S are written in a sequential programming language (as either separate programs or distinct procedures in the same program) and operating system primitives are used for program/process creation and interaction.

(3) A single concurrent program is used which retains the logical structure of T, P and S. No direct operating system support is required by the program, although a run-time support system is needed.

To illustrate these solutions, consider the Ada code to implement the simple embedded system. In order to simplify the structure of the control software, the following passive packages will be assumed to have been implemented:

```
package Data_Types is
  -- necessary type definitions
  type Temp_Reading is range 10..500;
  type Pressure_Reading is range 0..750;
  type Heater_Setting is (On, Off);
  type Pressure_Setting is range 0..9;
end Data_Types;
```

```
with Data_Types; use Data_Types;
package IO is
  -- procedures for data exchange with the environment
  procedure Read(TR : out Temp_Reading); -- from ADC
  procedure Read(PR : out Pressure_Reading);
    -- note, this is an example of overloading;
    -- two Reads are defined but they have a different
    -- parameter type; this is also the case with the
    -- following Writes
  procedure Write(HS : Heater_Setting);   -- to switch
  procedure Write(PS : Pressure_Setting); -- to DAC
  procedure Write(TR : Temp_Reading);     -- to screen
  procedure Write(PR : Pressure_Reading); -- to screen
end IO;

with Data_Types; use Data_Types;
package Control_Procedures is
  -- procedures for converting a reading into
  -- an appropriate setting for output
  procedure Temp_Convert(TR : Temp_Reading;
                         HS : out Heater_Setting);
  procedure Pressure_Convert(PR : Pressure_Reading;
                             PS : out Pressure_Setting);
end Control_Procedures;
```

2.4.1 Sequential solution

A simple sequential control program could have the following structure (it is known as a *cyclic executive*):

```
with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;
procedure Controller is
  TR : Temp_Reading;
  PR : Pressure_Reading;
  HS : Heater_Setting;
  PS : Pressure_Setting;
begin
  loop
    Read(TR);     -- from ADC
    Temp_Convert(TR,HS); -- convert reading to setting
    Write(HS);    -- to switch
    Write(TR);    -- to screen
    Read(PR);     -- as above for pressure
    Pressure_Convert(PR,PS);
    Write(PS);
    Write(PR);
  end loop; -- infinite loop, common in embedded software
end Controller;
```

This code has the immediate handicap that temperature and pressure readings must be taken at the same rate, which may not be in accordance with requirements. The use of counters and appropriate *if* statements will improve the situation, but it may still be necessary to split the computationally intensive sections (the procedures `Temp_Convert` and `Pressure_Convert`) into a number of distinct actions, and interleave these actions so as to meet a required balance of work. Even if this were done, there remains a serious drawback with this program structure: while waiting to read a temperature, no attention can be given to pressure (and vice versa). Moreover, if there is a system failure that results in, say, control never returning from the temperature `Read`, then in addition to this problem no further pressure `Reads` would be taken.

An improvement on this sequential program can be made by including two boolean functions in the package `IO`, `Ready_Temp` and `Ready_Pres`, to indicate the availability of an item to read. The control program then becomes

```
with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;
procedure Improved_Controller is
  TR : Temp_Reading;
  PR : Pressure_Reading;
  HS : Heater_Setting;
  PS : Pressure_Setting;
begin
  loop
    if Ready_Temp then
      Read(TR);
      Temp_Convert(TR,HS);
      Write(HS);    -- assuming write to be reliable
      Write(TR);
    end if;
    if Ready_Pres then
      Read(PR);
      Pressure_Convert(PR,PS);
      Write(PS);
      Write(PR);
    end if;
  end loop;
end Improved_Controller;
```

This solution is more reliable; unfortunately the program now spends a high proportion of its time in a ‘busy loop’ polling the input devices to see if they are ready. Busy-waits are, in general, unacceptably inefficient. They tie up the processor and make it very difficult to impose a queue discipline on waiting requests. Moreover, programs that rely on busy-waiting are difficult to design, understand or prove correct.

The major criticism that can be levelled at the sequential program is that no recognition is given to the fact that the pressure and temperature cycles are entirely independent subsystems.

2.4.2 Using operating system primitives

All operating systems provide facilities for creating concurrent processes. Usually, these processes execute a single sequential program; however, in recent years there has been a tendency to provide the facilities for processes to be created from within programs. Modern operating systems allow processes created from within the same program to have unrestricted access to shared memory (such processes are often called *threads*).

Consider a simple operating system which allows a new process/thread to be created and started by calling the following subprograms:

```
package Operating_System_Interface is
  type Thread_ID is private;
  type Thread is access procedure;

  function Create_Thread(Code : Thread) return Thread_ID;
  procedure Start(ID : Thread_ID);
  -- other subprograms for thread interaction
private
 type Thread_ID is ...;
end Operating_System_Interface;
```

The simple embedded system can now be implemented as follows. First the two controller procedures are placed in a package:

```
package Processes is
 procedure Pressure_Controller;
 procedure Temp_Controller;
end Processes;


with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;
package body Processes is
  procedure Temp_Controller is
    TR : Temp_Reading;
    HS : Heater_Setting;
  begin
    loop
      Read(TR);
      Temp_Convert(TR,HS);
      Write(HS);
      Write(TR);
    end loop;
  end Temp_Controller;
  procedure Pressure_Controller is
    PR : Pressure_Reading;
    PS : Pressure_Setting;
  begin
    loop
      Read(PR);
      Pressure_Convert(PR,PS);
      Write(PS);
      Write(PR);
    end loop;
  end Pressure_Controller;
end Processes;
```

Now the `Controller` procedure can be given:

```
with Operating_System_Interface;
use Operating_System_Interface;
with Processes; use Processes;
procedure Controller is
  TC, PC : Thread_ID;
begin
  -- create the threads
  -- 'Access returns a pointer to the procedure
  TC := Create_Thread(Temp_Controller'Access);
  PC := Create_Thread(Pressure_Controller'Access);
  Start(TC);
  Start(PC);
end Controller;
```

Procedures `Temp_Controller` and `Pressure_Controller` execute concurrently and each contains an indefinite loop within which the control cycle is defined. While one thread is suspended waiting for a read, the other may be executing; if they are both suspended, a busy loop is not executed.

Although this solution does have advantages over the cyclic executive solution, the lack of language support for expressing concurrency means that the program can become difficult to write and maintain. For the simple example given above, the added complexity is manageable. However, for large systems with many concurrent processes and potentially complex interactions between them, having a procedural interface obscures the structure of the program. For example, it is not obvious which procedures are really procedures or which ones are intended to be concurrent activities. Not only is this confusing for programmers, it is a problem for compiler writers who have to avoid some optimisations that are inappropriate when variables are being shared between multiple tasks.

2.4.3 Using a concurrent programming language

In a concurrent programming language, concurrent activities can be identified explicitly in the code:

```
with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;
procedure Controller is

  task Temp_Controller;
  task Pressure_Controller;

  task body Temp_Controller is
    TR : Temp_Reading;
    HS : Heater_Setting;
  begin
    loop
      Read(TR);
      Temp_Convert(TR,HS);
      Write(HS);
      Write(TR);
    end loop;
  end Temp_Controller;

  task body Pressure_Controller is
    PR : Pressure_Reading;
    PS : Pressure_Setting;
  begin
    loop
      Read(PR);
      Pressure_Convert(PR,PS);
      Write(PS);
      Write(PR);
    end loop;
  end Pressure_Controller;
begin
  null;
  -- Temp_Controller and Pressure_Controller
  -- have started their executions
end Controller;
```

The logic of the application is now reflected in the code; the inherent parallelism of the domain is represented by concurrently executing tasks in the program. The following refinement to the code allows the controllers to execute periodically. Let the pressure controller, `Pressure_Controller`, have a period of 30 ms; as the temperature in the environment changes slowly, a period of 70 ms is adequate for the temperature controller, `Temp_Controller`. The code (without `Screen_Controller`) is as follows:

```
with Ada.Real_Time; use Ada.Real_Time;
with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;
procedure Controller is
  task Temp_Controller;
  task Pressure_Controller;

  task body Temp_Controller is
    TR : Temp_Reading;
    HS : Heater_Setting;
    Next : Time;
    Interval : Time_Span := Milliseconds(30);
  begin
    Next := Clock;  -- start time
    loop
      Read(TR);
      Temp_Convert(TR,HS);
      Write(HS);
      Write(TR);
      Next := Next + Interval;
      delay until Next;
    end loop;
  end Temp_Controller;

  task body Pressure_Controller is
    PR : Pressure_Reading;
    PS : Pressure_Setting;
    Next : Time;
    Interval : Time_Span := Milliseconds(70);
  begin
    Next := Clock;  -- start time
    loop
      Read(PR);
      Pressure_Convert(PR,PS);
      Write(PS);
      Write(PR);
      Next := Next + Interval;
      delay until Next;
    end loop;
  end Pressure_Controller;
begin
  null;
end Controller;
```

Although an improvement, one major problem remains with this two-task solution. Both `Temp_Controller` and `Pressure_Controller` send data to the screen, but the screen is a resource that can only sensibly be accessed by one task at a time. In Figure 2.1, control over the screen was given to a third entity (S) which will need a representation in the program – `Screen_Controller`.

This entity may be a server or a protected resource (depending on the required behaviour of `Screen_Controller`). This has transposed the problem from one of concurrent access to a non-concurrent resource to one of inter-task communication, or at least communication between a task and some other concurrency primitive. The `Temp_Controller` and `Pressure_Controller` need to pass data to the `Screen_Controller`. Moreover, `Screen_Controller` must ensure that it deals with only one request at a time. These requirements and difficulties are of primary importance in the design of concurrent programs. Therefore, before considering in detail how Ada programmers address these problems, the next chapter will concentrate on inter-process communication and what structures concurrent languages have employed.

2.4.4 Operating-systems-defined versus language-defined concurrency

There has been a long debate amongst programmers, language designers and operating system designers as to whether it is appropriate to provide support for concurrency in a language or whether this should be provided by the operating system only. Arguments in favour of including concurrency in the programming languages include the following:

(1) It leads to more readable and maintainable programs.
(2) Compilers for languages that are unaware of potential concurrent executions of program components may introduce race conditions in an attempt to optimise code execution.
(3) There are many different types of operating system; even operating systems that conform to international standards have subtly different behaviours. Defining the concurrency in the language makes the program more portable.
(4) An embedded computer may not have any resident operating system available.

Arguments against concurrency in a language include the following:

(1) Different languages have different models of concurrency; it is easier to compose programs from different languages if they all use the same operating system model of concurrency.
(2) It may be difficult to implement a language's model of concurrency efficiently on top of an operating system's model.
(3) International operating system standards, such as POSIX and Linux, have emerged and, therefore, programs become more portable.

This debate will no doubt continue for some time. The Ada philosophy is that the advantages outweigh the disadvantages, and it therefore supports concurrency directly at the language level. Ada is not unique in taking this view. Languages such as Java and C# provide at least a standard interface as part of the language definition. Even C++ is moving in this direction.

2.5 Summary

A definition of concurrency has been given in this chapter, and a comparison has been made between concurrency and parallelism. Many uses of concurrency have been outlined. A model of a concurrent program has been given, in which active, passive, protected and server entities are defined. Issues of process representation and implementation have been illustrated by the use of an example of an embedded real-time system.

2.6 Further reading

G. R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison-Wesley, 2000.
J. Bacon, *Concurrent Systems*, 2nd Edition, Addison-Wesley, 1998.
M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, 2nd Edition, Addison-Wesley, 2006.

3
Inter-process communication

The major difficulties associated with concurrent programming arise from process interaction. Rarely are processes as independent of one another as they were in the simple example of the previous chapter. One of the main objectives of embedded systems design is to specify those activities that should be represented as processes (that is, active entities and servers), and those that are more accurately represented as protected entities (that is, resources). It is also critically important to indicate the nature of the interfaces between these concurrent objects. This chapter reviews several historically significant inter-process communication primitives: shared variables, semaphores, monitors and message passing. Before considering language primitives, however, it is necessary to discuss the inherent properties of inter-process communication. This discussion will be structured using the following headings:

- Data communication;
- Synchronisation;
- Deadlocks and indefinite postponements;
- System performance, correctness and reliability.

These are the themes that have influenced the design of the Ada tasking model.

As this model directly represents active and protected entities, there are two main forms of communication between active tasks:

(1) direct – task-to-task communication;
(2) indirect – communication via a protected resource.

Both these models are appropriate in Ada programs. In the following sections, however, we start by considering the problems of communicating indirectly by the use of only passive entities.

3.1 Data communication

The partitioning of a system into tasks invariably leads to the requirement that these tasks exchange data in order for the system to function correctly. For example, a device driver (a process with sole control over an external device) needs to receive requests from other processes and return data if appropriate.

Data communication is, in general, based upon either shared variables or message passing. Shared variables are passive entities to which more than one process has access; communication can therefore proceed by each process referencing these variables when appropriate. Message passing involves the explicit exchange of data between two processes by means of a message that passes from one process to the other. Ada provides for shared variables and a type of message-passing structure (called a rendezvous).

Although shared variables are a straightforward means of passing information between processes, their unrestricted use is unreliable and unsafe. Consider the following assignment:

```
X := X + 1;
```

This will most probably be implemented in three stages:

(1) Copy value of X into some local register.
(2) Add 1 to register.
(3) Store the value of the register in the address for X.

If two or more processes are assigning values to X, it is possible from the nature of the concurrency for unexpected values to be given to the shared variable. For example, let processes P1 and P2 execute the above assignment concurrently with X initially 0. A possible sequence of actions is:

(1) P1 copies value of X into its register (X(P1) = 0).
(2) P1 adds 1 to its register (X(P1) = 1).
(3) P2 copies value of X into its register (X(P2) = 0).
(4) P2 adds 1 to its register (X(P2) = 1).
(5) P2 stores its value of X (X = 1).
(6) P1 stores its value of X (X = 1).

This interleaving of the executions of P1 and P2 results in X having the final value of 1 rather than the expected value of 2. Only if the assignment statement for X is an indivisible operation would the integrity of the variable be assured. Because of this *multiple update problem*, shared variables are usually only employed in concert with operators that give indivisible actions (see for example, Section 3.7). Otherwise, the shared variable must be implemented as a protected resource.

3.2 Synchronisation

Although processes execute essentially independently, there are situations where it is necessary for two or more processes to coordinate their executions. For example, in order for a process to receive a message, it is necessary for another process to have first sent this message.

Definition: Synchronisation occurs when one process possesses knowledge about the state of another process.

In most instances, data communication will necessitate synchronisation. Indeed, with the Ada rendezvous, communication and synchronisation are closely related in the same basic mechanism.

With languages that allow communication through shared objects, there are two particularly important classes of synchronisation: *mutual exclusion* and *condition synchronisation*. The execution of a program implies the use of resources (files, devices, memory locations), many of which can only be safely used by one process at a time. A shared variable is itself an example of such a resource.

Definition: Mutual exclusion is a synchronisation that ensures that while one process is accessing a shared variable (or other such resource), no other process can gain access. The sequence of statements that manipulates the shared resource is called a *critical section*. It may be a single assignment (such as the `X := X + 1` described above) or it may be more complex (for example, a file update).

One means of defining mutual exclusion is to treat a critical section as an indivisible operation. The complete actions on the resource must therefore have been performed before any other process could execute any, possibly corrupting, action.

Definition: Condition synchronisation is the synchronisation that ensures that a process only accesses a resource when the resource is in a particular state.

Condition synchronisation is necessary when a process wishes to perform an operation that can only sensibly, or safely, be performed if another task has itself taken some action or is in some defined state. If two processes are communicating via a shared variable, then the receiver of the data must be able to know that the sender has made the appropriate assignment to this variable. In this case, the sender does not need to synchronise with the receiver; in other cases, the data communication may be in both directions, or the sender may wish to know that the receiver has taken the data. Here, both processes must perform condition synchronisation.

Another example of condition synchronisation comes with the use of buffers. Two processes that pass data between them may perform better if the communication is not direct but, rather, via a buffer. This has the advantage of de-coupling the processes and allows for small fluctuations in the speeds at which the two processes are working. The use of this structure (two processes communicating via a buffer) is commonplace in concurrent programs and is known as the *producer/consumer* system. Two condition synchronisations are necessary if a finite (bounded) buffer is used. Firstly, the producer process must not attempt to deposit data in the buffer if the buffer is full. Secondly, the consumer process cannot be allowed to extract objects from the buffer if the buffer is empty. If the buffer is serving more than one producer or consumer, then mutual exclusion over the buffer elements themselves will also be needed.

3.3 Deadlocks and indefinite postponements

The above synchronisations, although necessary, lead to difficulties that must be considered in the design of concurrent programs (rather than in the design of concurrent languages – in which it is usually not practical to remove the possibility of these difficulties arising). A deadlock is the most serious condition and entails a set of processes being in a state from which it is impossible for any of the processes to proceed. Consider two processes (P1 and P2) and two resources (R1 and R2), each of which must be accessed without interference. Access to the resources must be via critical sections that preclude further access. Let P1 have acquired access to R1 and P2 have acquired access to R2; the program structure may then allow the following interleaving:

```
P1 retains access to R1 but also requires R2

P2 retains access to R2 but also requires R1
```

Mutual exclusion will ensure that concurrent access does not take place, but unfortunately both P1 and P2 are deadlocked as a consequence: neither can proceed.

Four necessary conditions must hold if deadlock is to occur.

(1) *Mutual exclusion* – only one process can use a resource at once.
(2) *Hold and wait* – there must exist processes that are holding resources while waiting for others.
(3) *No preemption* – a resource can only be released voluntarily by a process.
(4) *Circular wait* – a circular chain of processes must exist such that each process holds resources which are being requested by the next process in the chain.

The testing of software rarely removes other than the most obvious deadlocks; they can occur infrequently but with devastating results. Two distinct approaches to the deadlock problem can be taken. One can attempt to prove that deadlocks are not possible in the particular program under investigation. Although difficult, this is clearly the correct approach to take and is helped by programming in an appropriate style (for example see the Ravenscar profile for Ada in Section 17.2).

Alternatively, one can attempt to avoid deadlocks whilst having contingency plans available if they do occur. These actions can be grouped together under three headings:

- deadlock avoidance;
- deadlock detection;
- recovery from deadlock.

Avoidance algorithms attempt to look ahead and stop the system moving into a state that will potentially lead to a deadlock. Detection and recovery are never painless and must involve, for real-time systems, the aborting (or backing off) of at least one process with the preemptive removal of resources from that process. Deadlocks are a particular problem with operating systems and considerable analysis has been undertaken on deadlocks within this context (see, for example, almost any book on operating systems).

Indefinite postponement (sometimes called *lockout* or *starvation*) is a less severe condition whereby a process that wishes to gain access to a resource, via a critical section, is never allowed to do so because there are always other processes gaining access before it. If entry to a critical section is in the order of the requests that have been made, then indefinite postponement is not possible. However, if processes have priorities, and if the order of entry is dependent on these priorities, then a low priority process may be postponed indefinitely by the existence of higher priority requests. Even if the postponement is not indefinite, but indeterminate, it may not be possible to make assertions about the program's behaviour.

An alternative view of indefinite postponements is obtained by considering the opposite criteria, namely those of *fairness*, which may be defined as equal chances for equal priority processes, and *liveness*. This latter property implies that if a process wishes to perform some action, then it will eventually be allowed to do so. In particular, if a process requests access to a critical section, then it will gain access in some finite time. If requests are being actively processed and there is a defined upper limit on how long a process will be delayed, then the synchronisation mechanism is said to be *bounded fair*. This property is needed in 'hard' real-time programs, where static analysis must be able to determine the worst-case response time of all actions that have deadlines imposed upon them.

3.4 System performance, correctness and reliability

Both mutual exclusion and condition synchronisation give rise to situations in which a process must be delayed; it cannot be allowed to continue its execution until some future event has occurred. Message-based systems similarly require processes to be suspended. A receiver process must wait until a message arrives and, in other circumstances, a sender process may wish to have it confirmed that the message has reached its destination before proceeding.

One method of 'holding back' a process would be to have it execute a busy-wait loop that tests a continuation flag. This approach was criticised in the previous chapter as being grossly inefficient. What is more, fairness is impossible to prove on a system using busy-waits. Rather, what is required is a mechanism by which a process is 'put to sleep', only to be awakened when circumstances are right for it to proceed. As this future event may be waited upon by a number of processes (although only one may proceed when it occurs, for example access to a critical section), it is necessary to queue processes and to have an appropriate run-time support system for managing this queue. The combination of fairness and process queues gives the foundation for an acceptable system performance. Particular communication methods, and implementation techniques, will nevertheless have varying effects upon this performance.

Given that the run-time performance of a concurrent program is heavily dependent upon the techniques used for implementation, it is important to reiterate that the functional correctness of a program should not depend on these issues. Two informal notions may help to clarify this point:

(1) The execution environment (hardware and run-time support system) is *minimal* if the program eventually undertakes all the input and output activities that are required of it.
(2) A program is functionally correct if, when running on any minimal execution environment, it produces output that conforms to that defined by the program's functional specification.

The topic of program specification, although crucially important, is outside the scope of this book.

A minimal execution environment may, of course, not be sufficient for a real-time system (where each system transaction may have a deadline defined). Its average performance may also not be satisfactory.

The notion of correctness applies across all minimal execution environments, be they single-processor with priority-based or EDF scheduling, single-processor with round-robin scheduling, multiprocessor shared-memory systems, or distributed systems. This makes comprehensive testing difficult. With an incorrect program, it may not be possible to produce invalid results on the available platform for testing. But it may fail on another platform or when the ‘current’ hardware is enhanced. It is therefore critically important that many of the known problems of concurrent programs are tackled during design. For example, the following simple program will probably produce the correct result (‘N = 40’) on most single-processor implementations (because they do not arbitrarily interleave task execution). Nevertheless, the program is incorrect as there is no protection against multiple concurrent updates.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Main is
  N : Integer := 0;
  task type Simple;
  T1, T2 : Simple; -- two tasks defined, both
                   -- execute according to the same body
  task body Simple is
  begin
    for I in 1 .. 20 loop
      N := N + 1;
    end loop;
  end Simple;
begin
  delay 60.0;
  -- any minimal system is assumed to have completed
  -- the two tasks in one minute
  Put("N="); Put(N);
end Main;
```

Reliability is of paramount importance with embedded systems, where software failure may be costly or dangerous. With inter-process communication, the model to be adopted in a language should both be well defined and lead to the production of reliable programs. The possibility of misinterpreting parts of the model should be eliminated, and implementation factors should not infringe upon the semantics of the language.

One aspect of reliability is of general significance: the effect of a single process failure upon the rest of the process system. Failure may be due to errors internal to a process or to external circumstances (for example, the process being aborted); in both cases the impact on all other processes should be minimal. A reliable model will allow a process to terminate without any effect upon other existing processes. However, attempts to communicate or synchronise with a terminated process must be catered for. A process failing whilst executing a critical section presents particular difficulties. A graceful termination may still lead to deadlock if it does not release its mutual exclusive hold over the critical section.

With pure message-based languages, reliability implies that systems should be resilient not only to process failure but also to message loss. This can be difficult. If process P1 sends a message to process P2, and expects a reply, what action should it take if the reply does not materialise? Does it assume that the original message was lost, or that the reply is lost, or that P2 has failed or is running slowly? An easy approach for the language is to assume that the run-time system implements a completely reliable communication system. This may take the form of support for an *atomic transaction*. An atomic transaction either is fully executed or is not executed at all. Therefore if the reply message is received, P2 has acted, once, upon the request; if a reply is not received then P2 has taken no action – it is as if P2 never received the original request. Implementation of atomic transactions is, however, expensive in distributed systems.

3.5 Dining philosophers problem

The dining philosophers problem concerns a system that does not contain computers, but whose behaviour illustrates well the points made in the above sections. Five Chinese philosophers are seated, permanently, around a circular table; between each pair of philosophers there is a single chopstick (so there are only five chopsticks). Each philosopher can be considered as an active concurrent process whose existence is entirely made up of either eating or thinking (Cogito ergo sum: ergo deinde consumo!).

Even though each philosopher is well versed in Chinese culture, they find it impossible to eat with only one chopstick. Therefore, at most two philosophers can be eating at any one time. The chopsticks are a scarce protected resource (that is, they cannot be broken in two). It is assumed that philosophers are too polite to lean across the table to obtain a chopstick (and too engrossed in thought to leave and find more) and thus they only make use of the chopsticks on either side of them. This philosopher system illustrates many of the important aspects of inter-process communication:

- Data communication – chopsticks may be passed directly from one philosopher to another or be held in a pool of available resources.
- Mutual exclusion – access to each chopstick must be via a critical section as two philosophers cannot have concurrent access to the same chopstick.
- Resource implementation – each chopstick could be implemented via a server that passes it to the philosophers, or as a protected resource (that is, with mutual exclusion).
- Condition synchronisation – a philosopher cannot eat until two chopsticks are free.
- Deadlocks – a solution of the form

```
loop
  pick_up_left_chopstick;
  pick_up_right_chopstick;
  eat;
  put_down_left_chopstick;
  put_down_right_chopstick;
  think;
end loop;
```

will lead to a deadlock if all philosophers pick up their left chopstick.

- Indefinite postponement – it is possible for two philosophers to starve (literally) the philosopher sitting between them. If they eat alternately, the middle philosopher will never have both chopsticks.
- Efficient waiting – should a philosopher who wishes to eat constantly check on the amount of food remaining on his or her neighbour's plate? Or should the philosopher sleep and rely on a neighbouring philosopher to wake him or her up when at least one chopstick is free?
- System reliability – will the death of a philosopher alarm the others unduly? (Yes, if it is while eating and the chopsticks depart with the dying philosopher.)

An Ada solution to the dining philosophers problem is given later in Section 6.11.

3.6 Shared variables and protected variables

From the above analysis, it is clear that the Ada tasking model should be reliable, have an efficient wait mechanism, provide for synchronisation, data communication and in particular mutual exclusion, and should enable software to be designed that is free from deadlocks and indefinite postponements. The simplest way that two Ada tasks can interact is via variables that are in scope for both tasks; however, the safe use of these shared variables is not straightforward, as was illustrated by the multiple update problem described above. It is possible to provide mutual exclusion by using only shared variables but a reliable solution that is free from deadlocks and indefinite postponements is not trivial. The following code implements Dekker's algorithm (Francez and Pnueli, 1978) for mutual exclusion:

```
procedure Dekker is
  task T1;
  task T2;
  type Flag is (Up, Down);
  Flag_1, Flag_2 : Flag := Down;
    -- 'up' implies intention to enter critical section
  Turn : Integer range 1..2 := 1;
    -- used to arbitrate between tasks if they both
    -- wish to enter their critical section concurrently

  task body T1 is
  begin
    loop
      Flag_1 := Up;
      while Flag_2 = Up loop
        if Turn = 2 then
          Flag_1 := Down;
          while Turn = 2 loop
            null;  -- busy-wait
          end loop;
          Flag_1 := Up;
        end if;
      end loop;
      -- critical section
      Turn := 2;
      Flag_1 := Down;
      -- rest of task
    end loop;
  end T1;

  task body T2 is
  begin
    loop
      Flag_2 := Up;
      while Flag_1 = Up loop
        if Turn = 1 then
          Flag_2 := Down;
          while Turn = 1 loop
            null;  -- busy-wait
          end loop;
          Flag_2 := Up;
        end if;
      end loop;
      -- critical section
      Turn := 1;
      Flag_2 := Down;
      -- rest of task
    end loop;
  end T2;
begin
  null;
end Dekker;
```

In order to protect the critical section, three extra shared variables are necessary.† When `T1` wishes to enter its critical section, it first announces this to `T2` by raising

† Note, that this solution assumes that the compiler does not attempt to optimise the code by keeping variables in a task-specific register whilst a task is manipulating them. See Section 7.13 for how a pragma can be used to prohibit this optimisation.

its flag (`Flag_1`); if `T2` is executing the 'rest of task' then `Flag_2` will be down, and `T1` can proceed safely into its critical section. However, if an interleaving of `T1` and `T2` leads to both flags being raised concurrently, then the shared variable `Turn` will arbitrate; if `Turn` has the value 2 then `T1` will relinquish its request to enter its critical section by lowering its flag. Only when `Turn` takes the value 1 (that is, when `T2` has left its critical section) will `T2` reset its flag. The switching of the value of `Turn` eliminates indefinite postponement.

This solution can be extended to more than two tasks but the protocols become even more complex. The program illustrates the difficulties of using shared variables as the basis for inter-process communication. The criticism can be summarised as follows:

- An unreliable (rogue) task that misuses shared variables will corrupt the entire system: hence the solution is unreliable.
- No other wait mechanism is provided, only busy-waiting is possible.
- Programming synchronisations and communications is non-trivial and error-prone.
- Testing programs may not examine rare interleavings that break mutual exclusion or lead to livelocks – a livelock is similar to a deadlock except that the processes involved are busy-waiting.
- With large programs, readability, and hence understanding, is poor.
- Analysis of code to prove the non-existence of deadlocks/livelocks is almost impossible with large programs.
- There may be no shared physical memory between the processes, as they may be executing in a distributed system.

The above shows that shared variables on their own are clearly inappropriate. They are passive entities and cannot easily be used to construct protected resources. Hence other methods have been derived. No concurrent programming language relies entirely upon shared variables. Semaphores and monitors attempt to ease the production of such resources. The alternative approach is to make all resources active (that is, servers) and use message passing.

As semaphores and monitors are important historically, and have influenced the development of Ada, they will be described briefly in the following sections.

3.7 Semaphores

Semaphores were originally designed by Dijkstra (1968) to provide simple primitives for the programming of mutual exclusion and condition synchronisation.

Definition: A semaphore is a non-negative integer variable that, apart from initialisation, can only be acted upon by two procedures: `Wait` and `Send` (called P and V by Dijkstra and `Wait` and `Signal` in other descriptions).

The actions of `Wait` and `Send` can be described as follows:

```
Send(S) executes    S.Sem := S.Sem + 1;

Wait(S) executes    if S.Sem = 0 then
                      suspend until S.Sem > 0;
                    end if;
                    S.Sem := S.Sem - 1;
```

The suspension implies, on implementation, a queue mechanism; a process executing a `Wait` on a zero semaphore will be suspended until the semaphore becomes non-zero. This will occur only when some other process executes a `Send` on that semaphore. The additional important property of `Send` and `Wait` is that they are *indivisible operators*. Because of this property, the mutual update of a semaphore (by the concurrent executions of `Wait` and `Send`) is guaranteed to be reliable.

An Ada package could define an abstract type for all semaphores that have the same initial value:

```
package Semaphore_Package is
  type Semaphore is limited private;
  procedure Wait(S : in out Semaphore);
  procedure Send(S : in out Semaphore);
private
  type Semaphore is
    record
      Sem : Natural := 1;
      ...
    end record;
end Semaphore_Package;
```

Mutual exclusion can now be coded quite easily:

```
with Semaphore_Package;
use Semaphore_Package;
procedure Mutual_Exclusion is
  Mutex : Semaphore;
  task T1;
  task T2;

  task body T1 is
  begin
    loop
      Wait(Mutex);
      -- critical section.
      Send(Mutex);
      -- rest of the task.
    end loop;
  end T1;

  task body T2 is
  begin
    loop
      Wait(Mutex);
      -- critical section.
      Send(Mutex);
      -- rest of the task.
    end loop;
  end T2;
begin
  null;
end Mutual_Exclusion;
```

The semaphore is initially set to the value one, so the first process (task `T1`, say) to execute `Wait` will not be delayed but will change the value of the semaphore to zero. If the other task now executes `Wait` it will be suspended; only when `T1` executes `Send` (that is, when it has left the critical section) will `T2` be allowed to continue. The generalisation of this program to any number of tasks presents no further difficulties. If `N` tasks are suspended on a zero semaphore, then a `Send` will unblock exactly one of them. `Send` will increment the semaphore; one blocked task will proceed and in doing so it will decrement the semaphore back to zero (on implementation, these pointless assignments to the semaphore can be eliminated in cases where a `Send` unblocks a waiting process).

The initial value of the semaphore sets the maximum concurrency through the critical section. If the value is one, then mutual exclusion is assured; alternatively if it has a value of, say, six, then up to six processes may have concurrent access. For condition synchronisation, an initial value of zero is often employed; the condition for which some process is waiting can be announced by executing a `Send`. By executing a `Wait` on a semaphore initialised to zero, the waiting process will be suspended until the condition has occurred.

Although semaphores give appropriate synchronisations that will allow, for example, safe data communication using shared variables, they can be criticised as being too low level and not sufficiently structured for reliable use. Just as with shared variables, a simple misuse of a semaphore, although less likely, will still corrupt the whole program. What is more, it cannot be assumed that all processes will make reference to the appropriate semaphore before entering their critical sections. If mutual exclusion is used to protect a resource, then reliability requires all calling processes to use the appropriate access protocols. A different approach would have the resource itself protecting usage – this is essentially the solution adopted with monitors. Finally, a semaphore solution is only partially reliable when faced with process failure. If a process terminates, or is aborted, while executing a critical section, then it will never execute `Send`; no other process will gain access and deadlock is almost certain to ensue. (Note in Ada, controlled variables can be used to help solve this problem, see Subsection 6.6.1.) The main advantage of semaphores is that they can be implemented efficiently.

3.8 Monitors

Monitors attempt to alleviate some of the problems with semaphores.

Definition: A monitor is an encapsulation of a resource definition where all operators that manipulate the resource execute under mutual exclusion.

In Ada terms, a monitor can be viewed as a package in which only procedures are defined in the specification; the resource is hidden in the monitor body and can only be accessed via these procedures. However, unlike a package, a monitor exercises control over calls to its external subprograms. Specifically, execution of monitor procedures is guaranteed, by definition, to be mutually exclusive. This ensures that variables declared in a monitor body can never be subject to concurrent access (this is not the case with ordinary packages). Monitors (in various guises) have been used in numerous programming languages over the years; they can be found in older languages like Mesa, Pascal Plus and Modula, through to the modern object-oriented programming languages like Java and C#. The protected type in Ada also bears some resemblance to the monitor (see Chapter 7).

If one considers a syntax for a monitor similar to that of an Ada package, then a simple resource control monitor would have the form

```
monitor SIMPLE_RESOURCE is     -- NOT VALID Ada
  -- this monitor controls access to 8 identical
  -- resources by providing acquire and release operations.
  procedure ACQUIRE;
  procedure RELEASE;
end SIMPLE_RESOURCE;

monitor body SIMPLE_RESOURCE is
  RESOURCE_MAX : constant := 8;
  R : INTEGER range 0..RESOURCE_MAX := RESOURCE_MAX;
                      -- R is the number of free resources.

  procedure ACQUIRE is
  begin
    if R = 0 then
      BLOCK;
    end if;
    R := R - 1;
  end ACQUIRE;
  procedure RELEASE is
  begin
    R := R + 1;
  end RELEASE;
end SIMPLE_RESOURCE;
```

Tasks that wish to make use of this resource need use no mutual exclusion protocols for the calling of either `ACQUIRE` or `RELEASE`. Mutual exclusion is catered for by only one execution of `ACQUIRE` or `RELEASE` being allowed at any one time.

The above code recognises, but does not deal with, the call of an `ACQUIRE` procedure when there are no resources available. This is an example of a condition synchronisation, for the `ACQUIRE` cannot be successful until a call of `RELEASE` is made. Although it would be possible to implement this synchronisation using a semaphore defined within the monitor, Hoare (1974) proposed a simpler primitive. This primitive is called a *condition variable* or *signal* and is again acted upon by two procedures, which by analogy will also be known here as WAIT and SEND. Unlike a semaphore, which has an integer value, a condition variable has no associated value. However, a simple way of understanding the properties of a condition variable is to think of it as a semaphore which always has the value zero. The actions of WAIT and SEND are therefore as follows:

```
WAIT (cond_var)
  always blocks calling process.
SEND (cond_var)
  will unblock a waiting process if there is one,
  otherwise it has no effect.
```

With condition variables, the simple monitor defined above can now be coded as

```
monitor body SIMPLE_RESOURCE is    -- NOT VALID Ada
  RESOURCE_MAX : constant := 8;
  R  : INTEGER range 0..RESOURCE_MAX := RESOURCE_MAX;
  CR : CONDITION_VARIABLE;

  procedure ACQUIRE is
  begin
    if R = 0 then WAIT(CR); end if;
    R := R - 1;
  end ACQUIRE;

  procedure RELEASE is
  begin
    R := R + 1;
    SEND(CR);
  end RELEASE;
end SIMPLE_RESOURCE;
```

If a process executes ACQUIRE when R=0, then it will become blocked and in doing so will release its mutual exclusive hold on the monitor so that other processes may call RELEASE or ACQUIRE. When RELEASE is called, the execution of SEND will unblock the waiting process, which can then proceed. This action could lead to a difficulty as there are now two processes active in the monitor: the unblocked process and the process that freed it. Different methods are used, by different monitor constructs, to ensure the integrity of the monitor in these circumstances. The simplest being that the execution of SEND must be the last action in the procedure. In effect, therefore, the process exits the monitor and passes mutual exclusive control to the unblocked process. If no process is blocked on the condition variable, then one process delayed upon entry to the monitor is freed. Hence, internally blocked processes are usually given priority over those blocked outside (attempting to gain access to) the monitor.

The monitor is a flexible programming aid that provides the means of tackling, reliably, most of the problems encountered with inter-process communication. Its use provides a clear distinction between synchronisation and data communication, the latter being provided by shared objects. The monitor, like a package, is a passive construction. There is no mechanism within the monitor for the dynamic control of the order of executions of incoming procedure calls. Rather, they must be handled in a predefined order and then be blocked if their complete execution is not, at that time, possible. A major criticism of monitor-based languages is that condition synchronisation outside the monitor must be provided by a further mechanism. In Modula, this is achieved by allowing signals to be a general language feature. However, signals (condition variables) suffer from the same criticism as semaphores: they are too low level for reliable use.

This section is completed by a further example. The simple resource control monitor described above is expanded so that requests state the number of resources required. Only when all these resources are available will control pass back to the calling process. Release of resources will cause all blocked processes to check, in turn, to see if enough resources are now available for them. If there are, they proceed; if not, they cycle back and become re-blocked on the same condition variable. The reason for giving this example here is that Ada's method of providing condition synchronisation (the use of *guards*) is not adequate for this problem. An extension to the model (the *requeue*) is needed. But this must wait until Chapter 8 before it can be described:

```
monitor RESOURCE_CONTROL is    -- NOT VALID Ada
  procedure ACQUIRE (AMOUNT : NATURAL);
  procedure RELEASE (AMOUNT : NATURAL);
end monitor;
```

```
monitor body RESOURCE_CONTROL is
  RESOURCE_MAX : constant := 8;
  AVAILABLE: INTEGER range 0..RESOURCE_MAX := RESOURCE_MAX;
  CR: CONDITION_VARIABLE;
  BLOCKED: NATURAL := 0; -- number of blocked processes
  QUEUED: NATURAL;       -- number of blocked processes at
                         -- the time of resource release
  procedure ACQUIRE (AMOUNT : NATURAL) is
  begin
    if AVAILABLE < AMOUNT then
      BLOCKED := BLOCKED + 1;
      loop
        WAIT(CR);
        QUEUED := QUEUED - 1;
        if AVAILABLE >= AMOUNT then
          BLOCKED := BLOCKED - 1;
          AVAILABLE := AVAILABLE - AMOUNT;
          if QUEUED > 0 then SEND(CR); end if;
          return;
        else
          if QUEUED > 0 then SEND(CR); end if;
        end if;
      end loop;
    else
      AVAILABLE := AVAILABLE - AMOUNT;
    end if;
  end ACQUIRE;

  procedure RELEASE (AMOUNT : NATURAL) is
  begin
    AVAILABLE := AVAILABLE + AMOUNT;
    QUEUED := BLOCKED;
    SEND(CR);
  end RELEASE;
end monitor;
```

This algorithm assumes a FIFO queue on the condition variable and has introduced unreliable elements in order to simplify the code (for example, it assumes that processes will only release resources that they have previously acquired). The actual nature of the resource and how it is passed to the calling process for use is similarly omitted for clarity.

Most monitors provide a ‘send all’ facility that allows all processes waiting on a condition variable to be made runnable and to contend for the monitor lock. With such a facility, the above code is significantly simplified.

```
monitor RESOURCE_CONTROL is   -- NOT VALID Ada
  procedure ACQUIRE (AMOUNT : NATURAL);
  procedure RELEASE (AMOUNT : NATURAL);
end monitor;
```

```
monitor body RESOURCE_CONTROL is
  RESOURCE_MAX: constant := 8;
  AVAILABLE: INTEGER range 0..RESOURCE_MAX := RESOURCE_MAX;
  CR: CONDITION_VARIABLE;

  procedure ACQUIRE (AMOUNT : NATURAL) is
  begin
    if AVAILABLE < AMOUNT then
      loop
        WAIT(CR);
        if AVAILABLE >= AMOUNT then
          AVAILABLE := AVAILABLE - AMOUNT;
          return;
        end if;
      end loop;
    else
      AVAILABLE := AVAILABLE - AMOUNT;
    end if;
  end ACQUIRE;

  procedure RELEASE (AMOUNT : NATURAL) is
  begin
    AVAILABLE := AVAILABLE + AMOUNT;
    SEND_ALL(CR);
  end RELEASE;
end monitor;
```

3.9 Message-based communication

Semaphores were introduced for synchronisation to protect shared variables; however, a possible extension to the semaphore idea would be for the semaphore itself to carry the data to be communicated between the synchronised processes. This is the basis of the design of message passing primitives and allows active entities to communicate directly. One process will SEND a message, another process will WAIT for it to arrive. If the process executing the SEND is delayed until the corresponding WAIT is executed, then the message passing is said to be *synchronous*. Alternatively, if the SEND process continues executing arbitrarily, then one has *asynchronous* message passing. The drawback with the asynchronous structure is that the receiving process cannot know the present state of the calling process; it only has information on some previous state. It is even possible that the process initiating the transfer no longer exists by the time the message is processed. To counter this, pairs of asynchronous messages (a SEND followed by a REPLY) can be used to simulate a synchronous message.

Process naming

One of the main issues to be addressed in the design of a message-based concurrent programming language is how destinations, and sources, are designated. The simplest form is for the unique process name to be used; this is called direct naming:

```
send <message> to <process-name>
```

A symmetrical form for the receiving process would be

```
wait <message> from <process-name>
```

Alternatively, an asymmetric form may be used if the receiver is only interested in the existence of a message rather than from whence it came:

```
wait <message>
```

The asymmetric form is particularly useful when the nature of the inter-process communication fits the *client/server* relationship. The server process renders some utility to any number of client processes (usually one client at a time). Therefore, the client must name the server in sending it a message. By comparison, the server will cycle round receiving requests and performing the service. This may necessitate the return of a reply message, but the address for this will be contained in the original message rather than in the syntax of the way it was received.

Where direct naming is inappropriate, *mailboxes* or *channels* may be used as intermediaries between the sending and receiving processes:

```
send <message> to <mailbox>

wait <message> from <mailbox>
```

Selective waiting

With all the above message structures, the receiving process, by executing a wait, commits itself to synchronisation and will be suspended until an actual message arrives. This is, in general, too restrictive; the receiving process may have a number of channels to choose from or it may be in a state that would make it inappropriate to process particular messages. For example, a buffer control process does not wish to wait on an ‘extract’ message if the buffer is empty. Dijkstra (1975) proposed that commands should be ‘guarded and selective’ in order to achieve a flexible program structure. A guarded command has the following form:

```
guard => statement
```

where guard is a boolean expression; if the guard is true then the statement may be executed. Selection is achieved through the use of an alternative statement which can be constructed as follows:

```
select
  G1 => S1;
or
  G2 => S2;
or
  ...
or
  Gn => Sn;
end select;
```

For example,

```
select
  BUFFER_NOT_FULL => WAIT <place_message>
or
  BUFFER_NOT_EMPTY => WAIT <extract_message>
end select;
```

If more than one guard is true, a non-deterministic choice is made from the open alternatives. An important aspect of the guard statement is whether or not the boolean expression can take information contained in the message into account. The language SR does allow a process to 'peek' at its incoming messages; Ada, as will be shown later, does not; although, as noted earlier, it does have a requeue facility.

The rendezvous

The combination of direct naming and synchronous communication is often called a *rendezvous*. Languages usually combine the rendezvous with guards and select statements. This structure has been used in CSP and occam. The motivations for inter-process communication being based on the rendezvous are fourfold:

(1) Data communication and process synchronisation should be considered as inseparable activities.
(2) The communication model should be simple and amenable to analysis.
(3) Direct communication, rather than via a protected third party, is a more straightforward abstraction.
(4) The use of passive shared variables is unreliable.

In CSP both communicating partners are treated equally. Assume that a variable (of type `VEC`) is to be passed between process `A` and process `B`; let `AR` and `BR` be of type `VEC`, then the transmission has the form

```
In process A
  B ! AR

In process B
  A ? BR
```

Each process names its partner; when A and B are ready to communicate (that is, when they execute the above statements), then a rendezvous is said to take place with data passing from the object AR in A to BR in B. The process that executes its command first will be delayed until the other process is ready to rendezvous. Once the communication is complete, the rendezvous is broken and both processes continue their executions independently and concurrently. In the CSP model, a rendezvous is restricted to unidirectional data communication.

Remote invocation/extended rendezvous

Many applications necessitate the coupling of messages in the form of a request followed by a reply. Process A sends a message to process B; B processes the message and sends a reply message to A. Because this type of communication is common, it is desirable to represent it directly in concurrent programming languages.

Procedures, in general, provide this form of processing; *in* parameters are processed to produce *out* parameters. Therefore, a simple construct is to allow a process to call a procedure defined in another process. However, rather than use the term *remote procedure call*, which is an implementation technique for allowing standard procedure calls to be made across a distributed system, a process defines an *entry*. The call of this entry is termed a *remote invocation* (remote as it comes from another process):

```
-- NOT VALID Ada
remote entry SERVICE (A : in SOME_TYPE;
                      B : out SOME_OTHER_TYPE) is
begin
  -- use value of A.
  -- produce value for B.
end SERVICE;
```

Unfortunately, this simple structure is not sufficient. If the entry updates any of the variables of its process, then reentrant access will be unsafe. There is a need to allow the owning process to exercise some control over the use of its entries. This can be achieved by replacing the entry code by an *accept* statement in the body of the process code:

```
if SOME_CONDITION then
  accept SERVICE (A : in SOME_TYPE;
                  B : out SOME_OTHER_TYPE) do
    ...
  end;
end if;
```

or

```
SOME_GUARD =>
  accept SERVICE(...) do
    ...
  end;
```

The code of SERVICE is now executed by the owner process and thus only one execution of SERVICE can take place at any one time.

The language DP (Distributed Processes) was the first to make use of this type of call. It enables an active process to 'export' a collection of 'entries' in a manner that recreates the rules governing the access of monitor procedures.

Ada also supports remote invocation as one of its inter-task communication methods. It differs from the rendezvous of CSP in the following:

(1) The naming is asymmetric – the calling task names (directly) the called task, but not vice versa.
(2) During the rendezvous, data may pass in both directions.

Nevertheless, Ada and CSP have the following features in common:

(1) Tasks wishing to rendezvous are blocked until their partners also indicate a wish to rendezvous.
(2) Guarded and selective commands allow flexible program structures.

The rendezvous in Ada, unlike in CSP, is not just a simple data transfer but may consist of quite complex processing in order to construct the reply parameters. Hence it is often known as an *extended rendezvous*. It takes the form of an *accept* statement as outlined above.

3.9.1 Asynchronous communication in Ada

One of the significant issues in comparing synchronous and asynchronous communication is that, given one of the abstractions, it is always possible to program the other. It has already been noted that two asynchronous messages can simulate a synchronous one. The use of a protected resource or server task between two other active tasks can also implement an asynchronous relationship. This 'duality' was exploited in Ada 83 by the language only giving direct support to synchronous communication (the rendezvous). Unfortunately, the inevitable poor efficiency of a task being used to provide asynchronous message passing is not acceptable in many applications. Indeed, some implementations of Ada attempted to give special 'efficient' support to this type of server task.

The current version of Ada directly supports asynchronous communication by the use of protected resources (called protected types). A protected resource can be placed between any two active tasks wishing to communicate asynchronously.

The implementation of such a protected resource can make use of guarded commands, and hence has equivalent functionality to the rendezvous. Interestingly, the ‘duality’ property has now been used to argue that the rendezvous could be dropped from Ada as synchronous communication can now be programmed using a protected resource. Fortunately, this view has not prevailed; both models are supported directly in Ada 2005.

The following chapters deal with the syntax and semantics of the Ada tasking model. Criticisms of Ada 83 were considerable. Ada 95 was more expressive and arguably easier to use, and led to more efficient program implementation. The Ada 2005 tasking model builds upon the strength of Ada 95 and provides more support for the real-time aspects of the model. It also provides better integration with the object-oriented programming model. Further assessment of the model will be made once the language features have been described in detail.

3.10 Summary

Process interactions require concurrent programming languages to support synchronisation and data communication. Communication can be based on either shared variables or message passing. This chapter has been concerned with the issues involved in defining appropriate inter-process communication primitives. Examples have been used to show how difficult it is to program mutual exclusion using only shared variables. Semaphores have been introduced to simplify these algorithms and to remove busy-waiting. Error conditions, in particular deadlock, livelock and indefinite postponement, have been defined and illustrated using the dining philosophers problem. Semaphores can be criticised as being too low level and error-prone in use. Monitors, by comparison, are a more structured means of providing access to shared variables.

The semantics of message-based communication have been defined by consideration of two main issues:

- the model of synchronisation; and
- the method of process naming.

Variations in the process synchronisation model arise from the semantics of the send operation. Three broad classifications exist:

- Asynchronous – sender process does not wait.
- Synchronous – sender process waits for message to be read.
- Remote invocation – sender process waits for message to be read, acted upon and a reply generated.

Process naming involves two distinct issues: direct or indirect, and symmetry.

In order to increase the expressive power of message-based concurrent programming languages, it is necessary to allow a process to choose between alternative communications. The language primitive that supports this facility is known as a *selective waiting* construct. Here a process can choose between different alternatives; on any particular execution some of these alternatives can be closed off by using a boolean guard.

3.11 Further reading

G. R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison-Wesley, 2000.

J. Bacon, *Concurrent Systems: Operating Systems, Database and Distributed Systems – An Integrated Approach*, Addison-Wesley, 1998.

M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, 2nd Edition, Addison-Wesley, 2006.

A. Burns and G.L. Davies, *Concurrent Programming*, Addison-Wesley, 1993.

D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 1999.

C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1986.

R. Milner, *Communication and Concurrency*, Prentice-Hall, 1989.

J. Peterson, A. Silberschatz and P. Galvin, *Operating System Concepts*, Addison-Wesley, 2001.

A.J. Wellings, *Concurrent and Real-Time Programming in Java*, Wiley, 2004.

4
Task types and objects

In Chapter 2, tasks were introduced informally in order to illustrate some of the properties of concurrent programs. The *Ada Reference Manual* defines the full syntax for a task type as follows:

```
task_type_declaration ::=
  task type defining_identifier [known_discriminant_part]
    [is  [new  interface_list with] task_definition];

task_definition ::=
    {task_item}
[ private
    {task_item ]
end [task_identifier];

task_item ::= entry_declaration | aspect_clause
```

The task body is declared as follows:

```
task_body ::=
  task body defining_identifier is
    declarative_part
  begin
    handled_sequence_of_statements
  end [task_identifier];
```

The full declaration of a task type consists of its specification and its body; the specification contains:

- The type name.
- A discriminant part – this defines the discrete or access parameters that can be passed to instances of the task type at their creation time.
- A list of interfaces – this defines the interfaces that are supported (implemented) by the task.
- A visible part – this defines the task type's entries and representation (aspect)

clauses which are visible to the user of the task type; it also includes the discriminant part.

- A private part – this defines the task type's entries and representation (aspect) clauses, which are invisible to the user of the task type.

Ada 2005 change: Tasks can now be defined to support Java-like interfaces – discussion of this topic is deferred until Chapter 10

Important notes: Two points are worth emphasising:

(1) Arbitrary parameters cannot be declared as discriminants; only parameters of a discrete type or of an access type.

(2) The visible entries define those parts of a task that can be accessed from other tasks; they indicate how other tasks can communicate directly with the task. Aspect clauses are intended solely for interrupt handling (see Subsection 12.2.1).

Example specifications are

```
task type Controller;
-- this task type has no entries; no other task can
-- communicate directly with tasks created from this type

task type Agent(Param: Integer);
-- this task type has no entries but task objects can be
-- passed an integer parameter at their creation time

task type Garage_Attendant(Pump : Pump_Number := 1) is
  -- objects of this task type will allow communication
  -- via two entries;
  -- the number of the pump the attendant is to serve
  -- is passed at task creation time; if no value
  -- is passed, a default of 1 will be used
  entry Serve_Leaded(G : Gallons);
  entry Serve_Unleaded(G : Gallons);
end Garage_Attendant;

task type Cashier is
  -- this task type has a single entry with two 'in'
  -- parameters and one 'out' parameter
  entry Pay(Owed : Money; Paid : Money; Change : out Money);
end Cashier;

task type Telephone_Operator is
  entry Directory_Enquiry(Person : in Name;
      Addr : in Address; Num : out Number);
end Telephone_Operator;
```

```
task type Colour_Printer is new Printer_Interface with
  entry Print(File_Name : String);
end Colour_Printer;
```

A full discussion of inter-task communication is included in Chapter 5. The remainder of this chapter focuses on task creation, activation, execution, finalisation, termination and task hierarchies.

4.1 Task creation

The value of an object of a task type is a task having the entries specified (if any), the values of the discriminants (if any) and an execution defined by the task body. This body may consist of some hidden data declarations and a sequence of statements. A task type can be regarded as a template from which actual tasks are created. Consider the following simple task type, which inputs characters from a user and counts the numbers of digits and alphabetic characters:

```
type User is (Andy, Alan, Neil);
task type Character_Count(Max: Integer := 100;
                          From : User := Andy);

task body Character_Count is
  use Ada.Text_IO;
  use User_IO;
  use Ada.Integer_Text_IO;
  Digit_Count, Alpha_Count, Rest : Natural := 0;
  Ch : Character;
begin
  for I in 1 .. Max loop
    Get_From_User(From, Ch);
      -- Get_From_User is defined in User_IO
    case Ch is
      when '0'..'9' =>
        Digit_Count := Digit_Count + 1;
      when 'a'..'z' | 'A'..'Z' =>
        Alpha_Count := Alpha_Count + 1;
      when others =>
        Rest := Rest + 1;
    end case;
    exit when Ch = '?';
  end loop;
  Put(Digit_Count);
      -- Put is defined in Ada.Integer_Text_IO.
  Put(Alpha_Count);
  Put(Rest);
end Character_Count;
```

All objects of type `Character_Count` will execute this sequence of statements; they will, however, all have their own separate copies of their local data

(`Digit_Count`, `Alpha_Count`, etc.). The discriminants (`Max`, `From`) can be viewed as data constants that either are initialised to the values passed at task creation time or are given default values (if no values are passed). As constants they cannot be altered in the task's body.

The following declarations illustrate how objects of a task type can be created:

```
Main_Controller : Controller;
Attendant1 : Garage_Attendant; -- pump 1 attendant
Attendant : Garage_Attendant(2);  -- pump 2 attendant
Input_Analyser : Character_Count(30, Neil);
                 --input 30 characters from Neil
Another_Analyser : Character_Count;
                 --input 100 characters from Andy
P06 : Colour_Printer;
```

Tasks can also be declared in structured data types as illustrated in the following array and record declarations:

```
type Garage_Forecourt is array (Pump_Number range <>)
     of Garage_Attendant;

type One_Pump_Garage is
record
  P : Garage_Attendant;
  C : Cashier;
end record;

type Garage (Max_No_Pumps : Pump_Number := 1) is
record
  Pumps : Garage_Forecourt(1 .. Max_No_Pumps);
  C1, C2 : Cashier;
end record;
```

Instances of these tasks will only be created when an instance of the array or record type is created:

```
BP_Station : Garage(12);
```

If a task has a discriminant that does not have a default value, it cannot be used in an array declaration. It can, however, be used in a record declaration if the record itself has a discriminant which is passed through to the task. For example, the following is valid:

```
type Named_Agent(Id : Integer) is
record
  Agent_Id : Integer := Id;
  The_Agent : Agent(Id);
end record;
```

Although arrays of tasks can be created easily, assigning values to their discriminants is awkward. For example, in the above declaration of a `Garage`, all tasks were created with the default Pump_Number. This is clearly not what is wanted; ideally all tasks should have been created with a different number. This effect can be achieved by calling a function with a side effect (although this is far from elegant):

```
package Count is
  function Assign_Pump_Number return Pump_Number;
end Count;

subtype New_Garage_Attendant is Garage_Attendant
          (Pump => Count.Assign_Pump_Number);

type Forecourt is array (1..10) of New_Garage_Attendant;
```

where the body of Count is

```
package body Count is
  Number : Pump_Number := 0;
  function Assign_Pump_Number return Pump_Number is
  begin
    Number := Number + 1;
    return Number;
  end Assign_Pump_Number;
end Count;
```

This will ensure that each task receives a different pump number, although it makes no guarantee that Pumps(1) will service Pump_Number 1 (because Ada does not define the order of initialisation of the array elements).

Note that the above example made use of a task subtype. During subtyping only the default value of the discriminant can be changed. No change can be made to the number or form of the entries, or to the structure of the discriminant part.

Task objects and types can be declared in any declarative part, including task bodies themselves. The entries are in scope immediately after the task specification that defines them.

The use of tasks normally necessitates the use of the rendezvous or calls to protected objects, discussion of which is deferred until later chapters. Some algorithmic tasks, however, require no direct communication. Consider the following procedure that will calculate both the sum and difference of two large integer arrays:

```
type Vector is array (1..10_000) of Integer;
type Vector_Prt is access all Vector;
type Vector_Const is access constant Vector;

procedure Sumdif(A, B : Vector_Const;
                 Sum : Vector_Prt;
                 Diff : Vector_Prt);
```

Although a sequential solution is quite straightforward, it is possible to define a concurrent structure that may, with hardware support, be more efficient. Consider the following two concurrent structures for `Sumdif`:

```
procedure Sumdif(A, B : Vector_Const;     -- Method 1
                 Sum : Vector_Prt;
                 Diff : Vector_Prt) is
  task type Minus;
  M : Minus;

  task body Minus is
  begin
    for I in Vector'Range loop
      Diff(I) := A(I) - B(I);
    end loop;
  end Minus;
begin
  for I in Vector'Range loop
    Sum(I) := A(I) + B(I);
  end loop;
end Sumdif; -- method 1

procedure Sumdif(A, B : Vector_Const;     -- Method 2
                  Sum : Vector_Prt;
                  Diff : Vector_Prt) is
  task type Minus;
  task type Plus;
  M : Minus;
  P : Plus;

  task body Minus is
  begin
    for I in Vector'Range loop
      Diff(I) := A(I) - B(I);
    end loop;
  end Minus;

  task body Plus is
  begin
    for I in Vector'Range loop
      Sum(I) := A(I) + B(I);
    end loop;
  end Plus;
begin
  null;
end Sumdif; -- method 2.
```

In the first method, there are two concurrent processes: the task and the main sequence of `Sumdif`, whereas in the second example three processes exist: the two tasks (`Minus` and `Plus`) and the main sequence. In the second method, however, the procedure itself consists of only the null statement. Both these structures are acceptable, the first involves fewer tasks, but the second has a more symmetric form. In each case, the tasks terminate naturally and the procedure itself terminates and returns appropriate values after the contained tasks have finished. (A detailed discussion on task termination is included later in Section 4.2.) Finally, this simple example illustrates that the concurrency deployed is only at an implementation level. The rest of the program, which makes use of `Sumdif`, is unaware that tasks have been employed. The body of the procedure may indeed have been coded in this manner only after the program was moved onto a hardware system that could exploit true parallelism.

Access discriminants

A task need not only manipulate data local to its body. It can access any data which is in scope (as illustrated in the previous example); furthermore, a pointer can also be passed to the data to be manipulated.

Consider, for example, a task that wishes to sort an array of integers; the array concerned being identified when the task is created. The following example illustrates how this can be done using an access discriminant:

```
type AI is array(Positive range <>) of Integer;
type PAI is access all AI;
task type Exchange_Sorter(A : access AI);
X : PAI := new AI(1 .. 100);  -- Initialised elsewhere
Sorter :  Exchange_Sorter(X);

task body Exchange_Sorter is
  Sorted : Boolean := False;
  Tmp : Integer;
begin
  while not Sorted loop
    Sorted := True;
    for I in A'First .. A'Last - 1 loop
      if A(I) < A(I+1) then
        Tmp := A(I);
        A(I) := A(I+1);
        A(I+1) := Tmp;
        Sorted := False;
      end if;
    end loop;
  end loop;
end Exchange_Sorter;
```

4.1.1 Anonymous task types

In the examples used above, a task type was declared followed by an instance of that type. This may appear long-winded and, indeed, Ada allows a shorthand form to be used that hides the explicit declaration of the type. There can, therefore, be anonymous task types in an identical way to there being anonymous array types. For the direct declaration of a single task, the reserved word 'type' is omitted from the specification along with any discriminant part:

```
single_task_declaration ::=
  task defining_identifier [is
      [new interface_list with] task_definition];
```

For example,

```
task Controller;
```

`Controller` now refers to the task object, not its type. The declaration of an anonymous task type is interpreted as being equivalent to the declaration of its type, followed immediately by the declaration of the object:

```
task type Controller_Type;
Controller : Controller_Type;
```

As no construct in Ada allows direct reference to the task's body, this form is unambiguous.

4.1.2 Task access types

If a task object has a defined type, it is possible to provide an access type for that task type:

```
task type Cashier is
  entry Pay(Owed : Money; Paid : Money; Change : out Money);
end Cashier;

type Cashier_Ptr is access Cashier;
```

Task types with associated access variables are often used where the dynamic generation of tasks is required during program execution. The task object itself is created by the evaluation of an allocator:

```
Pointer_Cashier : Cashier_Ptr := new Cashier;
```

A `Cashier` task object has been created and activated at the point of creation; its 'name' is `Pointer_Cashier.all` and its entry is designated `Pointer_Cashier.Pay`.

Dynamic task creation is an important feature of the language because it allows tasks to be created on demand. For many applications, this is better than trying to predict the required number or to impose an upper limit on it. Obviously, there is a limit to the number of tasks a system can support (usually due to memory limitations) but there are clear advantages in not having to fix a static limit artificially on, for example, the number of trackable objects in an air traffic control application.

As well as fully dynamic tasks, it is also possible to obtain a pointer to a task that has been created by an object declaration rather than by an allocator. If pointers to the task are required, then the task must be declared as being ‘aliased’ and an access attribute used. For example,

```
task type Garage_Attendant is ... -- as before
Attendant1 : aliased Garage_Attendant;
```

indicates that a pointer to `Attendant1` may be used:

```
type Ptr_Attendant is access all Garage_Attendant;
PA : Ptr_Attendant;
 ...
PA := Attendant1'Access;
```

After this assignment, there are two ways of referencing the task: `PA` and `Attendant1`.

Primes by sieve example

To give a more extensive example of the use of tasks (and task access types), a program will be developed that implements the ‘primes by sieve’ algorithm (otherwise known as the Sieve of Eratosthenes). The structure of the algorithm is as follows. A task `Odd` generates a continuous stream of odd integers. These are passed down a pipeline of `Sieve` tasks. Each `Sieve` task receives a stream of integers, the first one being a prime number that it keeps in its local variable `Prime`. The task then processes the rest of the numbers, checking each one to see if it can be divided exactly by `Prime`. If it can, it is thrown away; if it cannot, it is passed down the pipeline to the next task.

It is possible to define a static version of this algorithm with, say, N tasks in the pipeline. This will generate the first N primes. However, a more flexible program can be formed if each task creates the next task in the pipeline when it has an integer to pass on. Figure 4.1 illustrates the state of the pipeline when the prime number 11 is found.

In order for the tasks to pass data between themselves, they must be able to communicate with each other. As communication has not yet been directly addressed we shall assume, for now, that the `Sieve` tasks use buffers as the media for communication. A buffer is implemented as a protected type (see Section 7.3).

The `Sieve` task type will be defined with a parameter that is a static pointer to a buffer. New `Sieve` tasks will be created dynamically and so an access type is required:

```
task type Sieve(B : access Buffer);
type Sieve_Ptr is access Sieve;
```

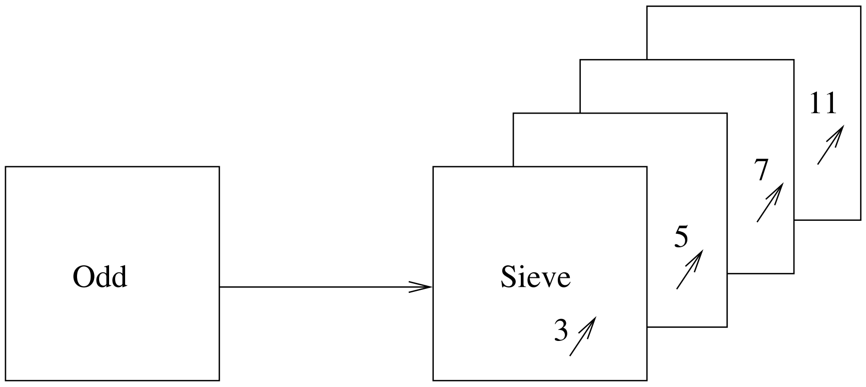


Fig. 4.1: Primes by sieve

```
function Get_New_Sieve(B : access Buffer)
         return Sieve_Ptr is
begin
  return new Sieve(B);
end Get_New_Sieve;
```

There is a single `Odd` task:

```
task Odd;

task body Odd is
  Limit : constant Positive := ...;
  Num : Positive;
  Buf : aliased Buffer;
  S : Sieve_Ptr := Get_New_Sieve(Buf'Access);
begin
  Num := 3;
  while Num <= Limit loop
    Buf.Put(Num);
    Num := Num + 2;
  end loop;
end Odd;
```

The code for the sieve tasks is as follows:

```
task body Sieve is
  New_Buf : aliased Buffer;
  Next_Sieve : Sieve_Ptr;
  Prime, Num : Natural;
begin
  B.Get(Prime);
  -- Prime is a prime number, which could be output
  loop
    B.Get(Num);
    exit when Num rem Prime /= 0;
  end loop;
  -- a number must be passed on and so a new Sieve task
  -- is created; note that it is necessary to use a
  -- function as a task type cannot be used directly
  -- within its own body to create a further task instance
  Next_Sieve := Get_New_Sieve(New_Buf'Access);
  New_Buf.Put(Num);
  loop
    B.Get(Num);
    if Num rem Prime /= 0 then
      New_Buf.Put(Num);
    end if;
  end loop;
end Sieve;
```

Note that this program will not terminate correctly. One way of programming termination, in this example, is to pass a 'close-down' token such as zero down the pipeline. Other methods will be considered in later chapters.

4.2 Task activation, execution, finalisation and termination

A task is said to be *created* by its elaboration.

Definitions: The execution of a task object has three main active phases:

(1) *Activation* – the elaboration of the declarative part, if any, of the task body (any local variables in the body of the task are created and initialised during activation).
(2) *Normal execution* – the execution of the statements within the body of the task.
(3) *Finalisation* – the execution of any finalisation code associated with any objects in its declarative part.

A task, in general, indicates its willingness to begin finalisation by executing its 'end' statement. A task may also begin its finalisation as a result of an unhandled exception, or by executing a select statement with a terminate alternative (see Section 6.6), or by being aborted (see Section 9.2).

Definition: A finished task is called *completed* or *terminated* depending on whether it has any active dependants (see Section 4.3).

4.2.1 Task activation

For static tasks, activation starts immediately after the complete elaboration of the declarative part in which they are defined:

```
declare
  task type T_Type;
  task A;          -- task A is created when this
                   -- declaration is elaborated
  B, C : T_Type;   -- tasks B and C are created when
                   -- these declarations are elaborated
  I, J : Integer;
  task body A is ...
  task body T_Type is ...
begin
  -- All tasks created in the above declarative region
  -- begin to activate as soon as the elaborations of
  -- the declarative region have finished.

  -- Sequence of statements: the first statement is executed
  -- once all tasks have finished their activation.
end;
```

Important notes:

The following points on task activation should be noted.

- All static tasks created within a single declarative region begin their activations immediately the region has completed elaboration (i.e., after ‘begin’ but before any following statements).
- The first statement following the declarative region is not executed until all tasks have finished their activation.
- Following activation, the execution of a task object is defined by the appropriate task body.
- A task need not wait for the activation of other concurrently created tasks before executing its body.
- A task may attempt to communicate with another task which, although created, has not yet been activated. The calling task will be delayed until the communication can take place.

If a task object is declared in a package specification, then it commences its execution after the elaboration of the declarative part of the package body:

```
package Client is
  task Agent;
end Client;

package body Client is
  task body Agent is separate;
begin
  -- Agent starts executing before 1st statement
  -- of this sequence begins executing.
  ...
end Client;
```

If the package does not contain initialisation code, then it acts as if there is a 'null' sequence.

Important notes: Dynamic tasks

(1) are activated immediately after the evaluation of the allocator (the **new** operator) that created them, and
(2) the task, which executed the statement responsible for their creation, is blocked until the tasks created have finished their activation.

The following stylised code illustrates the above points:

```
declare
  -- a declarative region executed (say) by task Parent
  task type T_Type;
  type Prt_T_Type is access T_Type;
  A1 : T_Type;   -- creation of A1
  A2 : T_Type;   -- creation of A2
  task body T_Type is ...
begin -- activation of A1, A2
  declare
    B : T_Type; -- creation of B
    C : Prt_T_Type := new T_Type;
      -- creation, activation of C.all
    D : Prt_T_Type;
  begin -- activation of B
    D := new T_Type; -- creation, activation of D.all
  end;
end;
```

In the above example, the tasks are created and activated by the `Parent` task in the following order:

(1) Task `A1` is created.
(2) Task `A2` is created.
(3) Tasks `A1` and `A2` are activated concurrently and the `Parent` waits for them to finish their activation.
(4) Assuming a successful activation: `A1`, `A2` and `Parent` proceed concurrently.
(5) `Parent` creates task `B`.
(6) `Parent` creates and activates task `C.all`, and waits for it to finish activation.
(7) `Parent` activates task `B`, and waits for it to finish activation.
(8) `Parent` creates and activates task `D.all`, and waits for it to finish activation.

Program errors during task creation and activation

Program errors can lead to exceptions being raised during these initial phases of a task's existence. If an exception is raised in the elaboration of a declarative part, then any task created during that elaboration becomes terminated and is never activated. For example if in the following, `Initial_Value` is a function that returns an integer value, then there is a possibility that it will return a value outside `Data_Range`. This would result in the exception `Constraint_Error` being raised in a declarative part which also declares a task. In this case the `Agent` task is never activated. As the exception is raised on elaboration of the declarative block, it cannot be handled within that block but must be caught at an outer level.

```
declare -- outer block
    ...
begin
  declare -- inner block
    task Agent;
    subtype Data_Range is Integer range 1 .. 10;
    Data_Object : Data_Range := Initial_Value;

    task body Agent is ...

  begin
    ...
  exception
    -- any exception handlers here will only catch
    -- exceptions raised during the 'begin .. exception',
    -- and not those raised in the elaboration
    -- of the declarative part
  end;

exception
  when Constraint_Error =>
    -- recovery routine for exception
    -- raised in the inner block
end;
```

If an exception is raised during a task's activation, then the task itself cannot handle the exception. The task is prevented from executing its body and hence becomes completed or terminated. If any objects have already been created, then they must be finalised (see Section 4.3). As the task itself cannot handle the exception, the language model requires the parent (creator) task or scope to deal with the situation: the predefined exception `Tasking_Error` is raised. In the case of dynamic task creation, the exception is raised after the statement that issued the allocator call. However, if the call is in a declarative part (as part of the initialisation of an object), the declarative part fails and the exception is raised in the surrounding block (or calling subprogram).

For static task creation, the exception is raised prior to the execution of the first statement of the declarative block. This exception is raised after all created tasks have finished their activation (whether successfully or not) and it is raised at most once. For tasks created statically during a declarative part, the associated exception handler must be at the outermost level of the declarative block; therefore little direct error recovery is possible. The attribute `Callable` can, however, at least be used to identify the rogue task (assuming the well-behaved tasks have not terminated correctly before the parent executes its exception handler):

```
declare
  task Child_1;
  task Child_2;
  task body Child_1 is ...
  task body Child_2 is ...
begin
  null;
exception
  when Tasking_Error =>
    if not Child_1'Callable then
      Put("Task Child_1 failed during activation");
    end if;
    if not Child_2'Callable then
      Put("Task Child_2 failed during activation");
    end if;
    if Child_1'Callable and Child_2'Callable then
      Put("Something strange is happening");
    end if;
end;
```

The boolean attribute `Callable` is defined to yield the value `True` if the designated task is neither `Completed`, `Terminated` nor `Abnormal`. (An abnormal task is one that has been aborted; the conditions necessary for an abnormal task to become terminated are discussed in Chapter 9.) In the above example, if task `Child_2` fails in its elaboration, then the declaring block will display an appropriate error message. Task `Child_1` will, however, be unaffected by this and will proceed in its execution. The declarative block will only exit if, and when, `Child_1` subsequently terminates.

Another task attribute is `Terminated`, which returns `True` if the named task has terminated; returns `False` otherwise.

Tasks states without task hierarchies

Figure 4.2 illustrates those task states and their transitions that have been introduced so far. The state transition diagram will be extended in this and later chapters. Note that 'activating' and 'executing' are definitions of the state of the task; they do not imply that the task is actually running on the processor. This is an im-

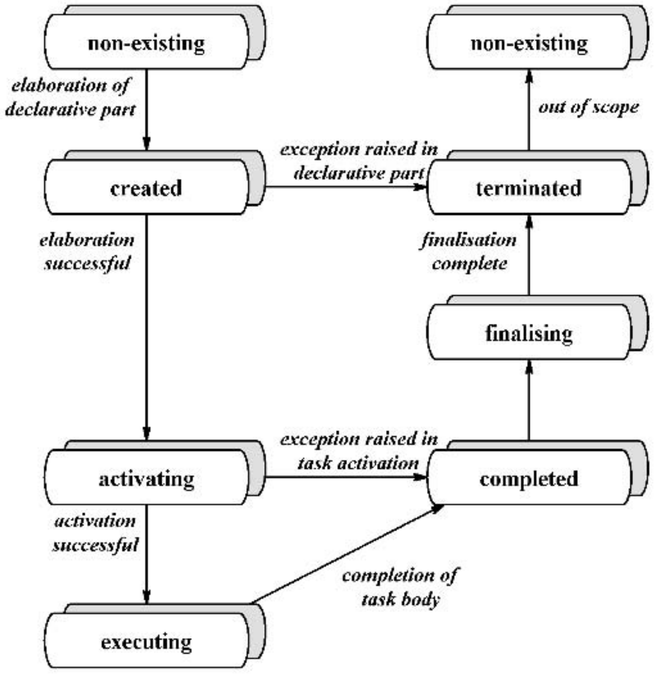


Fig. 4.2: Task states

plementation issue. Thus, for example, the state 'executing' means able to execute (make progress) if processing resources are made available to it.

4.3 Task hierarchies

Ada is a block structured language in which blocks may be nested within blocks. A task can be declared in any block; therefore it is possible for tasks to be declared within tasks (or blocks) which themselves are declared within other tasks (or blocks). This structure is called a task hierarchy. The creation and termination of tasks within a hierarchy affect the activation, execution, termination and finalisation of other tasks in the hierarchy.

4.3.1 Task creation with task hierarchies

Definitions: A task which is directly responsible for creating another task is called the *parent* of the task, and the task it creates is called the *child*.

When a parent creates a child, the parent's execution is suspended while it waits for the child to finish activating (either immediately if the child is created by an allocator, or after the elaboration of the associated declarative part). Once the child has finished its activation, the parent and the child proceed concurrently. If a task creates another task during its activation, then it must also wait for its child to activate before it can begin execution. We shall use the phrase *waiting for child tasks to activate* to indicate the state of a task when it is suspended waiting for children to finish their activation.

For example, consider the following program fragment:

```
task Grandparent;

task body Grandparent is
begin
  ...
  declare        -- inner block
    task Parent;
    task body Parent is

      task type Child;
      Child_1, Child_2 : Child;

      task body Child is
        -- declaration and initialisation
        -- of local variables etc
      begin
        ...
      end Child;

    begin
      ...
    end Parent;
  begin  -- inner block
    ...
  end;   -- inner block
  ...
end Grandparent;
```

The `Grandparent` task begins its execution, it enters an inner block where it is suspended waiting for its child (the `Parent` task) to be activated. During the `Parent` task's activation it creates the two child tasks `Child_1` and `Child_2`, and therefore `Parent` cannot finish its activation and begin its execution until these child tasks have finished their activation. Once *both* `Child_1` *and* `Child_2` have activated, `Parent` finishes its activation, and the `Grandparent` task is free to continue. Note that if `Child_1` finishes its activation before `Child_2`, it is not suspended but is free to execute.

The reason why parent tasks must be suspended whilst their children are activated was explained in the previous section; any problems with the creation and activation of the task must be reported to the parent. If the parent task continued to execute, it would be more difficult for it to respond to what is, in effect, an asynchronous exception. Once a child task is executing, it is considered to be independent from its parent. Any exception raised during its normal execution should be handled by the task itself. Failure to do so causes the task to become completed. Note, no other task will be automatically informed of the unhandled exception.

4.3.2 Task termination with task hierarchies

The parent of a child task is responsible for the creation of that task.

Definition: The *master* of a *dependant* task must wait for that task to terminate before it can itself terminate.

In many cases, the parent of a task is also the master of the task. For example, consider the following:

```
task Parent_And_Master;

task body Parent_And_Master is

  -- declaration and initialisation of local variables

  task Child_And_Dependant_1;

  task Child_And_Dependant_2;

  task body Child_And_Dependant_1 is
    -- declaration and initialisation of local variables
  begin
    ...
  end Child_And_Dependant_1;

  task body Child_And_Dependant_2 is
    -- declaration and initialisation of local variables
  begin
    ...
  end Child_And_Dependant_2;

begin
  ...
end Parent_And_Master;
```

In the above example, the `Parent_And_Master` task creates two child tasks called `Child_And_Dependant_1` and `Child_And_Dependant_2`. Once the child tasks have been activated, they are free to run concurrently with the parent. When the parent finishes its execution, it must wait for its dependant tasks to finish and execute their finalisation code. This is because the dependant tasks can potentially access the local variables of their master. If the master finishes and finalises its local variables (which would then disappear), the dependants would be accessing locations in memory that may have been reused by other tasks.

Definition: To avoid this situation, Ada forces the master task to wait for finalisation and termination of its dependants before it itself can finalise any variables (of which it is the master) and terminate. When a master has finished its execution but cannot terminate because its dependants are still executing, the master is said to be *completed*.

Of course, any dependant tasks cannot themselves terminate until all their dependant tasks have also terminated. For example, in the `Grandparent` example, task `Parent` must wait for `Child_1` and `Child_2` to terminate.

Important note: The master of a task need not be another task directly, but can be a subprogram or a declarative block within another task (or declarative block or main program). That block cannot be exited until all its dependant tasks have terminated and any dependant variables have been finalised.

For example,

```
  -- code within some block, task or procedure
Master:
declare  -- internal Master block

  -- declaration and initialisation of local variables

  -- declaration of any finalisation routines

  task Dependant_1;
  task Dependant_2;

  task body Dependant_1 is
    -- declaration and initialisation of local variables etc
  begin
    ...
    -- potentially can access local variables
    -- in the Master block
  end Dependant_1;

  task body Dependant_2 is
    -- declaration and initialisation of local variables
  begin
    ...
    -- potentially can access local variables
    -- in the Master block
  end Dependant_2;

begin  -- Master block
```

```
  ...
end; -- Master  block
```

On entry to the `Master` block above, the executing parent task is suspended whilst the child tasks are created and activated. Once this has occurred, the `Master` block itself is free to execute. However, before the block can exit it must wait for the dependant tasks to terminate (and any dependant variables to become finalised). When this has occurred, the task executing the block is free to finalise its dependant variables and then continue. We shall use the phrase *waiting for dependant task to terminate* to indicate the state of a task when it is suspended waiting for dependants to terminate and finalise.

Task termination and dynamic tasks

With dynamic task creation, the master of a task is not necessarily its parent.

Definition: The master of a task created by the evaluation of an allocator is the declarative region which contains the access type definition.

For example,

```
task type Dependant;
task body Dependant is ...;

declare  -- Outer Block, master of all tasks
         -- created using Dependant_Ptr
  type Dependant_Ptr is access Dependant;
  A : Dependant_Ptr;
begin
  ...
  declare           -- inner block
    B : Dependant;
    C : Dependant_Ptr:= new Dependant;
    D : Dependant_Ptr:= new Dependant;
  begin
    -- sequence of statements
    A := C;      -- A now points to C.all
  end;  -- must wait for B to terminate but
        -- not C.all or D.all
  ...
  -- C.all and D.all could still be active although
  -- the name C is out of scope, the task C.all can still
  -- be accessed via A.all; D.all cannot be accessed
  -- directly; it is anonymous

end; -- must wait for C.all and D.all to terminate
```

Although `B`, `C`.**all** and `D`.**all** are created within the inner block, only `B` has this block as its master; `C`.**all** and `D`.**all** are considered to be dependants of the outer block and therefore do not affect the termination of the inner block. The inner block is, however, the parent of C.**all** and D.**all** (as well as B).

Task termination and library tasks

Tasks declared in library-level packages have the main program as their master.

Important note: The *main program cannot terminate until all the library-level tasks have terminated.*

Consider the following program:

```
package Library_Of_Useful_Tasks is
  task type Agent(Size : Integer := 128);
  Default_Agent : Agent;
  ...
end Library_Of_Useful_Tasks;  -- a library package.

with Library_Of_Useful_Tasks; use Library_Of_Useful_Tasks;
procedure Main is
  My_Agent : Agent;
begin
  null;
end Main;
```

In Ada, there is a conceptual task (called the *environment task*) that calls the main procedure. Before it does this, it elaborates all library units named in the 'with' clauses by the main procedure. In the above example, this elaboration will cause task `Default_Agent` to be created and activated. Once the task has finished activation, the environment task calls the procedure `Main`, which creates and activates `My_Agent`. This task must terminate before the procedure can exit. The `Default_Agent` must also terminate before the environment task terminates and the whole program finishes. A similar argument applies to tasks created by an allocator whose access type is declared in a library unit: they also have the main program as their master. Note that an exception raised during the activation of `Default_Agent` cannot be handled – the program fails and the main procedure is never executed.

4.4 Task identification

One of the main uses of access variables is to provide another means of naming tasks. All task types in Ada are considered to be limited private. It is therefore not possible to pass a task, by assignment, to another data structure or program unit. For example, if `Robot_Arm` and `New_Arm` are two variables of the same access type (the access type being obtained from a task type), then the following is illegal:

```
Robot_Arm.all := New_Arm.all;  -- not legal Ada
```

However,

```
Robot_Arm := New_Arm;
```

is quite legal and means that `Robot_Arm` is now designating the same task as `New_Arm`.

Warning: Care must be exercised here as duplicated names can cause confusion and lead to programs that are difficult to understand.

In some circumstances, it is useful for a task to have a unique identifier (rather than a name). For example, a server task is not usually concerned with the type of the client tasks. Indeed, when communication and synchronisation are discussed in the next chapter, it will be seen that the server has no direct knowledge of who its clients are. However, there are occasions when a server needs to know that the client task it is communicating with is the same client task that it previously communicated with. Although the core Ada language provides no such facility, the Systems Programming Annex provides a mechanism by which a task can obtain its own unique identification. This can then be passed to other tasks:

```
package Ada.Task_Identification is

  type Task_Id is private;
  Null_Task_Id : constant Task_Id;

  function "=" (Left, Right : Task_Id) return Boolean;

  function Current_Task return Task_Id;
    -- returns unique id of calling task

  -- other functions not relevant to this discussion
private
  ...
end Ada.Task_Identification;
```

As well as this package, the Annex supports two attributes:

- For any prefix `T` of a task type, `T'Identity` returns a value of type `Task_Id` that equals the unique identifier of the task denoted by `T`.
- For any prefix `E` that denotes an entry declaration, `E'Caller` returns a value of type `Task_Id` that equals the unique identifier of the task whose entry call is being serviced. The attribute is only allowed inside an entry body or an accept statement (see Chapters 5 and 7).

Warning: Care must be taken when using task identifiers since there is no guarantee that, at some later time, the task will still be active or even in scope.

4.5 Task creation, communication and synchronisation within task finalisation

This chapter has discussed how a 'completed' task finalises any dependant variables before becoming 'terminated' and how a master block finalises any dependant variables before exiting. This act of finalisation can itself, in theory, be complex and involve task creation, termination, communication etc. However, this is not recommended, and issues resulting from this will not be discussed in this book. It should be noted, nevertheless, that it is a bounded error for a finalisation routine to propagate an exception.

Warning: Bounded errors in Ada are error conditions that have several possible defined outcomes, one of which is always the raising of the `Program_Error` exception.

Although task hierarchies are a powerful structuring tool, they can give rise to complex initialisation and termination behaviours. Wherever possible, these complexities should be avoided. For example, task failures during activation should be prevented (by assigning initial values to all local data variables in the execution part of the task rather than on variable declaration).

4.6 Summary

This chapter has concerned itself with the behaviour of Ada tasks. A number of states have been defined (summarised in Figure 4.3) that help to distinguish between the various phases of a task's execution. In general, a task is created by its parent, goes through a phase of activation while its declarative part is being elaborated, and then becomes an independent executing entity. Although in some applications, tasks will be expected to run 'forever', in terminating programs each task will first complete its code, then wait for its dependant tasks to terminate before finalising any controlled variables (and terminating).

Tasks can be characterised as being either static (that is, by declaring a variable of a task type) or dynamic. Dynamic tasks are created by an allocator referring to an access type. Because of different dependency relationships, the rules for activation and termination are different for these two forms of task.

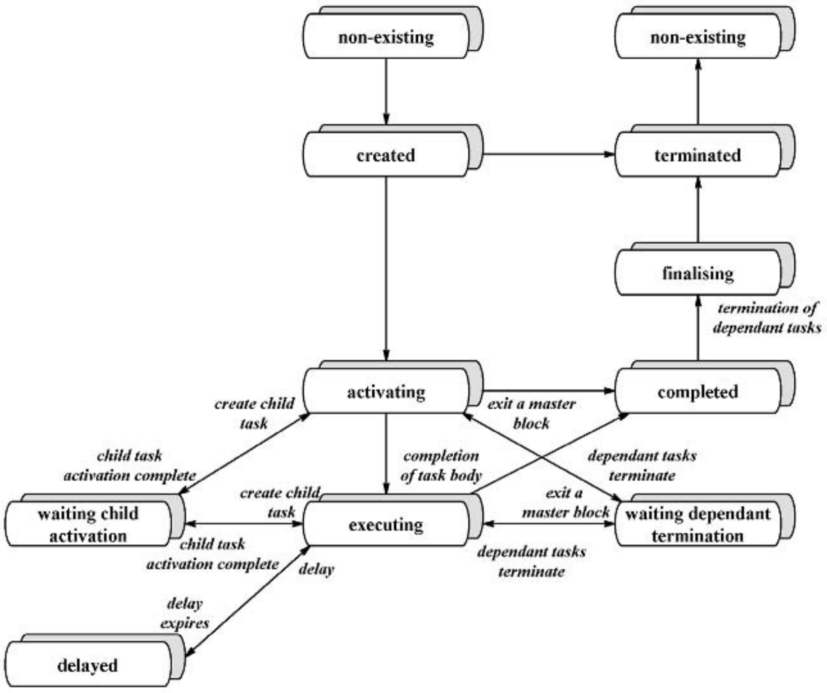


Fig. 4.3: Summary of task states and state transitions

When a task is created, it is possible to pass initialisation data to it via a discriminant. This allows tasks of the same type to be parameterised. Although any number of discriminants can be used, all must be of discrete or access type.

5
The rendezvous

A brief description of the rendezvous mechanism was given in Chapter 3. In this chapter, the standard rendezvous is explored in detail; more complex communication and synchronisation patterns are discussed in Chapter 6. The rendezvous is used for direct communication between tasks.

5.1 The basic model

The rendezvous model of Ada is based on a *client/server* model of interaction.

Definition: One task, the server, declares a set of services that it is prepared to offer to other tasks (the clients). It does this by declaring one or more public **entries** in its task specification. Each entry identifies the name of a service, the parameters that are required with any request and the results that will be returned.

For example, the following task models a telephone operator who provides a directory enquiry service:

```
task type Telephone_Operator is
  entry Directory_Enquiry(Person : in Name; Addr : in Address;
                          Num : out Number);
end Telephone_Operator;

An_Op : Telephone_Operator;
```

where `Name`, `Address` and `Number` are predeclared types in scope; and `Person` and `Addr` are passed to the operator (indicated by the keyword **in**) to identify the person whose telephone number is required, and `Num` is returned by the operator (indicated by the keyword **out**). Of course, the operator may also provide other services, such as logging faulty numbers.

Important note: A client task (also named the *calling task*) issues an ‘entry call’ on the server task (or *called task*) by identifying both the server and the required entry.

A call to the telephone operator is shown below:

```
-- client task
An_Op.Directory_Enquiry("STUART JONES",
                "10 MAIN STREET, YORK", Stuarts_Number);
```

Important note: A server indicates a willingness to provide the service at any particular time by executing an **accept** statement.

For example:

```
-- in the body of the server task
accept Directory_Enquiry(Person : in Name; Addr : in Address;
                         Num : out Number) do
    -- look up telephone number and assign the value to Num
end Directory_Enquiry;
```

For the communication to occur between the client and the server, both tasks must have issued their respective requests. When they have, the communication takes place; this is called a *rendezvous* because both tasks have to meet at the entry at the same time. When the rendezvous occurs, any **in** (and **in out**) parameters are passed to the server task from the client. The server task then executes the code inside the **accept** statement. When this statement finishes (by encountering its **end** statement, or via an explicit return), any **out** (and **in out**) parameters are passed back to the client and both tasks proceed independently and concurrently.

The following program fragment illustrates the client and server tasks:

```
task type Subscriber;

task type Telephone_Operator is
  entry Directory_Enquiry(Person : in Name; Addr : in Address;
                          Num : out Number);
end Telephone_Operator;

S1, S2, S3 : Subscriber; -- friends of Stuart Jones
An_Op : Telephone_Operator;

task body Subscriber is
  Stuarts_Number : Number;
begin
 ...
  An_Op.Directory_Enquiry("STUART JONES",
    "10 MAIN STREET, YORK", Stuarts_Number);
   -- phone Stuart
 ...
end Subscriber;
```

```
task body Telephone_Operator is
begin
  loop
    -- prepare to accept next call
    accept Directory_Enquiry(Person : in Name;
           Addr : in Address; Num : out Number) do
       -- look up telephone number and
       -- assign the value to Num
    end Directory_Enquiry;
    -- undertake housekeeping such as logging all calls
  end loop;
end Telephone_Operator;
```

It is quite possible that the client and server will not both be in a position to communicate at exactly the same time. For example, the operator may be willing to accept a service request but there may be no subscribers issuing an entry call. For the simple rendezvous case, the server is obliged to wait for a call; whilst it is waiting it frees up any processing resource (for example, the processor) it is using.

Definition: A task that is generally waiting for some event to occur is usually termed *suspended* or *blocked*.

Similarly, if a client issues a request and the server has not indicated that it is prepared to accept the request (because either it is already servicing another request or it is doing something else), then the client must wait.† Clients waiting for service at a particular entry are queued. The order of the queue depends on whether the Ada implementation supports the Real-Time Systems Annex. If it does not, then the queue is *first-in-first-out*; if it does, then other possibilities are allowed, such as priority queuing (see Section 13.4).

Figure 5.1 illustrates the synchronisation and communication that occur during the rendezvous.

5.2 The entry statement

A simplified formal description of the entry statement is given below:‡

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

† In Chapter 6, other mechanisms will be described which allow the server and the client to withdraw their offer of communication if they cannot enter into the rendezvous immediately or within a specified time period.
‡ In Ada 2005, explicit indication can also be given as to whether the entry is an overriding one or not – see Chapter 10.

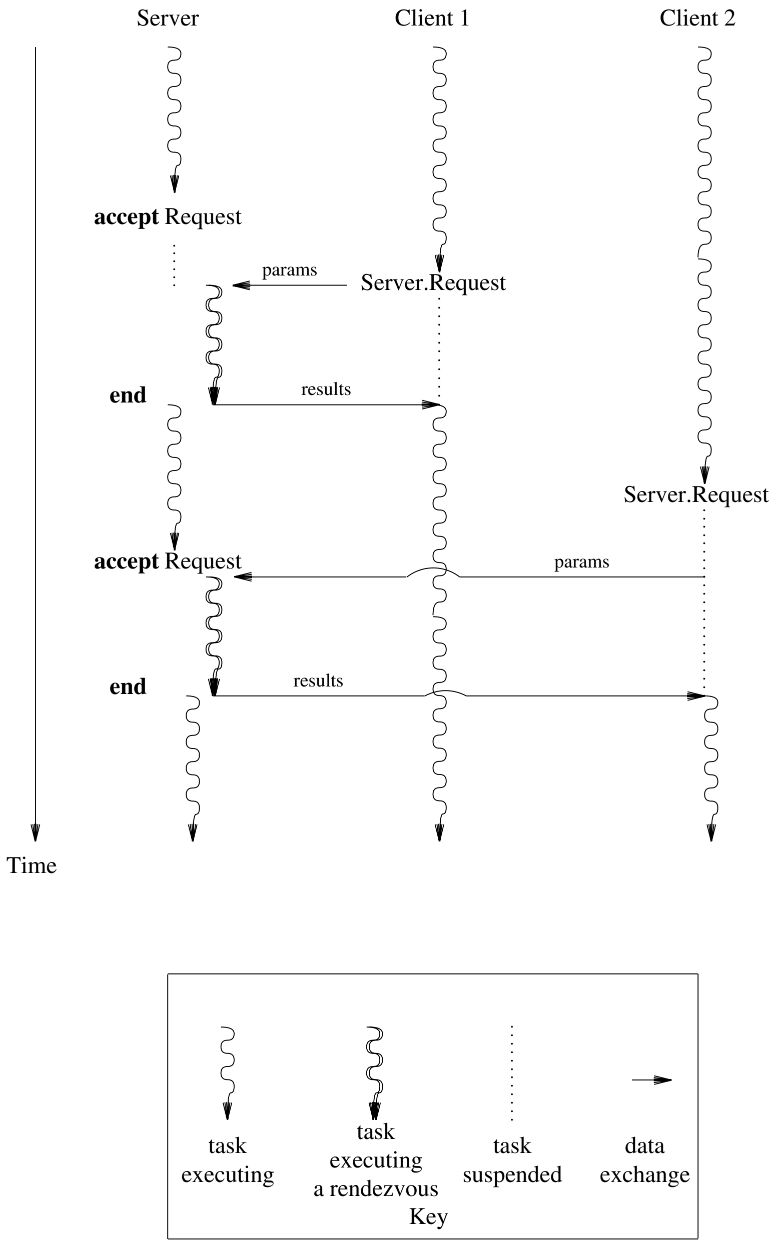


Fig. 5.1: Communication and synchronisation during a rendezvous

The parameter profile is the same as for Ada procedures (**in, out**, **in out** – with **in** being the default). Access parameters are not permitted, although parameters of any access type are, of course, allowed. Default parameters are also allowed.

The optional `discrete_subtype_definition` in the entry declaration is used to declare a family of distinct entries, all of which will have the same formal part (see Section 5.5).

A task can have several entries with the same name if the parameters to the entries are different (the entries are said to be *overloaded*). For example, the following task type overloads the `Directory_Enquiry` entry, allowing a subscriber to request a number giving a name and a postal or zip code instead of a name and an address:

```
task type Telephone_Operator is
  entry Directory_Enquiry(Person : in Name; Addr : in Address;
                          Num   : out Number);
  entry Directory_Enquiry(Person : in Name; Zip : in Postal_Code;
                          Num : out Number);
end Telephone_Operator;

An_Op : Telephone_Operator;
```

A single task entry (that is not a member of a family) will also overload a subprogram (because `Task_Name.Entry_Name` may be identical to `Package_Name.Subprogram_Name`) or an enumeration literal.

Unlike a package, a 'use' clause cannot be employed with a task to shorten the length of the calling identifiers; however, a procedure can rename an entry:

```
procedure Enquiry(Person : in Name; Addr : in Address;
          Num : out Number) renames An_Op.Directory_Enquiry;
```

A call to procedure `Enquiry` will result in an entry call to `An_Op.Directory_Enquiry`.

5.3 The accept statement

An accept statement specifies the actions to be performed when an entry is called.

Important note: For each and every entry defined in a task specification there must be at least one accept statement in the corresponding task body.

Interestingly, the ARM is not specific on this point and it is therefore possible that a compiler may not flag the lack of an appropriate accept statement as an error. Nevertheless, if there is no accept statement for some entry `E`, then a call to `E` will never be handled and the calling task may be blocked indefinitely. A task body, however, may contain more than one accept statement for the same entry. It should be noted that the accept statement must be placed directly in the task body; it cannot be placed in a procedure which is called by the task body – as such a procedure could be called from more than one task.

The formal part of the accept statement must conform exactly to the formal part of the corresponding entry:

```
accept_statement ::=
  accept entry_direct_name [(entry_index)] parameter_profile [do
    handled_sequence_of_statements
  end [entry_identifier]];
```

In general, therefore, the accept statement has the following form (although the family index and exception handling part are often absent):

```
accept Entry_Name(Family_Index)(P : Parameters) do
  -- sequence of statements
exception
  -- exception handling part
end Entry_Name;
```

The sequence of statements may contain subprogram calls, entry calls, protected object calls (see Chapter 7), accept statements (but not for the same entry) and inner blocks.

Important note: There are good reasons for making the accept statement as simple as possible. The code it contains should be only that which is necessary for the rendezvous. If it contains extra statements, then the calling task will be held up unnecessarily.

For example, if the telephone operator decided to perform its housekeeping operations inside the rendezvous, then the subscriber task would be further delayed before being allowed to continue. As a further example, consider a task that controls access to a single keyboard and screen. Let `Rec` be defined by

```
type Rec is
record
  I : Integer;
  F : Float;
  S : String (1..10);
end record;
```

and the task's structure by

```
task IO_Control is
  entry Get(R : out Rec);
  entry Put(R : Rec);
       ...
end IO_Control;
```

For `Put`, the accept statement might have the form

```
accept Put(R : Rec) do
  Put(R.I);  -- these procedures are constructed from Ada.Text_IO
  Put(R.F);
  Put(R.S);
end Put;
```

However, this ties up the rendezvous for the time it takes to output the three items; a better structure would be

```
accept Put(R : Rec) do
  Temp_Rec := R;
end Put;
Put(Temp_Rec.I); Put(Temp_Rec.F); Put(Temp_Rec.S);
```

With the `Get` entry, some user prompts are necessary; these must be undertaken during the rendezvous:

```
accept Get(R : out Rec) do
  Put("VALUE OF I?"); Get(R.I);
  Put("VALUE OF F?"); Get(R.F);
  Put("VALUE OF S?"); Get(R.S);
end Get;
```

User errors on I/O are unavoidable and exceptions are bound to occur from time to time. A reliable accept statement for `Get` would therefore be

```
accept Get(R : out Rec) do
  loop
    begin
      Put("VALUE OF I?"); Get(R.I);
      Put("VALUE OF F?"); Get(R.F);
      Put("VALUE OF S?"); Get(R.S);
      return;
    exception
      when others => Put("INVALID INPUT: START AGAIN");
    end;
  end loop;
end Get;
```

Note that this example makes use of a return statement to exit from the accept.

This formulation now represents a sizeable accept statement that will hold the rendezvous for a comparatively large amount of time. An alternative structure would be for the calling task to make a request (via another entry) for the record first, and later to make a separate entry call to `Get` the record. The calling task would therefore make two entry calls:

```
IO_Control.Request;
 ...
IO_Control.Get(Some_Rec);
```

The task `IO_Control` would then have the following code in its sequence of statements:

```
accept Request;
loop
  begin
    -- this is as before but with record variable Temp_Rec
  end;
end loop;
```

```
accept Get(R : out Rec) do
  R := Temp_Rec;
end Get;
```

This has simplified the accept statements but at the cost of a further rendezvous.

Primes by sieve example (revisited)

In the previous chapter an implementation of the 'primes by sieve' algorithm was given. In that program, the tasks communicated via buffer elements. Now that the rendezvous has been defined it is possible to give an alternative structure in which the tasks call each other directly; that is, each task rendezvouses with the next task in the pipeline. This, of course, saves on storage space for the buffer, but, perhaps, make the tasks more tightly synchronised than needed.

```
procedure Primes_By_Sieve is
  task type Sieve is
    entry Pass_On(Int : Integer);
  end Sieve;
  type Sieve_Ptr is access Sieve;

  task Odd;

  function Get_New_Sieve return Sieve_Ptr is
  begin
    return new Sieve;
  end Get_New_Sieve;

  task body Odd is
    Limit : constant Positive := ...;
    Num : Positive;
    S : Sieve_Ptr := new Sieve;
  begin
    Num := 3;
    while Num <= Limit loop
      S.Pass_On(Num);
      Num := Num + 2;
    end loop;
  end Odd;

  task body Sieve is
    New_Sieve : Sieve_Ptr;
    Prime, Num : Positive;
  begin
    accept Pass_On(Int : Integer) do
      Prime := Int;
    end Pass_On;
    -- Prime is a prime number, which could be output
    loop
      accept Pass_On(Int : Integer) do
        Num := Int;
      end Pass_On;
      exit when Num rem Prime /= 0;
    end loop;

    -- a number must be passed on and so a new Sieve task
    -- is created; note that it is necessary to use a
    -- function as a task type cannot be used directly
    -- within its own body
    New_Sieve := Get_New_Sieve;
    New_Sieve.Pass_On(Num);
    loop
      accept Pass_On(Int : Integer) do
        Num := Int;
      end Pass_On;
      if Num rem Prime /= 0 then
        New_Sieve.Pass_On(Num);
      end if;
    end loop;
  end Sieve;
begin -- procedure
  null;
end Primes_By_Sieve;
```

As with the example given in the previous chapter, this program still does not terminate correctly. The next chapter (see Section 6.6) will give the final correctly terminating algorithm.

Synchronisation without communication

Where the sequence of statements inside an accept statement is precisely null, then there is nothing to do within the rendezvous and the accept statement can be terminated by a semicolon following the formal part. This might be the case where only synchronisation is required. For example, consider the following simple telephone home security service, which, on receiving a specific entry call, will turn security lights on or off. Here, there is no need for data communication:

```
task type Security_Operator (House : House_Id) is
  entry Turn_Lights_On;
  entry Turn_Lights_Off;
end Security_Operator;

task body Security_Operator is
begin
  loop
    accept Turn_Lights_On;
    -- turn appropriate security lights on
    accept Turn_Lights_Off;
    -- turn appropriate security lights off
  end loop;
end Security_Operator;
```

In the above example, the discriminant informs the task which house to access.

5.4 The `Count` attribute

With each entry, a queue is defined for those tasks that are waiting to be accepted. Each of these queues has an associated attribute that allows the current length of the queue to be accessible to the owning task.

Important notes: `E'Count` returns a natural number representing the number of entry calls currently queued on the entry `E`, where `E` is either a single entry or a single entry of a family.
If `E` is an entry of task `T`, then `E'Count` can be used only within the body of `T` but not within a dependent program unit of that body

The attribute `Count` would appear to be a useful one in enabling a synchronisation task to exercise specific control over outstanding requests. Care must, however, be taken when using `Count` as its value not only will increase with the arrival of new entry calls but may decrease as a result of a timed entry call, an abort or an asynchronous transfer of control (see Sections 6.9, 9.2 and 9.3 respectively).

5.5 Entry families

Ada provides a facility that effectively allows the programmer to declare a one-dimensional array of entries.

Definition: Such an array is called a *family*; it is declared by specifying a discrete range in the entry declaration before the parameter specification.

For example, the following declares a multiplexer task with a family of three entries, each representing a channel on which data is to be received:

```
task Multiplexer is
  entry Channel(1..3)(X : in Data);
end Multiplexer;
```

Important note: In the body of a task declaring a family of entries, each member is treated as an individual entry and will, therefore, have its own accept statement.

For example, a possible body for the multiplexer task is shown below:

```
separate(Multiplex)
task body Multiplexer is
begin
  loop
    accept Channel(1)(X : in Data) do
      -- consume input data on channel 1
    end Channel;
    accept Channel(2)(X : in Data) do
      -- consume input data on channel 2
    end Channel;
    accept Channel(3)(X : in Data) do
      -- consume input data on channel 3
    end Channel;
  end loop;
end Multiplexer;
```

The task simply waits for communication on the first channel, then the second channel and then the third channel. Alternatively, the task could have been written as

```
task body Multiplexer is
begin
  loop
    for I in 1..3 loop
      accept Channel(I)(X : in Data) do
      -- consume input data on channel I
      end Channel;
    end loop;
  end loop;
end Multiplexer;
```

A client task of the multiplexer must now specify not only the task and its entry but also which particular member of the family it wishes to communicate with:

```
Multiplexer.Channel(2)(My_Data);
```

As well as providing a mechanism by which several identical entries can be conveniently defined, entry families give a way of processing entry calls in a user-defined order. For example, consider a server task that wishes to give priority to certain tasks. This could be achieved by the following:

```
task Server is
  entry Request_High_Priority(...);
  entry Request_Medium_Priority(...);
  entry Request_Low_Priority(...);
end Server;
```

Within the body of this task, calls to `Request_High_Priority` would be processed first, then calls to `Request_Medium_Priority` and then finally calls to `Request_Low_Priority`. The entries are distinct and therefore separate queues are supplied and supported for each. A more appropriate solution to this problem is possible, however, using a family of entries:

```
type Request_Priority is (High,Medium,Low);

task Server is
  entry Request(Request_Priority)(...);
end Server;
```

An entry call would then specify the priority of the request:

```
Server.Request(Medium)(...);
```

The structure of the task `Server` is such that only one call to `Request` should be processed at a time. If concurrent execution of `Request(High)`, `Requ-est(Medium)` and `Request(Low)` were allowed by the requirements of the system, then an array of `Server` tasks would be a more appropriate structure:

```
type Request_Priority is (High, Medium, Low);
task type Server is
  entry Request(...);
end Server;
Servers : array (Request_Priority) of Server;
```

A high priority task would then call

```
Servers(High).Request(...);
```

and so on.

5.6 Three-way synchronisation

Although a rendezvous can only take place between two tasks, nested accept statements can be used to tie together more than just two tasks. Consider a program in which a task (`Device`) simulates the actions of an external device, and where `Controller` is a task acting as a device driver and `User` is some task that wishes to use the (input) device:

```
procedure Three_Way is
  task User;
  task Device;
  task Controller is
    entry Doio (I : out Integer);
    entry Start;
    entry Completed (K : Integer);
  end Controller;

  task body User is ...;
    -- includes calls to Controller.Doio(...)

  task body Device is
    J : Integer;
  procedure Read (I : out Integer) is ...;
  begin
    loop
      Controller.Start;
      Read(J);
      Controller.Completed(J);
    end loop;
  end Device;
  task body Controller is
  begin
    loop
      accept Doio (I : out Integer) do
        accept Start;
        accept Completed (K : Integer) do
          I := K;
        end Completed;
      end Doio;
    end loop;
  end Controller;
begin
  null;
end Three_Way;
```

At the assignment of `K` to `I`, `User` is in rendezvous with `Controller`, and the rendezvous itself is in rendezvous with `Device`: three tasks are therefore synchronised.

A task can also issue an entry call to another task from within an accept statement. For example, consider the following tasks which model the customer interaction in a spare parts department for an automobile manufacturer. One task (`Customer_Service`) interfaces directly with the customer through an entry `Request_Part`. If this server task is unable to determine if the particular part is in stock, it communicates with a `Warehouse` task. If necessary, the requested part is placed on order:

```
task Warehouse is
  entry Enquiry(Item: Part_Number; In_Stock : out Boolean);
  ...
end Warehouse;

task body Warehouse is ...; -- details not shown

task Customer_Service is
  entry Request_Part(Order : Part_Number;
        Part : out Spare_Part; Order_Id : out Order_Number);
end Customer_Service;

task body Customer_Service is
  Found : Boolean;
  ...
begin
  loop
   ...
    accept Request_Part(Order : Part_Number;
           Part : out Spare_Part; Order_Id : out Order_Number) do
      ...
      if Found then
        Part := Found_Part;
        Order_Id := Null_Order;
      else
        Warehouse.Enquiry(Order, Found); -- entry call
        if Found then
          -- go and get part from warehouse
          Part := Found_Part;
          Order_Id := Null_Order;
        else
          -- make out order number
          Part := Null_Part;
          Order_Id := Next_Order_Number;
        end if;
      end if;
    end Request_Part;
  end loop;
end Customer_Service;
```

5.7 Private entries

So far in this chapter, all entries have been declared as public entries of the owning task. This makes them visible to all tasks for which the owning task's declaration is visible. In Ada, it is also possible to declare entries as private to the owning task. There are several reasons why the programmer might wish to do this:

(1) The task has several tasks declared internally; these internal tasks have access to the private entries.
(2) The entry is to be used internally by the task for requeuing purposes (see Chapter 8).

To illustrate the use of private entries, consider the controller program given earlier as an example of three-way communication. A possible way of rearranging this code is to have the device task contained within the controller task. This would be an appropriate decomposition for many systems, particularly when the controller has access to more than one device. With the new structure, only one of the controller's entries can be called from 'outside'. To enforce this restriction, private entries are used:

```
procedure Three_Way is
  task User;

  task Controller is
    entry Doio (I : out Integer);
  private
    entry Start;
    entry Completed (K : Integer);
  end Controller;

  task body User is ...;
    -- includes calls to Controller.Doio(...)
  task body Controller is
    task Device;
    task body Device is
      J : Integer;
      procedure Read (I : out Integer) is ...;
    begin
      loop
        Controller.Start;
        Read(J);
        Controller.Completed(J);
      end loop;
    end Device;
  begin
    loop
      accept Doio (I : out Integer) do
        accept Start;
        accept Completed (K : Integer) do
          I := K;
        end Completed;
      end Doio;
    end loop;
  end Controller;
begin
  null;
end Three_Way;
```

5.8 Exceptions and the rendezvous

In Section 5.3, it was indicated that exceptions may be raised during the rendezvous itself. In that discussion, all exceptions were trapped using a 'when others' exception handler.

Important note: It is possible for some exceptions to propagate outside the accept statement. If this happens, then the rendezvous is terminated and the exception is raised again in *both* the server (called) *and* the client (calling) task.

To illustrate the flow of exceptions between tasks engaged in a rendezvous, consider the following extension to the example given in Section 5.3.

```
accept Get(R : out Rec; Valid_Read : out Boolean) do
  loop
    begin
      Put("VALUE OF I?"); Get(R.I);
      Put("VALUE OF F?"); Get(R.F);
      Put("VALUE OF S?"); Get(R.S);
      Valid_Read := True;
      return;
    exception
```

```
      when Ada.Text_IO.Data_Error =>
        Put("INVALID INPUT: START AGAIN");
    end;
  end loop;
exception
  when Text_IO.Mode_Error => Valid_Read := False;
end Get;
```

If, during data input, the user types an inappropriate character, then Data_Error will be raised. As before, this is handled within the accept statement. Alternatively, if the file has been opened with the wrong mode, then the exception Mode_Error will be caught by the accept statement's handler and again it will not propagate.

The only other alternative is for an exception other than Data_Error or Mode_Error to be raised. For example, if the first call to Get causes Device_Error to be raised, then the exception will propagate out of the accept statement. Now the exception will be reraised in two tasks: the caller of this entry and the task owning the entry. If the caller happens not to have 'withed' Ada.Text_IO, then the exception will be anonymous in that task. In this situation, it can only be handled with a 'when others' handler.

The other interaction between exceptions and rendezvous occurs when a task attempts to call a task that has already terminated (completed or has become abnormal). In these situations, the caller task gets the exception Tasking_Error raised at the point of call. If the called task has already terminated, then the exception is raised immediately. It can also be the case that the called task becomes terminated while the caller is queued on an entry. Again, Tasking_Error is raised.

If a task calls an entry family giving an index outside the permitted range, Constraint_Error is raised at the point of call. The called task is unaffected.

5.9 Task states

In this chapter the basic rendezvous mechanism has been described. The synchronisation required to perform a rendezvous means that a task may be suspended. This introduces further states into the state transition diagram that was given in Section 4.6. Figure 5.2 summarises the states of an Ada task introduced so far in this book.

5.10 Summary

The rendezvous represents a high-level language abstraction for direct synchronous inter-task communication. Tasks declare entries that are called from other tasks.

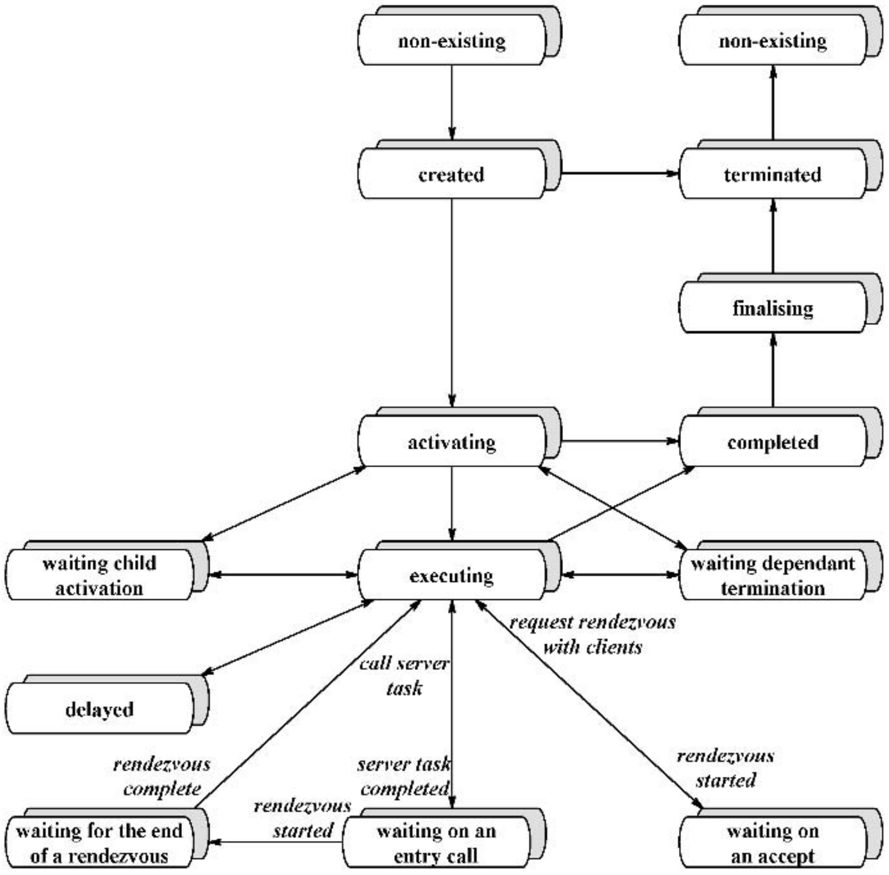


Fig. 5.2: Summary of task states and state transitions

Within the called task an accept statement defines the statements that are executed while the two tasks are linked in the rendezvous.

Ada's model for the rendezvous is a flexible one, allowing any number of parameters of any valid type to be passed. Moreover, it allows data to flow in the opposite direction to the rendezvous call.

Other issues covered in the chapter were entry families (these act like one-dimensional arrays of entries), private entries (which can help in visibility control) and exceptions. An exception raised in an accept statement either can be handled within the accept statement (in which case there are no further consequences), or can be allowed to propagate beyond the accept statement. In this latter case, the exception is raised in both the called and the calling task.

Although the basic model of the rendezvous is straightforward, some specific rules must be remembered when using this feature:

- Accept statements can only be placed in the body of a task.
- Nested accept statements for the same entry are not allowed.
- The `'Count` attribute can only be accessed from within the task that owns the entry.

6

The select statement and the rendezvous

The previous chapter presented the basic rendezvous model and detailed the language mechanisms by which two tasks can interact. In the basic model, a server task can only wait for a single rendezvous at any one time. Furthermore, once it has indicated that it wishes to communicate, it is committed to that communication and, if necessary, must wait indefinitely for the client to arrive at the rendezvous point. The same is true for the client task; in the basic model it can only issue a single entry call at any one time, and once issued the client is committed. This chapter shows how some of these restrictions can be removed by using the select statement.

The select statement has four forms:

```
select_statement ::= selective_accept | conditional_entry_call |
                     timed_entry_call | asynchronous_select
```

The first three of these forms are discussed in the context of the rendezvous mechanism. How the select statement can be applied to protected type entry calls and asynchronous transfer of control will be considered in Chapters 7 and 8 respectively.

6.1 Selective accept

Definition: The 'selective accept' form of the select statement allows a server task to

- wait for more than a single rendezvous at any one time;
- time out if no rendezvous is forthcoming within a specified period;
- withdraw its offer to communicate if no rendezvous is immediately available;
- terminate if no clients can possibly call its entries.

The syntax of the selective accept follows:

```
selective_accept ::=
  select
    [guard]
     selective_accept_alternative
{ or
    [guard]
    selective_accept_alternative }
[ else
    sequence_of_statements ]
  end select;

guard ::= when <condition> =>

selective_accept_alternative ::= accept_alternative
  | delay_alternative
  | terminate_alternative

accept_alternative ::=
   accept_statement [ sequence_of_statements ]

delay_alternative ::=
   delay_statement [ sequence_of_statements ]

terminate_alternative ::=
   terminate;
```

In the following sections, the various forms of the select statement will be explained and illustrated.

Waiting for more than a single rendezvous at any one time

A server task often wishes to provide more than a single service. Each service is represented by a separate entry declared in the task's specification. For example, if two services are offered, via entries `S1` and `S2`, then the following structure is often sufficient (that is, a loop containing a select statement that offers both services):

```
task Server is
  entry S1(...);
  entry S2(...);
end Server;

task body Server is
  ...
begin
  loop
    select
      accept S1(...) do
        -- code for this service
      end S1;
    or
      accept S2(...) do
        -- code for this service
      end S2;
    end select;
  end loop;
end Server;
```

On each execution of the loop, one of the accept statements will be executed.

To give a more illustrative example, consider the following `Telephone_Operator` task type. It provides three services: an enquiry entry requiring the name and address of a subscriber, an alternative enquiry entry requiring the name and postal (zip) code of a subscriber, and a fault-reporting service requiring the number of the faulty line. The task also has a private entry for use by its internal tasks:

```
task type Telephone_Operator is
  entry Directory_Enquiry(Person : in  Name; Addr : in Address;
                          Num : out Number);
  entry Directory_Enquiry(Person : in  Name;
        Zip : in Postal_Code; Num : out Number);
  entry Report_Fault(Num : Number);
private
  entry Allocate_Repair_Worker(Num : out Number);
end Telephone_Operator;
```

Without using the select statement, it is only possible to wait for one particular entry call at any one time. Clearly, this is not sensible; the operator may wish to wait for any one of its entries. The 'selective accept' form of the select statement allows it to do just this. Consider the following initial structure of the `Telephone_Operator` task's body:

```
task body Telephone_Operator is
  Workers : constant Integer := 10;
  Failed : Number;
begin
  loop
    -- prepare to accept next request
    select
      accept Directory_Enquiry(Person : in Name;
             Addr : in Address; Num : out Number) do
        -- look up telephone number and
        -- assign the value to Num
      end Directory_Enquiry;
    or
      accept Directory_Enquiry(Person : in  Name;
             Zip : in Postal_Code; Num : out Number) do
        -- look up telephone number and
        -- assign the value to Num
      end Directory_Enquiry;
    or
```

```
      accept Report_Fault(Num : Number) do
        Failed := Num;
      end Report_Fault;
      -- log faulty line and allocate repair worker
    end select;
    -- undertake housekeeping such as
    -- logging all calls
  end loop;
end Telephone_Operator;
```

In the above program fragment, the select statement allows the task to wait for a client task to call any one of its public entries. If none of the alternatives is immediately available (that is, there are no entry calls waiting on the accept alternatives), then the `Telephone_Operator` task must wait (it is suspended) until a call is made, at which time a rendezvous can take place.

Important note: It is possible that several clients may be waiting on one or more of the entries when the server task executes the select statement. In this case, the one chosen is implementation dependent. This means that the language itself does not define the order in which the requests are serviced. If the implementation is supporting the Real-Time Systems Annex, then certain orderings can be defined by the programmer (see Section 13.4). For general purpose concurrent programming, the programmer should assume that the order is arbitrary; that way the program cannot make any assumptions about the implementation of the language and thus it will be portable across different implementation approaches.

If tasks are queued on a single entry only, that entry is chosen and the rendezvous occurs with one of the client tasks; which client task is accepted again depends on whether the Real-Time Systems Annex is supported. By default, single queues are serviced on a first-come first-served basis.

It should be noted that statements can be placed after the accept statement in each arm of the select. For example, the comment 'log faulty line ...' in the above code fragment can be expanded into a sequence of statements. This sequence is executed after the rendezvous has occurred. It may, of course, include another rendezvous. For example,

```
task body Telephone_Operator is
  Workers : constant Integer := 10;
  Failed : Number;
  task type Repair_Worker;
  Work_Force : array (1 .. Workers) of Repair_Worker;
  task body Repair_Worker is ...;
begin
  loop
```

```
    -- prepare to accept next request
    select
      accept Directory_Enquiry(Person : in  Name;
             Addr : in Address; Num : out Number) do
         -- look up telephone number and
         -- assign the value to Num
      end Directory_Enquiry;
    or
      accept Directory_Enquiry(Person : in  Name;
             Zip : in Postal_Code; Num : out Number) do
         -- look up telephone number and
         -- assign the value to Num
      end Directory_Enquiry;
    or
      accept Report_Fault(Num : Number) do
        Failed := Num;
      end Report_Fault;
      -- log faulty line and allocate repair worker
      if New_Fault(Failed) then
          -- where New_Fault is a function in scope
        accept Allocate_Repair_Worker(Num : out Number) do
          Num := Failed;
        end Allocate_Repair_Worker;
      end if;
    end select;
    -- undertake housekeeping such as
    -- logging all calls
  end loop;
end Telephone_Operator;
```

Here, once a faulty number has been reported, the operator communicates the fault to a repair worker via a rendezvous on a private entry. The repair workers must indicate their availability by issuing an entry call on the operator.

Important note: The execution of the select statement involves the execution of one, and only one, accept alternative. This may then be followed by an arbitrary collection of statements. If none of the accept alternatives can be taken, the task executing the select is suspended.

6.2 Guarded alternatives

Each selective accept alternative can have a guard associated with it. This guard is a boolean expression which is evaluated when the select statement is executed. If the expression evaluates to true, the alternative is eligible for selection. If it is false, then the alternative is not eligible for selection during this execution of the select statement, even if clients are waiting on the associated entry. The general form for a guarded accept alternative is

```
select
  when Boolean_Expression =>
    accept S1(...) do
      -- code for service
    end S1;
    -- sequence of statements
or
  ...
end select;
```

Consider a `Telephone_Operator` task which will not accept a call to report a fault if all the repair workers have been allocated:

```
task body Telephone_Operator is
  ...
begin
  loop
    -- prepare to accept next request
    select
      accept Directory_Enquiry(Person : in Name;
             Addr : in Address; Num : out Number) do
         -- look up telephone number and
         -- assign the value to Num
      end Directory_Enquiry;
    or
      accept Directory_Enquiry(Person : in Name;
             Zip : in Postal_Code; Num : out Number) do
         -- look up telephone number and
         -- assign the value to Num
      end Directory_Enquiry;
    or
      when Workers_Available =>
        accept Report_Fault(Num : Number) do
          Failed := Num;
        end Report_Fault;
        ...
    end select;
    ...
  end loop;
end Telephone_Operator;
```

Warning: The following should be noted.

(1) The boolean expression is only evaluated once per execution of the select statement. If the expression accesses some shared data which might change whilst the select statement is waiting, then the change will go unnoticed until the select statement is re-executed. This includes use of the `'Count` attribute.

(2) It is considered to be an error (in the logic of the program) if a selective accept statement has a guard on each of its alternatives and all the guards evaluate to false. When this happens, the exception `Program_Error` is raised. Alternatives without guards are deemed to have 'true' guards.

Further discussion on guards is included in Section 6.5.

6.3 Delay alternative

Often it is the case that a server task cannot unreservedly commit itself to waiting for communication using one or more of its entries. The selective accept form of the select statement allows a server task to time-out if an entry call is not received within a certain period of time. The time-out is expressed using the delay statement and can therefore be a relative or an absolute delay. If the relative time expressed is zero or negative, or the absolute time has passed, then the delay alternative is equivalent to having an 'else part' (see next section). If the expression associated with the delay statement requires evaluation, then this is done at the same time as the guards are being analysed: that is, at the beginning of the execution of the select statement. More than one delay alternative is allowed, although, for any particular execution of the select statement, only the delay with the smallest time interval will act as the time-out.

Warning: Relative and absolute delay alternatives cannot be mixed (that is, a select statement can have one or more 'delay' alternatives, or one or more 'delay until' alternatives, but not both).

Generally, there are two reasons why a server task might wish to time-out waiting for an entry call; they are

(1) A task is required to execute periodically unless otherwise requested. For example, consider a periodic task that reads a sensor every ten seconds; however, it may be required to change its period during certain modes of the system's operation. This change should take effect immediately and the monitor should immediately read the sensor:

```
with Ada.Real_Time; use Ada.Real_Time;
  ...
task Sensor_Monitor is
  entry New_Period(Period : Time_Span);
end Sensor_Monitor;
```

```
task body Sensor_Monitor is
  Current_Period : Time_Span := To_Time_Span(10.0);
  Next_Cycle : Time := Clock +  Current_Period;
begin
  loop
    -- read sensor value and store in appropriate place
    select
      accept New_Period(Period : Time_Span) do
        Current_Period := Period;
      end New_Period;
      Next_Cycle := Clock + Current_Period;
    or
      delay until Next_Cycle;
      Next_Cycle := Next_Cycle + Current_Period;
    end select;
  end loop;
end Sensor_Monitor;
```

(2) The absence of the entry call within the specified time period indicates that an error condition has occurred (or some default action is required). For example, consider the following task that acts as a watchdog timer. The client task is meant to call the watchdog at least every ten seconds to indicate that all is well. If it fails to call in, the watchdog must raise an alarm:

```
task type Watchdog is
  entry All_Is_Well;
end Watchdog;

task body Watchdog is
begin
  loop
    select
      accept All_Is_Well;
    or
      delay 10.0;
      -- signal alarm, potentially the client has failed
      exit;
    end select;
  end loop;
  -- any further required action
end Watchdog;
```

Note that the delay time does not have to be a constant but could be a variable. Furthermore, the clock used is `Calendar.Clock`; there is no direct way to perform a relative delay using the `Real_Time.Clock`.

Guards and the delay alternative

In the examples shown so far, there has only been a single unguarded delay alternative. On occasions it may be appropriate to add a guard to a delay branch. For example, consider the case where each repair to a faulty telephone line is estimated to take no more than one hour:

```
task body Telephone_Operator is
   ...
begin
  loop
    -- prepare to accept next request
    select
      accept Directory_Enquiry(Person : in  Name;
             Addr : in Address; Num : out Number) do
         -- look up telephone number and assign the value to Num
      end Directory_Enquiry;
    or
      accept Directory_Enquiry(Person : in  Name;
             Zip : in Postal_Code; Num : out Number) do
         -- look up telephone number and assign the value to Num
      end Directory_Enquiry;
    or
      when Workers_Available =>
        accept Report_Fault(Num : Number) do
          Failed := Num;
        end Report_Fault;
        -- wait for a worker to call in
    or
      when not Workers_Available =>
        delay 3600.0;
        Workers_Available := True;
    end select;
    ...
  end loop;
end Telephone_Operator;
```

A guarded delay alternative whose guard evaluates to false on execution of the selection statement is not considered for selection. Hence, in the above example the time-out is not set if workers are available.

It may be appropriate to have more than one delay alternative in the same select statement. The same effect can be achieved by using only a single delay alternative, testing the boolean expressions just before the select statement and then setting an appropriate delay variable to the correct delay time; however, this may make the algorithm more difficult to understand.

In the general case, it is possible that two open delay alternatives have the same delay time specified. If this is the case and the delay time expires, then either alternative may be selected depending on the queuing policy in effect.

Unnecessary use of the delay alternative

On some occasions, the use of the delay alternative might seem appropriate but can be avoided by restructuring the select statement. Consider the telephone operator example again; the operator may not wish to spend too much time waiting for a repair worker to become free, because whilst it is waiting it cannot service any directory enquiry request. It may therefore decide to wait for only a certain period of time:

```
task body Telephone_Operator is
   ...
begin
  loop
    -- prepare to accept next request
    select
      accept Directory_Enquiry(...; Addr : in Address; ...) do
        ...
      end Directory_Enquiry;
    or
      accept Directory_Enquiry(...; Zip : in Postal_Code; ...) do
        ...
      end Directory_Enquiry;
    or
      accept Report_Fault( Num : Number ) do
      -- save details of fault
      end Report_Fault;
      -- log faulty line and allocate repair worker
      while Unallocated_Faults -- Unallocated_Faults in scope
      loop
        -- get next failed number
        select
          accept Allocate_Repair_Worker(Num : out Number) do
            Num := Failed;
          end Allocate_Repair_Worker;
          -- update record of failed unallocated numbers
        or
          delay 30.0;
          exit;
        end select;
      end loop;
    end select;
    -- undertake housekeeping such as
    -- logging all calls
  end loop;
end Telephone_Operator;
```

Although at first sight this seems an appropriate solution, it does cause some problems. In particular, if no workers are available, the telephone operator must keep track of the unallocated faults.

With the code shown above, the operator will only attempt to find a worker to repair a line when a new request has come in. A better solution would be

```
task body Telephone_Operator is
   ...
begin
  loop
    -- prepare to accept next request
    select
      accept Directory_Enquiry(...) do ... end Directory_Enquiry;
    or
      accept Directory_Enquiry(...) do ... end Directory_Enquiry;
    or
      accept Report_Fault( Num : Number ) do
      -- save details of fault
      end Report_Fault;
    or
      when Unallocated_Faults =>
        accept Allocate_Repair_Worker(Num : out Number) do
          -- get next failed number
          Num := Next_Fault;
        end Allocate_Repair_Worker;
        -- update record of failed unallocated numbers
    end select;
    ...
  end loop;
end Telephone_Operator;
```

This algorithm is now prepared to accept a call from a worker task when there are faults still to be allocated. It is simpler and avoids the use of the delay.

6.4 The else part

As well as allowing a server task to time-out in the absence of a call, the selective accept form of the select statement also allows the server to withdraw its offer to communicate if no call is immediately available.

Consider again the `Sensor_Monitor` task. Suppose that the requirement is that only when the task starts its next monitoring period does it check to see if its rate is to be increased or decreased:

```
task body Sensor_Monitor is
  Current_Period : Time_Span := To_Time_Span(10.0);
  Next_Cycle : Time := Clock + Current_Period;
begin
  loop
    select
      accept New_Period(Period : Time_Span) do
        Current_Period := Period;
      end accept;
    else
       null;
    end select;
    -- read sensor value and store in appropriate place
    Next_Cycle := Next_Cycle + Current_Period;
    delay until Next_Cycle;
  end loop;
end Sensor_Monitor;
```

In this case, as the sensor is to be read every time the task executes, the else part has a null sequence of statements. If the sensor were read only in the else part, the task would miss a reading on the changeover period.

Important note: An else part cannot be guarded and consequently only one else part may appear in a single select statement.

The delay alternative and the else part

If one compares the actions of the else clause and the delay alternative it is clear that to have both in the same select statement would be meaningless. The else clause defines an action to be taken, immediately, if no other alternative is executable. In contrast, the delay suspends the server task for some period of real-time. For these reasons, the language does not allow a select statement to contain a delay alternative and an else part. Interestingly, as type `Duration` has a range including 0.0, it is possible to delay for zero time. The following are therefore equivalent (for some sequence of statements `C`):

```
select                              select
  accept A;                           accept A;
or                                  or
  accept B;                           accept B;
else                                or
  C;                                  delay 0.0;
end select;                           C;
                                    end select;
```

It can therefore be argued that the else structure is redundant. However, its use does make the programmer's intention more explicit.

Finally in this section, two code fragments will be compared in order to illustrate the distinction between the delay alternative and the else part with a delay statement:

```
--S1                                -- S2
select                              select
  accept A;                           accept A;
or                                  else
  delay 10.0;                         delay 10.0;
end select;                         end select;
```

In the above, if there is an outstanding entry call on `A`, then these select statements will behave identically, that is, the call to `A` will be accepted. The distinction arises when there is no waiting call. In S1, the delay alternative will allow a call to A to be accepted if one arrives in the next ten seconds. S2 gives no such provision. If there is no outstanding call, the else part is executed; this happens to be an ordinary delay statement and so the task is delayed for ten seconds. If a call to A did arrive after, say, four seconds, S2 would not accept it (whereas S1 would). There is also a clear distinction between the behaviours of the following program fragments:

```
select                                select
  accept A;                             accept A;
or                                    or
  delay 10.0;                           delay 20.0;
  delay 10.0;
end select;                           end select;
```

The first example will accept a call to A only during the first ten seconds (it will then delay for a further ten seconds); the second example will accept a call during the entire twenty-second interval.

Warning: This apparent dual role for the delay keyword does cause confusion to programmers learning the language; it is unfortunate that another identifier such as 'time-out' was not used. However, this would increase the number of language keywords and hence complicate the language in other ways.

6.5 The correct use of guards

The boolean expression associated with a guard has no particular properties or restrictions.

Important note: It is strongly recommended that global variables (shared between tasks) should not be used in guards. To do so is likely to lead to unforeseen actions by the program; this is due to the fact that the guard is only evaluated once and is not 'retested' when an entry call is made or when the value of some component of the guard changes. The use of 'Count in a guard is also problematic if client tasks can be aborted (whilst on the entry queue) or if they issue timed entry calls.

Consider the following code, which controls calls to the entries Clock_In and Clock_Out. The specification of this task requires that Clock_In is not allowed before 08.30 hours (constant Start). Let Works_Clock be some function that provides the time in the correct form:

```
loop
  select
    when Works_Clock > Start =>
      accept Clock_In(N : Staff_Number) do
        ...
      end Clock_In;
  or
    accept Clock_Out(N : Staff_Number) do
        ...
    end Clock_Out;
  end select;
end loop;
```

This code, though superficially correct, is wrong. Consider the following:

(1) Call to `Clock_Out` at 8.20.
(2) Select statement is then immediately re-executed; the guard is `False` so `Clock_In` is deemed to be closed.
(3) Call to `Clock_In` at 8.45 – NOT ACCEPTED.

Indeed, no calls to `Clock_In` would be processed until a call to `Clock_Out` had terminated that particular execution of the select. A correct but inappropriate solution to this requirement would be to use an else clause:

```
loop
  select
    when Works_Clock > Start =>
      accept Clock_In(N : Staff_Number) do
      ...
      end Clock_In;
  or
    accept Clock_Out(N : Staff_Number) do
      ...
    end Clock_Out;
  else
    null;
  end select;
end loop;
```

Although this is now correct, in the sense that a call to `Clock_In` would be accepted at 8.45, it is very inefficient. This solution is, in effect, employing a busy-wait, which not only is wasteful of processor cycles but could lead to indefinite postponement of all other tasks on a single-processor system. The correct solution to this problem must employ a reference to `Works_Clock` inside the accept statement:

```
accept Clock_In(N : Staff_Number) do
  if Works_Clock < Start then
    Start_Time(N) := Start;
  else
    Start_Time(N) := Works_Clock;
  end if;
end Clock_In;
```

Important note: As the above discussion indicates, the else clause should only be used when absolutely necessary. Its existence in the language encourages the use of 'polling'. Polling is characterised by a task actively and repeatedly checking for the occurrence of some event. Unless a task can genuinely proceed with useful work in the situation where an entry call is not immediately outstanding, then the task should delay itself on a select statement without an else clause.

6.6 The terminate alternative

In general, server tasks only need to exist if there are clients that require their services. However, the very nature of the client/server model is that the server does not, in general, know the identities of its clients. Consequently, it is difficult for it to know when it can terminate. Conditions for termination can clearly be expressed as part of the program's logic, for example by using a special entry:

```
task General_Server is
  entry Service1(...);
  entry Service2(...);
  ...
  entry Terminate_Now(...);
end General_Server;

task body General_Server is
  ...
begin
  ...
  loop
    select
      accept Service1(...) do ... end;
    or
      accept Service2(...) do ... end;
    or
      ...
    or
      accept Terminate_Now;
      exit;
    end select;
  end loop;
end General_Server;
```

However, as this situation is common, a special select alternative is provided: the terminate alternative consists of only the single statement 'terminate' – which can be guarded. It is not possible to include a sequence of statements after the terminate alternative that the task can execute before terminating. This same effect can be achieved, however, by use of the finalisation facility (see Subsection 6.6.1

for a discussion on 'last wishes'). With the terminate alternative, the above example simply becomes

```
task General_Server is
  entry Service1(...);
  entry Service2(...);
  ...
end General_Server;

task body General_Server is
  ...
begin
  ...
  loop
    select
      accept Service1(...) do ... end;
    or
      accept Service2(...) do ... end;
    or
      ...
    or
      terminate;
    end select;
  end loop;
end General_Server;
```

Important notes: A server task which is suspended at a select statement with an open terminate alternative will become completed when the following conditions are satisfied:

- The task depends on a master whose execution is completed.
- Each task which depends on the master considered is either already terminated or similarly blocked at a select statement with an open terminate alternative.

When both the above conditions are satisfied, not only is the server task completed but so also are all the tasks that depend on the master being considered. Once these tasks are completed any associated finalisation code is executed.

For the server task to be completed, all remaining tasks that can call it must be suspended on appropriate select statements or have already completed. Therefore, there can be no outstanding entry calls (no task could have made such a call). Thus the terminate alternative cannot be selected if there is a queued entry call for any entry of the task. Consequently, it is not necessary, in normal circumstances, to guard the terminate alternative. The only real argument for using a guard is one of efficiency. On a multi-processor system, tests for termination can be expensive.

If the application knows that termination cannot be possible until at least some minimum condition holds, then there is a value in guarding the ‘terminate’.

Important note: As the inclusion of a terminate alternative indicates that on the execution of this select statement the server task may have no further work to do, it would be illogical to also include a delay alternative or an else clause in the same select statement. These possible combinations are therefore prohibited.

An example of the use of the terminate alternative comes from the primes by sieve program given in the previous chapter. The version given did not terminate when the total number of primes had been calculated. The `Odd` task terminated but the others in the pipeline remained suspended waiting for more integers to work upon. An appropriate modification allows termination to occur:

```
procedure Primes_By_Sieve is
  task type Sieve is
    entry Pass_On(Int : Integer);
  end Sieve;

  task Odd;

  type Sieve_Ptr is access Sieve;

  function Get_New_Sieve return Sieve_Ptr is
  begin
    return new Sieve;
  end Get_New_Sieve;

  task body Odd is
    Limit : constant Positive := ...;
    Num : Positive;
    S : Sieve_Ptr := new Sieve;
  begin
    Num := 3;
    while Num <= Limit loop
      S.Pass_On(Num);
      Num := Num + 2;
    end loop;
  end Odd;

  task body Sieve is
    New_Sieve : Sieve_Ptr;
    Prime, Num : Positive;
  begin
    accept Pass_On(Int : Integer) do
      Prime := Int;
    end Pass_On;
    -- Prime is a prime number, which could be output
    loop
      select
        accept Pass_On(Int : Integer) do
          Num := Int;
        end Pass_On;
      or
        terminate;
      end select;
      exit when Num rem Prime /= 0;
    end loop;
    -- a number must be passed on and
    -- so a new Sieve task is created
    New_Sieve := Get_New_Sieve;
    New_Sieve.Pass_On(Num);
    loop
      select
        accept Pass_On(Int : Integer) do
          Num := Int;
        end Pass_On;
      or
        terminate;
      end select;
      if Num rem Prime /= 0 then
        New_Sieve.Pass_On(Num);
      end if;
    end loop;
  end Sieve;

begin -- procedure
  null;
end Primes_By_Sieve;
```

6.6.1 Last wishes

There are many situations in which it is desirable to execute some finalisation code before a task terminates, either to close down some external resource (such as a file), or to 'clean up' some data structures that are shared with some later phase of execution of the program, or to produce a final report of some kind.

Important note: Ada's controlled types allow data objects to perform last wishes. A task cannot be derived from a controlled type, and therefore it is not possible to associate user-defined finalisation code with the task itself. Of course, any controlled objects defined in the task body will be finalised anyway; therefore finalisation of the task itself can be achieved by using a dummy controlled variable. It is a dummy in the sense that its only role is to invoke finalisation when the task is ready to terminate (and hence the variable is about to go out of scope).

To give a simple illustration of the use of last wishes, consider a server task that offers two services and wishes to print out, as it terminates, the total number of calls made upon each service. First, a new type is created from `Finalization.Limited_Controlled`:

```
with Ada.Finalization; use Ada;
package Counter is
  type Task_Last_Wishes is new Finalization.Limited_Controlled
    with record
      Count1, Count2 : Natural := 0;
    end record;
  overriding procedure Finalize(Tlw : in out Task_Last_Wishes);
end Counter;
```

The body of this simply outputs the required values.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
package body Counter is
  procedure Finalize(Tlw : in out Task_Last_Wishes) is
  begin
    Put("Calls on Service1:");
    Put(Tlw.Count1);
    Put(" Calls on Service2:");
    Put(Tlw.Count2);
    New_Line;
  end Finalize;
end Counter;
```

The specification of the server task is straightforward:

```
task Server is
 entry Service1(...);
 entry Service2(...);
end Server;

task body Server is separate;
```

The body of this task must have visibility of the `Counter` package; it is assumed that the task body is a separately compiled library unit (of procedure `Main`):

```
with Counter;
separate (Main)
task body Server is
  Last_Wishes :  Counter.Task_Last_Wishes;
begin
  -- initial housekeeping
  loop
    select
      accept Service1(...) do
        ...
      end Service1;
      Last_Wishes.Count1 :=  Last_Wishes.Count1 + 1;
    or
      accept Service2(...) do
        ...
      end Service2;
      Last_Wishes.Count2 :=  Last_Wishes.Count2 + 1;
    or
      terminate;
    end select;
    -- housekeeping
  end loop;
end Server;
```

When the conditions are right the server task will finalise the `Last_Wishes` variable, the two totals will be printed out and the task will then become terminated.

An alternative to using Ada's controlled types is to use the new Ada 2005 termination events. These are discussed in detail in Section 15.7.

6.7 The exception `Program_Error`

If all the accept alternatives have guards then it is possible that all the guards will be closed. Moreover, in this situation if the select statement does not contain an else clause, then it becomes impossible for the statement to be executed. This invidious position could be catered for in the following two ways:

(1) The situation is deemed to be an error.
(2) The select statement becomes equivalent to 'null' and the task continues execution.

As indicated earlier, the designers of Ada decided upon the first solution and the exception `Program_Error` is raised at the point of the select statement if no alternatives are open. In general, there will normally exist a relationship between the guards that will allow an analysis of the code to prove that the exception cannot be raised. There will, however, be situations where this is not the case. Consider the following examples:

```
1.    if A then                      2.    select
        accept B;                              when A =>
      end if;                                    accept B;
                                             end select;

3.    select
        when A =>
          accept B;
      else
        null;
      end select;
```

These three examples involve a single accept statement `B` and the condition `A`. If `A` is `False`, `B` should not be accepted, but if `A` is `True` the options are either

(a) accept an entry call before proceeding; or

(b) accept an entry call if and only if one is currently outstanding.

Example 1 caters for (a); example 3 caters for (b); example 2 is unnecessary and would cause an exception to be raised if `A` were `False`.

With two accept statements and two conditions the situation is more complex:

```
4.    if A then
        accept B;
      elsif C then
        accept D;
      end if;
```

```
5.    select
        when A =>
          accept B;
      or
        when C =>
          accept D;
      end select;
```

```
6.    select
        when A =>
          accept B;
      or
        when C =>
          accept D;
      else
        null;
      end select;
```

The equivalent of case (b) is again easily catered for (example 6) in that an open accept alternative will be taken if a call is outstanding. Example 4 does not, however, deal appropriately with the first case, for if `A` and `C` are both `True` then `B` will always be chosen in preference to `D` even when there is an outstanding call to `D` but not `B`. True indeterminacy can only be provided by the code in example 5; but this code will fail when `A` and `C` are both `False`. If `A` and `C` are not related, then it is clearly not possible to prove that `A` = `C` = `False` cannot occur. In these circumstances the select statement itself must be 'guarded'; the correct solution to the above problem is therefore

```
7.    if A or C then
        select
          when A =>
            accept B;
        or
          when C =>
            accept D;
        end select;
      end if;
```

6.8 Summary of the selective accept statement

The selective accept form of the select statement can be summarised as follows:

- A selective accept must contain at least one accept alternative (each alternative can be guarded).
- A selective accept may contain one, and only one, of the following:
  - a terminate alternative (possibly guarded); or
  - one or more absolute delay alternatives (each possibly guarded); or
  - one or more relative delay alternatives (each possibly guarded); or
  - an else part.

A select alternative is said to be 'open' if it does not contain a guard or if the boolean expression associated with the guard evaluates to `true`. Otherwise the alternative is 'closed'. On execution of the select statement, the first action to be taken is for all the guards to be evaluated and for all open delay expressions and entry family expressions to be evaluated; all open alternatives are thus determined. For this execution of the select statement, closed alternatives are no longer considered. If one or more open accept alternatives have an outstanding entry call, then one of these accept alternatives is chosen (the choice depends on the implementation and whether the Real-Time Systems Annex is being supported). The accept statement is executed, the rendezvous takes place, any statements following the accept (within that branch of the select statement) are executed and the execution of the select statement is then complete.

If, however, no open accept alternative has an outstanding entry call, then if there is an else part this will be executed. Otherwise, the select statement will suspend, waiting until:

- the arrival of an entry call associated with one of the open accept alternatives;
- the expiry of any open delay alternative; or
- the task becomes completed as a result of an open terminate option.

6.9 Conditional and timed entry calls

In the previous sections, it was shown how a server task could avoid unreservedly committing itself to accepting a single entry call by using a select statement. These facilities are not totally available to the client task, which can only issue a single entry call at any one time. However, the language does allow the client to avoid committing itself to the rendezvous by providing conditional and timed entry call facilities as part of the select statement.

6.9.1 Timed entry calls

A timed entry call issues an entry call which is cancelled if the call is not accepted within the specified period (relative or absolute). A simplified syntax description is given below.

```
timed_entry_call ::=
  select
    entry_call_alternative
  or
    delay_alternative
  end select;

entry_call_alternative ::= procedure_call_statement |
  entry_call_statement [sequence_of_statements]
```

Ada 2005 change: Ada 2005 now allows some types of procedure calls to appear in a timed entry call – see Section 10.2.

Note that only one delay alternative and one entry call can be specified.

Consider, for example, the following client of the telephone operator task:

```
task type Subscriber;

task body Subscriber is
  Stuarts_Number : Number;
begin
  loop
    ...
    select
      An_Op.Directory_Enquiry("STUART JONES",
            "10 MAIN STREET, YORK", Stuarts_Number);
      -- log the cost of a directory enquiry call
    or
      delay 10.0;
      -- phone up his parents and ask them,
      -- log the cost of a long distance call
    end select;
    ...
  end loop;
end Subscriber;
```

Here the task waits for ten seconds for the operator to accept the call. If the call is not accepted before the ten seconds have expired, the subscriber's call is cancelled and (in this case) the task attempts to obtain the required number via an alternative source.

Important note: The main point to note about the timed entry call is that it provides a facility whereby the client can cancel the call if it is not accepted within the specified period. *It makes no guarantee that the results from the call will be returned in that period.*

For example, consider the following telephone operator task:

```
task body Telephone_Operator is
  ...
begin
  loop
    -- prepare to accept next request
    select
      accept Directory_Enquiry(Person : in Name;
             Addr : in Address; Num : out Number) do
         delay 3600.0; -- take a lunch break
      end Directory_Enquiry;
    or
      ...
    end select;
    ...
  end loop;
end Telephone_Operator;
```

Here the operator, once it has accepted an enquiry request, delays for one hour before servicing the request. The client task is now forced to wait because the call has been accepted.

6.9.2 Conditional entry call

The conditional entry call allows the client to withdraw the offer to communicate if the server task is not prepared to accept the call immediately. It has the same meaning as a timed entry call, where the expiry time is immediate. The syntax is

```
conditional_entry_call ::=
  select
    entry_call_alternative
  else
    sequence_of_statements
  end select;
```

For example, a very impatient client of the telephone operator task may not be prepared to wait for a connection:

```
task type Subscriber;

task body Subscriber is
  Stuarts_Number : Number;
begin
  loop
    ...
    select
      An_Op.Directory_Enquiry("STUART JONES",
                              "10 MAIN STREET, YORK",
                               Stuarts_Number);
      -- log the cost of a directory enquiry call
    else
      -- phone up his parents and ask them,
      -- log the cost of a long distance call
    end select;
    ...
  end loop;
end Subscriber;
```

Clearly, it is possible for two clients to issue simultaneous conditional entry calls, or timed entry calls with immediate expiry times. There is therefore an obligation on the Ada run-time support system to make the commit operation an indivisible action so that only one client task can see the state of the server at any one time.

A conditional entry call should only be used when the task can genuinely do other productive work if the call is not accepted. Care should be taken not to use polling, or busy-wait solutions, unless they are explicitly required.

Important note: Note that the conditional entry call uses an ‘else’, the timed entry call an ‘or’. Moreover, they cannot be mixed, nor can two entry call statements be included. A client task cannot therefore wait for more than one entry call to be serviced (without using an asynchronous select statement – see Subsection 9.4.3).

6.10 Mutual exclusion and deadlocks

A number of the inherent difficulties associated with concurrent programming were considered in Chapter 3. Of paramount importance is ensuring the integrity of resources that should not be accessed by more than one task at a time (non-concurrent resources). This integrity is usually assured by defining a critical section of code that must be protected by mutual exclusion. With a server task, mutual exclusion can easily be constructed:

```
task type Server is
  entry A(...);
  entry B(...);
  ...
end T;

task body Server is
  -- The resource is represented by the definition
  -- of some appropriate data structure.
```

```
begin
  loop
    select
      accept A(...) do
        ...
      end A;
      -- housekeeping
    or
      accept B(...) do
        ...
      end B;
      -- housekeeping
    or
      ...
    end select;
    -- housekeeping
  end loop;
end Server;
```

Each task of type `Server` will define a new resource. For each of these resources, although entries `A`, `B` etc. give access, the semantics of the task body (with or without a select statement) ensure that only one accept statement at a time can be executing. The accept statement is itself the critical section. As long as the resource is defined within the task body, and is not accessible to ‘subtasks’ defined within the same body, then mutual exclusion is ensured. Mutual exclusion does not, however, extend to including the evaluations of the actual parameters to an entry call. The evaluations of such parameters in two distinct entry calls may therefore interfere with each other if shared variables are used directly or are influenced by side effects.

Deadlocks are another matter. Consider the following code:

```
task T1 is                              task T2 is
  entry A;                                entry B;
end T1;                                 end T2;

task body T1 is                         task body T2 is
begin                                   begin
  T2.B;                                   T1.A;
  accept A;                               accept B;
end T1;                                 end T2;
```

Clearly, each task will be placed on an entry queue for the other task. They will therefore not be in a position to accept the outstanding call. A task can even call its own entries and deadlock! Guidelines can reduce the possibility of deadlocks but inherent deadlocks should (must) be recognised early in the design of systems and evasive action taken.

Important note: The following points are pertinent:

- Tasks should be constructed to be either active entities or servers.
- Active tasks make entry calls but do not have entries.
- Server tasks accept entry calls but make no calls themselves.
- Entry calls from within a rendezvous should only be used when absolutely necessary.

Where a collection of servers is accessed by a number of tasks, then

(1) Tasks should use servers one at a time if possible.
(2) If (1) is not possible, then all tasks should access the servers in the same predefined order.
(3) If (1) and (2) are not possible, then the server tasks must be designed to take appropriate remedial action. For example, resources could be preemptively removed from active tasks using a time-out structure.

With deadlocks, there is no substitute for proving that they cannot occur in the first place.

To ensure liveness, select statements can be constructed to force the system to examine each queue in turn:

```
loop
  select
    accept A;
  else
    null;
  end select;

  select
    accept B;
  else
    null;
  end select;

  select
    accept A;
  or
    accept B;
  end select;
end loop;
```

This structure would, of course, become very tedious if a large number of entries were involved. The final select statement is needed to delay the server task if no outstanding entry calls exist (otherwise the task would loop around and poll the entry queues).

Important note: If mutual exclusion is provided by the use of protected objects and the priority ceiling protocol is in operation (see Section 13.3), then deadlock-free behaviour is guaranteed.

6.11 The dining philosophers

This section gives a solution to the dining philosophers program that was outlined in Section 3.5. It is assumed that the chopsticks are under the control of server tasks, as is the deadlock prevention rule that only allows $N - 1$ philosophers to consider eating at the same time. To obtain an array of static philosopher tasks (each with a unique discriminant) would require the side-effect initialisation function described in Section 4.1. In the following code, an alternative structure is used: a static array of a task access type. Each task is then created with the correct discriminant:

```
procedure Dining_Philosophers is
  package Activities is
    procedure Think;
    procedure Eat;
  end Activities;

  N : constant := 5;  -- number of philosophers.
  type Philosophers_Range is range 0 .. N-1;

  task type Phil(P : Philosophers_Range);
  type Philosopher is access Phil;

  task type Chopstick_Control is
    entry Pick_Up;
    entry Put_Down;
  end Chopstick_Control;

  task Deadlock_Prevention is
    entry Enters;
    entry Leaves;
  end Deadlock_Prevention;

  Chopsticks : array(Philosophers_Range) of Chopstick_Control;
  Philosophers : array(Philosophers_Range) of Philosopher;

  package body Activities is separate;
  task body Phil is separate;
  task body Chopstick_Control is separate;
  task body Deadlock_Prevention is separate;

begin
 for P in Philosophers_Range loop
    Philosophers(P) := new Phil(P);
 end loop;
end Dining_Philosophers;
```

The procedures `Think` and `Eat` in the package `Activities` will consist of some appropriate delay statement and will be called concurrently by the philosophers, who are structured as active tasks. Each chopstick, being a shared resource, is represented by a server task:

```
separate (Dining_Philosophers)
task body Chopstick_Control is
begin
  loop
    accept Pick_Up;
    accept Put_Down;
  end loop;
end Chopstick_Control;
```

Deadlocks are prevented by noting that they can only occur if all the philosophers wish to eat at the same time. By stopping just one philosopher from entering the eating stage, deadlocks are prevented; moreover, as only one philosopher task is delayed (and therefore freed when a single philosopher stops eating), liveness is preserved:

```
separate (Dining_Philosophers)
task body Deadlock_Prevention is
  Max : constant Integer := N - 1;
  People_Eating : Integer range 0..Max := 0;
begin
  loop
    select
      when People_Eating < Max =>
        accept Enters;
        People_Eating := People_Eating + 1;
    or
      accept Leaves;
      People_Eating := People_Eating - 1;
    end select;
  end loop;
end Deadlock_Prevention;
```

The philosophers themselves are necessarily programmed as tasks and have a simple life-cycle. In order to know which chopsticks to pick up, each philosopher must know its own identity. This is achieved by passing a unique array index during task creation. The philosopher tasks therefore make calls upon the server agents and are not themselves called:

```
separate (Dining_Philosophers)
task body Phil is
  Chop_Stick1, Chop_Stick2 : Philosophers_Range;
  use Activities;
```

```
begin
  Chop_Stick1 := P;
  Chop_Stick2 := (Chop_Stick1 + 1) mod N;
  loop
    Think;
    Deadlock_Prevention.Enters;
    Chopsticks(Chop_Stick1).Pick_Up;
    Chopsticks(Chop_Stick2).Pick_Up;
    Eat;
    Chopsticks(Chop_Stick1).Put_Down;
    Chopsticks(Chop_Stick2).Put_Down;
    Deadlock_Prevention.Leaves;
  end loop;
end Phil;
```

The structure used above ensures an indefinite execution of the program. If a limited execution is desired, then the `Philosopher` would need to exit from its life-cycle after a number of iterations or after some duration of time. The synchronisation tasks would all have 'or terminate' alternatives on their select statements.

The reliability of this solution is typical of many concurrent Ada programs. Firstly, it can be seen that the failure of any of the server tasks would be disastrous for the program as a whole. A `Philosopher` could, however, terminate without affecting the system unless he or she happened to have control of a chopstick at that time. Resources can, in general, be programmed to put themselves back in the resource pool in this eventuality. Assume that there is an upper limit of `Interval` on the time it takes for a philosopher to eat, such that if a chopstick is not returned within that period the philosopher can be assumed to have died (of over-eating!). The body for the task `Chopstick_Control` would then take the following form (including a termination alternative):

```
task body Chopstick_Control is
begin
  loop
    select
      accept Pick_Up;
    or
      terminate;
    end select;
    select
      accept Put_Down;
    or
      delay Interval;
      -- As the philosopher has not called Put_Down he or
      -- she is assumed to be 'dead', the chopsticks can
      -- therefore be rescued.
    end select;
  end loop;
end Chopstick_Control;
```

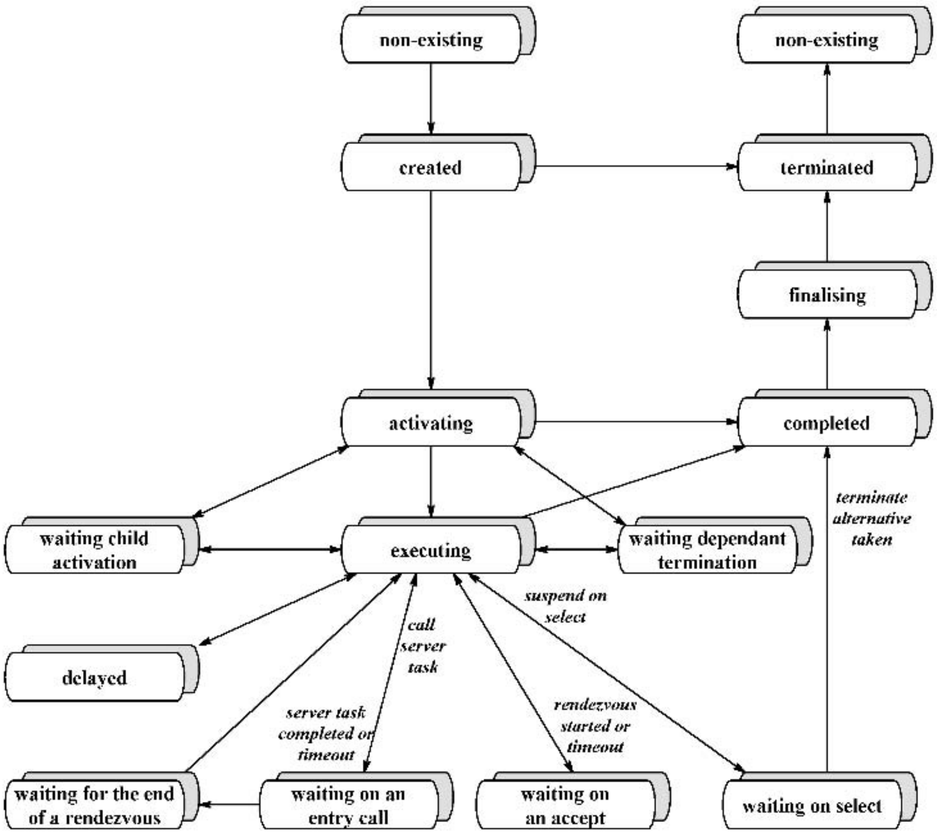


Fig. 6.1: Summary of task states and state transitions

6.12 Task states

In this chapter the basic rendezvous mechanism has been extended. The select statement has introduced further states into the state transition diagram. Figure 6.1 summarises the states of an Ada task introduced so far in this book.

6.13 Summary

In order to increase the expressive power of rendezvous-based communications, Ada provides the select statement. This allows a task to choose between alternative actions. The fundamental use of the select statement is to allow a server task to choose to respond to one of any number of possible entry calls. The server task does not need to give a fixed ordering, it can choose any of the accept statements that have outstanding entry calls.

There are a number of different variations of the select statement. Not only can a server task wait for more than one call, it can choose to limit its wait to a period of real-time; indeed it can choose not to wait at all if there are no outstanding calls.

The select statement is also used to support a form of termination that is very useful in concurrent programs with hierarchical task structures. A task will terminate if it is suspended on a select statement, with a terminate alternative, if all tasks that could call it either are already terminated or are similarly suspended on this type of select statement. If a task wishes to perform some last rites before terminating, then it can use a controlled variable or the new Ada 2005 termination event feature.

In addition to selecting between different incoming calls, the select statement can also allow an external outgoing call to be made conditionally (that is, if the call cannot be accepted immediately, it is cancelled) or be 'offered' for only a limited period of time. In the latter case if the call is not accepted within the defined time interval it is cancelled.

Although flexible, the select statement does not allow a task to select between more than one outgoing entry call (without resorting to the asynchronous variety), or to choose between an accept statement and an entry call.

7
Protected objects and data-oriented communication

The problem of sharing resources between processes was briefly discussed in Chapter 3. Two requirements were identified as being essential: mutual exclusion and condition synchronisation. This chapter discusses various ways in which these requirements can be met in Ada without having to encapsulate the resource in a server task and without having to use the rendezvous. Ada gives direct support to protected data by the *protected object* feature, the discussion of which is the main focus of this chapter. However, the language does also support the notions of atomic and volatile data, which are covered in Section 7.13.

7.1 Protected objects

A protected object in Ada encapsulates data items and allows access to them only via protected subprograms or protected entries. The language guarantees that these subprograms and entries will be executed in a manner that ensures that the data is updated under mutual exclusion. Consequently, they are rather like monitors found in previous concurrent programming languages (as described in Chapter 3).

A protected unit may be declared as a type or as a single instance; it has a specification and a body. The specification of a protected unit has the following syntax (as with a task, there are visible and private parts):

```
protected_type_declaration::=
  protected type defining_identifier
      [known_discriminant_part] is
      [new interface_list with] protected_definition;

single_protected_declaration ::=
      protected defining_identifier
      is protected_definition;
```

```
protected_definition ::=
    protected_operation_declaration
[  private
    { protected_element_declaration } ]
end [protected_identifier]

protected_operation_declaration ::=
   subprogram_declaration |  entry_declaration |
   aspect_clause
protected_element_declaration ::=
    protected_operation_declaration |
    component_declaration
```

Thus a protected type has a specification that can contain functions, procedures and entries. It may also contain aspect clauses which can be used to map protected procedures to interrupts (see Section 12.1). Similar to tasks and records, the discriminant can only be of a discrete or access type.

Ada 2005 change: As with task specifications, protected types in Ada 2005 can now support Java-like interfaces. See Chapter 10 for a full discussion.

The body, which may be compiled separately from the specification, is declared using the following syntax:

```
protected_body ::=
  protected body defining_identifier is
    { protected_operation_item }
  end [protected_identifier];

protected_operation_identifier ::=
    subprogram_declaration | subprogram_body | entry_body

entry_body ::=
  entry defining_identifier entry_body_formal_part
              entry_barrier is
    declarative_part
  begin
    handled_sequence_of_statements
  end [entry_identifier];

entry_body_formal_part ::=
   [(entry_index_specification)] parameter_profile

entry_barrier ::= when condition

entry_index_specification ::=
   for defining_identifier in discrete_subtype_definition
```

A protected type is a ‘limited type’, and therefore there are no predefined assignment or comparison operators (the same is true for task types).

The following sections define the full semantics of protected types and objects, and give illustrative examples of their use.

7.2 Mutual exclusion

The following declaration illustrates how protected types can be used to provide simple mutual exclusion:

```
protected type Shared_Integer(Initial_Value : Integer) is
  function Read return Integer;
  procedure Write(New_Value : Integer);
  procedure Increment(By : Integer);
private
  The_Data : Integer := Initial_Value;
end Shared_Integer;

My_Data : Shared_Integer(42);
```

The above protected type encapsulates a shared integer. The object declaration, `My_Data`, declares an instance of the protected type and passes the initial value for the encapsulated data. The encapsulated data can now only be accessed by the three subprograms: `Read`, `Write` and `Increment`.

Definition: A protected procedure provides mutually exclusive read/write access to the data encapsulated.

In this case, concurrent calls to the procedure `Write` or `Increment` will be executed in mutual exclusion; that is, only one can be executing at any one time. If, for example, a task attempts to call `Write` whilst another task is already executing the procedure (or already executing a call to `Read` or `Increment`), it is unable to enter the `Write` procedure until the other task has exited.

Definition: Protected functions provide concurrent read-only access to the encapsulated data.

In the above example, this means that many calls to `Read` can be executed simultaneously. However, calls to a protected function are still executed mutually exclusively with calls to a protected procedure. A `Read` call cannot be executed if there is a currently executing procedure call; a procedure call cannot be executed if there are one or more concurrently executing function calls.

The core language does not define the order in which tasks waiting to execute protected functions and protected procedures are executed. If, however, the Real-Time Systems Annex is being supported, certain assumptions can be made about the order of execution (see Section 13.4).

The body of Shared_Integer is simply

```
protected body Shared_Integer is
  function Read return Integer is
  begin
    return The_Data;
  end Read;

  procedure Write(New_Value : Integer) is
  begin
    The_Data := New_Value;
  end Write;

  procedure Increment(By : Integer) is
  begin
    The_Data := The_Data + By;
  end Increment;
end Shared_Integer;
```

Without the abstraction of a protected object this shared integer would have had to be implemented as a server task with the read and write operations being coded as entries:†

```
task body Shared_Integer is
  The_Data : Integer := Initial_Value;
begin
  loop
    select
      accept Read(New_Value : out Integer) do
        New_Value := The_Data;
      end Read;
    or
      accept Write(New_Value : Integer) do
        The_Data := New_Value;
      end Write;
    or
      accept Increment(By : Integer) do
        The_Data := The_Data + By;
      end Increment;
    or
      terminate;
    end select;
  end loop;
end Shared_Integer;
```

Whereas the task has an explicit select statement and loop construct, the protected object has an entirely implicit structure. It does not even need to concern itself with termination. A protected object is passive, and will thus cease to exist when its scope disappears.

† Note that concurrent calls to Read are now not supported.

Protected objects and tasks are, however, alike in that they are both limited private data types. Single protected objects can also be declared (cf. anonymous task types):

```
protected My_Data is
  function Read return Integer;
  procedure Write (New_Value : Integer);
private
  The_Data : Integer := Initial_Value;
end My_Data;
```

Note that `Initial_Value` is now a variable in scope rather than a discriminant.

7.3 Condition synchronisation

Condition synchronisation using protected types is provided by the protected entry facility.

Definition: A protected entry is similar to a protected procedure in that it is guaranteed to execute in mutual exclusion and has read/write access to the encapsulated data. However, a protected entry is guarded by a boolean expression (called a barrier) inside the body of the protected object; if this barrier evaluates to false when the entry call is made, the calling task is suspended until the barrier evaluates to true and no other tasks are currently active inside the protected object.

Consider a bounded buffer shared between several tasks. The data must be placed in, and retrieved from, the buffer under mutual exclusion. Furthermore, condition synchronisation is required because a calling task attempting to place data into the buffer, when the buffer is full, must be suspended until there is space in the buffer; also a retrieving task must be suspended when the buffer is empty. The specification of the buffer is

```
type Data_Item is ...
Buffer_Size : constant Integer := 10;
type Index is mod Buffer_Size;
subtype Count is Natural range 0 .. Buffer_Size;
type Buffer is array (Index) of Data_Item;

protected type Bounded_Buffer is
  entry Get(Item : out Data_Item);
  entry Put(Item : in Data_Item);
private
  First : Index := Index'First;
  Last : Index := Index'Last;
  Number_In_Buffer : Count := 0;
  Buf : Buffer;
end Bounded_Buffer;
```

```
My_Buffer : Bounded_Buffer;
```

Two entries have been declared; these represent the public interface of the buffer. The data items declared in the private part are those items which must be accessed under mutual exclusion. In this case, the buffer is an array and is accessed via two indices; there is also a count indicating the number of items in the buffer.

The body of this protected type is given below:

```
protected body Bounded_Buffer is
  entry Get(Item : out Data_Item)
      when Number_In_Buffer /= 0 is
  begin
    Item := Buf(First);
    First := First + 1;
    Number_In_Buffer := Number_In_Buffer - 1;
  end Get;

  entry Put(Item : in Data_Item)
      when Number_In_Buffer /= Buffer_Size is
  begin
    Last := Last + 1;
    Buf(Last) := Item;
    Number_In_Buffer := Number_In_Buffer + 1;
  end Put;
end Bounded_Buffer;
```

The `Get` entry is guarded by the barrier '**when** `Number_In_Buffer /= 0`'; only when this evaluates to true can a task execute the `Get` entry; similarly with the `Put` entry. Barriers in protected objects have a similar function to guards in task select statements. They define a precondition; only when they evaluate to true can the entry be executed.

Important note: It is possible that more than one task may be queued on a particular protected entry. *As with task entry queues, a protected entry queue is, by default, ordered in a first-in-first-out fashion; however, other queuing disciplines are allowed* (see Section 13.4.)

In the `Bounded_Buffer` example, when the buffer is empty, several tasks may queue up waiting on the `Get` entry.

In summary, protected objects can define three types of operations: entries, functions and procedures. An entry should be used if there are synchronisation constraints associated with the call; otherwise: if the operation requires only read access to the internal state of the protected object, then a function should be used; otherwise, a procedure is the appropriate operation type.

7.4 Entry calls and barriers

To issue a call to a protected object, a task simply names the object and the required subprogram or entry. For example, to place some data into the above bounded buffer requires the calling task to do

```
My_Buffer.Put(Some_Item);
```

As with task entry calls, the caller can use the select statement to issue a *timed* or *conditional* entry call.

Important note: At any instant in time, a protected entry is either open or closed. It is open if, when checked, the boolean expression evaluates to true; otherwise it is closed. Generally, the protected entry barriers of a protected object are evaluated when:

- a task calls one of its protected entries and the associated barrier references a variable or an attribute that might have changed since the barrier was last evaluated; or
- a task executes and leaves a protected procedure or protected entry and there are tasks queued on entries that have barriers which reference variables or attributes that might have changed since the barriers were last evaluated.

Barriers are not evaluated as a result of a protected function call.

Warning: A program should not use shared variables in barriers and should not rely on the exact circumstances in which barriers are re-evaluated (that is, an implementation may evaluate them more often than is strictly necessary). However, note that it is safe to use the Count attribute in a barrier (compared with the use of this attribute as part of a guard within the select statement – see Section 6.5). This is because adding and subtracting a task to and from an entry queue is a protected action (see below).

When a task calls a protected entry or a protected subprogram, the protected object may already be locked: if one or more tasks are executing protected functions inside the protected object, the object is said to have an active **read lock**; if a task is executing a protected procedure or a protected entry, the object is said to have an active **read/write lock**. The following actions take place when a task attempts to enter a protected object (in the order given):

(1) If the protected object has an active read lock and the call is a function call, the function is executed and step (14) is executed afterwards.

(2) If the protected object has an active read lock and the call is an entry or a procedure call, the call is delayed while there are tasks active in the protected object. †

(3) If the protected object has an active read/write lock, the call is delayed whilst there are tasks with conflicting access requirements active in the protected object.

(4) If the protected object has no active lock and the call is a function call, the protected object read lock becomes active and step (5) is executed.

(5) The function is executed; step (14) is then executed.

(6) If the protected object has no active lock and the call is a procedure or entry call, the protected object read/write lock becomes active and step (7) is executed.

(7) If the call is a procedure call, the procedure is executed and step (10) is then executed.

(8) If the call is an entry call, its associated barrier is evaluated (if necessary) and if true the entry body is executed; step (10) is then executed.

(9) If the barrier is false, the call is placed on a queue associated with the barrier (any timed or conditional entry calls are considered now) and step (10) is executed.

(10) Any entry barriers with tasks queued whose barriers reference variables or attributes which might have changed since they were last evaluated are re-evaluated, and step (11) is executed.

(11) If there are any open entries, one is chosen (the core language does not define which one, although selection policies can be defined by the programmer if the Real-Time Systems Annex is supported, see Chapter 13), the associated entry body is executed, and step (10) is executed.

(12) If no barriers with queued tasks are open, then step (13) is executed.

(13) If one or more tasks are delayed awaiting access to the protected object then either a single task that requires the read/write lock is allowed to enter, or all tasks that require the read lock are allowed to enter and step (5), (7) or (8) is executed (by the associated task); otherwise the access protocol is finished.

(14) If no tasks are active in the protected object, step (13) is executed; otherwise the access protocol is finished.

† Note that the semantics do not require the task to be queued or the run-time scheduler to be called. There is no suspended state associated with attempting to gain access to a protected object. Although this at first may seem strange, the Real-Time Systems Annex defines an implementation model which guarantees that the task can never be delayed when attempting to enter a protected object (see Section 13.3).

Important note: The main point to note about the actions performed above is that when a protected procedure or entry is executed (or when a task queues on an entry), entry barriers are re-evaluated and, potentially, entry bodies are executed. *It is not defined which task executes these entry bodies; it may be the task that issues the associated entry call, or the current task that caused the entry to become open.*

This issue will be discussed further in Chapter 13 when the Real-Time Systems Annex is considered. Furthermore, timed and conditional entry calls are not considered until the task is placed on an entry queue; any delay in accessing the protected object is not considered to be a suspension.

A task that is already active in a protected object can call a subprogram in the same protected object; this is considered to be an internal call and therefore executes immediately.

The `Count` *attribute*

Protected object entries, like task entries, have a `Count` attribute defined that gives the current number of tasks queued on the specified entry.

Important note: Even if a task is destined to end up on an entry queue (due to the barrier being closed), it requires the read/write lock to be placed on the queue. Moreover, having been put on the queue, the `Count` attribute will have changed (for that queue) and hence any barriers that have made reference to that attribute will need to be re-evaluated. Similarly, removal of a task from an entry queue (as a result of a timed entry call or because the caller has been aborted) requires the read/write lock.

This is reflected in step (10) above. To give an example of the use of the `Count` attribute consider a protected object that blocks calling tasks until five are queued; they are then all released:

```
protected Blocker is
  entry Proceed;
private
  Release : Boolean := False;
end Blocker;

protected body Blocker is
  entry Proceed when Proceed'Count = 5 or Release is
  begin
    if Proceed'Count = 0 then
```

```
      Release := False;
    else
      Release := True;
    end if;
  end Proceed;
end Blocker;
```

When the fifth task calls `Blocker.Proceed`, it will find that the barrier evaluates to false and hence it will be blocked. But the barrier will then be re-evaluated (as `'Count` has changed). It now evaluates to true and the first task in the queue will execute the entry. The `Release` boolean variable ensures that all the other four tasks will then pass through the barrier. The last one, however, ensures that the barrier is set to false again. Note that whilst releasing the five tasks, it is not possible for another task to become queued on the entry. This is because queuing a task on an entry requires the read/write lock, and this is not released until the fifth task has left the protected object.

To give a more concrete example of the use of protected objects consider the telephone example used in previous chapters. Some receiver has decided to make use of an answering phone; if a caller cannot make a direct connection (rendezvous) within a specified time interval, then a message is left on the answering phone. This phone has a finite capacity and hence may become full. The telephone is represented by a package:

```
with Message_Type; use Message_Type;
package Telephone is
  procedure Call(M : Message);
    -- for some appropriate message type
  Not_Operational : exception;
end Telephone;
```

The body of this package contains the task `Client` representing the receiver, a protected object (`Answering_Machine`) and the implementation of the procedure `Call`:

```
package body Telephone is
  Timeout_On_Client : constant Duration := 15.0;
  Timeout_On_Answering_Machine : constant Duration := 2.0;
  Max_Messages : constant Positive := 64;
  type Message_Set is ...;

  task Client is
    entry Call(M : Message);
  end Client;

  protected Answering_Machine is
    entry Call(M : Message);
    procedure Replay(MS : out Message_Set);
  private
```

```
    Tape : Message_Set;
    Messages_On_Tape : Natural := 0;
  end Answering_Machine;

  procedure Call(M : Message) is
  begin
    select
      Client.Call(M);
    or
      delay Timeout_On_Client;
      select
        Answering_Machine.Call(M);
      or
        delay Timeout_On_Answering_Machine;
        raise Not_Operational;
      end select;
    end select;
  exception
    when Tasking_Error => -- client no longer callable
      raise Not_Operational;
  end Call;

  protected body Answering_Machine is
    entry Call(M : Message)
        when Messages_On_Tape < Max_Messages is
    begin
      Messages_On_Tape := Messages_On_Tape + 1;
      -- put message on Tape
    end Call;

    procedure Replay(MS : out Message_Set) is
    begin
      -- rewind Tape
      MS := Tape;
      Messages_On_Tape := 0;
    end Replay;
  end Answering_Machine;

  task body Client is separate;
    -- includes calls to Answering_Machine.Replay
end Telephone;
```

Note that this example makes use of a timed entry call on a protected entry. The semantics of this feature are identical to those of a timed entry call on a task. Conditional entry calls can also be made.

7.5 Private entries and entry families

So far this chapter has considered the basic protected type. As with tasks, protected objects may have private entries. These are not directly visible to users of the protected object. They may be used during requeue operations (see Chapter 8).

A protected type can also declare a family of entries by placing a discrete subtype definition in the specification of the entry declaration. Unlike task entry families, however, the programmer need not provide a separate entry body for each member of the family. The barrier associated with the entry can use the index of the family (usually to index into an array of booleans). Consider the following:

```
type Family is Integer range 1 .. 3;

protected Controller is
  entry Request(Family)(...);
end Controller;

task Server is
  entry Service(Family)(...);
end Server;
```

In the case of the `Server` task it is necessary, inside the body of the task, to execute an accept statement naming each member of the family. For example, to wait for a request from any member requires a select statement:

```
task body Server is
begin
  ...
  loop
    select
      when Some_Guard_1 =>
        accept Service(1)(...) do
          ...
        end Service;
    or
      when Some_Guard_2 =>
        accept Service(2)(...) do
          ...
        end Service;
    or
      when Some_Guard_3 =>
        accept Service(3)(...) do
          ...
        end Service;
    end select;
    ...
  end loop;
end Server;
```

For the protected body, it is not necessary to enumerate all the members of the family (indeed, the programmer is not allowed to do so). Instead, a shorthand notation is provided:

```
protected body Controller is
  entry Request(for I in Family )(...)
      when Some_Barrier_Using(I) is
  begin
    ...
  end Request;
end Controller;
```

This is notionally equivalent to

```
-- Not Valid Ada
protected body Controller is
  entry Request(1)(...) when Some_Barrier_Using(1) is
  begin
    ...
  end Request;

  entry Request(2)(...) when Some_Barrier_Using(2) is
  begin
    ...
  end Request;

  entry Request(3)(...) when Some_Barrier_Using(3) is
  begin
    ...
  end Request;
end Controller;
```

For example, the following defines a protected type which provides a group communication facility. The type `Group` defines several communications groups. The protected procedure `Send` transmits a `Data_Item` to a particular group. The family of entries `Receive` allows a task to wait for a `Data_Item` on a particular group:

```
type Group is range 1 .. 10;
type Group_Data_Arrived is array(Group) of Boolean;

protected type Group_Controller is
  procedure Send(To_Group : Group;
                 This_Data : Data_Item);
  entry Receive(Group)(Data : out Data_Item);
private
  Arrived : Group_Data_Arrived := (others => False);
  The_Data : Data_Item;
end Group_Controller;

My_Controller : Group_Controller;

protected body Group_Controller is
  procedure Send(To_Group : Group; This_Data : Data_Item) is
  begin
    if Receive(To_Group)'Count > 0 then
      Arrived(To_Group) := True;
      The_Data := This_Data;
    end if;
  end Send;
  entry Receive(for From in Group)(Data : out Data_Item)
                 when Arrived(From) is
     -- this is a family of entries
  begin
    if Receive(From)'Count = 0 then
       Arrived(From) := False;
    end if;
    Data := The_Data;
  end Receive;
end Group_Controller;
```

When a task sends data to a particular group, the `Send` procedure looks to see if any tasks are waiting on the `Receive` entry for that particular member of the family. If tasks are waiting, it sets a boolean flag associated with the member to true.† On exit from the procedure, the barriers associated with the `Receive` family entries are re-evaluated. The appropriate group's boolean evaluates to true and so that entry of the family is open. The entry body is executed and, if only one task is waiting, the boolean is set to false; in either case the data is sent to the first queued task. If the guard is still open, the entry body is executed again, and so on until all tasks queued for the group are released with the multi-cast value. Note that, once a task is executing inside the protected object, no other task can join an entry queue or be removed from an entry queue.

There are other ways of manipulating the barrier on the entry family (for example saving the identifier of the group during the `Send` operation and comparing the family index with the group identifier). However, the given solution is easily extendible if the message is to be sent to more than one group.

It is useful to dwell on one further feature of the above code. In general, a call of `Send` may release blocked tasks on `Receive`. But it cannot just set the barrier to true on `Receive`: it must first check to see if any task is blocked (using the `'Count` attribute). Moreover, the last task to be released must set the barrier again to false. These checks must be made, as a change to a barrier value is persistent. This can be compared with a signal on a monitor's condition variable (see Chapter 3), which has either an immediate effect or no effect at all.

7.6 Restrictions on protected objects

In general, code executed inside a protected object should be as brief as possible. This is because whilst the code is being executed other tasks are delayed when they try to gain access to the protected object. The Ada language clearly cannot enforce a maximum length of execution time for a protected action. However, it does try to ensure that a task cannot be blocked indefinitely waiting to gain access

† Note that the `Count` attribute can only be applied to a specific family member.

to a protected procedure or a protected function. (As a protected entry has a barrier that must evaluate to true before the operation can be executed, the language can make no guarantees that it will ever evaluate to true.)

Definition: The ARM defines it to be a bounded error to call a *potentially blocking* operation from within a protected action. The following operations are defined to be potentially blocking:

- a select statement;
- an accept statement;
- an entry call statement;
- a delay statement;
- task creation or activation;
- a call on a subprogram whose body contains a potentially blocking operation.

It is also a bounded error to call an external procedure, from a protected object, which in turn calls back to the protected object. This is because the calling task still holds the read/write lock and attempts to acquire the lock again.

For example,

```
protected P is
  procedure X;
  procedure Y;
end P;

procedure External is
begin
  P.Y; -- or P.X
end External;

protected body P is
  procedure X is
  begin
    External;
  end X;

  procedure Y is ...
end P;
```

Of course, there may be several levels of indirection before the call is returned to the protected object. The ARM distinguishes between external and internal calls by the use, or not, of the full protected object name. So, for example, a call to `P.Y` from within `X` would be deemed an external call and therefore a bounded error, whilst a call of just `Y` from within `X` is an internal call and is thus legal. A similar situation applies to internal and external requeue operations (see Chapter 8). In contrast, it is always a bounded error to call an entry, either internal or external, from a protected object.

Warning: If this bounded error is detected, `Program_Error` is raised. If not detected, the bounded error might result in deadlock.

Important note: A call to an external protected subprogram is NOT considered a potentially blocking action.

This might seem strange, as a task may have to wait to gain access to an external subprogram because some other task is already active inside the protected object. However, as the other task cannot block inside the target protected object, the time the first task must wait will be bounded (and hopefully small).

In general, it may be difficult for the programmer to know whether subprograms in library packages are potentially blocking or not. Consequently, all subprogram declarations should contain comments to indicate whether they block. The ARM indicates that all language defined library units which are declared *pure* contain no potentially blocking operations. Furthermore, package `Calendar` does not contain any potentially blocking operations. However, all input and output related packages are potentially blocking.

Currently, the way protected actions have been defined (in the core language), it is possible for two tasks to deadlock by one calling protected object A, which calls a protected subprogram in protected object B, and the other calling protected object B, which calls a subprogram in protected object A. Clearly, there is an order of execution whereby it is possible for the first task to execute a protected action in A and then for the other task to execute a protected action in B. Now neither task can complete its protected action until the other has completed. This problem is common to all concurrent programming languages that allow tasks to request and allocate non-sharable resources (see discussion in the previous chapter). However, the problem can be avoided in Ada if the Real-Time Systems Annex is supported (see Section 13.3).

7.7 Access variables and protected types

In common with all types in Ada, it is possible to declare an access type to a protected type. This enables pointers to instances of protected types to be passed as parameters to subprograms and entries. Consider, by way of an example, a protected type that implements a broadcast of aircraft altitude data (Section 11.8 will generalise the facility to illustrate a generic broadcast facility):

```
  protected type Broadcast is
    procedure Send(This_Altitude : Altitude);
    entry Receive(An_Altitude : out Altitude);
  private
    Altitude_Arrived : Boolean := False;
    The_Altitude : Altitude;
  end Broadcast;
```

Suppose now that there are various tasks on the aircraft which can measure the altitude by different means (barometric pressure, radar altimeter, inertial navigation etc.). It is possible to implement the following name server:

```
with Broadcasts; use Broadcasts;
package Name_Server is
  type Ptr_Broadcast is access all Broadcast;
  subtype Group_Name is String(1 .. 20);

  procedure Register(G : Ptr_Broadcast; Name : Group_Name);
  function Find(Name : Group_Name) return Ptr_Broadcast;
end Name_Server;

package body Name_Server is separate;
  -- details of no importance here
```

The above package declares an access type for the `Broadcast` protected type and allows clients to register and retrieve names. It is now possible for the various tasks to register their names and broadcast altitude readings to those tasks that are interested. For example,

```
task Barometric_Pressure_Reader;
task body Barometric_Pressure_Reader is
  My_Group : Ptr_Broadcast := new Broadcast;
  Altitude_Reading : Altitude;
begin
  Name_Server.Register(My_Group, "Barometric_Pressure ");
  loop
    ...
    -- periodically
    My_Group.Send(Altitude_Reading);
  end loop;
end Barometric_Pressure_Reader;

task Auto_Pilot;
task body Auto_Pilot is
  Bp_Reader :  Ptr_Broadcast;
  Current_Altitude : Altitude;
begin
  Bp_Reader := Name_Server.Find("Barometric_Pressure ");
  ...
  select
    Bp_Reader.Receive(Current_Altitude);
```

```
      -- get new reading if available
  or
    delay 0.1;
  end select;
  ...
end Auto_Pilot;
```

Of course, as with task access variables, an instance of a protected type can be declared with the aliased keyword, in which case a pointer can be obtained using the `'Access` attribute:

```
task Barometric_Pressure_Reader;
task body Barometric_Pressure_Reader is
  My_Group : aliased Broadcast;
  Altitude_Reading : Altitude;
begin
  Name_Server.Register(My_Group'Access, "Barometric_Pressure ");
  loop
    ...
    -- periodically
    My_Group.Send(Altitude_Reading);
  end loop;
end Barometric_Pressure_Reader;
```

As well as declaring access types for protected objects, Ada also allows the programmer to declare an access type to a protected subprogram:

```
access_to_subprogram_definition ::=
  access [ protected ] procedure parameter_profile |
  access [protected] function parameter_and_result_profile
```

Examples of this facility will be given later in Chapter 11; however, it should be noted that access types to protected entries are not allowed.

7.8 Elaboration, finalisation and exceptions

A protected object is elaborated when it comes into scope in the usual way. However, a protected object cannot simply go out of scope if there are still tasks queued on its entries. Finalisation of a protected object requires that any tasks left on entry queues have the exception `Program_Error` raised. Although this may seem unusual, there are two situations where it can happen:

- A protected object, created by an allocator, is subject to unchecked deallocation via the access pointer to it.
- A task calls an entry in another task which requeues the first task on a protected object which then goes out of scope (this possibility is discussed again in Section 8.3).

In an earlier discussion, it was mentioned that the exception `Program_Error` can be raised when a protected action issued a potentially blocking operation. Other exceptions can be raised during the execution of protected operations:

- any exception raised during the evaluation of a barrier results in `Program_Error` being raised in all tasks currently waiting on ALL the entry queues associated with the protected object containing the barrier;
- any exception raised whilst executing a protected subprogram or entry, and not handled by the protected subprogram or entry, is propagated to the task that issued the protected call (as if the call were a normal subprogram call).

7.9 Shared data

This discussion on protected types is concluded by returning to their most common usage – the provision of mutual exclusion over some shared data. At the beginning of the chapter a protected shared integer was used to illustrate a simple protected object. The generalisation of this structure is a generic package that defines an appropriate protected type. Note that it is not possible to have a generic protected object (or generic task):

```
generic
  type Data_Item is private;
  Default_Data : Data_Item;
package Shared_Data_Template is
  protected type Shared_Data is
    function Read return Data_Item;
    procedure Write(New_Value : in Data_Item);
  private
    The_Data : Data_Item := Default_Data;
  end Shared_Data;
end Shared_Data_Template;
```

```
package body Shared_Data_Template is
  protected body Shared_Data is
    function Read return Data_Item is
    begin
      return The_Data;
    end Read;

    procedure Write(New_Value : in Data_Item) is
    begin
      The_Data := New_Value;
    end Write;
  end Shared_Data;
end Shared_Data_Template;
```

Note that, as a discriminant can only be of access or discrete type, the default value of the data item must be assigned from the generic parameter. And hence, all objects of the same instantiated type must have the same initial value.

7.10 The readers and writers problem

One of the motivations for the notion of a protected object is that a single mechanism provides both mutual exclusion and condition synchronisation. Unfortunately, mutual exclusion is too strong a constraint for many synchronisation protocols. A commonly used example of such protocols is the readers and writers problem. Consider a (non-sharable) resource such as a file. Because of multiple update difficulties, the necessary synchronisations are such that if one task is writing to the file, then no other task should be either writing or reading. If, however, there is no writer task, then any number of tasks can have read access.

As just illustrated, a standard protected object can implement the readers/writers algorithm if the read operation is encoded as a function and the write as a procedure. There are, however, two drawbacks with this simple approach:

(1) The programmer cannot easily control the order of access to the protected object; specifically, it is not possible to give preference to write operations over reads.
(2) If the read or write operations are potentially blocking, then they cannot be made from within a protected object.

To overcome these difficulties, the protected object must be used to implement an access control protocol for the read and write operations (rather than encapsulate them). The following code does this whilst giving preference to writes over reads:

```
with Data_Items; use Data_Items;
package Readers_Writers is
  procedure Read (I : out Item);  -- for some type Item
  procedure Write (I : Item);
end Readers_Writers;

package body Readers_Writers is
  procedure Read_File(I : out Item) is separate;
  procedure Write_File(I : Item) is separate;

  protected Control is
    entry Start_Read;
    procedure Stop_Read;
    entry Start_Write;
    procedure Stop_Write;
  private
    Readers : Natural := 0; -- Number of current readers
    Writers : Boolean := False; -- Writers present
  end Control;
  procedure Read (I : out Item) is
  begin
    Control.Start_Read; -- entry protocol
      Read_File(I);
    Control.Stop_Read;  -- exit protocol
  end Read;

  procedure Write (I : Item) is
  begin
    Control.Start_Write; -- entry protocol
      Write_File(I);
    Control.Stop_Write;  -- exit protocol
  end Write;

  protected body Control is
    entry Start_Read when not Writers and
                Start_Write'Count = 0 is
    begin
      Readers := Readers + 1;
    end Start_Read;

    procedure Stop_Read is
    begin
      Readers := Readers - 1;
    end Stop_Read;

    entry Start_Write when not Writers and Readers = 0 is
    begin
      Writers := True;
    end Start_Write;

    procedure Stop_Write is
    begin
      Writers := False;
    end Stop_Write;
  end Control;
end Readers_Writers;
```

Important note: The readers/writers problem provides a good example of how to solve complex synchronisation problems. The operations that have the synchronisation constraints are bracketed by entry and exit protocols. The entry protocol ensures that the conditions are right for the operation to be performed. The exit protocol ensures that any used variables are left in an appropriate state to allow the next operation to proceed.

The entry protocol for a writer ensures that a writer can only begin writing if there is no writer currently writing and there are no current readers. The use of

`'Count` on the `Start_Read` barrier ensures that waiting writers are given preference. The exit protocol for both readers and writers is a non-blocking call to announce the termination of that operation.

This package has addressed both the criticisms that were apparent with the simple use of a protected object. However, a slightly simpler form is available if the `Write_File` procedure is non-blocking (and hence can be made from within the protected control object):

```
package body Readers_Writers is
  procedure Write_File(I : Item) is separate;
  procedure Read_File(I : out Item) is separate;

  protected Control is
    entry Start_Read;
    procedure Stop_Read;
    entry Write(I : Item);
  private
    Readers : Natural := 0;
  end Control;

  procedure Read (I : out Item) is
  begin
    Control.Start_Read;
      Read_File(I);
    Control.Stop_Read;
  end Read;

  procedure Write (I : Item) is
  begin
    Control.Write(I);
  end Write;

  protected body Control is
    entry Start_Read when Write'Count = 0 is
    begin
      Readers := Readers + 1;
    end Start_Read;

    procedure Stop_Read is
    begin
      Readers := Readers - 1;
    end Stop_Read;

    entry Write(I : Item) when Readers = 0 is
    begin
      Write_File(I);
    end Write;
  end Control;
end Readers_Writers;
```

It should be noted that neither of these solutions is resilient to the readers and writers failing whilst executing their critical sections. For example, a reader process which fails in `Read_File` will cause all write operations to be suspended indefinitely. A more robust version of these algorithms is given in Section 9.5.

7.11 The specification of synchronisation agents

The readers and writers problem illustrates the use of a package, with procedure specifications, to encapsulate the necessary synchronisations for some resource usage. The package specification does not, however, give any indication of the protocols that the package body will implement. Indeed, it is usually not clear that a task can be delayed by calling one of these apparently straightforward procedures. The use of comments can help in the readability and understanding of such packages. Moreover, they form a valuable aid in the verification of programs. The simplest property that a synchronisation package can have is that it will lead to the potential blocking of callers. It is recommended that such potential blocking should always be indicated:

```
package Readers_Writers is
  procedure Read(I : out Item);
    -- potentially blocking
  procedure Write(I : Item);
    -- potentially blocking
end Readers_Writers;
```

For more informative specifications, a more formal notation can be used in the comments. For example, path expressions can be employed to state quite explicitly the protocol that is, or should be, supported by the package body. The following comment implies mutual exclusion:

```
--| path 1 : (Read, Write) end
```

Here one procedure is permitted at a time and it is either a `Read` or a `Write`. In general, the readers/writers algorithm allows a number of simultaneous reads, so that the associated path expression becomes

```
--| path 1 : ([Read], Write) end
```

The square brackets imply ‘de-restriction’. If a specific sequence of calls is required, then a semicolon can replace the comma:

```
--| path 1 : (Write; Read) end
```

This now implies that there must be a strict sequence to the calls of the procedures; first `Write`, then `Read`, then `Write`, etc. The appropriate comment for the bounded buffer example can now be developed. Firstly, there must be a `Put` before a `Get`, so:

```
--| path 1 : (Put; Get) end
```

But this is too restrictive for there can be up to `N` `Puts` before a `Get` (for buffer size `N`), that is, `Put` can get ahead of `Get` by up to `N` calls:

```
--| path N : (Put; Get) end
```

Unfortunately, this has now removed the mutual exclusion property. For a buffer, a call to `Put` can be concurrent with a call to `Get` but not with another call to `Put`; therefore the `Puts` and `Gets` must be protected from themselves:

```
--| path N : (1 : (Put); 1 : (Get)) end
```

If the construct were a stack, then mutual exclusion would have to extend to both subprograms. This is indicated by giving a second path restriction – the implication being that both must be obeyed:

```
--| path N : (Put; Get), 1 : (Put, Get) end
```

The first part allows calls to `Put` to get ahead of calls to `Get` by `N`; the second part states that mutual exclusion on both procedures is necessary.

Path expressions were first defined by Campbell and Habermann (1974), since when a number of extensions and variations have been proposed. It is recommended that if a package is hiding a synchronisation agent (a protected object or a server task), then some form of formal comment should be used to describe the embedded synchronisations. This comment will be useful both for the design of the system, where the package body may later be coded to the specification described in the comment, and in the general readability and maintainability of the program. Of course, it may also be possible to generate the synchronisation code automatically from the path expressions.

7.12 Shared variables

Passing data safely between two tasks can be achieved via the rendezvous or via protected objects. However, two tasks may often safely pass shared data between them because they are synchronised in some other way and it is not possible for the data to be read by one task whilst it is being written by another. Unfortunately (in this instance), compilers usually attempt to optimise the code, and variables may be kept in registers; consequently a write to a shared variable may not result in the data being written to memory immediately. Hence there is the potential that a reader of the shared variable may not obtain the most up-to-date value. Indeed, if the shared variable is being kept in a register, then the synchronisation implied by the program no longer holds and the variable may be written just when the data is being read, giving an internally inconsistent result.

The Ada language defines the conditions under which it is safe to read and write to shared variables outside the rendezvous or protected objects. All other situations are deemed to result in erroneous program execution. The safe conditions are

- Where one task writes/reads a variable before activating another task, which then reads/writes the same variable, for example

```
The_Data : Shared_Variable;

task type Reader;
task Writer;

task body Reader is
  Param : Shared_Variable := The_Data;
begin
  ...
end Reader;

task body Writer is
begin
  ...
  The_Data := ...; -- pass parameter to Reader
  declare
    A_Reader : Reader;
  begin
    ...
  end;
end Writer;
```

- Where the activation of one task writes/reads the variable and the task awaiting completion of the activation reads/writes the same variable, for example

```
The_Data : Shared_Variable;

task type Reader;
task Writer;

task body Reader is
  Param : Shared_Variable := Read_And_Update(The_Data);
begin
  ...
end Reader;

task body Writer is
begin
  ...
  The_Data := ...; -- pass parameter
  declare
    A_Reader : Reader;
  begin
    -- updated value now available
  end;
end Writer;
```

- Where one task writes/reads the variable and another task waits for the termination of the task and then reads/writes the same variable, for example

```
The_Data : Shared_Variable;

task type Writer;
task Reader;

task body Writer is
begin
  ...
  The_Data := ...; --pass back some results
  ...
end Writer;

task body Reader is
  Param : Shared_Variable;
begin
  ...
  declare
    A_Writer : Writer;
  begin
    ...
  end;
  Param := The_Data;  -- The_Data has now been written
end Reader;
```

- Where one task writes/reads the variable before making an entry call on another task, and the other task reads/writes the variable during the corresponding entry body or accept statement, for example

```
The_Data : Shared_Variable;

task Reader is
  entry Synchronise;
end Reader;

task Writer;

task body Reader is
  Param : Shared_Variable;
begin
  ...
  accept Synchronise do
    Param := The_Data;
  end;
  ...
end Reader;

task body Writer is
begin
  ...
  The_Data := ...;
  Reader.Synchronise;
  ...
end Writer;
```

- Where one task writes/reads a shared variable during an accept statement and the calling task reads/writes the variable after the corresponding entry call has returned, for example

```
The_Data : Shared_Variable;

task Updater is
  entry Synchronise;
end Updater;

task Writer_And_Reader;

task body Updater is
begin
  ...
  accept Synchronise do
    The_Data := The_Data + ...;
  end;
  ...
end Updater;

task body Writer_And_Reader is
begin
  ...
  The_Data := ...;
  Updater.Synchronise;
  if The_Data = ...; -- The_Data has now been updated
  ...
end Writer_And_Reader;
```

- Where one task writes/reads a variable whilst executing a protected procedure body or entry, and the other task reads/writes the variable as part of a later execution of an entry body of the same protected body, for example

```
The_Data : Shared_Variable;

protected Updater is
  procedure Write;
  entry Read;
private
  Data_Written : Boolean := False;
end Updater;

protected body Updater is
  procedure Write is
  begin
    The_Data := ...;
    Data_Written := True;
  end Write;
  entry Read when Data_Written is
  begin
    if The_Data = ...;
  end;
end Updater;
```

Although data can be safely passed during the above operations, such mechanisms must be used with care as they are error-prone and lead to obscure programs.

7.13 Volatile and atomic data

If the Systems Programming Annex is supported, there are extra facilities which can be used to control shared variables between *unsynchronised* tasks. They come in the form of extra pragmas which can be applied to certain objects or type declarations.

Pragma `Volatile`

The purpose of indicating that a shared data item is volatile is to ensure that the compiler does not optimise the code and keep the item in a register whilst it is manipulating it. Pragma `Volatile` ensures that all reads and writes go directly to memory.

Pragma `Volatile` can be applied to

- An object declaration, for example

```
I : Integer;
pragma Volatile(I); -- object I is volatile

J : Some_Record_Type;
pragma Volatile(J); -- object J is volatile

K : Some_Array_Type;
pragma Volatile(K); -- object K is volatile
```

- A non-inherited component declaration, for example

```
type Record_With_Volatile_Component is tagged
  record
    Component_1 : Integer;
    Component_2 : Float;
    Component_3 : Some_Enumeration_Type;
  end record;
pragma Volatile(Record_With_Volatile_Component.Component_2);
-- all Component_2 elements of all objects of
-- type Record_With_Volatile_Component will be volatile
```

```
type Inherited_Record is new
     Record_With_Volatile_Component with
  record
    Component_4 : Some_Type;
  end record;

-- Note that the following is ILLEGAL
pragma Volatile(Inherited_Record.Component_3);

-- However, this is legal:
pragma Volatile(Inherited_Record.Component_4);
```

- A full type declaration (as opposed to a private type or an incomplete type), for example

```
type Volatile_Data is range 1 .. 100;
pragma Volatile(Volatile_Data);
-- all objects created from the type will be volatile

type Volatile_Record is
  record
    Component_1 : Integer;
    Component_2 : Float;
    Component_3 : Some_Enumeration_Type;
  end record;
pragma Volatile(Volatile_Record);
-- all objects created will be volatile, also all
-- components will be volatile (similarly for arrays)
```

Pragma `Volatile_Components`

Pragma `Volatile_Components` applies to components of an array, for example

```
type My_Array is array(1..10) of Integer;
pragma Volatile_Component(My_Array);
  -- all array components are volatile for all
  -- objects of type My_Array
```

Pragma `Atomic` *and pragma* `Atomic_Components`

Whilst pragma `Volatile` indicates that all reads and writes must be directed straight to memory, pragma `Atomic` imposes the further restriction that they must be indivisible. That is, if two tasks attempt to read and write the shared variable at the same time, then the result must be internally consistent. Consider the following example:

```
subtype Axis is Integer range 0 .. 100;
type Point is
  record
    X : Axis;
    Y : Axis;
  end record;

Diagonal : Point := (0,0);
-- invariant X = Y
pragma Volatile(Diagonal);

procedure Draw(X : Point) is ...
function New_Point return Point is ...

task Plot_Point;

task Update_Point;

task body Plot_Point is
begin
  loop
    Draw(Diagonal);
    ...
  end loop;
end Plot_Point;

task body Update_Point is
begin
  loop
    ...
    Diagonal := New_Point;
  end loop;
end Update_Point;
```

In this example the two tasks both have access to the shared variable `Diagonal`. Suppose the values returned from the `New_Point` function are {(0,0), (1,1), (2,2), ...}. Although the shared variable has been declared as volatile, this does not stop the read and the write operations becoming interleaved. It is quite possible for the `Plot_Point` task to draw a point (0,1), (2,1) and so on. Indeed, the program is erroneous.

To cure this problem, the `Diagonal` variable must be made atomic:

```
pragma Atomic(Diagonal);
```

This ensures not only that all reads and writes are directed to memory (that is, the variable is volatile), but also that reads and writes are indivisible; they cannot be interleaved.

Important note: An implementation is not required to support atomic operations for all types of variable; however, if not supported for a particular object, the pragma must be rejected by the compiler.

The same classes of item as supported by the `Volatile` and `Volatile_Components` pragmas are potentially supportable as `Atomic` and `Atomic_Components`.

Mutual exclusion and Simpson's algorithm

Many implementations of the Systems Programming Annex will only support pragma `Atomic` on variables which can be read or written as atomic actions at the machine assembly language level. Typically, this restricts the size of an object to a single word in length. However, using just this basic property it is possible to construct an algorithm which will guarantee that any size data item can be accessed without interference and without having to resort to busy-waiting. One such algorithm, due to Simpson (1990), is given here for two tasks:

```
generic
  type Data is private;
  Initial_Value : Data;
package Simpsons_Algorithm is
  procedure Write(Item : Data); -- non-blocking
  procedure Read (Item : out Data); -- non-blocking
end Simpsons_Algorithm;


package body Simpsons_Algorithm is

  type Slot is (First, Second);

  Four_Slot : array (Slot, Slot) of Data :=
             (First => (Initial_Value,Initial_Value),
              Second => (Initial_Value,Initial_Value));
  pragma Volatile(Four_Slot);

  Next_Slot : array(Slot) of Slot := (First, First);
  pragma Volatile(Next_Slot);

  Latest : Slot := First;
  pragma Atomic(Latest);
  Reading : Slot := First;
  pragma Atomic(Reading);

  procedure Write(Item : Data) is
    Pair, Index : Slot;
  begin
    if Reading = First then
      Pair := Second;
    else
      Pair := First;
    end if;
    if Latest = First then
      Index := Second;
    else
      Index := First;
    end if;
    Four_Slot(Pair, Index) := Item;
    Next_Slot(Pair) := Index;
    Latest := Pair;
  end Write;

  procedure Read(Item : out Data) is
    Pair, Index : Slot;
  begin
    Pair := Latest;
    Reading := Pair;
    Index := Next_Slot(Pair);
    Item := Four_Slot(Pair, Index);
  end Read;
end Simpsons_Algorithm;
```

The algorithm works by keeping four slots for the data: two banks of two slots. The reader and the writer never access the same bank of slots at the same time. The atomic variable `Latest` contains the index of the bank to which the last data item was written and the `Next_Slot` array indexed by this value indicates which slot in that bank contains the data.

Consider some arbitrary time when the latest value of the data item is in `Four_Slot(Second, First)`. In this case, `Latest` equals `Second` and `Next (Second) = First`. Assume also that this is the last value read. If another read request comes in and is interleaved with a write request, the write request will choose the first bank of slots and the first slot, and so on. Thus it is possible that the reader will obtain an old value but never an inconsistent one. If the write comes in again before the read has finished, it will write to the first bank and second slot, and then the first bank and first slot. When the reader next comes in, it will obtain the last value that was completely written (that is, the value written by the last full invocation of `Write`). A full proof of this algorithm is given by Simpson (1990).

7.14 Task states

The task state diagram given in the previous three chapters can be extended again to include the new states that protected objects have introduced (see Figure 7.1). Note the important point made earlier in this chapter: waiting to gain access to a protected object is not a blocked state. Thus the only new state is ‘waiting on a protected entry call’; this will occur when a barrier evaluates to false.

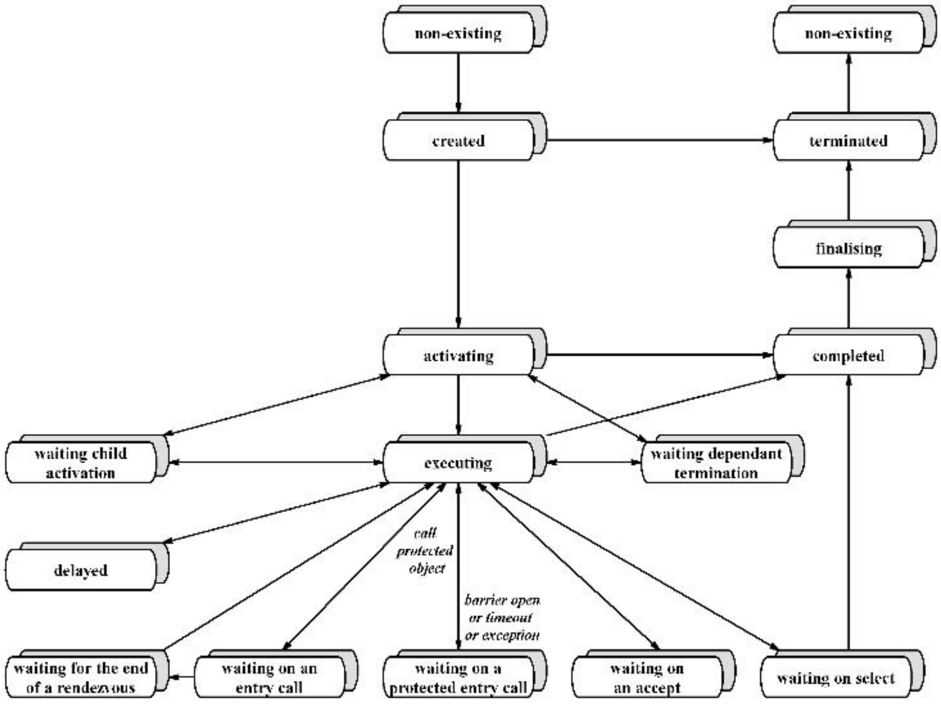


Fig. 7.1: Summary of task states and state transitions

7.15 Summary

Ada gives direct support to protected data by the abstraction of a *protected object*. A protected object encapsulates the data and allows access only via protected subprograms or protected entries. The language guarantees that these subprograms and entries will be executed in a manner that ensures the data is updated under mutual exclusion.

Protected functions allow concurrent access, as they only have a read-only impact on the protected data. Protected procedures and entries can write to the data and hence are only allowed access one at a time. A procedure can always complete its execution inside a protected object (once it has the read/write lock on the data).

By comparison, a protected entry first has to evaluate a boolean barrier. If the barrier evaluates to false, then the calling task is blocked. Whenever a procedure or entry completes, a check is made to see if a blocked task can now proceed (that is, the associated barrier is now true).

In addition to protected objects, Ada also gives direct support to shared variables. It does this by the provision of a number of pragmas: `Volatile`, `Volatile_Components`, `Atomic` and `Atomic_Components`. A volatile variable must be located in just one memory location; temporary copies that the compiler might otherwise use to improve the efficiency of the generated code are not allowed. An atomic variable must, additionally, have indivisible read and write operations. Although these pragmas are defined by the ARM it is up to any implementation to decide which types they can be applied to.

8

Avoidance synchronisation and the requeue facility

The models of synchronisation discussed in the previous four chapters have the common feature that they are based on avoidance synchronisation. Guards or barriers are used to prevent rendezvous and task-protected object interactions when the conditions are not appropriate for the communications event to start. Indeed, one of the key features of the tasking model is the consistent use of avoidance to control synchronisation. The use of guards and barriers represents a high-level abstract means of expressing and enforcing necessary synchronisations; and as such they can be compared favourably with the use of low-level primitives such as semaphores or monitor signals (see Chapter 3). This chapter starts by giving a more systematic assessment of avoidance synchronisation in order to motivate the requirement for 'requeue'. It then describes the syntax and semantics of the requeue statement and gives examples of its use.

8.1 The need for requeue

Different language features are often compared in terms of their *expressive power* and *ease of use* (*usability*). Expressive power is the more objective criterion, and is concerned with the ability of language features to allow application requirements to be programmed directly. Ease of use is more subjective, and includes the ease with which the features under investigation interact with each other and with other language primitives.

In her evaluation of synchronisation primitives, Bloom (1979) used the following criteria to evaluate and compare the expressive power and usability of different language models. She identified several issues that need to be addressed when determining the order of interaction between synchronising agents; a list of such issues is as follows:

(a) type of service requested;

(b) order of request arrival;
(c) internal state of the receiver (including the history of its usage);
(d) priority of the caller;
(e) parameters to the request.

It should be clear that the Ada model described so far deals adequately with the first three situations. The fourth will be considered in Chapter 13, when real-time issues are discussed in detail. Here, attention is focused on the fifth issue, which causes some difficulties for avoidance synchronisation mechanisms.

8.1.1 The resource allocation problem

Resource allocation is a fundamental problem in all aspects of concurrent programming. Its consideration exercises all Bloom's criteria and forms an appropriate basis for assessing the synchronisation mechanisms of concurrent languages.

Consider the problem of constructing a resource controller that allocates some resource to a group of client agents. There are a number of instances of the resource but the number is bounded; contention is possible and must be catered for in the design of the program. If the client tasks only require a single instance of the resource, then the problem is straightforward. For example, in the following, the resource (although not directly represented) can be encoded as a protected object:

```
protected Resource_Controller is
  entry Allocate(R : out Resource);
  procedure Release(R : Resource);
private
  Free : Natural := Max;
  ...
end Resource_Controller;

protected body Resource_Controller is
  entry Allocate(R : out Resource) when Free > 0 is
  begin
    Free := Free - 1;
    ...
  end Allocate;
  procedure Release(R : Resource) is
  begin
    Free := Free + 1;
    ...
  end Release;
end Resource_Controller;
```

To generalise this code requires the caller to state how many resources are required (up to some maximum). The semantics required (to help avoid deadlocks) are that either all requested resources are assigned to the caller or none are (and the caller blocks until the resources are free).

This resource allocation problem is difficult to program with avoidance synchronisation. In order to determine the size of the request, the communication must be accepted and the parameter read. But if, having read the parameter, the internal state of the resource controller is such that there are currently not enough resources available, then the communication must be terminated and the client must try again. To prevent polling, a different entry must be tried. A detailed examination of this problem has shown that an acceptable solution is not available if avoidance synchronisation only is used. Note that a solution is possible, so the issue is one of ease of use rather than expressive power. Nevertheless, the elegance of this solution is poor when compared with the monitor solution given in Section 3.8. A monitor uses condition synchronisation (not avoidance synchronisation) and it is therefore trivially easy to block the caller after the parameter has been read but before it leaves the monitor.

Using entry families

A possible solution in Ada (without requeue) to the resource allocation problem assumes that the number of distinct requests is relatively small and can be represented by a family of entries. Each entry in the family is guarded by the boolean expression `F <= Free`, where `F` is the family index:

```
type Request_Range is range 1 .. Max;

protected Resource_Controller is
  entry Allocate(Request_Range)(R : out Resource);
  procedure Release(R : Resource; Amount : Request_Range);
private
  Free : Request_Range := Request_Range'Last;
  ...
end Resource_Controller;

protected body Resource_Controller is
  entry Allocate(for F in Request_Range)(R : out Resource)
              when F <= Free is
  begin
    Free := Free - F;
   ...
  end Allocate;

  procedure Release(R : Resource; Amount : Request_Range) is
  begin
   ...
    Free := Free + Amount;
  end Release;
end Resource_Controller;
```

Although this solution is concise, there are two potential problems:

(1) This may not be practical for a large number of resources, as there needs to be Max entry queues; these must be serviced individually, which could be inefficient. Also, for server tasks there is no equivalent syntactical form for families and hence an excessively long select statement must be used.
(2) It is difficult to allocate the resources selectively – when several requests can be serviced, an arbitrary choice between them is made (note that if the Real-Time Systems Annex is supported, the request can be serviced in a priority order; if all calling tasks have the same priority, then the family is serviced from the smallest index to the largest).

The latter problem can be ameliorated by having each entry in the family guarded by its own boolean, and then selectively setting the booleans to True. For example, if on freeing new resources it is required to service the largest request first, the following algorithm can be used. Note that a request to Allocate that can be satisfied immediately is always accepted. Hence a boolean variable (Normal) is needed to distinguish between a normal allocation and a phase of allocations following a Release:

```
type Request_Range is range 1 .. Max;
type Bools is array(Request_Range) of Boolean;

protected Resource_Controller is
  entry Allocate(Request_Range)(R : out Resource);
  procedure Release(R : Resource; Amount : Request_Range);
private
  Free : Request_Range := Request_Range'Last;
  Barrier : Bools := (others => False);
  Normal : Boolean := True;
end Resource_Controller;

protected body Resource_Controller is
  entry Allocate(for F in Request_Range)(R : out Resource)
        when F <= Free and (Normal or Barrier(F)) is
  begin
    Free := Free - F;
    if not Normal then
      Barrier(F) := False;
      Normal := True;
      for I in reverse 1 .. F loop
        if Allocate(I)'Count /= 0 and I <= Free then
          Barrier(I) := True;
          Normal := False;
          exit;
        end if;
      end loop;
    end if;
   ...
  end Allocate;
  procedure Release(R : Resource;
                    Amount : Request_Range) is
  begin
    Free := Free + Amount;
    for I in reverse 1 .. Free loop
      if Allocate(I)'Count /= 0 then
        Barrier(I) := True;
        Normal := False;
        exit;
      end if;
    end loop;
    ...
  end Release;
end Resource_Controller;
```

Note that the loop bound in entry `Allocate` is `F` not `Free` as there cannot be a task queued on `Allocate` requiring more than `F` instances of the resource.

The correctness of this algorithm relies on the property that requests already queued upon `Allocate` (when resources are released) are serviced before any new call to `Allocate` (from outside the resources controller). It also relies on tasks not removing themselves from entry queues, that is, after the count attribute has been read (and the associated barrier raised) but before the released task actually executes `Allocate`. Furthermore, it assumes that tasks only ever release resources that they have acquired.

The double interaction solution

One possible solution to the resource allocation problem, which does not rely on a family of entries, is for the resource controller to reject calls that cannot be satisfied. In this approach, the client must first request resources and, if refused, try again. To avoid continuously requesting resources when no new resources are available, the client calls a different entry from the original request entry:

```
type Request_Range is range 1 .. Max;

protected Resource_Controller is
  entry Allocate(R : out Resource; Amount : Request_Range;
                 Ok : out Boolean);
  entry Try_Again(R : out Resource; Amount : Request_Range;
                  Ok : out Boolean);
  procedure Release(R : Resource; Amount : Request_Range);
private
  Free : Request_Range := Request_Range'Last;
  New_Resources_Released : Boolean := False;
  ...
end Resource_Controller;

protected body Resource_Controller is
```

```
  entry Allocate(R : out Resource; Amount : Request_Range;
                 Ok : out Boolean) when Free > 0 is
  begin
    if Amount <= Free then
      Free := Free - Amount;
      Ok := True;
      -- allocate
    else
      Ok := False;
    end if;
  end Allocate;

  entry Try_Again(R : out Resource; Amount : Request_Range;
        Ok : out Boolean) when New_Resources_Released is
  begin
    if Try_Again'Count = 0 then
      New_Resources_Released := False;
    end if;
    if Amount <= Free then
      Free := Free - Amount;
      Ok := True;
      -- allocate
    else
      Ok := False;
    end if;
  end Try_Again;

  procedure Release(R : Resource; Amount : Request_Range) is
  begin
    Free := Free + Amount;
    -- free resources
    if Try_Again'Count > 0 then
      New_Resources_Released := True;
    end if;
  end Release;
end Resource_Controller;
```

To use this controller, each client must then make the following calls:

```
Resource_Controller.Allocate(Res,N,Done);
while not Done loop
  Resource_Controller.Try_Again(Res,N,Done);
end loop;
```

Even this code is not entirely satisfactory, for the following reasons:

(1) The clients must `Try_Again` for their resources each time any resources are released; this is inefficient.

(2) If a client is tardy in calling `Try_Again`, it may miss the opportunity to acquire its resources (as only those tasks queued on `Try_Again`, at the point when new resources become available, are considered).

(3) It is difficult to allocate the resources selectively – when several requests can be serviced, then they are serviced in a FIFO order.

An alternative approach is to require the resource controller to record outstanding requests:

```
type Request_Range is range 1 .. Max;
protected Resource_Controller is
  entry Allocate(R : out Resource; Amount : Request_Range;
                 Ok : out Boolean);
  entry Try_Again(R : out Resource; Amount : Request_Range;
                  Ok : out Boolean);
  procedure Release(R : Resource; Amount : Request_Range);
private
  Free : Request_Range := Request_Range'Last;
  New_Resources_Released : Boolean := False;
  ...
end Resource_Controller;

protected body Resource_Controller is

  procedure Log_Request(Amount : Request_Range) is
  begin
    -- store details of request
  end Log_Request;

  procedure Done_Request(Amount : Request_Range) is
  begin
    -- remove details of request
  end Done_Request;

  function Outstanding_Requests return Boolean is
  begin
    -- returns True if there are outstanding requests to
    -- be serviced
  end Outstanding_Requests;

  procedure Seen_Request(Amount : Request_Range) is
  begin
    -- log details of failed request
  end Seen_Request;

  function More_Outstanding_Requests return Boolean is
  begin
    -- returns True if there are outstanding requests
    -- to be serviced which have not been considered
    -- this time around
  end More_Outstanding_Requests;

  entry Allocate(R : out Resource; Amount : Request_Range;
                 Ok : out Boolean)
        when Free > 0 and not New_Resources_Released is
  begin
```

```
    if Amount <= Free then
      Free := Free - Amount;
      Ok := True;
      -- allocate
    else
      Ok := False;
      Log_Request(Amount);
    end if;
  end Allocate;

  entry Try_Again(R : out Resource; Amount : Request_Range;
        Ok : out Boolean) when New_Resources_Released is
  begin
    if Amount <= Free then
      Free := Free - Amount;
      Ok := True;
      Done_Request(Amount);
      -- allocate
    else
      Ok := False;
      Seen_Request(Amount);
    end if;
    if not More_Outstanding_Requests then
      New_Resources_Released := False;
    end if;
  end Try_Again;

  procedure Release(R : Resource; Amount : Request_Range) is
  begin
    Free := Free + Amount;
    -- free resources
    if Outstanding_Requests then
      New_Resources_Released := True;
    end if;
  end Release;
end Resource_Controller;
```

In order to ensure that tasks waiting on the `Try_Again` entry are serviced before new requests, it is necessary to guard the `Allocate` entry. Unfortunately, this algorithm then breaks down if the client does not make the call to `Try_Again` (due, for example, to being aborted or suffering an asynchronous transfer of control – see Section 9.3). To solve this problem, it is necessary to encapsulate the double interaction in a procedure and provide a dummy controlled variable (as was done in Subsection 6.6.1) which, during finalisations, informs the resource controller (via a new protected procedure `Done_Waiting`) that it is no longer interested:

```
type Resource_Recovery is new
     Finalization.Limited_Controlled with null record;

procedure Finalize(Rr : in out Resource_Recovery) is
begin
```

```
  Resource_Controller.Done_Waiting;
end Finalize;

procedure Allocate(R : out Resource; Amount : Request_Range) is
  Got : Boolean;
  Protection : Resource_Recovery;
begin
  Resource_Controller.Allocate(R, Amount, Got);
  while not Got loop
    Resource_Controller.Try_Again(R, Amount, Got);
  end loop;
end Allocate;
```

Note that with this solution, the `Done_Waiting` routine will be called *every time* the procedure `Allocate` is left (either normally or because of task abortion). The resource controller will therefore have to keep track of the actual client tasks rather than just the requests. It can do this by using task identifiers provided by the Systems Programming Annex. The controller can then determine if a task executing `Done_Waiting` has an outstanding request.

Even with this solution, the controller still has difficulty in allocating resources selectively. However, the fundamental problem with this approach is that the task must make a double interaction with the resource controller even though only a single logical action is being undertaken.

8.1.2 Solutions using language support

Two methods have been proposed to increase the effectiveness of avoidance synchronisation. One of these, requeue, has been incorporated into Ada. The other approach, which is less general purpose, is to allow the guard/barrier to have access to 'in' parameters. This approach is adopted in the language SR. The resource control problem is easily coded with this approach; for example, using Ada-like syntax:

```
type Request_Range is range 1..Max;

protected Resource_Controller is
  entry Allocate(R : out Resource; Amount : Request_Range);
  procedure Release(R : Resource; Amount : Request_Range);
private
  Free : Request_Range := Request_Range'Last;
  ...
end Resource_Controller;

protected body Resource_Controller is
  entry Allocate(R : out Resource; Amount : Request_Range)
        when Amount <= Free is  -- Not Legal Ada
  begin
    Free := Free - Amount;
  end Allocate;

  procedure Release(R : Resource; Amount : Request_Range) is
  begin
    Free := Free + Amount;
  end Release;
end Resource_Controller;
```

The main drawback with this approach is implementational efficiency. It is no longer possible to evaluate a barrier once per entry; each task's placement on the entry queue will lead to a barrier evaluation. However, optimisations are possible that would allow a compiler to recognise when 'in' parameters were not being used; efficient code could then be produced.

The second solution to this problem is to provide a requeue facility.

Important note: The key notion behind requeue is to move the task (which has been through one guard or barrier – we shall use the term 'guard' in this discussion) to 'beyond' another guard.

For an analogy, consider a person (task) waiting to enter a room (protected object) which has one or more doors (guarded entries) giving access to the room. Once inside, the person can be ejected (requeued) from the room and once again be placed behind a (potentially closed) door.

Important note: Ada allows requeues between task entries and protected object entries. A requeue can be to the same entry, to another entry in the same unit, or to another unit altogether. Requeues from task entries to protected object entries (and vice versa) are allowed. However, the main use of requeue is to send the calling task to a different entry of the same unit from which the requeue was executed.

The resource control problem provides illustrative examples of the application of requeue. One solution is given now; some variations are considered later in this chapter (Section 8.4).

Requeue example – concurrent solution to the resource control problem

One of the problems with the double interaction solution was that a task could be delayed (say, due to preemption) before it could requeue on the `Try_Again` entry. Consequently, when new resources became available it was not in a position to have them allocated. Requeue allows a task to be ejected from a protected object and placed back on an entry queue as an atomic operation. It is therefore not possible for the task to miss the newly available resources.

In the following algorithm, an unsuccessful request is now requeued on to a private entry (called Assign) of the protected object. The caller of this protected object now makes a single call on Allocate. Whenever resources are released, a note is taken of how many tasks are on the Assign entry. This number of tasks can then retry to either obtain their allocations or be requeued back onto the same Assign entry. The last task to retry closes the barrier:

```
type Request_Range is range 1 .. Max;

protected Resource_Controller is
  entry Allocate(R : out Resource; Amount : Request_Range);
  procedure Release(R : Resource; Amount : Request_Range);
private
  entry Assign(R : out Resource; Amount : Request_Range);
  Free : Request_Range := Request_Range'Last;
  New_Resources_Released : Boolean := False;
  To_Try : Natural := 0;
 ...
end Resource_Controller;

protected body Resource_Controller is
  entry Allocate(R : out Resource; Amount : Request_Range)
        when Free > 0 is
  begin
    if Amount <= Free then
      Free := Free - Amount;
      -- allocate
    else
      requeue Assign;
    end if;
  end Allocate;

  entry Assign(R : out Resource; Amount : Request_Range)
     when New_Resources_Released is
  begin
    To_Try := To_Try - 1;
    if To_Try = 0 then
      New_Resources_Released := False;
    end if;
    if Amount <= Free then
      Free := Free - Amount;
      -- allocate
    else
      requeue Assign;
    end if;
  end Assign;

  procedure Release(R : Resource; Amount : Request_Range) is
  begin
    Free := Free + Amount;
    -- free resources
    if Assign'Count > 0 then
```

```
      To_Try := Assign'Count;
      New_Resources_Released := True;
    end if;
  end Release;
end Resource_Controller;
```

Note that this will only work if the `Assign` entry queuing discipline is FIFO. When priorities are used, two entry queues are needed. Tasks must requeue from one entry to the other (and back again after the next release). This is illustrated in the example given in Section 8.4.

Finally, it should be observed that a more efficient algorithm can be derived if the protected object records the smallest outstanding request. The barrier should then only be set to true in `Release` (or remain true in `Assign`) if `Free >= Smallest`.

Even with this style of solution, it is difficult to give priority to certain requests other than in FIFO or task priority order. As indicated earlier, to program this level of control requires a family of entries. However, with requeue, a more straightforward solution can be given (as compared with the earlier code that did not use requeue):

```
type Request_Range is range 1 .. Max;
type Bools is array(Request_Range) of Boolean;

protected Resource_Controller is
  entry Allocate(Request_Range)(R : out Resource);
  procedure Release(R : Resource; Amount : Request_Range);
private
  entry Assign(Request_Range)(R : out Resource);
  Free : Request_Range := Request_Range'Last;
  Barrier : Bools := (others => False);
  ...
end Resource_Controller;

protected body Resource_Controller is

  entry Allocate(for F in Request_Range)(R : out Resource)
              when True is
  begin
    if F <= Free then
      Free := Free - F;
      ...
    else
      requeue Assign(F);
    end if;
  end Allocate;

  entry Assign(for F in Request_Range)(R : out Resource)
              when Barrier(F) is
  begin
```

```
    Free := Free - F;
    Barrier(F) := False;
    for I in reverse 1 .. F loop
      if Allocate(I)'Count /= 0 and I <= Free then
        Barrier(I) := True;
        exit;
      end if;
    end loop;
   ...
  end Assign;

  procedure Release(R : Resource; Amount : Request_Range) is
  begin
    Free := Free + Amount;
    for I in reverse 1 .. Free loop
      if Assign(I)'Count /= 0 then
        Barrier(I) := True;
        exit;
      end if;
    end loop;
    ...
  end Release;
end Resource_Controller;
```

This algorithm not only is more straightforward than the one given earlier, but also has the advantage that it is resilient to a task removing itself from an entry queue (after the count attribute has acknowledged its presence). Once a task has been requeued it cannot be aborted or subject to a time-out on the entry call – see following discussion.

8.2 Semantics of requeue

Warning: It is important to appreciate that requeue is not a simple call. If procedure P calls procedure Q, then, after Q has finished, control is passed back to P. But if entry X requeues on entry Y, then control never returns to X. After Y has completed, control passes back to the object that called X. Hence, when an entry or accept body executes a requeue, that body is 'completed, finalized and left' (ARM, Section 9.5.4).

One consequence of this is that when a requeue is from one protected object to another then mutual exclusion on the original object is given up once the task is queued. Other tasks waiting to enter the first object will be able to do so. However, a requeue to the same protected object will retain the mutual exclusion lock (if the target entry is open).

Important note: The entry named in a requeue statement (called the *target* entry) must either have no parameters or have a parameter profile that is equivalent (that is, type conformant) with that of the entry (or accept) statement from which the requeue is made.

However, this does not mean that the values of the parameters cannot be changed prior to the requeue action.

For example, in the resource control program the parameters of `Assign` are identical to those of `Allocate`. Because of this rule, it is not necessary to give the actual parameters with the call; indeed it is forbidden to do so (in case the programmer tries to change them). Hence if the target entry has no parameters, then no information is passed; if it has parameters, then the corresponding parameters in the entity that is executing the requeue are mapped across.

The syntax for requeue is defined as

```
requeue_statement ::= requeue entry_name [with abort]
```

The optional 'with abort' clause has two uses. When a task is on an entry queue it will remain there until serviced unless it made a timed entry call or is aborted (see next chapter). Once the task has been accepted (or starts to execute the entry of a protected object), the time-out is cancelled and the effect of any abort attempt is postponed until the task comes out of the entry. There is, however, a question as to what should happen with requeue. Consider the time-out issue; clearly two views can be taken:

(1) As the first call has been accepted, the time-out should now be cancelled (that is, it cannot have an effect).
(2) If the requeue puts the calling task back onto an entry queue, then time-out expiry should again be possible.

A similar argument can be made with abort; if the task is again on an entry queue it should be abortable. The requeue statement allows both views to be programmed; the default does not allow further time-outs or aborts, the addition of the 'with abort' clause enables the task to be removed from the second entry.

Important note: The real issue (in deciding whether to use 'with abort' or not) is whether the protected object (or client task) having requeued the task expects it to be there when the guard/barrier is next open. If the correct behaviour of the object requires the task's presence, then 'with abort' should *not* be used.

The combination of time-outs and requeue can give rise to somewhat unanticipated behaviour. For example, a call of

```
select
  T.E1;
or
  delay 10.0;
end select;
```

on a task that does a requeue

```
select -- in task T
  accept E1 do
    -- lots of computing taking 2 seconds
    requeue E2 with abort;
  end E1;
or
  ...
end select;
```

will fail if E1 is not accepted in 10.0 seconds, will not time out between 10.0 and 12.0 seconds (if the call is accepted just before 10.0 seconds has elapsed) but will time out after 12.0 seconds. Hence, the selective entry call may take the delay alternative some time after the value given in the delay alternative.

Now consider the following program fragment:

```
task Server is
  entry Service(Param: Parameter; Res: out Result);
private
  entry Service_Done(Param : Parameter;
                     Res : out Result);
end Server;

task body Server is
  P : Parameter;
begin
  ...
  accept Service(Param : Parameter; Res : out Result) do
    P := Param;
    requeue Service_Done with abort;
  end Service;
  -- perform required service
  -- which potentially may take a long time
  select
    accept Service_Done(Param : Parameter;
                        Res : out Result) do
      Res := ...;
    end Service_Done;
  else
    -- client has not waited
    -- throw away the result
  end select;
  ...
end Server;
```

Important note: Requeuing a request inside a rendezvous 'with abort' allows a client effectively to time-out on the results being returned from the rendezvous, rather than on the start of the rendezvous.

This section concludes with a further example. In a real-time system it may be necessary to restrict the number of interrupts that can be raised from any particular source. An excessive amount of interrupt handling may undermine the schedulability of some application task. One way to do this is to use a protected object as the interrupt handler but to turn on interrupts only when a calling task (the second-level interrupt handler) indicates that it is acceptable to do so. This calling task must therefore turn on interrupts and then be suspended until an interrupt actually occurs. The use of requeue makes this straightforward:

```
protected Interrupt_Interface is
  pragma Interrupt_Priority(Interrupt_Level);
  entry Wait_For_Interrupt;
private
  procedure Interrupt_Handler;
      -- mapped onto interrupt source
  pragma Attach_Interrupt(Interrupt_Handler, ...);
  entry Private_Wait;
  Open : Boolean := False;
end Interrupt_Interface;

protected body Interrupt_Interface is
  procedure Interrupt_Handler is
  begin
    Open := True;
  end Interrupt_Handler;

  entry Wait_For_Interrupt when True is
  begin
    -- turn on interrupts
    requeue Private_Wait;
  end Wait_For_Interrupt;

  entry Private_Wait when Open is
  begin
    -- handle interrupt
    Open := False;
    -- turn off interrupts
  end Private_Wait;
end Interrupt_Interface;
```

A similar structure could be used to record the time at which an entry call on a protected object is made:

```
entry Call(...) when True is
begin
```

```
  Time := Clock;
  requeue Private_Call;
end Call;
```

Note that these examples use an entry with a ‘when true’ barrier. All protected entries must have a barrier, but when requeue is used it may be necessary to have this null barrier.

8.3 Requeuing to other entities

Although requeuing to the same entity represents the normal use of requeue, there are situations in which the full generality of this language feature is useful.

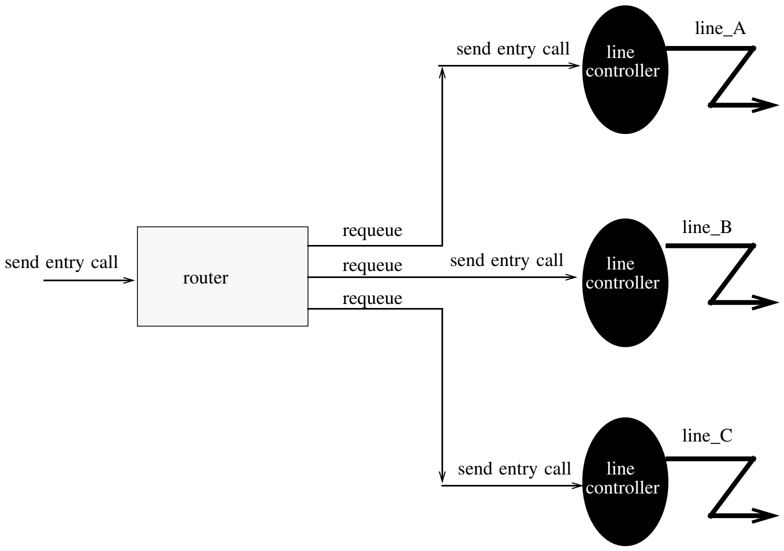


Fig. 8.1: A network router

Consider the situation in which resources are controlled by a hierarchy of objects. For example, a network router might have a choice of three communication lines on which to forward messages: `Line_A` is the preferred route, but if it becomes overloaded `Line_B` can be used; if this also becomes overloaded `Line_C` can be used. Each line is controlled by a server task; it is an active entity as it has housekeeping operations to perform. A protected unit acts as an interface to the router; it decides which of the three channels should be used and then uses requeue to pass the request to the appropriate server. The structure of the solution is illustrated in Figure 8.1, and the program fragment is given below:

```
type Line_Id is (Line_A, Line_B, Line_C);
type Line_Status is array (Line_Id) of Boolean;

task type Line_Controller(Id : Line_Id) is
  entry Request(...);
end Line_Controller;

protected Router is
  entry Send(...);
  procedure Overloaded(Line : Line_Id);
  procedure Clear(Line : Line_Id);
private
  Ok : Line_Status := (others => True);
end Router;

La : Line_Controller(Line_A);
Lb : Line_Controller(Line_B);
Lc : Line_Controller(Line_C);

task body Line_Controller is
...
begin
  loop
    select
      accept Request(...) do
        -- service request
      end Request;
    or
      terminate;
    end select;
    -- housekeeping including possibly
    Router.Overloaded(Id);  -- or
    Router.Clear(Id);
  end loop;
end Line_Controller;

protected body Router is
  entry Send(...) when Ok(Line_A) or Ok(Line_B) or Ok(Line_C) is
  begin
    if Ok(Line_A) then
      requeue La.Request with abort;
    elsif Ok(Line_B) then
      requeue Lb.Request with abort;
    else
      requeue Lc.Request with abort;
    end if;
  end Send;

  procedure Overloaded(Line : Line_Id) is
  begin
    Ok(Line) := False;
  end Overloaded;
  procedure Clear(Line : Line_Id) is
  begin
    Ok(Line) := True;
  end Clear;
end Router;
```

Important note: When requeuing from one protected object to another it is important to understand that while the call is being evaluated a mutual exclusion lock is held on both protected objects. It is therefore a bounded error to execute an external requeue request back to the requesting object (it would inevitably lead to deadlock):

```
protected P is
  entry X;
  entry Y;
end P;

protected body P is
  entry X when ... is
  begin
    requeue Y;   -- valid internal requeue
    requeue P.Y; -- invalid external requeue
  end X;

  entry Y when ...
end P;
```

Moreover, the programmer cannot assume, when a requeue has taken place from one protected object to another, that any tasks released in the original object will execute before the entry in the destination object. Indeed, an implementation is most likely to behave as follows:

```
protected P is
  entry X;
  entry Y;
private
  Barrier_Down : Boolean := False;
end P;

protected Q is
  entry Z;
end Q;

protected body P is
  entry X when Barrier_Down is
  begin
    ...
  end X;
  entry Y when True is
  begin
    Barrier_Down := True;
    requeue Q.Z;
  end Y;
end P;
```

Assume that a task first calls `P.X` and is blocked. Next, another task calls `P.Y`. If the entry `Z` in `Q` is open, then `Z` will be executed before the original call of `X` is completed.

Finally in this section, we show another form of interaction to be aware of. This involves the requeuing from a task to a protected object declared within the task body:

```
task T is
  entry E;
  ...
end T;

task body T is
  protected P is
    entry E;
    procedure L;
  end P;
  protected body P is ...
begin
  loop
    select
      accept E do
        requeue P.E;
      end E;
    or
      ...
    or
      terminate;
    end select;
    ...
    exit when ...;
  end loop;
end T;
```

With this code a call on `T.E` is requeued on to a protected object where it may be blocked (to be freed again by a call of `L` from some other entry of `T`). Task `T` cannot terminate (with the terminate alternative) while there is a task blocked on the protected object as there must be a non-terminated task remaining that can call `T` (that is, the blocked task when it is next freed). However, if `T` terminates for some other reason, then `P` must go out of scope and hence the task on its entry queue will have `Program_Error` raised at its point of call of `T.E`. Of course, if `P` were declared outside `T`, then this would not happen.

8.4 Real-time solutions to the resource control problem

Consider again the resource control problem in the case where entry queues are ordered by priority. The top priority task will now be put back at the front of the queue and other tasks will not be able to proceed. To allow other tasks to still make progress in this situation requires two queues. Whenever resources are released, unsuccessful tasks are moved from one queue to the other. A family of two is used in the following code:

```
type Request_Range is range 1 .. Max;
type Family is range 1 .. 2;
type Bools is array(Family) of Boolean;

protected Resource_Controller is
  entry Allocate(R : out Resource; Amount : Request_Range);
  procedure Release(R : Resource; Amount : Request_Range);
private
  entry Assign(Family)(R : out Resource; Amount : Request_Range);
  Free : Request_Range := Request_Range'Last;
  New_Resources_Released : Bools := (others => False);
  Queue_At : Family := 1;
  ...
end Resource_Controller;

protected body Resource_Controller is
  entry Allocate(R : out Resource; Amount : Request_Range)
        when Free > 0 is
  begin
    if Amount <= Free then
      Free := Free - Amount;
      -- allocate resources
    else
      requeue Assign(Queue_At);
    end if;
  end Allocate;

  entry Assign(for F in Family)
              (R : out Resource; Amount : Request_Range)
        when New_Resources_Released(F) is
  begin
    -- for this code to execute F
    -- must not equal Queue_At
    if Assign(F)'Count = 0 then
      New_Resources_Released(F) := False;
    end if;
    if Amount <= Free then
      Free := Free - Amount;
      -- allocate resources
    else
      requeue Assign(Queue_At);
    end if;
  end Assign;
  procedure Release(R : Resource; Amount : Request_Range) is
  begin
    Free := Free + Amount;
    -- free resources
    if Assign(Queue_At)'Count > 0 then
      New_Resources_Released(Queue_At) := True;
      if Queue_At = 1 then
        Queue_At := 2;
      else
        Queue_At := 1;
      end if;
    end if;
  end Release;
end Resource_Controller;
```

An alternative real-time model is that the highest priority process receives the resource as soon as possible and, consequently, is blocked for a bounded minimum time. Here, resources are not given to lower priority waiting tasks, even though there are enough resources to satisfy their requests. The solution to this problem is not immediately obvious. Consider first a flawed attempt; it assumes that the entry queues are ordered according to the priority of the queued tasks:

```
-- Flawed priority-driven resource allocation algorithm
type Request_Range is range 1 .. Max;

protected Resource_Controller is
  entry Allocate(R : out Resource; Amount : Request_Range);
  procedure Release(R : Resource; Amount : Request_Range);
private
  entry Assign(R : out Resource; Amount : Request_Range);
  Free : Request_Range := Request_Range'Last;
  New_Resources_Released : Boolean := False;
end Resource_Controller;

protected body Resource_Controller is
 entry Allocate(R : out Resource; Amount : Request_Range)
               when Free > 0 is
 begin
   if Amount <= Free then
     Free := Free - Amount;
     -- allocate
   else
     New_Resources_Released := False;
     requeue Assign;
   end if;
 end Allocate;

 entry Assign(R : out Resource; Amount : Request_Range)
             when New_Resources_Released is
 begin
   if Amount <= Free then
     Free := Free - Amount;
     -- allocate
   else
     New_Resources_Released := False;
     requeue Assign;
   end if;
 end Assign;

 procedure Release(R : Resource; Amount : Request_Range) is
 begin
    Free := Free + Amount;
    -- free resources
    New_Resources_Released := True;
 end Release;
end Resource_Controller;
```

The problem with this 'solution' is that a low priority task could be given some free resources (when it calls `Allocate`) when a higher priority task is queued on `Assign` (because there are not enough resources to satisfy its request). This breaks our requirement that the highest priority task must be serviced first. An alternative approach would be to have only a single entry for `Allocate` and `Assign`. Consider the following algorithm:

```
type Request_Range is range 1 .. Max;

protected Resource_Controller is
  entry Allocate(R : out Resource; Amount : Request_Range);
  procedure Release(R : Resource; Amount : Request_Range);
private
  Free : Request_Range := Request_Range'Last;
  Blocked : Natural := 0;
  ...
end Resource_Controller;

protected body Resource_Controller is
  entry Allocate(R : out Resource; Amount : Request_Range)
        when Free > 0 and Blocked /= Allocate'Count is
  begin
    if Amount <= Free then
      Free := Free - Amount;
      -- allocate
    else
      Blocked := Allocate'Count + 1;
      requeue Allocate with abort;
    end if;
  end Allocate;

  procedure Release(R : Resource; Amount : Request_Range) is
  begin
    Free := Free + Amount;
    -- free resources
    Blocked := 0;
  end Release;
end Resource_Controller;
```

Here, a note is taken (in `Blocked`) by the controller of the number of tasks on the `Allocate` entry. This is then used as part of the barrier. When the number queued (given by `'Count`) changes (either because new tasks arrive or because currently queued tasks time-out or are aborted), the barrier becomes open and the controller checks again to see if it can service the highest priority task.

The above algorithm makes use of the `Count` attribute within the barrier of an entry. This is a powerful programming technique; and a number of examples of its use are given elsewhere in this book. However, it should be noted that such usage will usually lead to the barrier expression being evaluated twice. Once as the call arrives, and (if the barrier evaluates to false) again when it is queued (as `'Count` has now increased).

8.5 Entry families and server tasks

Earlier in this chapter a family of entries was used with a protected object. With server tasks, families are not as easy to use. This difficulty will be illustrated with a simple server that wishes to give priority to certain classes of user task:

```
type A_Priority is (High, Medium, Low);
task Server is
  entry Request(A_Priority)(...);
end Server;
```

a typical call would be

```
Server.Request(Low)(...);
```

Within the body of the task, a select statement is constructed so that priority is given to calls coming in on the `High` family entry:

```
task body Server is
  Empty : Boolean;
begin
  loop
    select
      accept Request(High)(...) do
        ...
      end Request;
    or
      when Request(High)'Count = 0 =>
        accept Request(Medium)(...) do
          ...
        end Request;
    or
      when Request(High)'Count   = 0  and
           Request(Medium)'Count = 0  =>
        accept Request(Low)(...) do
```

```
          ...
        end Request;
    or
      terminate;
    end select;
  end loop;
end Server;
```

One criticism of the above code is that for a large family the necessary guards become somewhat excessive (for a 1,000-member family, the guard on the lowest value would need to contain 999 boolean evaluations!).

Important note: If the Real-Time Systems Annex is being supported, then it is possible to define a queuing discipline that will use the textual ordering of the select statement to give priority to those accept statements that appear first.

With the Real-Time Annex, the above code is simplified to the following:

```
select  -- with priority queuing and all callers having
        -- the same priority
  accept Request(High)(...) do
    ...
  end Request;
or
  ...
or
  accept Request(Low)(...) do
    ...
  end Request;
or
  terminate;
end select;
```

However, each member of the family still has to be written out individually. Ideally, a syntactical form equivalent to that available to protected objects could be used:

```
select (for P in A_Priority)  -- Not Valid Ada
  accept Request(P)(...) do
    ...
  end Request;
or
  terminate;
end select;
```

Unfortunately, no such syntax is available. An alternative structure involves looping through all the possible values of the family:

```
task body Server is
  ...
begin
```

```
  loop
    for P in A_Priority loop
      select
        accept Request(P)(...) do
            ...
        end Request;
        exit;
      else
        null;
      end select;
    end loop;
  end loop;
end Server;
```

The task loops through the values of `A_Priority` in the order required until it finds an outstanding entry call. Having accepted this call it exits from the inner loop and tries to find a high priority entry again. If no entries are outstanding, then it returns via the outer loop to try again.

This last point immediately raises a question about this 'solution', for it uses a busy-wait loop, which is polling for requests. A reliable and efficient algorithm must separate the acceptance of outstanding entry calls from waiting for the first new entry call to arrive. If there are no outstanding calls, then the server task must be suspended on a select statement that will accept the first incoming call, whatever its priority:

```
task body Server is
  Empty : Boolean;   -- no outstanding calls.
begin
  loop
    loop
      Empty := True;
      for P in A_Priority loop
        select
          accept Request(P)(...) do
            ...
          end Request;
          Empty := False;
          exit;
        else
          null;
        end select;
      end loop;
      exit when Empty;
      -- will only exit when no requests
      -- have been found.
    end loop;
    select
      accept Request(High)(...) do
        ...
      end Request;
    or
```

```
      accept Request(Medium)(...) do
        ...
      end Request;
    or
      accept Request(Low)(...) do
        ...
      end Request;
    or
      terminate;
    end select;
  end loop;
end Server;
```

The second half of this task body, which will accept any incoming call (or terminate), must, of necessity, name each family entry explicitly and will be lengthy for large families as a consequence. If the family size is such that this length is a problem, then a structure using requeue is possible:

```
task Server is
  entry Request(Pri : A_Priority; ...);
private
  entry Waiting(A_Priority)(Pri : A_Priority; ...);
end Server;
task body Server is
  Empty : Boolean;
  ...
begin
  loop
    select
      accept Request(Pri : A_Priority; ...) do
        requeue Waiting(Pri);
      end Request;
    or
      terminate;
    end select;
    loop
      Empty := True;
      for P in A_Priority loop
        select
          accept Request(Pri : A_Priority; ...)  do
            requeue Waiting(Pri);
          end Request;
          Empty := False;
          exit;
        or
          when Request'Count = 0 =>
            accept Waiting(P)(Pri : A_Priority; ...) do
              ...
            end Waiting;
          Empty := False;
          exit;
        else
```

```
          null;
        end select;
      end loop;
      exit when Empty;
    end loop;
  end loop;
end Server;
```

Note that the parameter lists for `Waiting` and `Request` have to be identical so that a requeue with parameters between them can take place.

8.6 Extended example

This chapter concludes with a further example that illustrates the power of requeue as a concurrent programming primitive. The problem is one of simulating/representing the behaviour of travellers on a circular railway (metro). There are N stations on the circular track and one train (with a small finite capacity). Travellers arrive at one station and are transported to their requested destination. A full trip back to the original station is allowed.

The design of the concurrent representation of this situation consists of active entities for each traveller and an active entity for the train itself. Stations represent resources. Travellers are blocked at the station until the train arrives. The program will then requeue the passengers (if there are seats on the train) to the destination station of choice. When the train subsequently arrives at that station, the travellers disembark and continue on their way (which in the following program means taking another trip). The system is illustrated in Figure 8.2.

Each station will be represented by a protected object. This object will have a number of procedures and entries:

(1) `Arrive` – entry called by passengers when they arrive at the station (they are blocked until the train arrives).
(2) `Stopping` – procedure called by the train when it stops at the station.
(3) `Alight` – entry requeued by passengers wishing to alight at that station.
(4) `Boarding` – procedure called by the train to allow passengers onto the train.
(5) `Closedoors` – procedure called by the train to indicate that it is about to leave the station.

The code for the station definition is thus

```
type Station_Address is range 1..N;
type Passengers is range 0..Max;
Capacity : constant Positive := 10;
           -- capacity of the small train
```

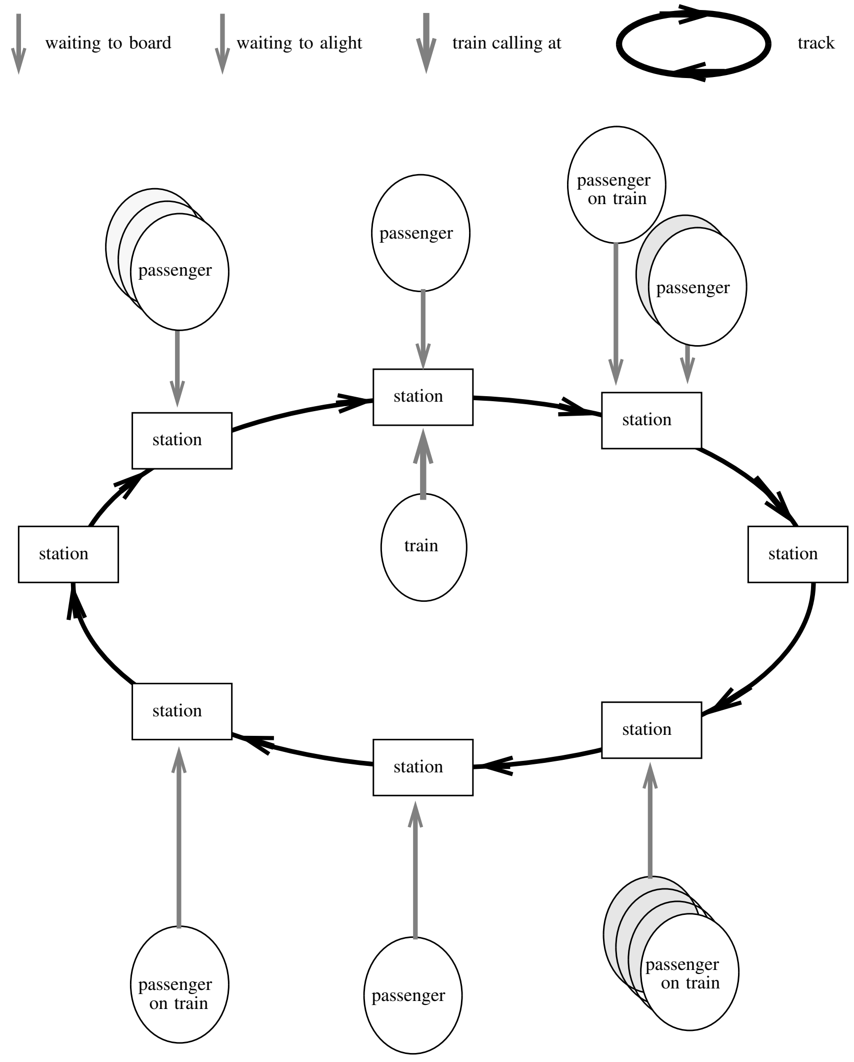


Fig. 8.2: The requeue metro

```
protected type Station is
  entry Arrive(Destination : Station_Address);
  procedure Stopping(P : Passengers);
  entry Alight(Destination : Station_Address);
  procedure Boarding;
  procedure Closedoors(P : out Passengers);
private
  On_Train : Passengers;
  Trainboarding : Boolean := False;
  Trainstopped : Boolean := False;
end Station;
```

The set of stations is represented as an array:

```
Stations : array(Station_Address) of Station;
```

The code for the body of `Station` can now be given:

```
protected body Station is
  entry Arrive(Destination : Station_Address)
    when Trainboarding and then On_Train < Capacity is
  begin
    On_Train := On_Train + 1;
    requeue Stations(Destination).Alight;
  end Arrive;

  procedure Stopping(P : Passengers) is
  begin
    On_Train := P;
    Trainstopped := True;
  end Stopping;

  entry Alight(Destination : Station_Address)
        when Trainstopped is
  begin
    On_Train := On_Train - 1;
    -- passenger has arrived
  end Alight;

  procedure Boarding is
  begin
    Trainstopped := False;
    Trainboarding := True;
  end Boarding;

  procedure Closedoors(P : out Passengers) is
  begin
    P := On_Train;
    Trainboarding := False;
  end Closedoors;
end Station;
```

Passengers just call on the stations and take trips:

```
task type Clients;

Travellers : array(Passengers) of Clients;

task body Clients is
  Home, Away : Station_Address;
begin
  -- choose home
  loop
```

```
    -- choose away
    Stations(Home).Arrive(Away);
    Home := Away;
  end loop;
end Clients;
```

The train must repeat a fixed set of operations for each station:

```
task Train;

task body Train is
  Volume: Passengers := 0;
  Travel_Times: array(Station_Address) of Duration := ...;
begin
  loop -- forever or until re-nationalised
    for S in Station_Address loop
      Stations(S).Stopping(Volume);
      Stations(S).Boarding;
      Stations(S).Closedoors(Volume);
      delay Travel_Times(S);
    end loop;
  end loop;
end Train;
```

Note that the train interacts with each station in three distinct phases. First the train stops at the station with a current passenger load of `Volume`; this number is copied and held internally, to the station, in `On_Train`. People are released at this stage and thus `On_Train` is decreased. Only when all such alighting passengers have left the train (and the station) can the call of `Boarding` be accepted. This second action of the train closes the barrier for passengers getting off, and opens the barrier for passengers getting on (up to the capacity of the train). Again, only when all boarding passengers are on the train can the train task call `Closedoors`; this returns the new value of `Volume` and closes the barrier for boarding passengers.

The key to this solution is that the train task cannot re-enter the protected object until all passenger tasks released by the lowering of barriers have themselves left the protected object. It would be possible for the station object to export only a single procedure for the train, and to requeue the train call internally through the three phases. This is left as an exercise for the reader, as is the addition of extra trains.

8.7 Task states

The requeue feature has only introduced a minor change to the state transition diagram given in previous chapters. It is now possible to move from executable to waiting on a protected (or task) entry call by a requeue operation.

8.8 Summary

The ability to program various forms of synchronisation is a fundamental requirement in any concurrent programming language. Ada's basic mechanism for programming inter-task synchronisation is the *guard*. If it is necessary for a task to prevent itself from proceeding, until some required system state is achieved, then it must call an entry in either another task or a protected object. In this called entity a guard (or barrier) is evaluated, and if found to be false the calling task is suspended. The use of guards supports avoidance synchronisation; progress is prevented if conditions are not favourable.

Unfortunately, the simple use of guards does not deal with all necessary synchronisations. There are situations in which some level of interaction is needed before the decision can be made as to whether a task should be suspended. A simple example of this is the need to evaluate incoming parameters before the suspension decision can be made. Ada has extended the expressive power and usability of avoidance synchronisation by introducing the requeue facility. This allows a task to be returned to an entry so that it must again pass though a guard before proceeding.

The requeue facility is a general one. It allows a task to be requeued back to the same entry as before, to a different entry in the same server task or protected object, or to a different entity altogether. It is possible to requeue from a protected object to a task and vice versa. Guards plus requeue represents a powerful, but structured, means of programming all forms of task synchronisation.

9

Exceptions, abort and asynchronous transfer of control

This chapter considers three related topics: exceptions (and their handling), the abort statement and asynchronous transfer of control. They are related in that they all divert a task away from its current execution path and force it to execute some other section of code.

The main motivation for these facilities comes from the use of concurrent programming techniques in real-time systems. Here there are often stringent timing and reliability requirements which necessitate that the language provides facilities additional to those that support concurrent execution and communication and synchronisation.

9.1 Exceptions

In Chapters 4–8, exceptions were discussed in the context of task creation, task communication and synchronisation. This section summarises, in the one place, the complete interaction between tasking and exceptions.

Unhandled exceptions during task execution

An exception that is raised and not handled in an executing task will cause that task to become completed. This will *not* have a direct effect on the rest of the program (that is, the exception is *not* propagated to any other task). Of course, a task which tries to communicate with a completed task will get `Tasking_Error` raised.

Ada 2005 change: The silent completion of a task as a result of an unhandled exception can cause problems in high integrity systems. For this reason, Ada 2005 has added a general mechanism that allows the program to be notified when a task terminates (normally or as a result of an exception or an abort). This is discussed further in Section 15.7.

Exceptions during elaboration of a declarative block

An exception may be raised during elaboration of a declarative part. For example, assigning an initial value to an object which falls outside the permissible range of values will result in `Constraint_Error` being raised. Any tasks which would have been created during the elaboration are not activated. See Section 4.2 for further details and an example.

Exceptions during activation of a task

An exception may be raised during the activation of a task. For example, assigning an initial value (outside the permissible range of values) to an object declared in the task body's declarative part would result in `Constraint_Error` being raised. The task is unable to handle this exception and therefore becomes completed. This results in the exception `Tasking_Error` being raised in the parent task after all its child tasks have been activated. Even if more than one of its children become completed during activation, the exception is only raised once. See Subsection 4.2.1 for further details.

Exceptions in task finalisation

It is a bounded error for an exception to be raised during any finalisation routine. If one does occur during the finalisation of a task, or of a master block containing tasks, then the most likely consequence is that the exception will be lost and the finalisation of other objects continued.

Exceptions and the rendezvous

The consequences of exceptions being raised during the rendezvous have been discussed in Section 5.8. The model is essentially that if an exception is not handled within the rendezvous, it is propagated to both the server and the client task.

The following exceptions may also be raised when a task attempts a rendezvous:

- `Tasking_Error` is raised when a client task attempts to rendezvous with a server task that has already completed, terminated or become abnormal, or that becomes completed, terminated or abnormal before the rendezvous starts. `Tasking_Error` is also raised in the client if the server becomes abnormal during the rendezvous.
- `Constraint_Error` may be raised when the client or the server attempts to rendezvous using an entry family and the index used falls outside the range of the family.

- `Program_Error` is raised when the server task executes a select statement and all arms of the select statement are closed (because all are guarded and all guards evaluate to false) and there is no 'else' part. See Section 6.7.

Exceptions and protected objects

When a task interacts with a protected object, the following conditions will raise the `Program_Error` exception:

- the evaluation of a barrier that results in an exception condition being detected;
- a protected action that issues a potentially blocking operation;
- a task is queued on a protected entry and the protected object is finalised.

Important note: If a barrier gives rise to an exception, then all tasks waiting on all entries of that protected object get `Program_Error` raised.

Any exception raised whilst executing a protected subprogram or entry and not handled by the protected subprogram or entry is propagated to the task that issued the protected call (as if the call were a normal subprogram call).†

As with task entry families, a client task may suffer a `Constraint_Error` if a family index is out of range.

Exceptions and abort-deferred operations

Certain operations are defined by the language to be abort-deferred. These operations have restrictions placed on them which, if violated, may result in the exception `Program_Error` being raised. See Subsection 9.2.1.

Exceptions and the asynchronous select statement

Although the asynchronous select statement introduces no new exception scopes, the interactions between exceptions and the tasks involved can be quite subtle. This issue is discussed in full later in this chapter (Subsection 9.3.2).

Exceptions in interrupt handlers

If an exception is propagated from an interrupt handler that is invoked by an interrupt, the exception has no effect.

† In common with all protected action calls, the barriers are re-evaluated before the exception propagates out beyond the protected object (see Section 7.4).

9.2 The abort statement

Raising an exception in an errant task is an appropriate response to an error condition when:

- a task itself detects the error condition and can explicitly raise (and handle) the exception; or
- the environment in which the task executes detects an error as a result of an action being performed by the task (e.g. array bounds violation) – in this case, one of the predefined standard exceptions can be raised.

However, it may be the case that the error condition has been detected by another task. In this situation, it is not appropriate for an exception to be raised in the errant task, as the task may not be in a position to handle the error. Indeed, the task may long since have executed the code that caused the original error to occur. To help program these situations, Ada provides two facilities: the abort statement and the asynchronous select statement. The abort statement is considered in this section; the asynchronous select statement is discussed in Subsection 9.3.2.

The abort statement is intended for use in response to those error conditions where recovery by the errant task is deemed to be impossible.

Important note: Tasks that are aborted are said to become *abnormal*, and are thus prevented from interacting with any other task.

Ideally, an abnormal task will stop executing immediately. However, some implementations may not be able to facilitate immediate shut down, and hence all the ARM requires is that the task terminate before it next interacts with other tasks. Note that the Real-Time Systems Annex does require 'immediate' to be just that (see Section 17.5) on a single processor system.

The syntax of the abort statement is simply

```
abort_statement ::= abort task_name {,task_name};
```

Any task may abort any other named task by executing this statement (tasks named in the same statement need not even be of the same type):

```
abort Operator;
abort Philosopher(1), Philosopher(3);
```

Important note: If a task is the target of this statement, then it becomes abnormal; any non-completed tasks that depend upon an aborted task also become abnormal. Once all named tasks are marked as abnormal, then the abort statement is complete, and the task executing the abort can continue. It does not wait until the named tasks have actually terminated.

After a task has been marked as abnormal, execution of its body is aborted. This means that the execution of every construct in the task body is aborted, unless it is involved in the execution of an *abort-deferred operation*. The execution of an abort-deferred operation is allowed to complete before it is aborted. The same rules for aborting a task body also apply to aborting a sequence of statements in the asynchronous select statement discussed later, in Section 9.3.

If a construct, which has been aborted, is blocked outside an abort-deferred operation (other than at an entry call), the construct becomes abnormal and is immediately completed. Other constructs must complete no later than the next *abort completion point* (if any) that occurs outside an abort-deferred operation.

Definition: An abort completion point occurs at:

- the end of activation of a task;
- the point where the execution initiates the activation of another task;
- the start or end of an entry call, accept statement, delay statement or abort statement;
- the start of the execution of a select statement, or of the sequence of statements of an exception handler.

Warning: A consequence of this rule is that an abnormal task which does not reach any of the above points need not be terminated! It could be abnormal but still loop round updating shared variables, calling protected procedures and using processor cycles. There is no way of ensuring that such a task is forced to complete, although on a traditional uniprocessor system it would be normal to terminate the task at once. As indicated earlier, the Real-Time Systems Annex does require an implementation to document any aspects of implementation which might delay the completion of an aborted construct.

9.2.1 Abort-deferred operations

The issue of how immediate an abort action should be is a complex one. It can be argued that when aborted, if a task is updating some shared data structure (perhaps inside a protected object), it should be shielded from the imposed abort. But if it is shielded, then the whole point of using abort is diminished. Ada favours the view that certain actions must be protected in order that the integrity of the remaining tasks and their data be assured.

Important note: The following operations are defined to be abort-deferred:

- a protected action;
- waiting for an entry call to complete;
- waiting for termination of dependent tasks;
- the execution of an 'initialize' procedure, a 'finalize' procedure, or an assignment operation of an object with a controlled part.

Note that there are some further restrictions on controlled objects if their controlling procedures are to be abort-deferred (see ARM, 9.8). For example, it is a bounded error for such a procedure to create a task whose master is contained within the procedure.

9.2.2 Use of the abort statement

The use of the abort statement is clearly an extreme response to an error condition and one which carries the following warning:

Warning: An abort statement should be used only in situations requiring unconditional termination (ARM 9.8).

It was available in Ada 83 under the assumption that it was provided for emergency use only. Its overuse could severely hinder program understanding and validation. Nevertheless, the fact that a task can abort any other task introduces an interesting circular argument to this rationale. Hoare (1979) has been particularly critical of this language feature, remarking

The existence of this statement causes intolerable overheads in the implementation of every other feature of tasking. Its 'successful' use depends on a valid process aborting a wild one before the wild one aborts a valid process – or does any other serious damage. The probability of this is negligible. If processes can go wild, we are much safer without aborts.

Nevertheless, the ability to abort a task is considered to be a valid requirement for real-time systems. The current version of the language makes every effort to ensure that the facility can be used as safely as possible, given its inherently dangerous nature.

9.3 Asynchronous transfer of control

Definition: An asynchronous transfer of control (ATC) is where the flow of execution in one task is changed (in a controlled manner) by the action of another task.

The presence of an asynchronous transfer of control facility within the Ada language, like the abort statement, is controversial, as it complicates the language's semantics and increases the complexity of the run-time support system. This section thus first considers the application requirements which justify the inclusion of such a facility. Following this, the asynchronous select statement is described and then examples of its use are given.

9.3.1 The user need for ATC

The fundamental requirement for an asynchronous transfer of control facility is to enable a task to respond *quickly* to an asynchronous event. The emphasis here is on a quick response; clearly a task can always respond to an event by simply polling or waiting for that event. The notification of the event could be mapped onto a task entry call or a protected object subprogram or entry call. The handling task, when it is ready to receive the event, simply issues the corresponding accept statement or protected object entry/subprogram call.

Unfortunately there are occasions when polling for events or waiting for the event to occur is inadequate. These include the following:

- Error recovery

  A fault may have occurred that requires a task to alter its flow of control as a consequence. For example, a hardware fault may mean that the task will never finish its planned execution because the preconditions under which it started no longer hold; the task may never reach its polling point. Also, a timing fault might have occurred, which means that the task will no longer meet the deadline for the delivery of its service. In both these situations, the task must be informed that an error has been detected and the task must undertake some error recovery as quickly as possible.
- Mode changes

  A real-time system often has several modes of operation. For example, a fly-by-wire civil aircraft may have a take-off mode, a cruising mode and a landing mode. On many occasions, changes between modes can be carefully managed and will occur at well-defined points in the system's execution; as in a normal flight plan for a civil aircraft. Unfortunately, in some application areas, mode changes are expected but cannot be planned. For example, a fault may lead to an aircraft abandoning its take-off and entering into an emergency mode of operation; an accident in a manufacturing process may require an immediate mode change to ensure an orderly shutdown of the plant. In these situations, tasks must be quickly and safely informed that the mode in which they are operating has changed, and that they now need to undertake a different set of actions.

- Scheduling using partial/imprecise computations

  There are many algorithms where the accuracy of the results depends on how much time can be allocated to their calculation. For example, numerical computations, statistical estimations and heuristic searches may all produce an initial estimation of the required result, and then refine that result to a greater accuracy. At run-time, a certain amount of time can be allocated to an algorithm, and then, when that time has been used, the tasks must be interrupted to stop further refinement of the result.
- User interrupts

  In a general interactive computing environment, users often wish to stop the current processing because they have detected an error condition and wish to start again.

In all the above situations, it is possible to abort the task and recreate it, passing information as to why the task was aborted. Indeed, it can be argued that this facility is adequate and can be implemented efficiently enough for its use in time critical situations. Furthermore, many implementations of ATC will use a 'two-thread model', where the code to be interrupted is contained in one thread of control and the code waiting for the interrupt notification is in another thread. On receipt of the notification, the latter thread aborts the former and recreates it. However, requiring the programmer to code these interactions directly is error-prone, and it has been decided that special language support is preferable. The programmer can use ATC to define a sequence of statements that can be aborted if a specified triggering event occurs.

9.3.2 The asynchronous select statement

The select statement was introduced in Chapter 6 as having four forms: a selective accept (to support the server side of the rendezvous), a timed entry call (to either a task or a protected entry), a conditional entry call (also to a task or a protected entry), and an asynchronous select. The first three forms have been discussed in Chapters 6 and 7. Here attention is focused on the last form, which has the following syntax definition:

```
asynchronous_select ::=
  select
    triggering_alternative
  then abort
    abortable_part
  end select;
```

```
triggering_alternative ::=
  triggering_statement [sequence_of_statements]
triggering_statement ::=
  entry_call_statement | delay_statement

abortable_part ::= sequence_of_statements
```

Important note: There is a restriction on the sequence of statements that can appear in the abortable part. It must not contain an accept statement.

The reason for this is to keep the implementation as simple as possible.

The execution of the asynchronous select begins with the issuing of the triggering entry call or the issuing of the triggering delay. If the triggering statement is an entry call, the parameters are evaluated as normal and the call issued. If the call is queued (or requeued with abort), then the sequence of statements in the abortable part is executed. If the abortable part completes before the completion of the entry call, an attempt is made to cancel the entry call and, if successful, the execution of the asynchronous select statement is finished.

Similarly, if the triggering statement is a delay statement, the delay time is calculated, and if it has not passed, the abortable part is executed. If this finishes before the delay time expires, the delay is cancelled and the execution of the asynchronous select statement is finished.

If the cancellation of the triggering event fails, because the protected action or rendezvous has started, or has been requeued (without abort), then the asynchronous select statement waits for the triggering statement to complete before executing the optional sequence of statements following the triggering statement.

If the triggering statement completes (other than due to cancellation, that is, the delay time expires or the rendezvous or protected action starts and finishes) before the execution of the abortable part completes, the abortable part is aborted and any finalisation code is executed. When these activities have finished, the optional sequence of statements following the triggering statement is then executed.

Clearly, it is possible for the triggering event to occur even before the abortable part has started its execution. In this case the abortable part is not executed and therefore not aborted.

Consider the following example:

```
task Server is
  entry Atc_Event;
end Server;

task To_Be_Interrupted;

task body Server is
begin
```

```
  ...
  accept Atc_Event do
    Seq2;
  end Atc_Event;
  ...
end Server;

task To_Be_Interrupted is
begin
  ...
  select  -- ATC statement
    Server.Atc_Event;
    Seq3;
  then abort
    Seq1;
  end select;
  Seq4;
  ...
end To_Be_Interrupted;
```

When the above ATC statement is executed, the statements which are executed will depend on the order of events that occur:

```
if the rendezvous is available immediately then
   Server.Atc_Event is issued
   Seq2 is executed
   Seq3 is executed
   Seq4 is executed
elsif no rendezvous starts before Seq1 finishes then
   Server.Atc_Event is issued
   Seq1 is executed
   Server.Atc_Event is cancelled
   Seq4 is executed
elsif the rendezvous finishes before Seq1 finishes then
   Server.Atc_Event is issued
   partial execution of Seq1 occurs concurrently with Seq2
   Seq1 is aborted and finalised
   Seq3 is executed
   Seq4 is executed
else (the rendezvous finishes after Seq1 finishes)
   Server.Atc_Event is issued
   Seq1 is executed concurrently with partial execution of Seq2
   Server.Atc_Event cancellation is attempted
   execution of Seq2 completes
   Seq3 is executed
   Seq4 is executed
end if
```

Note that there is a race condition between `Seq1` finishing and the rendezvous finishing. The situation could occur where `Seq1` does finish but is nevertheless aborted.

If Seq1 contains an abort-deferred operation (such as a call to a protected procedure), then its cancellation will not occur until the operation is completed.

The above discussion has concentrated on the concurrent behaviour of Seq1 and the triggering rendezvous. Indeed, on a multi-processor implementation it could be the case that Seq1 and Seq2 are executing in parallel. However, on a single processor system, the triggering event will only ever occur if the action that causes it has a higher priority than Seq1. The normal behaviour will thus be the preemption of Seq1 by Seq2. When Seq2 (the triggering rendezvous) completes, Seq1 will be aborted before it can execute again. And hence the ATC is 'immediate' (unless an abort-deferred operation is in progress).

Exceptions and ATC

With the asynchronous select statement, two activities are potentially concurrent: the abortable part may execute concurrently with the triggering action (when the action is an entry call). In either one of these activities, exceptions may be raised and unhandled. Therefore, at first sight it may appear that potentially two exceptions can be propagated simultaneously from the select statement. However, this is not the case; one of the exceptions is deemed to be lost and hence only one exception is propagated. Consider the following example:

```
E1, E2 : exception;
task Server is
  entry Atc_Event;
end Server;

task To_Be_Interrupted;

task body Server is
begin
  ...
  accept Atc_Event do
    Seq2; -- including raise E2;
  end Atc_Event;
  ...
end Server;

task To_Be_Interrupted is
begin
  ...
  select  -- ATC statement
    Server.Atc_Event;
    Seq3;
  then abort
    Seq1; -- including raise E1;
  end select;
exception
```

```
  when E1 =>
    Seq4;
  when E2 =>
    Seq5;
  when others =>
    Seq6;
end To_Be_Interrupted;
```

When the above ATC statement is executed, the result will be

```
if the rendezvous is available immediately then
   Server.Atc_Event is issued
   Seq2 is executed and raises E2
   E2 is propagated from the select statement
         (Seq3 is not executed)
   Seq5 is executed
elsif no rendezvous starts before Seq1 finishes then
   Server.Atc_Event is issued
   Seq1 is executed and raises E1
   Server.Atc_Event is cancelled
   E1 is propagated from the select statement
   Seq4 is executed
elsif the rendezvous finishes before Seq1 finishes then
   Server.Atc_Event is issued
   partial execution of Seq1 occurs concurrently with Seq2
   Seq1 is aborted and finalised
   E2 is propagated from the select statement
   E1 is lost if it was ever raised
   Seq5 is executed
elsif the rendezvous finishes after Seq1 finishes
   Server.Atc_Event is issued
   Seq1 executes concurrently with the partial execution of Seq2
   Server.Atc_Event is cancelled
   partial execution of Seq2 occurs
   E2 is propagated from the select statement
   E1 is lost
   Seq5 is executed
else the called task terminates before the rendezvous starts
            or before it is cancelled
   Server.Atc_Event is issued
   Seq1 is aborted (or never starts)
   Tasking_Error is propagated from the select statement
   E1 is lost if it was ever raised
   Seq6 is executed
end if
```

Generally, if the triggering event of an asynchronous select statement is taken, then any unhandled exception raised in the abortable part is lost. Of course, any unhandled exception raised by `Seq3` will always be propagated.

9.3.3 Examples of asynchronous transfer of control

The examples presented in this subsection are derived from the application requirements mentioned earlier in Subsection 9.3.1.

Error recovery

Before recovery can be initiated to handle a fault, the fault must cause a detectable error to occur in the system. Once the error has been detected, some form of damage assessment and damage confinement must be undertaken before error recovery can begin. The details of these activities are application dependent; however, typically, a set of tasks might need to be informed of the fault. The following code fragment illustrates the approach:

```
with Broadcasts; -- see Chapter 11
...
type Error_Id is (Err1, Err2, Err3);
  -- some appropriate identifier

package Error_Notification is new Broadcasts(Error_Id);
Error_Occurred : Error_Notification.Broadcast;
    -- a protected object

task Error_Monitor;

-- all tasks interested in the error have the following structure
task type Interested_Party;

task body Interested_Party is
  Reason : Error_Id;
begin
  loop
    ...
    select
      Error_Occurred.Receive(Reason);
        -- a protected entry call
      case Reason is
        when Err1 =>
          -- appropriate recovery action
        when Err2 =>
          -- appropriate recovery action
        when Err3 =>
          -- appropriate recovery action
      end case;
    then abort
      loop
        -- normal operation
      end loop;
    end select;
  end loop;
end Interested_Party;
```

```
task body Error_Monitor is
  Error : Error_Id;
begin
  ...
  -- when error detected
  Error_Occurred.Send(Error);
  -- a protected procedure call
  ...
end Error_Monitor;
```

The above code fragment makes use of a generic package, introduced in Chapter 11, which provides a general purpose broadcast facility (via a protected object). The `Error_Monitoring` task detects the error condition and sends the broadcast to all those tasks that are listening. Those tasks executing within the select statement will receive the error notification, but those outside will not (in this case). The use of a different communication abstraction for error notification (such as a persistent signal) would ensure that all interested tasks received notification eventually.

The above example illustrates an interesting compromise between demanding that a task polls for the event or explicitly waits for the event, and forcing the task to respond immediately to the event. In Ada, the task must indicate explicitly that it is prepared to have its flow of control changed by executing the *select then abort* statement; however, once it has done this, it is free to continue executing. Any race conditions that result between the triggering event being signalled and the task issuing the *select then abort* statement must be catered for by the programmer. Furthermore, any section of code that should not be aborted must be encapsulated in a protected object, thus making it an abort-deferred operation.

Deadline overrun detection

If a task has a deadline associated with part of its execution, then the *select then abort* statement can be used to detect a deadline overrun. For example, the following task must undertake some action before a certain time:

```
with Ada.Real_Time; use Ada.Real_Time;
task Critical;

task body Critical is
  Deadline : Real_Time.Time := ...; -- some appropriate value
begin
  ...
  select
    delay until Deadline;
    -- recovery action
  then abort
    -- time-critical section of code
  end select;
  ...
end Critical;
```

Alternatively, the task may wish the action to be performed within a certain period of time:

```
with Ada.Calendar; use Ada.Calendar;
task Critical;

task body Critical is
  Within_Deadline : Duration := ...; -- some appropriate value
begin
  ...
  select
    delay Within_Deadline;
    -- recovery action
  then abort
    -- time-critical section of code
  end select;
  ...
end Critical;
```

Chapter 14 will show how Ada 2005 has generalised this approach to cater for Earliest-Deadline-First (EDF) scheduling.

Mode changes

Consider a periodic task in an embedded application which can operate in two modes. In a non-critical mode, the task wishes to read a sensor every ten seconds, perform some exact, but extensive, calculation and output some value to an actuator. However, in a critical mode it is required to read the sensor every second, undertake a simple inexact calculation and then output this value. A mode change is signalled via a persistent signal (see Subsection 11.4.2). The following program fragment illustrates how the periodic task can be structured:

```
with Persistent_Signals; use Persistent_Signals;
with Ada.Calendar; use Ada.Calendar;
  ...

type Mode is (Non_Critical, Critical);
Change_Mode : Persistent_Signal;

task Sensor_Monitor;

task body Sensor_Monitor is
  Current_Mode : Mode := Non_Critical;
  Next_Time : Time := Clock;
  Critical_Period : constant Duration := 1.0;
  Non_Critical_Period : constant Duration := 10.0;
  Current_Period : Duration := Non_Critical_Period;
begin
  loop
    select
      Change_Mode.Wait;
      if Current_Mode = Critical then
        Current_Mode := Non_Critical;
        Current_Period := Non_Critical_Period;
      else
        Current_Mode := Critical;
        Current_Period := Critical_Period;
      end if;
    then abort
      loop
        -- read sensor
        -- perform appropriate calculation and
        -- output to actuator
        Next_Time := Next_Time + Current_Period;
        delay until Next_Time;
      end loop;
    end select;
  end loop;
end Sensor_Monitor;
```

If the output to the actuator involves a number of operations (or there is more than one actuator to set), then this action could be encapsulated in a call to a protected object. This would force the action to be an abort-deferred operation.

Partial/imprecise computations

Partial or imprecise computations are those that can produce intermediate results of varying accuracy. Moreover, the accuracy of these results does not decrease as the execution of the tasks continues. With these tasks, it is usual to define a minimum precision that is needed by the application.

A task which is to perform a partial computation can place its data into a protected object. The client task can retrieve the data from the protected object. Using a protected object to encapsulate the passing of the result ensures that no inconsistent value is returned to the client due to the task receiving an ATC in the middle of the update.

The following illustrates the approach:

```
with Persistent_Signals; use Persistent_Signals;
with Ada.Real_Time; use Ada.Real_Time;
  ...
Out_Of_Time : Persistent_Signal;
```

```
protected Shared_Data is
  procedure Write(D : Data);
  entry Read(D : out Data);
private
  The_Data : Data;
  Data_Available : Boolean := False;
end Shared_Data;

task Client;
task Imprecise_Server;

protected body Shared_Data is
  procedure Write(D : Data) is
  begin
    The_Data := D;
    Data_Available := True; -- indicates that the
                            -- data is available
  end Write;

  entry Read(D : out Data) when Data_Available is
  begin
    D := The_Data;
    Data_Available := False;
  end Read;
end Shared_Data;

task body Client is
  ...
begin
  loop
    ...
    Out_Of_Time.Send;
    Shared_Data.Read(Result);
    ...
  end loop;
end Client;

task body Imprecise_Server is
  -- declaration of Result
begin
  loop
    ...
    -- produce result with minimum required precision
    Shared_Data.Write(Result);
    select
      Out_Of_Time.Wait;
    then abort
      -- compute refined Result
      -- potentially this may contain
      loop
        -- next iteration
        ...
        Shared_Data.Write(Result);
        exit when Best_Possible_Result_Obtained;
      end loop;
    end select;
  end loop;
end Imprecise_Server;
```

Again, the use of a call on a protected object in the abortable sequence of statements ensures that a consistent value is written.

9.4 Understanding the asynchronous select statement

Although the asynchronous select statement is simple in its syntactic representation, it can result in complex situations arising at run-time. To illustrate these, several contrived examples are now presented and explained.

9.4.1 Interaction with the delay statement

Consider the following example:

```
  task A;                          task B;

  task body A is                   task body B is
    T : Time;                          T : Time;
    D : Duration;                      D : Duration;
  begin                            begin
    ...                              ...
    select                           select
      delay until T;                   delay D;
    then abort                       then abort
      delay D;                         delay until T;
    end select;                      end select;
  end A;                           end B;
```

In the above example, two tasks have an asynchronous select statement with a very similar structure. In both cases the result will be that the task will be delayed either until a certain time (`T`) in the future or until a certain interval has expired (`D`); whichever occurs first will result in the other being cancelled or aborted. However, note that after the delay has expired in the abortable part, the task must be rescheduled before the abortable part can complete and the triggering delay can be cancelled. The triggering delay could expire before this occurs.

9.4.2 Comparison with timed entry calls

Consider the following three tasks, all of which execute a form of timed entry call:

```
task A;               task B;               task C;

task body A is        task body B is        task body C is
  T: Time;              T: Time;              T: Time;
begin                 begin                 begin
  ...                   ...                   ...
  select                select                select
    delay until T;        Server.Entry1;        Server.Entry1;
    S2;                   S1;                   S1;
  then abort            then abort            or
    Server.Entry1;        delay until T;        delay until T;
    S1;                   S2;                   S2;
  end select;           end select;           end select;
end A;                end B;                end C;
```

The execution of the three tasks can be compared according to whether the rendezvous starts or finishes before the time-out occurs:

- Rendezvous with `Server` starts and finishes before the time-out
  `A` executes the rendezvous, and then attempts to execute `S1`; if `S1` does not complete before the time-out, it is abandoned and `S2` is executed.
  `B` executes the rendezvous, and then `S1`.
  `C` executes the rendezvous, and then `S1`.
- The rendezvous starts before the time-out but finishes after the time-out
  `A` executes the rendezvous and `S2`.
  `B` executes the rendezvous, part of `S2` and all of `S1`.†
  `C` executes the rendezvous and `S1`.
- The time-out occurs before the rendezvous is started
  `A` executes `S2`.
  `B` executes `S2` (part or all of it depending if the rendezvous completes before it finishes) and possibly the rendezvous and `S1`.
  `C` executes `S2`.

Interestingly, the semantics of the timed entry call of task `C` can be derived exactly using an asynchronous select statement:

```
task body C is
  T: Time;
begin
  ...
  Occurred := False;
  select
    delay until T;
  then abort
    Server.Entry1(Occurred); -- Occurred set to True in Server
  end select;
```

† Note that on a single processor system, `S2` might not actually get to execute if the processor is busy with the rendezvous and the rendezvous is non-blocking.

```
  if Occurred then
    S1;
  else
    S2;
  end if;
end C;
```

9.4.3 Multiple entry calls

The following example illustrates what happens if the triggering event of an asynchronous select statement is an entry call and the abortable part is a single entry call:

```
task A;

task body A is
  ...
begin
  ...
  select
    C.Entry1;
  then abort
    D.Entry1;
  end select;
end A;
```

```
task B;

task body B is
  ...                       ...
begin
  ...
  select
    D.Entry1;
  then abort
    C.Entry1;
  end select;
end B;
```

In many ways the two tasks above are the same: both appear to wait on an entry call to either task `D` or `C`. Consider the following three cases:

(1) `C.Entry1` becomes ready first (after the asynchronous select statement has been evaluated):

Task `A` will rendezvous with task `C`, and possibly with task `D` (if the rendezvous becomes available before the rendezvous with `C` completes).

Task `B` will rendezvous with task `C`, and possibly with task `D` (if the rendezvous becomes available before the rendezvous with `C` completes).

(2) `D.Entry1` becomes ready first (after the asynchronous select statement has been evaluated):

Task `A` will rendezvous with `D`, and possibly with `C` (if the rendezvous becomes available before the rendezvous with `D` completes).

Task `B` will rendezvous with `D`, and possibly with `C` (if the rendezvous becomes available before the rendezvous with `D` completes).

(3) `C.Entry1` and `D.Entry1` are both ready when the asynchronous select statement is evaluated:

Task `A` will rendezvous with `C` only.

Task `B` will rendezvous with `D` only.

Now consider the following:

```
task A;

task body A is
begin
  ...
  select
    B.Entry1;
    Seq2;
  then abort
    B.Entry1;
    Seq1;
  end select;
  ...
end A;
```

If B is ready when the select statement is executed, Seq2 will be executed. If the task has to wait for the same entry call both in the abortable part and as the triggering event, then in this case the entry queuing policy will dictate which arm of the select completes first. For FIFO or priority scheduling, the triggering entry will complete and Seq2 will again be executed.

The following example continues this point. Task C is called either by task A or by task B, but not both:

```
                                            task C is
                                              entry Entry1;
                                              entry Entry2;
task A;               task B;               end C;

task body A is        task body B is        task body C is
  T: Time;              T: Time;            begin
begin                 begin                   ...
  ...                   ...                   select
  select                select                  accept Entry1 do
    C.Entry1;             C.Entry2;                 ...
  then abort            then abort              end Entry1;
    C.Entry2;             C.Entry1;           or
  end select;           end select;             accept Entry2 do
  ...                   ...                       ...
end A;                end B;                    end Entry2;
                                              end select;
                                              ...
                                            end C;
```

Here when C is waiting at the select statement, A will rendezvous with Entry1, whereas B will rendezvous with Entry2. However if C is not initially waiting at the select statement, then it is not defined what will happen (it will depend on the implementation of the selective accept statement).

Finally, consider the case where task C has the following structure:

```
task body C is
begin
  ...
    accept Entry1 do
      accept Entry2 do
        ...
      end Entry2;
    end Entry1;
  ...
end C;
```

Here for task A, if C is already waiting at the outer accept statement, the system will deadlock (because Entry2 will never be called). If C arrives later than A, then the rendezvous will be completed. For task B, both rendezvous will occur.

9.4.4 Interaction with requeue

A triggering event of an asynchronous select statement can contain a requeue statement. Consider the following tasks:

```
task A;

task body A is
begin
  ...
  select
    B.Entry1;
  then abort;
    Seq;
  end select;
  ...
end A;
```

```
task B is
  entry Entry1;
  entry Entry2;
end B;

task body B is
begin
  ...
  accept Entry1 do
    requeue Entry2;
    -- potentially 'with abort'
  end Entry1
  ...
  accept Entry2 do
    ...
  end Entry2;
end B;
```

When the asynchronous select statement of task A is executed, there are several possible scenarios:

(1) If Entry1 of task B is available and the task is requeued with abort, then Seq will not begin its execution until the requeue has occurred.
(2) If Entry1 of task B is available and the task is requeued without abort, then Seq will never be executed.
(3) If Entry1 of task B becomes available after A has begun its execution of Seq, then if it is requeued with abort, the entry call is cancelled when Seq has finished.

(4) If `Entry1` of task `B` becomes available after `A` has begun its execution of `Seq`, then if it is requeued without abort, the select must wait for the rendezvous with `Entry2` to complete, even when `Seq` has finished.

9.4.5 Nested ATC

The sequence of statements within the abortable part of an asynchronous select statement can contain an asynchronous select statement. Consider the following example:

```
task A;

task body A is
begin
  ...
  select
    B.Entry1;
  then abort
    select  -- nested ATC
      C.Entry1;
    then abort
      Seq;
    end select;
  end select;
  ...
end A;
```

Here, task `A` will wait for an entry call to become complete from task `B` or `C`. If none arrives before `Seq` has finished its execution, both will be cancelled.

Note that if the same task and entry are mentioned in a nested ATC then, with most queuing disciplines, the outer triggering call will be handled first.

9.5 A robust readers and writers algorithm

In Section 7.10, two solutions were given to the *readers and writers problem*. Although a single protected object with a function for read and a procedure for write will, in some senses, solve this problem, it was pointed out that such a solution would not be adequate if preference were to be given to write operations, or if the actual read and write operations of the shared resource were potentially blocking (and hence could not be made from within a protected object). We conclude this chapter by giving a robust version of the first of the two solutions given in Section 7.10.

The difficulty with the solution in Section 7.10 is that it can lead to deadlock if client tasks failed in their execution of the read or write operations on the resource. Recall that the readers and writers protocol was controlled by a protected object:

```
protected Control is
  entry Start_Read;
  procedure Stop_Read;
  entry Start_Write;
  procedure Stop_Write;
private
  Readers : Natural := 0; -- Number of current readers
  Writers : Boolean := False; -- Writers present
end Control;
```

Client tasks called one of the following:

```
procedure Read(I : out Item) is
begin
  Control.Start_Read;
    Read_File(I);
  Control.Stop_Read;
end Read;

procedure Write(I : Item) is
begin
  Control.Start_Write;
    Write_File(I);
  Control.Stop_Write;
end Write;
```

The first difficulty with these procedures is potential exceptions being propagated from the file I/O operations. These can easily be caught.

```
procedure Read(I : out Item) is
begin
  Control.Start_Read;
    Read_File(I);
  Control.Stop_Read;
exception
  when others => Control.Stop_Read;
end Read;

procedure Write(I : Item) is
begin
  Control.Start_Write;
    Write_File(I);
  Control.Stop_Write;
exception
  when others => Control.Stop_Write;
end Write;
```

The next problem is that of an I/O operation that does not return. To solve this, it would be possible to set a time bound of, say, ten seconds for a write operation:

```
procedure Write(I : Item; Failed : out Boolean) is
begin
  Control.Start_Write;
  select
    delay 10.0;
    Failed := True;
  then abort
    Failed := False;
    Write_File(I);
  end select;
  Control.Stop_Write;
exception
  when others =>
    Control.Stop_Write;
    Failed := True;
end Write;
```

One of the motivations behind this solution of the readers and writers problem is to give preference to write operations. In situations where reads can take some time, and writes are not frequent, it may be desirable to abort an incomplete read action in order for an update to occur as soon as possible. This can be programmed using a nested ATC (one for time-out on failure, the other to enable a read to be abandoned):

```
procedure Read(I : out Item; Failed : out Boolean) is
begin
  loop
    Control.Start_Read;
    select
      Control.Abandon_Read;
      Control.Stop_Read;
    then abort
      select
        delay 10.0;
        Failed := True;
      then abort
        Read_File(I);
        Failed := False;
      end select;
      exit;
    end select;
  end loop;
  Control.Stop_Read;
exception
  when others =>
    Control.Stop_Read;
    Failed := True;
end Read;
```

This solution is now resilient to any failure of the file I/O operations. The controlling protected object is a refinement of that given in Chapter 7:

```
protected Control is
  entry Start_Read;
  procedure Stop_Read;
  entry Abandon_Read;
  entry Start_Write;
  procedure Stop_Write;
private
  Readers : Natural := 0; -- Number of current readers
  Writers : Boolean := False; -- Writers present
end Control;

protected body Control is
  entry Start_Read when not Writers is
  begin
    Readers := Readers + 1;
  end Start_Read;

  procedure Stop_Read is
  begin
    Readers := Readers - 1;
  end Stop_Read;

  entry Abandon_Read when Start_Write'Count /= 0 is
  begin
    null;
  end;

  entry Start_Write when not Writers and Readers = 0 is
  begin
    Writers := True;
  end Start_Write;

  procedure Stop_Write is
  begin
    Writers := False;
  end Stop_Write;
end Control;
```

A further refinement to this code is possible if the controller must be protected against the client tasks aborting their read or write operations; for example, if a read is requested and the procedure is then aborted. To provide protection against this eventuality requires the use of an abort-deferred region. The easiest way of obtaining this is via a controlled variable:

```
type Read_Protocol is new Ada.Finalization.Limited_Controlled
    with null record;

procedure Initialize(RP : in out Read_Protocol) is
begin
  Control.Start_Read;
end Initialize;

procedure Finalize(RP : in out Read_Protocol) is
begin
  Control.Stop_Read;
end Finalize;
```

```
procedure Read(I : out Item; Failed : out Boolean) is
begin
  loop
    declare
      Read_Pro : Read_Protocol;
    begin
      select
        Control.Abandon_Read;
      then abort
        select
          delay 10.0;
          Failed := True;
        then abort
          Read_File(I);
          Failed := False;
        end select;
        exit;
      end select;
    end; -- declare block
  end loop;
exception
  when others =>
    Failed := True;
end Read;
```

Note that there is no longer any need for the exception handler to call `Control.Stop_Read` as the finalisation procedure will always execute on exit from the inner block.

9.6 Task states

The task state diagram given in the early chapters can be further extended (see Figure 9.1) to include the new state introduced by the abort statement: abnormal. Modelling ATC in state transition diagrams is difficult, as a task can be waiting for one or more triggering events and be executing at the same time (and therefore calling protected objects, creating child tasks etc.). It is assumed that ATCs are handled as an attribute of a task and therefore not represented directly in the state transition diagrams.

9.7 Summary

The normal behaviour of a multi-tasking Ada program is for each task to make progress by following a valid execution path from activation to either termination or a repeating, indefinite sequence. This chapter has considered the ways in which this normal behaviour can be subverted.

Within a task, exceptions cause the current flow of control to be interrupted and

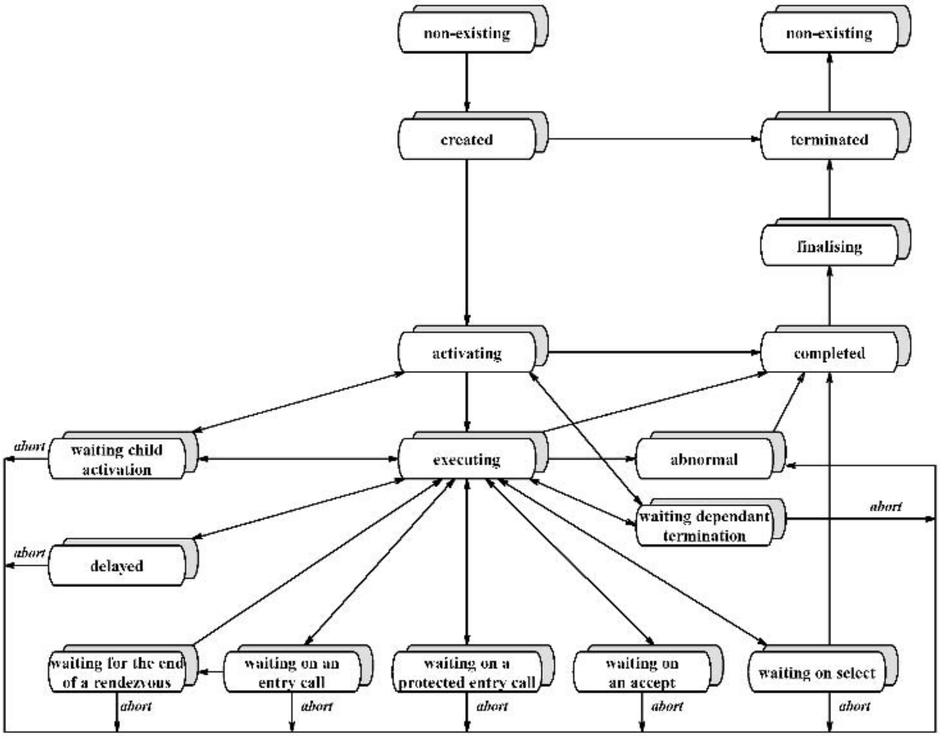


Fig. 9.1: Summary of task states and state transitions

passed to an appropriate exception handler. This is a standard feature of sequential Ada and hence it has not been explained in detail.

The situations in which the tasking model and the exception model interact are, however, summarised in this chapter. Many of the details of these interactions have been discussed in earlier chapters.

The most severe effect that one task can have upon another is to abort it. This is an extreme and irreversible action, and one that is not universally agreed as being appropriate in a concurrent programming language. Although a task cannot protect itself directly from an abort instruction, it will always complete correctly any protected procedures or entries that it is executing.

A more controlled way for one task to affect another indirectly is via an asynchronous transfer of control (ATC). Here a task explicitly (via a call to another task or protected object) indicates that it is prepared to be interrupted while executing a particular sequence of instructions. Examples of the use of ATCs were given in this chapter, as was a discussion of the interactions between ATCs and the delay statement, timed entry calls, multiple entry calls and requeue.

10
Object-oriented programming and tasking

The Ada language addresses many aspects of software engineering, but it is beyond the scope of this book to discuss in detail its support for such topics as programming in the large, reuse and so on. Rather, the goal of this book is to discuss, in depth, the Ada model of concurrency and how it is affected by other areas of the language, for example exception handling. This chapter explores the interaction between the object-oriented programming (OOP) facilities and tasking.

Ada 83 was defined long before object-oriented programming became popular, and it is a credit to the Ada 95 language designers that they managed to introduce OOP facilities without having to alter the basic structure of an Ada program. However, the Ada 95 model had several limitations. In particular, it did not support the standard prefix notation (also called the *distinguised receiver syntax*), `Object_Name.Method_Name(Params)`, found in most OOP languages. Instead, it required the object name to be given as part of the parameter list of the method. Although this had some advantages,† it caused some confusion when programmers moved from languages like C++ and Java to Ada. Ada 2005 now allows the standard OOP notation with the object name prefixing the subprogram name when it is called.

Ada 95 also didn't attempt to integrate the language's support for concurrent programming directly into the OOP model. Instead, the models were orthogonal and paradigms had to be created to allow the benefits of OOP to be available in a concurrent environment. These paradigms had inherent limitations and proposals were developed to add OOP facilities directly into, for example, the protected type mechanism. Unfortunately, these proposals added another layer of complexity to the language and they did not receive widespread support.

Since Ada 95, object-oriented programming techniques have also advanced and the notion of an *interface* has emerged as the preferred mechanism for acquiring

† For example, it allows symmetric binary operators (A+B rather than A.plus(B)) and allows dispatching on function results.

many of the benefits of multiple inheritance without most of the problems. The introduction of this mechanism into Ada 2005 also allows tasks and protected types to support interfaces. Whilst this doesn't give the full power of extensible tasks and extensible protected types, it does give much of the functionality. Indeed, the introduction of a *synchronized* interface is essential if OOP is to be fully integrated with real-time analysis techniques.‡

This chapter provides a brief introduction to Ada 2005's new OOP model and explores in depth how it can be used with the concurrency model. The combination of interfaces and generics presents powerful mechanisms that allow general concurrency utilities to be implemented. The next chapter illustrates some of these utilities.

10.1 The Ada 2005 OOP model

This section provides a brief introduction to

- the new prefix notation for object method calls,
- the role of interfaces in the derivation of tagged types and
- the difference between limited and non-limited interfaces.

The goal here is to provide enough understanding of the new Ada OOP model so that the later material on using interfaces with Ada's task and protected types, including the development of concurrency utilities, can be understood. The further reading section is included at the end of the chapter for those readers who wish to understand the full power of the new OOP model.

10.1.1 The prefix notation

In Ada, one way of representing an extensible object type (called a class in languages like Java) is shown below.

```
package Objects is
  type Object_Type is tagged private;

  procedure Op1(O : in out Object_Type; P : Parameter_Type);
  ... -- other operations

private
  type Object_Type is tagged
  record
    ... -- the type's attributes (instance variables)
  end record;
end Objects;
```

‡ One of the problems with dispatching using Java interfaces is that it is not possible to determine statically if an implementing object is causing blocking to the calling thread. The Ada 2005 approach makes this much clearer.

Instances of these are created in the usual way and the subprograms (methods) are called as shown below.

```
declare
  My_Object : Object_Type; -- a new instance of the object type
  My_Parameter : Parameter_Type;
begin
    ...
    Op1(My_Object, My_Parameter);
       -- object passed as a parameter to the subprogram
    ...
end;
```

Ada 2005 still allows this style but also supports the following.

```
declare
  My_Object : Object_Type;
  My_Parameter : Parameter_Type;
begin
    ...
    My_Object.Op1(My_Parameter);
       -- object name prefixes the subprogram name
    ...
end;
```

This makes the program easier to understand and eases the transition for programmers used to other OOP languages such as C++ and Java.

Important note: The introduction of the prefix notion is more than just syntactic sugar. It eases the naming of methods when complex inheritance hierarchies are used, as there is no need to name all the associated packages in 'with' clauses.

10.1.2 Interfaces

An interface is a type that has no state. In Ada 95, it could be represented as an abstract tagged type without any components. For example, the following abstract type provides the interface needed to print an object. Any types that extend from this type must provide an implementation of the `Print` subprogram. Hence objects of all types in the hierarchy rooted in the `Printable` type can be printed. These can collectively be accessed by the `Any_Printable` type given below.

```
package Printable is
  type Printable_Interface is abstract tagged null record;
  procedure Print(X : in Printable_Interface) is abstract;

  type Any_Printable is access all Printable_Interface'Class;
end Printable;
```

Ada 2005 change: Ada 95 does not allow multiple inheritance; therefore, it is not possible to have a derived type that can support multiple interfaces via the abstract tagged type mechanism. Ada 2005 has solved this problem by introducing a language construct that is essentially an abstract tagged typed with null components. This is called an **interface**. All primitive operations on an interface type must be abstract or null – Ada 2005 has also introduced syntax for a null procedure: e.g. '**procedure** `X (P: Parameter_Type)` **is null**`;`'. This acts as shorthand for a procedure with a single enclosed null statement.

The above interface is represented in Ada 2005 as:

```
package Printable is
  type Printable_Interface is interface;
  procedure Print(X: in Printable_Interface) is abstract;

  type Any_Printable is access all Printable_Interface'Class;
end Printable;
```

Warning: Although the inspiration for Ada interfaces comes from the Java language, they cannot be used in exactly the same way as Java interfaces but should be thought of as abstract types. In particular, it is not possible to define an arbitrary method that takes as a parameter an interface (as it is in Java). The equivalent in Ada 2005 is to define the parameter type as a *class-wide type* or *class-wide access type* rooted at the Ada interface (as illustrated above with access types). Hence, Ada makes explicit what Java leaves as implicit.

For example, a Java-like comparable interface would be represented as:

```
package Comparable is
  type Comparable_Interface is interface;
  type Any_Comparable is access all Comparable_Interface'Class;
  function Compares(This: Comparable_Interface;
            To_That: in Comparable_Interface'Class)
            return Boolean is abstract;
end Comparable;
```

Here the first parameter to the `Compares` method is the controlling (dispatching) parameter. The second parameter represents the object to be compared with. The function returns true if the object components are equal in value.

Now consider an object that wants to support the above two interfaces (or *implement* them, as Java calls it). It simply declares a new tagged type from the two interface types and provides the primitive operations needed to support the interfaces.

```
with Printable; use Printable;
with Comparable; use Comparable;
package Printable_and_Comparable is
  type Object is new Printable_Interface and
                     Comparable_Interface with private;

  overriding procedure Print (This : Object);
  overriding function Compares (This : Object;
             To_That : Comparable_Interface'Class) return Boolean;
private
  type Object is new Printable_Interface and
                     Comparable_Interface with record
    X : Integer := 0;  -- say
  end record;
end Printable_and_Comparable;
```

The implementation is shown below:

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Tags; use Ada.Tags;
package body Printable_and_Comparable is
  procedure Print (This : Object) is
  begin
     Put_Line (This.X'Img);
  end Print;

  function Compares (This : Object;
           To_That : Comparable_Interface'Class)
           return Boolean is
  begin
    return To_That in Object'Class and then
           This.X = Object(To_That).X;
  end Compares;
end Printable_and_Comparable;
```

Of course, if there is a pre-existing type that needs to become printable and comparable:

```
package My_Objects is
  type My_Object is tagged record
    X : Integer;
  end record;
end My_Objects;
```

It is possible to create a new tagged type that is derived from the `My_Object` tagged type that can now be printed and compared.

```
with Printable, Comparable, My_Objects;
use Printable, Comparable, My_Objects;
package My_Printable_And_Comparable_Objects is
  type My_Printable_And_Comparable_Object is new My_Object and
       Printable_Interface and Comparable_Interface with
       null record;
  overriding
  procedure Print(X : My_Printable_And_Comparable_Object);
  overriding
  function Compares(This : My_Printable_And_Comparable_Object;
                    To_That : in Comparable_Interface'Class)
                    return Boolean;
end My_Printable_And_Comparable_Objects;
```

Note that so far, equality is defined to be when an object has the same value in its X field, irrespective of whether it has other fields present or not. If equality was defined to be only for objects of the exact same type then the following would be needed.

```
function Alternative_Compares(This : Object;
         To_That : Comparable_Interface'Class)
         return Boolean is
begin
   return To_That'Tag = Object'Tag  and then
          This = Object(To_That);
end Alternative_Compares;
```

Of course, this is the standard '=' operator in Ada.

Interfaces are incorporated into the standard Ada type declaration syntax:

```
type_declaration ::= full_type_declaration
                     | incomplete_type_declaration
                     | private_type_declaration
                     | private_extension_declaration

full_type_declaration ::=
     type defining_identifier [known_discriminant_part] is
       type_definition;
       | task_type_declaration
       | protected_type_declaration

type_definition ::=
     enumeration_type_definition | integer_type_definition
     | real_type_definition | array_type_definition
     | record_type_definition | access_type_definition
     | derived_type_definition | interface_type_definition

derived_type_definition::=
           [abstract] [limited] new parent_subtype_indication
             [[and interface_list] record_extension_part]


interface_type_definition ::=
    [limited | task | protected | synchronized] interface
             [and interface_list]

interface_list ::= interface_subtype_mark
                   {and interface_subtype_mark}
```

where `interface_subtype_mark` is essentially the name of an interface type.

Important note: Formally, a tagged type can be derived from zero or one parent tagged type plus zero or more interface types. If a non-interface tagged type is present it must be the first in the list. Ada 2005 calls the other types **progenitors**. Hence interfaces themselves can be derived from other interfaces.

10.1.3 Limited interfaces

In Ada, a limited type is a type that does not support the assignment statement and has no predefined equality operator. Interfaces can be limited; they are, however, not limited by default. A non-limited interface has a predefined equality operator available, whereas a limited interface does not.

Important note: A new interface can be composed of a mixture of limited and non-limited interfaces but if any one of them is non-limited then the resulting interface must be specified as *non-limited*. This is because it must allow the equality and assignment operations implied by the non-limited interface.

Similar rules apply to derived types that implement one or more interfaces. The resulting type must be non-limited if any of its associated interfaces are non-limited.

10.1.4 Synchronized, protected and task interfaces

This section focuses on how Ada 2005's interface concept has been applied to the Ada tasking model. The goal is to enable all task and protected types to have access to some of the OOP mechanisms. In order to provide this integration (something that was lacking in Ada 95), Ada 2005 provides three further forms of limited interfaces: *synchronized*, *protected* and *task*.

Important note: The key idea of a synchronized interface is that there is some implied synchronization between the task that calls an operation from an interface and the object that implements the interface.

Synchronisation in Ada is achieved via two main mechanisms: the rendezvous and a protected action (call of an entry or protected subprogram). Hence, a task type or a protected type can implement a synchronized interface. As illustrated in Chapters 5, 6 and 7, the rendezvous provides control-oriented synchronisation and the protected type provides data-oriented synchronisation. Where the programmer is not concerned with the type of synchronisation, a synchronized interface is the appropriate abstraction. For situations where the programmer requires control-oriented (or data-oriented) synchronisation, task interfaces (or protected interfaces) should be used explicitly.

Important notes:

- All task and protected interfaces are also synchronized interfaces, and all synchronized interfaces are also limited interfaces.
- A synchronized interface can be implemented by either a task or a protected type.
- A protected interface can **only** be implemented by a protected type.
- A task interface can **only** be implemented by a task type.
- The controlling (dispatching) parameter in a synchronized (or protected or task) interface must be the first parameter. If an operation is to be implemented as an entry (or a procedure within a protected object), the parameter must be of mode 'in out' or 'out', or an access type.

Interfaces of various types can be composed as long as no conflict arises. Hence, given

```
type Limited_Interface is limited interface;
... -- primitive operations

type Synchronized_Interface is synchronized interface;
... -- primitive operations

type Protected_Interface is protected interface;
... -- primitive operations

type Task_Interface is task interface;
... -- primitive operations
```

the following are all permitted:

```
type New_Synchronized_Interface is synchronized interface and
         Synchronized_Interface and Limited_Interface;
... -- primitive operations

type New_Protected_Interface is protected interface and
         Protected_Interface and New_Synchronized_Interface;
... -- primitive operations

type New_Task_Interface is task interface and
         Task_Interface and New_Synchronized_Interface;
... -- primitive operations
```

Important note: There is a hierarchy in the types of limited interfaces: limited comes before synchronized, which comes before task/protected. Task and protected interfaces have equal position in the hierarchy. Two or more interfaces can be composed as long as the resulting interface is not before any of the component interfaces in the hierarchy.

Hence a limited interface *cannot* be composed from a synchronized, protected or task interface. A synchronized interface *cannot* be composed from a task or protected interface; and a task and a protected interface *cannot* be composed from each other.

Calling operations on objects that implement limited interfaces

Operations on interfaces that are defined as limited, synchronized, protected or task may be implemented directly by a protected or task type. The following should be noted.

(1) A call to any operation on an object that implements a synchronized (or protected or task) interface may be blocked until the right conditions are in place for that operation to be executed. If the operation is implemented by a protected procedure/function then this blocking will be bounded. If the operation is implemented by a task or a protected entry then the blocking may be indefinite. Consequently, Ada 2005 allows a call to an operation defined by a synchronized (or protected or task) interface to be placed in a 'time or conditional entry call' statement or in a 'select then abort' statement but not in a 'requeue' statement.

(2) A task or protected type can also implement a limited interface. Consequently, calls to objects that implement limited interfaces may also block! Ada 2005, therefore, allows them to be placed in a 'time or conditional entry call' statement or in a 'select then abort' (ATC) statement.

(3) Any call to an operation on an object that implements a limited (or synchronized or protected or task) interface that dispatches to a non-entry call is deemed to have been 'accepted' immediately and, therefore, can never time-out.

10.2 Tasks and interfaces

Recall from Chapter 4, the syntax of a task declaration is:

```
task_type_declaration ::=
  task type defining_identifier [known_discriminant_part]
    [is  [new  interface_list with] task_definition];
```

```
task_definition ::=
    {task_item}
[ private
    {task_item}]
end [task_identifier];

task_item ::= entry_declaration | aspect_clause
```

Hence, a task type can now be declared to 'implement' zero, one or more combinations of limited, synchronized and task interfaces. So for example, given the following interface declaration

```
package Simple_Task_Interface is
  type Simple_TI is task interface;
  procedure OP(TI : in out Simple_TI) is abstract;
end Simple_Task_Interface;
```

a task type declaration derived from this interface would be as follows:

```
task type Example_Task_Type is new Simple_TI with
  entry OP;
end Example_Task_Type;
```

Note that the task interface contains subprogram declarations even though the task implements them as entries. It is not possible to put an entry declaration in an interface definition.

Important note: In Ada 2005, it is not possible to declare a task type that is extended from another task type. A task type can only be derived from one or more interfaces.

Consider the following example that defines an interface that allows a parameterless procedure to be executed concurrently with the caller:

```
package Asynchronous_Execution is
   type Parameterless_Procedure is access procedure;
   type Async_Identifier is private;

   Invalid_Join_Id : exception;

   type Async_Executor is task interface;
   type Any_Async_Executor is access all Async_Executor'Class;

   procedure Fork(AE : in out Async_Executor;
                  This : Parameterless_Procedure;
                  Id : out Async_Identifier) is abstract;
   procedure Join(AE : in out Async_Executor;
                  Id : Async_Identifier) is abstract;
private
   type Async_Identifier is new Integer;
end Asynchronous_Execution;
```

Here, a task interface is an appropriate abstraction as any implementation will need to use a task to obtain the parallel execution. The definition includes a general access type, as often a pointer to the actual task object is required.

The procedure `Fork` takes an access parameter to a parameterless procedure. When called, it returns 'immediately' (passing back an identifier that can be used by the caller to name the associated parallel execution) and then invokes the procedure in parallel with the calling task. When the calling task wishes to wait for the forked routine to finish, it invokes the `Join` procedure passing the associated identifier. Any mismatch between the identifier returned by the `Fork` procedure and that expected by the `Join` procedure results in the `Invalid_Join_Id` exception being raised.

Now consider the following implementation of this interface.

```
package Asynchronous_Execution.Default is
   task type Single_Shot_Agent is new Async_Executor with
     overriding entry Fork(This : Parameterless_Procedure;
                           Id : out Async_Identifier);
     overriding entry Join(Id : Async_Identifier);
   end Single_Shot_Agent;
end Asynchronous_Execution.Default;
```

The `Single_Shot_Agent` task type is derived from the `Async_Executor` interface. The two entries implement the required interface.

Important note: Ada 2005 allows the programmer to explicitly indicate when subprograms and entries are intended to override the primitive operations of a type (or interface). This allows simple spelling errors in operation names to be detected at compile-time.

The trivial implementation of the agent task is shown below. As can be seen, the task terminates, once it has joined with the calling task.

```
package body Asynchronous_Execution.Default is
  task body Single_Shot_Agent is
    My_Procedure : Parameterless_Procedure;
    My_Id : Async_Identifier := 0;
  begin
    accept Fork(This : Parameterless_Procedure;
                Id : out Async_Identifier) do
      My_Procedure := This;
      Id := My_Id;
    end Fork;
    My_Procedure.all;
    loop
      begin
        accept Join(Id : Async_Identifier) do
          if Id /= My_Id then
            raise Invalid_Join_Id;
          end if;
        end Join;
        exit;
      exception
        when Invalid_Join_Id =>
          null;
      end;
    end loop;
  end Single_Shot_Agent;
end Asynchronous_Execution.Default;
```

An alternative implementation of this interface provides a reusable agent.

```
package Asynchronous_Execution.Reusable is
  task type Reusable_Agent is new Async_Executor with
    overriding entry Fork(This : Parameterless_Procedure;
                          Id : out Async_Identifier);
    overriding entry Join(Id : Async_Identifier);
  end Reusable_Agent;
end Asynchronous_Execution.Reusable;


package body Asynchronous_Execution.Reusable is
  task body Reusable_Agent is
    My_Procedure : Parameterless_Procedure;
    My_Id : Async_Identifier := 0;
  begin
    loop
      select
        accept Fork(This : Parameterless_Procedure;
                    Id : out Async_Identifier) do
          My_Procedure := This;
          Id := My_Id;
        end Fork;
      or
        terminate;
      end select;
      My_Procedure.all;
      loop
        begin
          accept Join(Id : Async_Identifier) do
            if Id /= My_Id then
              raise Invalid_Join_Id;
            end if;
          end Join;
          exit;
        exception
          when Invalid_Join_Id =>
            null;
        end;
      end loop;
      My_Id := My_Id + 1;
    end loop;
  end Reusable_Agent;
end Asynchronous_Execution.Reusable;
```

Important note: As a task type cannot be derived from other task types, it is not possible to share the code between the above two implementations for the `Async_Executor` interface.

Another implementation might provide a timeout on the calling of the `Join` procedure relative to when the fork occurred.

```
package Asynchronous_Execution.Reusable_With_Timeout is
  task type Reusable_Agent is new Async_Executor with
    entry Set_Timeout(D : Duration);
    overriding entry Fork(This : Parameterless_Procedure;
                          Id : out Async_Identifier);
    overriding entry Join(Id : Async_Identifier);
  end Reusable_Agent;
end Asynchronous_Execution.Reusable_With_Timeout;


with Ada.Calendar; use Ada.Calendar;
package body Asynchronous_Execution.Reusable_With_Timeout is
  task body Reusable_Agent is
    My_Procedure : Parameterless_Procedure;
    My_Timeout : Duration;
    My_Id : Async_Identifier := 0;
    Fork_Time : Time;
  begin
    accept Set_Timeout(D : Duration) do
      My_timeout := D;
    end Set_Timeout;
    loop
      select
        accept Fork(This : Parameterless_Procedure;
                    Id : out Async_Identifier) do
          My_Procedure := This;
          Id := My_Id;
          Fork_Time := Clock;
        end Fork;
      or
        terminate;
      end select;
      My_Procedure.all;
      loop
        begin
          select
            accept Join(Id : Async_Identifier) do
              if Id /= My_Id then
                raise Invalid_Join_Id;
              end if;
            end Join;
          or
            delay until Fork_Time + My_Timeout;
          end select;
          exit;
        exception
          when Invalid_Join_Id =>
            null;
        end;
      end loop;
      My_Id := My_Id + 1;
    end loop;
  end Reusable_Agent;
end Asynchronous_Execution.Reusable_With_Timeout;
```

Note, in the above example, the task provides an additional entry to set the timeout value.

Important note: As a consequence of a procedure in a limited interface being implementable by a task (or protected) entry, Ada 2005 now allows a timeout to be associated with the procedure call. Hence, the following could be used by the client.

```
My_Agent : Async_Executor'Class := ...;

select
  My_Agent.Join(...);
  -- This can be any procedure call that uses a limited interface
or
  delay 10.0;
end select;
```

10.2.1 Image processing example

To illustrate the use of the above task interfaces, consider an image processing system using range images. The image to be processed begins as an array of points whose values represent the distance from the camera to a corresponding point in the scene being viewed. A typical first step in the processing is the calculation of the surface normals at each point of the array. This operation is ideally suited for parallel computation.

The problem is configured so that the image acquisition is performed in a package, `Image_Acq`, and the data is held in shared memory. Overall control is performed by a `Manager` task that processes segments of the image in parallel via calls to the `Calc_Seg` package.

A package, `Images`, contains the type declarations for the images and image segments. It also contains the farm of worker tasks which support the `Async_Executor` interface. Note, at this stage no commitment has been made to the actual worker task type to be used. Notice also the use of the pragmas to ensure that the shared data is properly accessed.

```
with Asynchronous_Execution;
package Images is
  type Seg_Index is range 1 .. 8;
  type Image_Index is range 1 .. 4;
  type Image_Seg is array(Seg_Index, Seg_Index) of Integer;
  type Image is array (Image_Index, Image_Index) of Image_Seg;
  pragma Volatile_Components(Image);

  type Farm is array(Image_Index, Image_Index) of
       Asynchronous_Execution.Any_Async_Executor;
  pragma Atomic_Components(Farm);
  My_Farm : Farm;
end Images;
```

The `Image_Acq` package contains the code to take the picture, and stores the resulting image so that it can be accessed by the worker tasks:

```
with Images; use Images;
package Image_Acq is
  procedure Get_The_Picture;
  Picture : Image;
end Image_Acq;
```

The `Manager` package contains the code for managing the parallel execution. Worker tasks can request segments to be processed and return the results. The segments are identified by indices into the `Picture` image.

```
with Images; use Images;
with Asynchronous_Execution; use Asynchronous_Execution;
package Manager is
  procedure Get_Segment_Index(I, J : out Image_Index);
end Manager;
```

The body of `Manager` contains a single task. The `Controller` first takes the picture and then distributes the segments of the image to the worker tasks by calling the `Fork` method. At this time, run-time dispatching occurs to the agent that has been used to provide the parallel execution. The `Controller` task is also responsible for collecting the results:

```
with Image_Acq;
with Calc_Seg;
package body Manager is
  task Controller is
    entry Get_Segment_Index(I, J : out Image_Index);
  end Controller;

  procedure Get_Segment_Index(I, J : out Image_Index) is
  begin
    Controller.Get_Segment_Index(I,J);
  end Get_Segment_Index;
  task body Controller is
    type Worker_Ids is array(Image_Index, Image_Index) of
         Asynchronous_Execution.Async_Identifier;
    Ids : Worker_Ids;
  begin
    loop

      -- The image acquisition process is often of a repetitive
      -- nature, so some sort of loop structure would be used
      -- here. The actual nature of the loop is not of
      -- significance to this example, so no details are given.
      -- Whatever computation and communication are required
      -- take place before image acquisition.

      Image_Acq.Get_The_Picture; -- this could delay the task

      -- Start the process of image acquisition. Initiate the
      -- worker tasks.

      for Major_Index in Image_Index loop
        for Minor_Index in Image_Index loop
           My_Farm(Major_Index, Minor_Index).Fork(Calc_Seg.
                Do_Normal'Access, IDs(Major_Index, Minor_Index));
        end loop;
      end loop;

      -- Pass the segments to be processed for normal calculation,
      -- whatever computation and communication take place
      -- during image processing.

      for Major_Index in Image_Index loop
        for Minor_Index in Image_Index loop
          accept Get_Segment_Index(I, J : out Image_Index) do
            I := Major_Index;
            J := Minor_Index;
          end Get_Segment_Index;
        end loop;
      end loop;

      -- Now wait for completion

      for Major_Index in Image_Index loop
        for Minor_Index in Image_Index loop
          My_Farm(Major_Index, Minor_Index).Join(
                IDs(Major_Index, Minor_Index));
        end loop;
      end loop;

      -- Whatever computation and communication
      -- take place after image processing.
    end loop;
  end Controller;
end Manager;
```

The Calc_Seg package is a package which provides the code to be used by the worker tasks to process the image segments and calculate the surface normals:

```
package Calc_Seg is
  procedure Do_Normal;
end Calc_Seg;


with Images; use Images;
with Manager;
with Image_Acq;
package body Calc_Seg is
  procedure Do_Normal is
    I, J : Image_Index;
  begin
      -- getting a new segment to process whenever
      -- Image_Acq has a new image to be processed.
      Manager.Get_Segment_Index(I, J);
      -- Get a segment to work on. Do calculation
      -- of normals, leaving result in Picture.
      -- Picture(I,J) := ...;
  end Do_Normal;
end Calc_Seg;
```

Finally, a main program decides on which worker task type to use and how many. For example, the following uses one task per segment.

```
with Manager; use Manager;
with Images; use Images;
with Asynchronous_Execution.Reusable;
procedure Image_System is
begin
   for Major_Index in Image_Index loop
     for Minor_Index in Image_Index loop
        My_Farm(Major_Index, Minor_Index) :=
            new Asynchronous_Execution.Reusable.Reusable_Agent;
     end loop;
   end loop;
end Image_System;
```

However, if the application is run on a multiprocessor system with a limited number of processors, it may be more efficient to limit the number of work tasks to the number of processors.

10.3 Protected types and interfaces

Protected interfaces are interfaces that can only be implemented by protected types. They should be used when

- the programmer wants data-oriented rather than control-oriented synchronisation,
- there is a level of indirection between the tasks needing the synchronisation or
- the programmer wishes to ensure that the required synchronisation is implemented by a passive synchronisation agent rather than an active one.

For example, often a task needs to wait for a signal from another task before it can proceed. There are various types of signals. With all types of signals, it is essential to separate the sending and the waiting interface in order to ensure that the tasks are able to call only their appropriate operation. In this case, a signal is sent via a `Send` operation and is received by the `Wait` operation.

```
package Signals is
  type Signal_Sender is protected interface;
  procedure Send(S : in out Signal_Sender) is abstract;
  type Any_Signal_Sender is access all Signal_Sender'Class;

  type Signal_Waiter is protected interface;
  procedure Wait(S : in out Signal_Waiter) is abstract;
  type Any_Signal_Waiter is access all Signal_Waiter'Class;
end Signals;
```

Transient signals are signals that release one or more waiting tasks but they are lost if no tasks are waiting. They are illustrated below (the protected type `Transient_Signal` releases a single task, whereas the `Pulse` protected type releases all waiting tasks).

```
package Signals.Transient_Signals is
  protected type Transient_Signal is new Signal_Sender and
            Signal_Waiter with
    overriding procedure Send;
    overriding entry Wait;
  private
    Arrived : Boolean := False;
  end Transient_Signal;
  type Transient_Signal_Access is access all Transient_Signal;

  protected type Pulse is new Signal_Sender and Signal_Waiter with
    overriding procedure Send;
    overriding entry Wait;
  private
    Arrived : Boolean := False;
  end Pulse;
  type Pulse_Signal_Access is access all Pulse;
end Signals.Transient_Signals;
```

The body is given below.

```
package body Signals.Transient_Signals is
  protected body Transient_Signal is
```

```
    procedure Send is
    begin
      Arrived := Transient_Signal.Wait'Count > 0;
    end Send;

    entry Wait when Arrived is
    begin
      Arrived := False;
    end Wait;
  end Transient_Signal;

  protected body Pulse is
    procedure Send is
    begin
       Arrived := Pulse.Wait'Count > 0;
    end Send;

    entry Wait when Arrived is
    begin
      Arrived := Pulse.Wait'Count > 0;
    end Wait;
  end Pulse;
end Signals.Transient_Signals;
```

A persistent signal (sometimes called a latch or a gate) is a signal that remains set until a single task has received it.

```
package Signals.Persistent_Signals is
  protected type Persistent_Signal is new Signal_Sender and
            Signal_Waiter with
    overriding procedure Send;
    overriding entry Wait;
  private
    Arrived : Boolean := False;
  end Persistent_Signal;
  type Persistent_Signal_Access is access all Persistent_Signal;
end Signals.Persistent_Signals;
```

The body is given below.

```
package body Signals.Persistent_Signals is
  protected body Persistent_Signal is
    procedure Send is
    begin
      Arrived := True;
    end Send;

    entry Wait when Arrived is
    begin
      Arrived := False;
    end Wait;
  end Persistent_Signal;
end Signals.Persistent_Signals;
```

Disk controller example

As an example of the use of a persistent signal, consider the following package, which provides access to a disk.

```
with Signals; use Signals;
with System;
package Disk_Controller is
  Max_No_Blocks_On_Disk : constant Positive := 10_000; -- say
  type Block_Number is range 0 .. Max_No_Blocks_On_Disk;

  -- various operations including
  procedure Async_Write(To_Block : Block_Number;
            From : System.Address; Size : Positive;
            Done : out Any_Signal_Waiter);
end Disk_Controller;
```

One of the functions that it provides for is asynchronous output. That is, the calling task is not blocked when it requests that data be written to the disk. Instead, the package returns an access variable to a persistent signal (the waiter interface). When the output is complete, the controller sends the associated signal. Inside this package, the output may be queued and actually not written to the disk for some time.

```
with Signals.Persistent_Signals; use Signals.Persistent_Signals;
package body Disk_Controller is
  -- various internal data structures, including

  To_Send : Signals.Persistent_Signals.Persistent_Signal_Access;

  -- various operations including
  procedure Async_Write(To_Block : Block_Number;
            From : System.Address; Size : Positive;
            Done : out Any_Signal_Waiter) is
  begin
    ...
    To_Send := new Persistent_Signal;
    Done := Any_Signal_Waiter(To_Send);
  end Async_Write;

  -- at a later time, in some task or operation of
  -- the disk controller
  To_Send.Send;
end Disk_Controller;
```

The client of the package can proceed as soon as the procedure returns and later check to see if the output has been written:

```
with Disk_Controller; use Disk_Controller;
with System;
with Signals; use Signals;
  ...
  Output_Done : Any_Signal_Waiter;
  Block_Id : Block_Number;
  From_Address : System.Address;
  Data_Size : Positive;
  ...
  -- set up data and parameters for transfer

  Async_Write(Block_Id, From_Address, Data_Size, Output_Done);
  ...
  -- when the time has come to check that
  -- the output is complete
  select
    Output_Done.Wait;
  else
    -- output not complete,
    -- perhaps initiate some recovery action
  end select;
```

Important note: Ada does not require automatic garbage collection; therefore, care must be taken when protected objects are created dynamically from an access type, as the protected object will still exist even though the output operation has completed. This problem is common to all objects that are created dynamically and techniques such as unchecked deallocation must be used.

An alternative to the disk controller object creating the persistent object is to let the client decide what type of signal to use and to pass the sender interface with the write request. The disk controller class becomes

```
with Signals; use Signals;
with System;
package Disk_Controller is
  Max_No_Blocks_On_Disk : constant Positive := 10_000; -- say
  type Block_Number is range 0 .. Max_No_Blocks_On_Disk;

  -- various operations including
  procedure Async_Write(To_Block : Block_Number;
            From : System.Address; Size : Positive;
            Done : in Any_Signal_Sender);
              -- Note Done is now an 'in' parameter and
              -- of type Any_Signal_Sender
end Disk_Controller;
```

In the above, the disk controller expects the client to provide its own synchronisation agent. All that the controller requires is an interface for signalling when the required operation is complete. The disk controller's body becomes:

```
package body Disk_Controller is
  -- various operations including
  To_Send : Signals.Any_Signal_Sender;
  procedure Async_Write(To_Block : Block_Number;
            From : System.Address; Size : Positive;
            Done : in Signals.Any_Signal_Sender) is
  begin
    ...
    To_Send := Done;
  end Async_Write;

  -- at a later time, in some task or operation of
  -- the disk controller
  To_Send.Send;
end Disk_Controller;
```

The client body becomes:

```
with Disk_Controller; use Disk_Controller;
with System;
with Signals; use Signals;
with Signals.Persistent_Signals; use Signals.Persistent_Signals;
  ...
  Output_Done : Persistent_Signal_Access := new Persistent_Signal;
  Block_Id : Block_Number;
  From_Address : System.Address;
  Data_Size : Positive;
  ...
  -- set up data and parameters for transfer

  Async_Write(Block_Id, From_Address, Data_Size,
              Any_Signal_Sender(Output_Done));
  ...
  -- when the time has come to check that
  -- the output is complete
  select
    Output_Done.Wait;
  else
    -- output not complete,
    -- perhaps initiate some recovery action
  end select;
  ...
```

10.4 Synchronized interfaces

Task and protected interfaces are the correct abstractions to use when the programmer wishes to commit to a particular implementation strategy. In some circumstances, this may not be appropriate. For example, there are various communication paradigms that all have at their heart some form of buffer. They, therefore, all have buffer-like operations in common. Some programs will use these paradigms and will not care whether the implementation uses a mailbox, a link or whatever.

Some will require a task in the implementations, others will just need a protected object. Synchronized interfaces allow the programmer to defer the commitment to a particular paradigm and its implementation approach.

Consider the operations that can be performed on all buffers:

```
generic
  type Element is private;
  Max_Capacity : Positive;
package Buffers is
  type Buffer is synchronized interface;
  type Any_Buffer is access all Buffer'Class;
  procedure Put(Buf : in out Buffer; Item : in Element)
            is abstract;
  procedure Get(Buf : in out Buffer; Item : out Element)
            is abstract;
end Buffers;
```

Now consider the various types that can implement this interface. First a mailbox:

```
with Buffers;
generic
package Buffers.Mailboxes is
  subtype Capacity_Range is Natural range 0 .. Max_Capacity - 1;
  subtype Count is Integer range 0 .. Max_Capacity;
  type Buffer_Store is array(Capacity_Range) of Element;

  protected type Mailbox is new Buffer with
    overriding entry Put(Item : in Element);
    overriding entry Get(Item : out Element);
  private
    First : Capacity_Range := Capacity_Range'First;
    Last : Capacity_Range := Capacity_Range'Last;
    Number_In_Buffer : Count := 0;
    Box_Store : Buffer_Store;
  end Mailbox;
end Buffers.Mailboxes;
```

Next an active buffer:

```
with Buffers;
generic
package Buffers.Active_Buffers is
  task type Active_Buffer is new Buffer with
    overriding entry Put(Item : in Element);
    overriding entry Get(Item : out Element);
  end Active_Buffer;
end Buffers.Active_Buffers;
```

Others implementations are shown in the next chapter.

10.5 Summary

This chapter has discussed the extensions to the Ada 95 OOP model that have been introduced in Ada 2005. The new facilities include:

- A new prefix notation for object method calls.
- Concise syntax for the declaration of null procedures.
- Explicit language support for interfaces.

Interfaces have the following properties:

- Interfaces can be synchronized, protected, task, limited or non-limited.
- All task and protected interfaces are also synchronized interfaces, and all synchronized interfaces are also limited interfaces.
- Interfaces can be composed of a mixture of limited (or synchronized, protected, task) and non-limited interfaces but if any one of them is non-limited then the resulting interface must be specified as non-limited.
- A synchronized interface can be implemented by either a task or a protected type.
- A protected interface can **only** be implemented by a protected type.
- A task interface can **only** be implemented by a task type.
- Calls to operations in limited interfaces can be used within a select statement where an entry call is expected.

The use of interfaces allows the programmer to defer decisions about synchronisation and to group tasks (and protected types) into common classes. When it is appropriate for a collection of tasks (and protected types) to have the same operations, the use of an interface provides the required program structure.

10.6 Further reading

J. Barnes, *Programming in Ada 2005*, Addison-Wesley, 2006.

D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 1997.

A.J. Wellings, *Concurrent and Real-Time Programming in Java*, Wiley, 2004.

11
Concurrency utilities

Ada 2005 provides a very flexible framework for object-oriented concurrent programming. It is possible to use this framework to implement a range of reusable concurrency abstractions. The aim of this chapter is to illustrate how a set of concurrency utilities can be constructed. The ones discussed include examples of how to define abstractions that facilitate

- communication and synchronisation between tasks;
- the construction, execution and termination of concurrent activities.

Using a predefined set, instead of developing new ones each time, has several advantages including

- Reduced programming effort and increased productivity – reusing pre-written software components will improve programmer effectiveness.
- Increased performance – publicly available utilities will be scrutinised by the Ada community and will be optimized for performance.
- Increased reliability – publicly available utilities will be scrutinised by the Ada community and bugs will be detected and removed.
- Improved maintainability – programs that use standard concurrency utilities will be easier to understand and maintain than those that use bespoke ones.

Warning: It is beyond the scope of this book to provide a definitive set of concurrency utilities (that might be suitable for public distribution), rather the goal is to illustrate the expressive power of the language and initiate discussion within the Ada community towards such a library.

11.1 Communication and synchronisation abstractions

There are many communication and synchronisation paradigms; this chapter will illustrate the following:

- Semaphores – A standard counting semaphore providing wait (P) and signal (V) operations, along with binary semaphores and quantity semaphores variations.
- Locks – The standard mutual exclusion and read-write locking mechanisms.
- Signals – A mechanism to support condition synchronisation between tasks without passing data parameters. The main operations are wait and send. Both persistent signals and transient signals are shown.
- Event variables – Event variables are bivalued state variables (*up* or *down*). Tasks can *set* (assign to *up*), *reset* (assign to *down*) or *toggle* an event. Any tasks *waiting* for the event to become *up* (or *down*) are released by a call of *set* (or *reset*); *toggle* can also release *waiting* tasks.
- Buffers – Buffers support the standard bounded buffer abstraction.
- Blackboards – Blackboards are similar to events except that they allow arbitrary data to be transmitted between the signalling and the waiting task.
- Broadcasts – The broadcast entry supports the standard broadcast paradigm.
- Barriers – The barrier entry provides a pure synchronisation mechanism; it allows a set number of tasks to block until all are present. They are then all released.

11.2 Semaphores

It is often claimed that semaphores are among the most flexible low-level synchronisation primitives. There are various types of semaphores but they all share the following common interface. Here, the sender and the receiver interface are separated out. Note, that to avoid committing to a particular implementation strategy, synchronized interfaces are used:

```
package Semaphores is
  type Signal_Interface is synchronized interface;
  procedure Signal(Sem : in out Signal_Interface) is abstract;
  type Any_Signal_Interface is access all Signal_Interface'Class;

  type Wait_Interface is synchronized interface;
  procedure Wait(Sem : in out Wait_Interface) is abstract;
  type Any_Wait_Interface is access all Wait_Interface'Class;

  type Semaphore_Interface is synchronized interface and
       Signal_Interface and Wait_Interface;
  type Any_Semaphore_Interface is access all
       Semaphore_Interface'Class;
end Semaphores;
```

The most common semaphore is a **counting** semaphore. This can be initialised to any value greater that 0. Calls to the `Wait` subprogram are blocked when the value of the semaphore is zero. Calls to `Signal` will increment the semaphore and this will result in a blocked task being released.

Two types of counting semaphores can be defined, one that does not have any deadlock prevention mechanism and one that does. The one that does not, is based on a protected type; the one that does, is based on a task.

```
package Semaphores.Counting is
  protected type Counting_Semaphore (Initial : Natural) is
           new Semaphore_Interface with
    overriding entry Wait;
    overriding procedure Signal;
  private
    Value : Natural := Initial;
  end Counting_Semaphore;

  task type Counting_Semaphore_With_Deadlock_Prevention (
            Initial : Natural;
            Timeout_In_Milliseconds : Positive) is
       new Semaphore_Interface with
    overriding entry Wait;
    overriding entry Signal;
  end Counting_Semaphore_With_Deadlock_Prevention;
end Semaphores.Counting;
```

Warning: With the protected type implementation, if a task fails to call the signal operation whilst holding a semaphore (because of either an unhandled exception or an ATC or an abort), the semaphore will become deadlocked. The tasking implementation can be used to help avoid this problem.

The approach to deadlock prevention is based on a timeout (hence, the necessity to use a task). Note to obtain termination, the task has to be structured with two select statements (as a delay branch and a termination branch cannot appear in the same select statement). In contrast, the protected type implementation is very straightforward.

```
package body Semaphores.Counting is
  protected body Counting_Semaphore is
    entry Wait when Value > 0 is
    begin
      Value := Value - 1;
    end Wait;

    procedure Signal is
    begin
```

```
      Value := Value + 1;
    end Signal;
  end Counting_Semaphore;

  task body Counting_Semaphore_With_Deadlock_Prevention is
    Value : Natural := Initial;
  begin
    loop
       if Value = 0 then
         select
           accept Signal;
           Value := Value + 1;
         or
           delay Timeout_In_Milliseconds * 0.001;
           Value := Value + 1;
         end select;
       else
         select
           accept Wait;
           Value := Value - 1;
         or
           accept Signal;
           Value := Value + 1;
         or
           terminate;
         end select;
      end if;
    end loop;
  end Counting_Semaphore_With_Deadlock_Prevention;
end Semaphores.Counting;
```

An example use of a semaphore is given below:

```
package Restricted_Tunnel_Control is
  -- This package allows a certain number of cars to enter
  -- a tunnel

  procedure Enter_Protocol; -- potentially blocking operation
  procedure Exit_Protocol;
end Restricted_Tunnel_Control;


with Semaphores.Counting; use Semaphores.Counting;
package body Restricted_Tunnel_Control is
  Max_Allowed : constant Natural := 10;
  -- maximum number of cars allowed in the tunnel

  Tunnel_Control : Counting_Semaphore(Max_Allowed);

  procedure Enter_Protocol is
  begin
    Tunnel_Control.Wait;
  end Enter_Protocol;
  procedure Exit_Protocol is
  begin
    Tunnel_Control.Signal;
  end Exit_Protocol;
end Restricted_Tunnel_Control;
```

In the above example, cars wishing to enter the tunnel must first call the `Enter_Protocol` procedure. This is a potentially blocking procedure as it calls an entry in a `Semaphore` protected object; only ten cars are allowed in the tunnel at any one time. The cars call `Exit_Protocol` as they leave the tunnel.

11.2.1 Binary semaphores

Although semaphores can be programmed in Ada, it is often more appropriate to program the actual synchronisation required directly rather than with a semaphore. For example, consider binary semaphores (which only take the values 0 and 1)

```
package Semaphores.Binary is
  type Binary_Value is range 0 .. 1;

  protected type Binary_Semaphore (Initial : Binary_Value) is
            new Semaphore_Interface with
    overriding entry Wait;
    overriding procedure Signal;
  private
    Free : Binary_Value := Initial;
  end Binary_Semaphore;

  task type Binary_Semaphore_With_Deadlock_Prevention (
            Initial : Binary_Value;
            Timeout_In_Milliseconds : Positive) is
            new Semaphore_Interface with
    overriding entry Wait;
    overriding entry Signal;
  end Binary_Semaphore_With_Deadlock_Prevention;
end Semaphores.Binary;
```

and their associated implementations

```
package body Semaphores.Binary is
  protected body Binary_Semaphore is
    entry Wait when Free = 1 is
    begin
      Free := 0;
    end Wait;

    procedure Signal is
    begin
      Free := 1;
    end Signal;
  end Binary_Semaphore;
  task body Binary_Semaphore_With_Deadlock_Prevention is
    Free : Binary_Value := Initial;
  begin
    loop
      if Free = 0 then
        select
          accept Signal;
          Free := 1;
        or
          delay Timeout_In_Milliseconds * 0.001;
          Free := 1;
        end select;
      else
        select
          accept Wait;
          Free := 0;
        or
          accept Signal;
          Free := 1;
        or
          terminate;
        end select;
      end if;
    end loop;
  end Binary_Semaphore_With_Deadlock_Prevention;
end Semaphores.Binary;
```

To obtain mutual exclusion using a semaphore (without deadlock detection) requires the following:

```
with Semaphores.Binary; use Semaphores.Binary;
  ...
  Mutex : Binary_Semaphore(1);
  -- declaration of data requiring
  -- mutually exclusive access
  ...
  procedure Mutual_Exclusion is
  begin
    Mutex.Wait;
     -- code to be executed in mutual exclusion
    Mutex.Signal;
  end Mutual_Exclusion;
```

This can, however, more efficiently, more safely and more elegantly be written as:

```
  protected Mutual_Exclusion is
    procedure Operation;
  private
     -- declaration of data requiring
     -- mutually exclusive access
  end Mutual_Exclusion;
  protected body Mutual_Exclusion is
    procedure Operation is
    begin
      -- code to be executed in mutual exclusion
    end Operation;
  end Mutual_Exclusion;
```

Important note: Many communication paradigms are necessary because Ada does not allow a task to block from within a protected object. Hence, the above approach only works if the 'code to be executed in mutual exclusion' is non-blocking. If it is blocking, then either the code has to be encapsulated within a task's accept statement (which can block), or the semaphore abstraction (or some other locking mechanism) must be used.
Where code is non-blocking, greater efficiency may be obtained by coding the required synchronisation directly using protected objects. This is because *abstraction inversions* may occur. An abstraction inversion occurs when a low-level synchronisation mechanism is implemented by a higher-level synchronisation mechanism. In the above example, semaphores are low level yet they are being implemented by protected objects, which in effect provide supports for the higher-level monitor abstraction. This higher-level abstraction (monitor) is often implemented in the run-time support system using the same lower-level mechanism (semaphore) that the programmer is trying to support.

11.2.2 Quantity semaphores

Another variation on the normal definition of a semaphore is the *quantity semaphore*. With this primitive, the amount to be decremented by the wait operation (and incremented by the signal operation) is not fixed at one but is given as a parameter to the procedures.

```
package Semaphores.Quantity is
  type Quantity_Signal_Interface is synchronized interface and
                  Signal_Interface;
  procedure Signal(Sem : in out Quantity_Signal_Interface;
                  By : Positive) is abstract;

  type Quantity_Wait_Interface is synchronized interface and
                Wait_Interface;
  procedure Wait(Sem : in out Quantity_Wait_Interface;
                By : Positive) is abstract;
  type Quantity_Semaphore_Interface is synchronized interface and
       Quantity_Signal_Interface and Quantity_Wait_Interface;

  type Any_Quantity_Semaphore_Interface is access all
       Quantity_Semaphore_Interface'Class;
end Semaphores.Quantity;
```

In essence, the protected object implementation of a quantity semaphore is equivalent to the resource control problem discussed in the previous chapter. An efficient implementation, derived from the two-queue algorithm (presented in Section 8.4), can now be given. The two queues are represented as a family with a boolean index. The boolean variable `First_Queue` indicates if the first queue is the currently active one.

```
package Semaphores.Quantity.Passive is
  protected type Quantity_Semaphore (Initial : Natural) is
            new Quantity_Semaphore_Interface with
    overriding entry Wait;
    overriding procedure Signal;
    overriding entry Wait(By : Positive);
    overriding procedure Signal(By : Positive);
  private
    entry Waiting(Boolean)(By : Positive);
    First_Queue : Boolean := True;
    Value : Natural := Initial;
  end Quantity_Semaphore;
end Semaphores.Quantity.Passive;


with Ada.Real_Time; use Ada.Real_time;
package body Semaphores.Quantity.Passive is
  protected body Quantity_Semaphore is
    entry Wait when Value > 0 is
    begin
      Value := Value - 1;
    end Wait;

    entry Wait(By : Positive) when Value > 0 is
    begin
      if By <= Value then
        Value := Value - By;
      else
        requeue Waiting(First_Queue) with abort;
      end if;
    end Wait;

    entry Waiting(for Q in Boolean)(By : Positive)
          when Q xor First_Queue is
    begin
      if By <= Value then
        Value := Value - By;
      else
        requeue Waiting(First_Queue);
      end if;
    end Waiting;

    procedure Signal is
    begin
      Value := Value + 1;
      First_Queue := not First_Queue;
    end Signal;

    procedure Signal(By : Positive) is
    begin
      Value := Value + By;
      First_Queue := not First_Queue;
    end Signal;
  end Quantity_Semaphore;
end Semaphores.Quantity.Passive;
```

This time, the task implementation of the abstraction is more straightforward (in terms of manipulating the guards).

```
package Semaphores.Quantity.Active is
  task type Quantity_Semaphore_With_Deadlock_Prevention (
            Initial : Integer;
            Timeout_In_Milliseconds : Positive) is
            new Quantity_Semaphore_Interface with
    overriding entry Wait;
    overriding entry Signal;
    overriding entry Wait(By : Positive);
    overriding entry Signal(By : Positive);
  private
    entry Waiting(Boolean)(By : Positive);
  end Quantity_Semaphore_With_Deadlock_Prevention;
end Semaphores.Quantity.Active;


package body Semaphores.Quantity.Active is
  task body Quantity_Semaphore_With_Deadlock_Prevention is
    Value : Natural := Initial;
    First_Queue : Boolean := True;
  begin
    loop
      if (Waiting(True)'Count > 0 or Waiting(False)'Count > 0)
                           and Value < Initial then
        select
          when Value > 0 =>
            accept Wait;
            Value := Value - 1;
        or
          accept Wait(By : Positive) do
            if Value >= By then
              Value := Value - By;
            else
              requeue Waiting(First_Queue);
            end if;
          end Wait;
        or
          when First_Queue =>
            accept Waiting(True)(By : Positive) do
              if Value >= By then
                Value := Value - By;
              else
                requeue Waiting(False);
              end if;
            end Waiting;
        or
          when not First_Queue =>
            accept Waiting(False)(By : Positive) do
              if Value >= By then
                Value := Value - By;
              else
                requeue Waiting(True);
              end if;
            end Waiting;
        or
          accept Signal;
          Value := Value + 1;
          First_Queue := not First_Queue;
        or
          accept Signal(By : Positive) do
            Value := Value + By;
            First_Queue := not First_Queue;
          end Signal;
        or
          delay Timeout_In_Milliseconds * 0.001;
          Value := Initial;
          First_Queue := not First_Queue;
        end select;
      else
        select
          when Value > 0 =>
            accept Wait;
            Value := Value - 1;
        or
          accept Wait(By : Positive) do
            if Value >= By then
              Value := Value - By;
            else
              requeue Waiting(First_Queue);
            end if;
          end Wait;
        or
          accept Signal;
          Value := Value + 1;
          First_Queue := not First_Queue;
        or
```

```
          accept Signal(By : Positive) do
            Value := Value + By;
            First_Queue := not First_Queue;
          end Signal;
       or
         when First_Queue =>
           accept Waiting(True)(By : Positive) do
             if Value >= By then
               Value := Value - By;
             else
               requeue Waiting(False);
             end if;
           end Waiting;
       or
         when not First_Queue =>
           accept Waiting(False)(By : Positive) do
             if Value >= By then
               Value := Value - By;
             else
                requeue Waiting(True);
             end if;
           end Waiting;
        or
           terminate;
        end select;
      end if;
    end loop;
  end Quantity_Semaphore_With_Deadlock_Prevention;
end Semaphores.Quantity.Active;
```

Note, that it is difficult to determine exactly what to do when the timeout expires. Here, the semaphore is reinitialised to its default value.

11.3 Locks

One of the main problems with semaphores is that they are not very structured. A general purpose locking mechanism can provide better encapsulation of the code to be executed under exclusion constraints. First, the interface to the locking mechanism is defined. Two types are provided: one which supports mutual exclusion and one that supports read-write locks.

```
package Lock_Mechanisms is
  type Mutex_Lock_Mechanism_Interface is
           synchronized interface;
  procedure Lock(L : in out Mutex_Lock_Mechanism_Interface)
           is abstract;
  procedure Unlock(L : in out Mutex_Lock_Mechanism_Interface)
           is abstract;
  type Any_Mutual_Exclusion_Lock_Mechanism is access all
           Mutex_Lock_Mechanism_Interface'Class;
  type RW_Lock_Mechanism_Interface is synchronized interface;
  procedure Write_Lock(L : in out RW_Lock_Mechanism_Interface)
            is abstract;
  procedure Write_Unlock(L : in out RW_Lock_Mechanism_Interface)
            is abstract;
  procedure Read_Lock(L : in out RW_Lock_Mechanism_Interface)
            is abstract;
  procedure Read_Unlock(L : in out RW_Lock_Mechanism_Interface)
            is abstract;
  type Any_Read_Write_Lock_Mechanism is access all
            RW_Lock_Mechanism_Interface'Class;
end Lock_Mechanisms;
```

They are defined as synchronized interfaces to allow both active and passive mechanisms. Here, only locks based on protected types are considered. First the mechanism for a simple mutual exclusion lock:

```
package Lock_Mechanisms.Simple_Mutual_Exclusion_Locks is
  protected type Simple_Mutex_Lock_Mechanism is new
            Mutex_Lock_Mechanism_Interface with
    overriding entry Lock;
    overriding entry Unlock;
  private
    Locked : Boolean := False;
  end Simple_Mutex_Lock_Mechanism;
end Lock_Mechanisms.Simple_Mutual_Exclusion_Locks;
```

The implementation just uses a boolean variable to indicate if the lock is being held.

```
package body Lock_Mechanisms.Simple_Mutual_Exclusion_Locks is
  protected body Simple_Mutex_Lock_Mechanism is
    entry Lock when not Locked is
    begin
      Locked := True;
    end Lock;

    entry Unlock when Locked is
    begin
      Locked := False;
    end Unlock;
  end Simple_Mutex_Lock_Mechanism;
end Lock_Mechanisms.Simple_Mutual_Exclusion_Locks;
```

The read-write lock mechanism is a little more complex. Here, as multiple readers are allowed, a count of the number of active readers is needed.

```
package Lock_Mechanisms.Simple_Read_Write_Locks is
  protected type Simple_RW_Lock_Mechanism is new
            RW_Lock_Mechanism_Interface with
    overriding entry Write_Lock;
    overriding entry Write_Unlock;
    overriding entry Read_Lock;
    overriding entry Read_Unlock;
  private
    Write_Locked : Boolean := False;
    Read_Count : Integer := 0;
  end Simple_RW_Lock_Mechanism;
end Lock_Mechanisms.Simple_Read_Write_Locks;
```

The implementation uses a combination of the booleans and counts to ensure the execution constraints on the lock and unlock operations are satisfied.

```
package body Lock_Mechanisms.Simple_Read_Write_Locks is
  protected body Simple_RW_Lock_Mechanism is
    entry Write_Lock when not Write_Locked and Read_Count = 0 is
    begin
      Write_Locked := True;
    end Write_Lock;

    entry Write_Unlock when Write_Locked is
    begin
      Write_Locked := False;
    end Write_Unlock;

    entry Read_Lock when not Write_Locked is
    begin
      Read_Count := Read_Count + 1;
    end Read_Lock;

    entry Read_Unlock when Read_Count > 0 is
    begin
      Read_Count := Read_Count - 1;
    end Read_Unlock;
  end Simple_RW_Lock_Mechanism;
end Lock_Mechanisms.Simple_Read_Write_Locks;
```

Having got the basic mechanisms needed to support the locks, the main programmer's API can be defined. Here, it is assumed that the code to be executed whilst holding the lock is encapsulated within a parameterless procedure.

```
package Locks is
  type Parameterless_Procedure is access procedure;

  type Mutex_Lock_Interface is limited interface;
  procedure Lock_And_Execute(
            L : in out Mutex_Lock_Interface;
            This : Parameterless_Procedure) is abstract;
  type Any_Mutex_Lock is access all
       Mutex_Lock_Interface'Class;

  type RW_Lock_Interface is limited interface;
  procedure Write_Lock_And_Execute(
            L : in out RW_Lock_Interface;
            This: Parameterless_Procedure) is abstract;
  procedure Read_Lock_And_Execute(
            L : in out  RW_Lock_Interface;
            This : Parameterless_Procedure) is abstract;
  type Any_RW_Lock is access all RW_Lock_Interface'Class;
end Locks;
```

The main challenge in implementing this interface is to ensure that errors that occur whilst executing the parameterless procedure do not result in the lock becoming deadlocked. Consider the following specification that is going to provide passive locks.

```
with Lock_Mechanisms.Simple_Mutual_Exclusion_Locks;
use Lock_Mechanisms.Simple_Mutual_Exclusion_Locks;
with Lock_Mechanisms.Simple_Read_Write_Locks;
use Lock_Mechanisms.Simple_Read_Write_Locks;
package Locks.Passive_Locks is
  type Passive_Mutex_Lock is limited new
    Mutex_Lock_Interface with private;
  overriding procedure Lock_And_Execute(
    L : in out Passive_Mutex_Lock; This : Parameterless_Procedure);

  type Passive_RW_Lock is limited new
    RW_Lock_Interface with private;
  overriding procedure Write_Lock_And_Execute(
    L : in out Passive_RW_Lock; This : Parameterless_Procedure);
  overriding procedure Read_Lock_And_Execute(
    L : in out Passive_RW_Lock; This : Parameterless_Procedure);
private
   type Passive_Mutex_Lock is limited new
            Mutex_Lock_Interface with
   record
     The_Lock : Simple_Mutex_Lock_Mechanism;
   end record;

   type Passive_RW_Lock is limited new
            RW_Lock_Interface with
   record
     The_Lock : Simple_RW_Lock_Mechanism;
   end record;
end Locks.Passive_Locks;
```

The private parts of the data structures contain the actual locks to be used. For example, the `Passive_Mutual_Exclusion_Lock` contains a `Simple_Mutual_Exclusion_Lock_Mechanism`. Now the `Lock_And_Execute` procedure can surround calls to the parameterless procedure with calls to lock and unlock. Unfortunately, this simple approach would not be robust in the face of unhandled exceptions, ATCs and aborts. To make it robust requires the use of Ada's finalization mechanism. Consider the following.

```
with Ada.Finalization; use Ada.Finalization;
package Locks.Passive_Locks.Deadlock_Controls is
  type Mutex_Lock_Deadlock_Control(
       Lock : access Passive_Mutex_Lock) is new
                  Limited_Controlled with null record;
  overriding procedure Initialize(
       Object : in out Mutex_Lock_Deadlock_Control);
  overriding procedure Finalize(
       Object : in out Mutex_Lock_Deadlock_Control);

  type Read_Lock_Deadlock_Control
       (Lock : access Passive_RW_Lock) is new
                   Limited_Controlled with null record;
  overriding procedure Initialize(
             Object : in out Read_Lock_Deadlock_Control);
  overriding procedure Finalize(
             Object : in out Read_Lock_Deadlock_Control);

  type Write_Lock_Deadlock_Control(
       Lock : access Passive_RW_Lock) is new
       Limited_Controlled with null record;
  overriding procedure Initialize(
       Object : in out Write_Lock_Deadlock_Control);
  overriding procedure Finalize(
       Object : in out Write_Lock_Deadlock_Control);
end Locks.Passive_Locks.Deadlock_Controls;
```

The above package contains some types that are extensions of Ada's `Limited_Controlled`, which can be used to define initialisers and finalisers. They take as discriminants the data structures containing the locks. Note that as the package is a child package it can access the private types of the parent. The body of the `Initialize` procedures will obtain the required locks, and the `Finalize` procedures will release the required locks.

```
package body Locks.Passive_Locks.Deadlock_Controls is
  procedure Initialize(
    Object : in out Mutex_Lock_Deadlock_Control) is
  begin
    Object.Lock.The_Lock.Lock;
  end Initialize;

  procedure Finalize(
    Object : in out Mutex_Lock_Deadlock_Control) is
  begin
    Object.Lock.The_Lock.Unlock;
  end Finalize;

  procedure Initialize(
    Object : in out Write_Lock_Deadlock_Control) is
  begin
    Object.Lock.The_Lock.Write_Lock;
  end Initialize;
  procedure Finalize(
    Object : in out Write_Lock_Deadlock_Control) is
  begin
    Object.Lock.The_Lock.Write_Unlock;
  end Finalize;

  procedure Initialize(
    Object : in out Read_Lock_Deadlock_Control) is
  begin
    Object.Lock.The_Lock.Read_Lock;
  end Initialize;

  procedure Finalize(
    Object : in out Read_Lock_Deadlock_Control) is
  begin
    Object.Lock.The_Lock.Read_Unlock;
  end Finalize;
end Locks.Passive_Locks.Deadlock_Controls;
```

Now the body of the `Locks.Passive_Locks` can be written:

```
with Locks.Passive_Locks.Deadlock_Controls;
use  Locks.Passive_Locks.Deadlock_Controls;
package body Locks.Passive_Locks is
  procedure Lock_And_Execute(L : in out Passive_Mutex_Lock;
            This : Parameterless_Procedure) is
    Ensure_Locking : Mutex_Lock_Deadlock_Control(
                   L'Unchecked_Access);
  begin
    This.all;
  end Lock_And_Execute;

  procedure Write_Lock_And_Execute(L : in out Passive_RW_Lock;
    This : Parameterless_Procedure) is
    Ensure_Lock : Write_Lock_Deadlock_Control(L'Unchecked_Access);
  begin
    This.all;
  end Write_Lock_And_Execute;

  procedure Read_Lock_And_Execute(L : in out Passive_RW_Lock;
    This : Parameterless_Procedure) is
    Ensure_Lock : Read_Lock_Deadlock_Control(L'Unchecked_Access);
  begin
    This.all;
  end Read_Lock_And_Execute;
end Locks.Passive_Locks;
```

Note that each of the ‘lock and execute’ procedures declares a variable of a type that extends the `Limited_Controlled` type. When the procedure is called, the `Initialize` method for that variable will automatically be called which will get the appropriate lock. Similarly, when the procedure returns (irrespective of whether it is a normal or an abnormal return), the `Finalize` procedure will be called that will release the appropriate lock.

Although the above approach allows deadlocks to be avoided, it does not protect the integrity of any resources manipulated from the parameterless procedure. This is the programmer's responsibility.

11.4 Signals

In the previous chapter, signals were used as examples for explaining protected interfaces. The package specifications are repeated here for completeness.

```
package Signals is
  type Signal_Sender is protected interface;
  procedure Send(S : in out Signal_Sender) is abstract;
  type Any_Signal_Sender is access all Signal_Sender'Class;

  type Signal_Waiter is protected interface;
  procedure Wait(S : in out Signal_Waiter) is abstract;
  type Any_Signal_Waiter is access all Signal_Waiter'Class;
end Signals;
```

11.4.1 Transient signals

A transient signal (or pulse) is a signal which releases one or more waiting tasks but is lost if no tasks are waiting.

```
package Signals.Transient_Signals is
  protected type Transient_Signal is new Signal_Sender and
            Signal_Waiter with
    overriding procedure Send;
    overriding entry Wait;
  private
    Arrived : Boolean := False;
  end Transient_Signal;
  type Transient_Signal_Access is access all Transient_Signal;

  protected type Pulse is new Signal_Sender and Signal_Waiter with
    overriding procedure Send;
    overriding entry Wait;
  private
    Arrived : Boolean := False;
  end Pulse;
  type Pulse_Signal_Access is access all Pulse;
end Signals.Transient_Signals;
```

Warning: Transient signals inevitably suffer from race conditions, as a task that calls the wait operation just after the signal has been sent will have to wait for the next signal. Also, if the sender fails, the signal may never be sent. Timeout can be used by the receiver to help alleviate this problem.

11.4.2 Persistent signals

A persistent signal is a signal which remains set until a single task has received it. It, therefore, allows better control over the race conditions associated with transient signals. However, timeouts are still required to avoid deadlocks as a result of the failure of the sending task.

```
package Signals.Persistent_Signals is
  protected type Persistent_Signal is new Signal_Sender and
            Signal_Waiter with
    overriding procedure Send;
    overriding entry Wait;
  private
    Arrived : Boolean := False;
  end Persistent_Signal;
  type Persistent_Signal_Access is access all Persistent_Signal;
end Signals.Persistent_Signals;
```

11.5 Event variables

As previously mentioned, an event variable is a bivalued variable (up or down). A task can wait for an event to be set or reset. The following package specifies the interface:

```
package Events is
  type Event_State is (Up, Down);

  type Event_Interface is synchronized interface;
  procedure Set(E : in out Event_Interface) is abstract;
  procedure Reset(E : in out Event_Interface) is abstract;
  procedure Toggle(E : in out Event_Interface) is abstract;
  function State(E : in Event_Interface)
           return Event_State is abstract;
  procedure Wait(E : in out Event_Interface;
                 S : Event_State) is abstract;
  type Any_Event is access all Event_Interface'Class;

  protected type Event(Initial : Event_State) is
                 new Event_Interface with
    overriding procedure Set;
    overriding procedure Reset;
    overriding procedure Toggle;
    overriding function State return Event_State;
    overriding entry Wait(S : Event_State);
  private
    entry Wait_Up(S : Event_State);
    entry Wait_Down(S : Event_State);
    Value : Event_State := Initial;
  end Event;
end Events;
```

Events variables may be created using the protected type `Event`. The procedure `Set` causes the event variable to go into the `Up` state; the procedure `Reset` causes it to go into the `Down` state. The `Toggle` procedure simply changes the state of the event variable from `Up` to `Down`, or from `Down` to `Up`. The function `State` returns the current state of the event variable.

Synchronisation with an event variable is achieved using the `Wait` entry. This will suspend the caller until the state of the event variable is that required by the caller. Two private entries are used to help implement the required synchronisation, as shown in the package body below:

```
package body Events is
  protected body Event is
    procedure Set is
    begin
      Value := Up;
    end Set;

    procedure Reset is
    begin
      Value := Down;
    end Reset;

    procedure Toggle is
    begin
      if Value = Down then
        Value := Up;
      else
        Value := Down;
      end if;
    end Toggle;

    function State return Event_State is
    begin
      return Value;
    end State;

    entry Wait(S : Event_State) when True is
    begin
      if S = Up then
        requeue Wait_Up with abort;
      else
        requeue Wait_Down with abort;
      end if;
    end Wait;

    entry Wait_Up(S : Event_State) when Value = Up is
    begin
      null;
    end Wait_Up;
    entry Wait_Down(S : Event_State) when Value = Down is
    begin
      null;
    end Wait_Down;
  end Event;
end Events;
```

11.6 Buffers

The basic bounded buffer has already been described in Section 7.3; a generic version is presented here:

```
generic
  type Element is private;
package Buffers is
  type Buffer is synchronized interface;
  type Any_Buffer is access all Buffer'Class;
  procedure Put(Buf : in out Buffer; Item : in Element)
            is abstract;
  procedure Get(Buf : in out Buffer; Item : out Element)
            is abstract;
end Buffers;
```

As with semaphores, both passive and active components are appropriate.

```
with Buffers;
generic
package Buffers.Mailboxes is
  type Buffer_Store is array (Positive range <>) of Element;

  protected type Mailbox(Max_Capacity : Positive) is
                 new Buffer with
    overriding entry Put(Item : in Element);
    overriding entry Get(Item : out Element);
  private
    First : Positive := 1;
    Last : Positive := Max_Capacity;
    Number_In_Buffer : Natural := 0;
    Box_Store : Buffer_Store(1..Max_Capacity);
  end Mailbox;
end Buffers.Mailboxes;

package body Buffers.Mailboxes is
  protected body Mailbox is
    entry Get(Item : out Element)
              when Number_In_Buffer /= 0 is
    begin
      Item := Box_Store(First);
      First := First mod Max_Capacity + 1;
      Number_In_Buffer := Number_In_Buffer - 1;
    end Get;
    entry Put(Item : in Element)
          when Number_In_Buffer /= Max_Capacity is
    begin
      Last := Last mod Max_Capacity + 1;
      Box_Store(Last) := Item;
      Number_In_Buffer := Number_In_Buffer + 1;
    end Put;

    function Buffer_Capacity return Positive is
    begin
      return Max_Capacity;
    end Buffer_Capacity;
  end Mailbox;
end Buffers.Mailboxes;


with Buffers;
generic
package Buffers.Active_Buffers is
  task type Active_Buffer is new Buffer with
    overriding entry Put(Item : in Element);
    overriding entry Get(Item : out Element);
  end Active_Buffer;
end Buffers.Active_Buffers;


package body Buffers.Active_Buffers is
  subtype Capacity_Range is Natural range 0 .. Max_Capacity - 1;
  subtype Count is Integer range 0 .. Max_Capacity;
  type Buffer_Store is array(Capacity_Range) of Element;

  task body Active_Buffer is
    First : Capacity_Range := Capacity_Range'First;
    Last : Capacity_Range := Capacity_Range'Last;
    Number_In_Buffer : Count := 0;
    Box_Store : Buffer_Store;
  begin
    loop
      select
        when Number_In_Buffer /= Max_Capacity =>
          accept Put(Item : in Element) do
            Last := (Last + 1) mod Max_Capacity;
            Box_Store(Last) := Item;
            Number_In_Buffer := Number_In_Buffer + 1;
          end Put;
      or
        when Number_In_Buffer /= 0 =>
          accept Get(Item : out Element) do
            Item := Box_Store(First);
            First := (First + 1) mod Max_Capacity;
            Number_In_Buffer := Number_In_Buffer - 1;
          end Get;
      or
```

```
        terminate;
      end select;
      -- log activity in database
    end loop;
  end Active_Buffer;
end Buffers.Active_Buffers;
```

The buffer abstraction is one in which the data, once read, is destroyed. If the data is to be retained, then the blackboard abstraction is more appropriate.

11.7 Blackboards

The blackboard abstraction can be viewed as being similar to either the events abstraction with data transfer, or the buffer abstraction with a non-destructive read and the facility to invalidate the data. Each notional item of the buffer is represented as a single blackboard:

```
generic
  type Message is private;
package Blackboards is
  type Blackboard_State is (Valid, Invalid);

  type Reader_Interface is synchronized interface;
  function State (RI : Reader_Interface)
                 return Blackboard_State is abstract;
  procedure Read(RI : in out Reader_Interface;
                M : out Message) is abstract;
  type Any_Blackboard_Reader_Interface is
                access all Reader_Interface'Class;

  type Writer_Interface is synchronized interface;
  function State (WI : Writer_Interface)
                 return Blackboard_State is abstract;
  procedure Write(WI : in out Writer_Interface;
                 M : Message) is abstract;
  procedure Clear (WI : in out Writer_Interface) is abstract;
  type Any_Blackboard_Writer_Interface is
       access all Writer_Interface'Class;

  type Blackboard_Interface is synchronized interface and
       Reader_Interface and Writer_Interface;
  type Any_Blackboard is access all Blackboard_Interface'Class;

  protected type Blackboard is new Blackboard_Interface with
    overriding procedure Write(M : Message);
    overriding procedure Clear;
    overriding function State return Blackboard_State;
    overriding entry Read(M : out Message);
  private
    The_Message : Message;
    Status : Blackboard_State := Invalid;
  end Blackboard;
end Blackboards;
```

Items are placed on a `Blackboard` by calling `Write`; they are deleted by calling `Clear`. The entry `Read` will block the caller until data on the blackboard is valid (that is, there is data present). The function `State` indicates whether the blackboard currently has data.

The body of the package follows:

```
package body Blackboards is
  protected body Blackboard is
    procedure Write(M : Message) is
    begin
      The_Message := M;
      Status := Valid;
    end Write;
    procedure Clear is
    begin
      Status := Invalid;
    end Clear;
    function State return Blackboard_State is
    begin
      return Status;
    end State;
    entry Read (M : out Message) when Status = Valid is
    begin
      M := The_Message;
    end Read;
  end Blackboard;
end Blackboards;
```

A simpler form of blackboard does not have a clear operation; all data is preserved until overwritten. `Read` would then be non-blocking and hence could be implemented as a subprogram (assuming that the backboard is initialised to some appropriate value).

11.8 Broadcasts

The use of a broadcast facility was shown in Section 7.7 Here the facility is generalised and the implementation is shown (which is similar in structure to a transient signal with `Send_All`):

```
generic
 type Message is private;
package Broadcasts is
  type Sender_Interface is synchronized interface;
  procedure Send(SI : in out Sender_Interface;
                 This_Message : in Message) is abstract;
  type Any_Broadcast_Send is access all Sender_Interface'Class;
  type Receiver_Interface is synchronized interface;
  procedure Receive(RI : in out Receiver_Interface;
                    A_Message : out Message) is abstract;
  type Any_Broadcast_Receiver is access all
      Receiver_Interface'Class;

  type Broadcast_Interface is synchronized interface
       and Sender_Interface and Receiver_Interface;
  type Any_Broadcast is access all Broadcast_Interface'Class;

  protected type Broadcast is new Broadcast_Interface with
    overriding procedure Send(This_Message : Message);
    overriding entry Receive(A_Message : out Message);
  private
    Message_Arrived : Boolean := False;
    The_Message : Message;
  end Broadcast;
end Broadcasts;
```

The generic package allows any type of message to be sent in the broadcast. Only tasks that are waiting will receive the message.

```
package body Broadcasts is
  protected body Broadcast is
    procedure Send(This_Message : Message) is
    begin
      if Broadcast.Receive'Count > 0 then
        Message_Arrived := True;
        The_Message := This_Message;
      end if;
    end Send;

    entry Receive(A_Message : out Message)
       when Message_Arrived is
    begin
      if Broadcast.Receive'Count = 0 then
        Message_Arrived := False;
      end if;
      A_Message := The_Message;
    end Receive;
  end Broadcast;
end Broadcasts;
```

11.8.1 Multicast to a group

The term, broadcast, has so far been used to indicate that the data should be sent to any task that is waiting. Often the term, broadcast (or, more correctly, 'multicast'), is used to indicate that the data should be sent to a specific group of tasks. In this situation, *all* tasks in the group should receive the data, not just those that happen to be waiting when the data is sent. This is slightly more difficult to achieve; all potential recipients must be known (say via their task identifiers) and only when all have received one item of data is another item allowed to be transmitted.

The following package specification defines the multicast interface. Tasks which are interested in receiving from a group must join the group explicitly and when they are no longer interested they must leave the group:

```
with Ada.Task_Identification; use Ada.Task_Identification;
generic
  type Message is private;
package Multicasts is
  Max_Group_Size : constant Positive := ...;
    --set to some appropriate value
  type Group_Range is range 1 .. Max_Group_Size;
  Default_Group_Size : constant Group_Range := ...;
    --set to some appropriate value

  type Multicast_Interface is synchronized interface;
  procedure Join_Group(MI : in out Multicast_Interface)
            is abstract;
  procedure Leave_Group(MI : in out Multicast_Interface)
            is abstract;
  procedure Send(MI : in out Multicast_Interface;
                 This_Message : Message) is abstract;
  procedure Receive(MI : in out Multicast_Interface;
                    A_Message : out Message) is abstract;
  type Any_Multicast is access all Multicast_Interface'Class;

  type Task_Status is private;
  type Group_Tasks is array(Group_Range range <>) of Task_Status;

  Already_Member, Not_Member, Group_Full, Group_Empty : exception;

  protected type Multicast(Max_Size : Group_Range) is
           new Multicast_Interface  with
    overriding procedure Join_Group;
      -- raises Already_Member, Group_Too_Large
    overriding procedure Leave_Group;
      -- raises Not_Member
    overriding entry Send(This_Message : Message);
      -- raises Group_Empty (or ignore Send if
      -- empty group is acceptable)
    overriding entry Receive(A_Message : out Message);
      -- raises Not_Member
  private
    entry Wait_Next_Message(A_Message : out Message);
    Message_Available : Boolean := False;
    New_Message_Arrived : Boolean := False;
    The_Message : Message;
    Ok_To_Send : Boolean := True;
    Group : Group_Tasks(Group_Range'Range);
  end Multicast;
```

```
private
  type Task_Status is
    record
      Id : Task_Id := Null_Task_Id;
      Received : Boolean := False;
    end record;
end Multicasts;
```

Inside the body of the `Multicasts` package, there are various housekeeping subprograms which keep track of which tasks have received which messages.

Note the use of task identifiers to identify the calling task uniquely:

```
package body Multicasts is
  protected body Multicast is
    function All_Received return Boolean is
    begin
      -- check if all tasks in the
      -- Group have received the data
      for I in Group'Range loop
        if Group(I).Id /= Null_Task_Id and then
                    not Group(I).Received then
          return False;
        end if;
      end loop;
      return True;
    end All_Received;

    function Already_Received(Id : Task_Id) return Boolean is
    begin
      -- check if task has received the data
      for I in Group'Range loop
        if Group(I).Id = Id then
          if Group(I).Received then
            return True;
          else
            return false;
          end if;
        end if;
      end loop;
      raise Not_Member;
    end Already_Received;

    procedure Log_Received(Id : Task_Id) is
    begin
      -- log that task has received the data
      for I in Group'Range loop
        if Group(I).Id = Id then
          if Group(I).Received then
            raise Program_Error;
          else
            Group(I).Received := True;
            return;
          end if;
        end if;
      end loop;
      raise Not_Member;
    end Log_Received;

    procedure Clear_Received is
    begin
      -- set all Boolean flags in Group array to False
      for I in Group'Range loop
        Group(I).Received  := False;
      end loop;
    end Clear_Received;

    procedure In_Group(Yes : out Boolean;
                            Index : out Group_Range) is
    begin
      for I in Group'Range loop
        if Group(I).Id = Current_Task then
          Yes := True;
          Index := I;
          return;
        end if;
      end loop;
      Yes := False;
    end In_Group;

    procedure Find_Empty(Found : out Boolean;
                         Slot : out Group_Range) is
    begin
      for I in Group'Range loop
        if Group(I).Id = Null_Task_Id then
          Slot := I;
          Found := True;
          return;
        end if;
      end loop;
      Found := False;
    end Find_Empty;

    procedure Join_Group is
      I : Group_Range;
      Found : Boolean;
    begin
      -- save ID of Current_Task in Group array
      In_Group(Found, I);
      if Found then raise Already_Member; end if;
      Find_Empty(Found, I);
      if not Found then raise Group_Full; end if;
      Group(I).Id := Current_Task;
    end Join_Group;

    procedure Leave_Group is
      I : Group_Range;
      Found : Boolean;
    begin
      -- delete ID of Current_Task in Group array
      In_Group(Found, I);
      if not Found then raise Not_Member; end if;
      Group(I).id := Null_Task_ID;
      if Message_Available and All_Received then
        Message_Available := False;
        Ok_To_Send := True;
        Clear_Received;
      end if;
    end Leave_Group;

    entry Send(This_Message : Message) when Ok_To_Send is
    begin
      The_Message := This_Message;
      Ok_To_Send := False;
      New_Message_Arrived := True;
      Log_Received(Multicast.Send'Caller);
          -- sender automatically receives
    end Send;

    entry Receive(A_Message : out Message) when True is
      Id : Task_Id :=  Multicast.Receive'Caller;
      I : Group_Range;
      Found : Boolean;
    begin
      -- if not member raise exception
      In_Group(Found, I);
      if not Found then
         raise Not_Member;
      end if;
      if Message_Available then
        if Already_Received(Id) then
          requeue Wait_Next_Message with abort;
        else
          Log_Received(Id);
        end if;
        A_Message := The_Message;
        if All_Received then
          Message_Available := False;
          Ok_To_Send := True;
          Clear_Received;
        end if;
      else
        requeue Wait_Next_Message with abort;
      end if;
    end Receive;

    entry Wait_Next_Message(A_Message : out Message)
          when New_Message_Arrived is
      Id : Task_Id := Wait_Next_Message'Caller;
    begin
```

```
      Log_Received(Id);
      A_Message := The_Message;
      if All_Received then
        Ok_To_Send := True;
        New_Message_Arrived := False;
        Clear_Received;
      elsif Wait_Next_Message'Count=0 then
        New_Message_Arrived := False;
        Message_Available := True;
      else
        null;
      end if;
    end Wait_Next_Message;
  end Multicast;
end Multicasts;
```

In the above algorithm, it is not possible to send another multicast message until the previous message has been received by everyone. Compare this with the previous algorithm, where only those waiting received the message. There is clearly a solution in between these, where the message remains available for a certain period of time or where more than one message is buffered; this is left as an exercise for the reader.

A task wishing to receive a broadcast message must first join the group. Assume that a group has already been created, for example

```
-- package to multicast the current aircraft altitude
-- to all interested parties
package Altitude_Multicasts is new Multicasts(Altitude);
```

Assume now that a task, which controls some aspect of an aircraft, is implemented so as to react to any new altitude setting multicast from some sensor reading routine. It might take the following form:

```
with Altitude_Multicasts; use Altitude_Multicasts;
  ...
  Barometric_Pressure_Reading : Multicast;
  ...
  task Surface_Controller;

  task body Surface_Controller is
    -- declaration of Current_Altitude etc
  begin
    ...
    Barometric_Pressure_Reading.Join_Group;
    loop
      Barometric_Pressure_Reading.Receive(Current_Altitude);
      -- use Current_Altitude
    end loop;
    Barometric_Pressure_Reading.Leave_Group;
  end Surface_Controller;
```

11.9 Barriers

A barrier simply blocks several tasks, until all have arrived. In this case, no data is passed but a form of multicast could be programmed that passes data as well. Tasks wishing to block at a barrier simply call the `Wait` subprogram; the two functions return information about the barrier:

```
package Barriers is
  Max_Capacity : constant Positive := ...;
    -- set to an appropriate value
  type Capacity_Range is range 1 .. Max_Capacity;

  type Barrier_Interface is synchronized interface;
  procedure Wait(BI : in out Barrier_Interface) is abstract;
  function Capacity(BI : Barrier_Interface)
           return Capacity_Range is abstract;
  function Value(BI : Barrier_Interface)
           return Capacity_Range is abstract;

  type Any_Barrier is access all Barrier_Interface'Class;

  protected type Barrier(Needed : Capacity_Range) is
            new Barrier_Interface with
    entry Wait;
    function Capacity return Capacity_Range;
    function Value return Capacity_Range;
  private
    Release_All : Boolean := False;
  end Barrier;
end Barriers;


package body Barriers is
  protected body Barrier is
    entry Wait when Barrier.Wait'Count = Needed or
                          Release_All is
    begin
      Release_All :=  Barrier.Wait'Count > 0;
    end Wait;

    function Capacity return Capacity_Range is
    begin
      return Needed;
    end Capacity;

    function Value return Capacity_Range is
    begin
      return Needed - Barrier.Wait'Count;
    end Value;
  end Barrier;
end Barriers;
```

11.10 Concurrent execution abstractions

This section illustrates how concurrent execution abstractions can be created in Ada 2005. The focus is on the invocation, execution and control of asynchronous activities rather than the scheduling aspects. Real-time utilities will be consider later in Chapter 16. The goal here is to generalise the approach defined in section 10.2 for creating concurrent execution engines.

The following packages are defined in this section.

- `Callables` – A generic package that defines a `Callable` interface; code that is to be executed concurrently must support this interface.
- `Futures` – All values returned from concurrent activities are stored in objects for later access. These objects must support one of the interfaces defined in this package. Two types of futures are defined: one that can be reused and one that is not.
- `Default_Reusable_Futures` – This package defines a protected type that provides the default implementation of a reusable future.
- `Executors` – This package provides the interface for all approaches (execution engines) that support concurrent execution of one form or another.
- `Thread_Pool_Executors` – This package provides a pool of tasks to execute the concurrent activities.
- `Executor.Completion_Services` – This is a package that defines an interface for an executor that also manages the futures such that applications can just obtain the next completed future.
- `Completion_Pool_Executors` – This package provides a pool of tasks that execute the concurrent activities and manages the returned futures.

Figure 11.1 illustrates the packages that support the concurrent execution utilities.

From a user's perspective, the code that they wish to have executed concurrently must be represented as an Ada object that implements the `Callable` interface defined in the `Callables` package. They then must decide which execution engine to use (either a thread pool or a completion pool as defined in this chapter). They then have to provide an appropriate container (called a future in this chapter) in which the execution engine can store the result for later access by the application. Following this, the user must instantiate the appropriate generic packages to implement the chosen execution engine. A full example is given in Section 11.14.

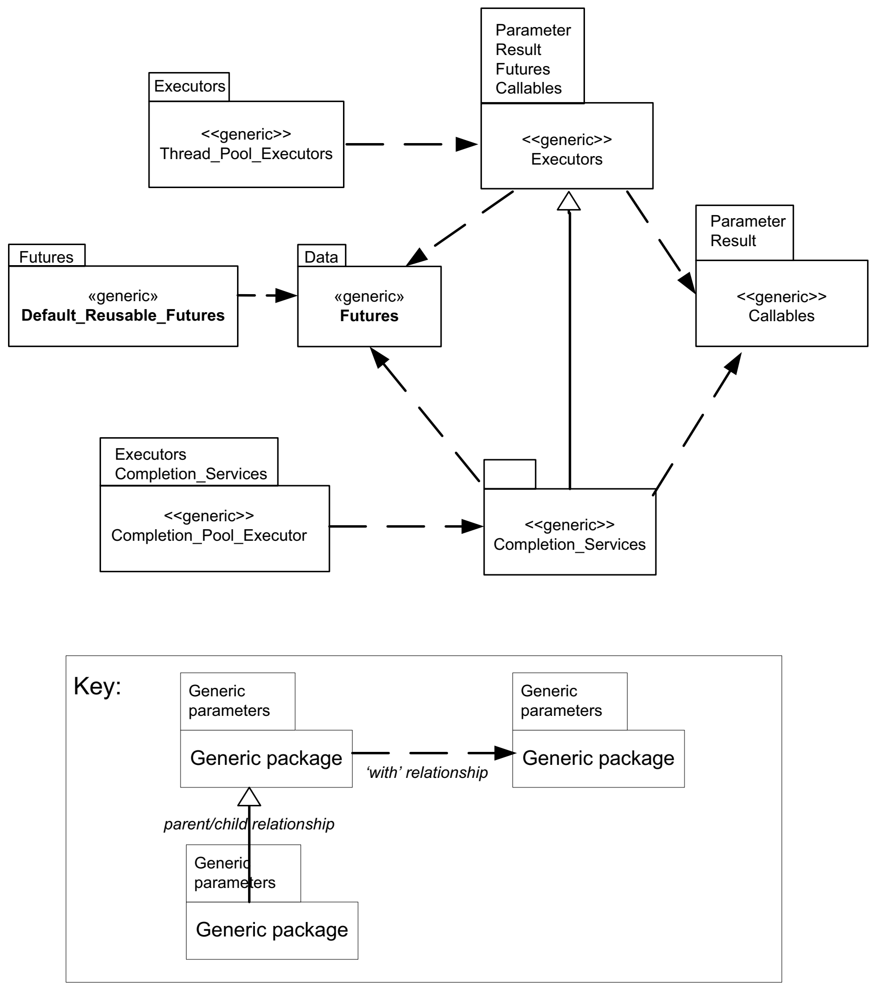

Executors
<<generic>> Thread_Pool_Executors
Parameter Result Futures Callables
<<generic>> Executors
Futures
«generic» Default_Reusable_Futures
Data
«generic» Futures
Parameter Result
<<generic>> Callables
Executors Completion_Services
<<generic>> Completion_Pool_Executor
<<generic>> Completion_Services
Key:
Generic parameters
Generic package
'with' relationship
Generic parameters
Generic package
parent/child relationship
Generic parameters
Generic package

Fig. 11.1: Packages to support concurrent execution utilities

11.11 Callables and futures

For generality, it is assumed that the code to be executed concurrently takes a parameter and returns a result. The following generic package, declaring a `Callable` interface, is used to define the signature of the code.

```
generic
 type Parameter is limited private;
 type Result is limited private;
package Callables is
  type Callable is interface;
  function Call(C : Callable; P : Parameter) return Result
           is abstract;
  type Any_Callable is access all Callable'Class;
end Callables;
```

The return value is called a **future** and is stored in a synchronized type for later access. There are two types of futures: ones that cannot be reused (`Future`) and ones that can (`Reusable_Future`).

```
generic
  type Data is private;
package Futures is
  type Future is synchronized interface;
  procedure Get(F : in out Future; D : out Data) is abstract;
  procedure Set(F : in out Future; D : Data) is abstract;
  type Any_Future is access all Future'Class;

  type Reusable_Future is synchronized interface and Future;
  procedure Allocate(F : in out Reusable_Future) is abstract;
  procedure Cancel(F : in out Reusable_Future) is abstract;
  overriding procedure Get(F : in out Reusable_Future;
                           D : out Data) is abstract;
  overriding procedure Set(F : in out Reusable_Future;
                           D : Data) is abstract;
  type Any_Reusable_Future is access all Reusable_Future'Class;
end Futures;
```

A default reusable future illustrates how futures can be implemented as protected types.

```
with Futures;
generic
  with package My_Futures is new Futures(<>);
  use My_Futures;
package Default_Reusable_Futures is
  protected type Default_Future is new Reusable_Future with
    overriding entry Get(D : out Data);
    overriding entry Set(D : Data);
    overriding entry Cancel;
    overriding entry Allocate;
  private
    Allocated : Boolean := False;
    Value : Data;
    Valid : Boolean := False;
  end Default_Future;
end Default_Reusable_Futures;
```

A `Default_Future` is initially not allocated and the data is not valid. To set the data in the future, it first has to be allocated. Once allocated, the data can be set. When the data is retrieved (via `Get`), the future is automatically available for reallocation. `Cancel` allows a future, which contains data that is no longer required, to be reset.

```
package body Default_Reusable_Futures is
  protected body Default_Future is
    entry Get(D : out Data) when Valid and Allocated is
    begin
      D := Value;
      Valid := False;
      Allocated := False;
    end Get;

    entry Set(D : Data) when not Valid and Allocated is
    begin
      Value := D;
      Valid := True;
    end Set;

    entry Cancel when Allocated and Valid is
    begin
      Valid := False;
      Allocated := False;
    end Cancel;

    entry Allocate when not Allocated is
    begin
      Allocated := True;
      Valid := False;
    end Allocate;
  end Default_Future;
end Default_Reusable_Futures;
```

Figure 11.2 illustrates the hierarchy of future types.

11.12 Executors

There are many different approaches to achieving concurrent execution. Here, they are encapsulated by the `Executor` generic interface. As well as being parameterised with the data type, the generic also has generic package parameters to allow access to the associated `Callables` and `Futures`.

```
with Callables;
with Futures;
generic
  type Parameter is private;
  type Result is private;
  with package My_Futures is new Futures(Result);
  with package My_Callables is new Callables(Parameter, Result);
package Executors is
  type Executor is limited interface;
  procedure Submit(Exec : in out Executor;
            Command : My_Callables.Any_Callable;
            Params : Parameter;
            My_Future : My_Futures.Any_Future)
            is abstract;
  type Any_Executor is access all Executor'Class;
end Executors;
```

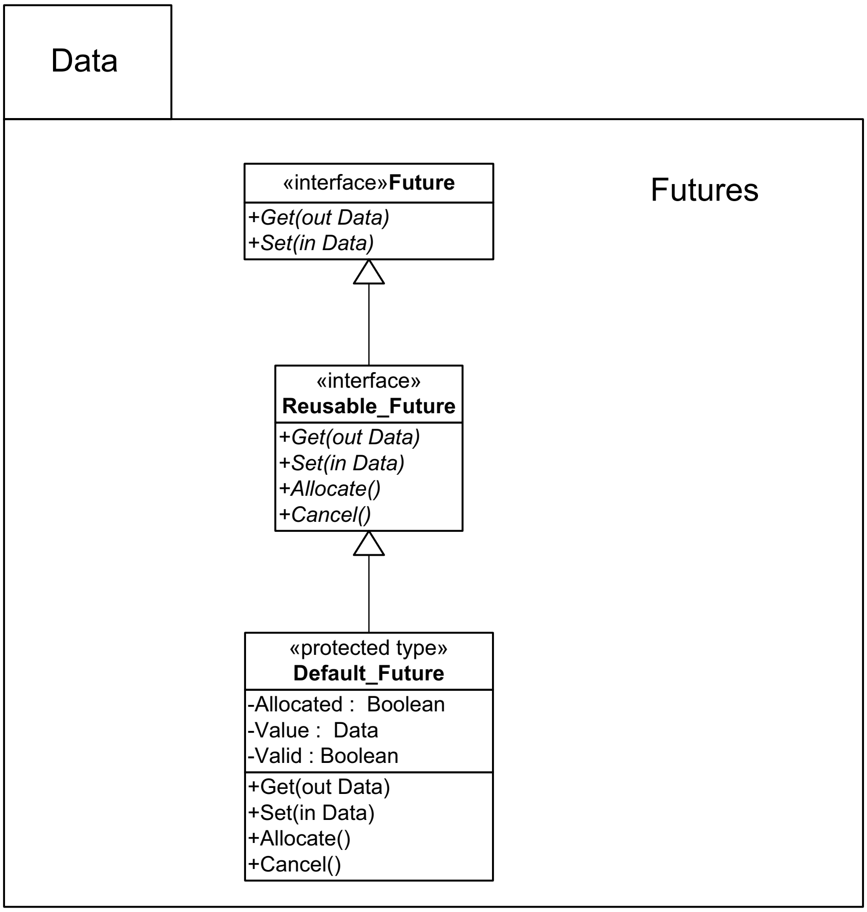


Fig. 11.2: Packages to support concurrent execution utilities

A call to the `Submit` procedure will execute the `Command` at some time in the future. No commitment is made as to whether it will execute in a new task, in a pooled task or even in the called task. A `Future` is provided by the client indicating where the result should be placed.

As an example of an executor, consider one that provides a thread pool to execute the commands.

```
with Ada.Finalization; use Ada.Finalization;
with Executors;
with Buffers, Buffers.Mailboxes;
generic
  with package My_Executors is new Executors(<>);
  use My_Executors;
package Thread_Pool_Executors is
  type Thread_Pool_Executor(Pool_Size : Positive) is new
        Limited_Controlled and Executor with private;
  overriding procedure Submit(
            Exec : in out Thread_Pool_Executor;
            Command : My_Callables.Any_Callable;
            Params : Parameter;
            My_Future : in  My_Futures.Any_Future);
  type Any_Thread_Pool_Executor is access all
       Thread_Pool_Executor'Class;
private
  procedure Initialize(TPE : in out Thread_Pool_Executor);
  package My_Buffers is new Buffers(Positive);
  package My_Mailboxes is new My_Buffers.Mailboxes;
  use My_Mailboxes;

  task type Worker is
     entry Fork(This : My_Callables.Any_Callable;
          Params : Parameter; Result : My_Futures.Any_Future);
  end Worker;

  type My_Worker_Pool is array (Positive range <>) of Worker;

  type Thread_Pool_Executor(Pool_Size : Positive) is
       new Limited_Controlled and Executor with record
    Controller : MailBox(Pool_Size);
    Pool : My_Worker_Pool(1 .. Pool_Size);
  end record;
end Thread_Pool_Executors;
```

The `Thread_Pool_Executor` is an extension of Ada's `Limited_Controlled` type and it implements the `Executor` interface. The size of the pool is a discriminant of the type. For simplicity, a client task is held if there are no workers currently available in the thread pool.

The private part of the package defines an array of worker tasks, along with a mailbox implementation of a bounded buffer (that is used to control access to the worker tasks). These are encapsulated inside the `Thread_Pool_Executor` tagged type. The body of the package shows how objects of this type are initialised using the `Initialize` procedure. Note the use of task attributes to allow the worker tasks to be told which thread pool they are associated with.

```
with Ada.Task_Attributes;
with Default_Reusable_Futures;
package body Thread_Pool_Executors is
  package My_Attributes is new Ada.Task_Attributes(
          Any_Thread_Pool_Executor, null);
  package My_Default_Futures is new Default_Reusable_Futures(
          My_Futures);

  procedure Initialize(TPE : in out Thread_Pool_Executor) is
    use My_Futures;
  begin
    for I in TPE.Pool'Range loop
```

```
      My_Attributes.Set_Value(TPE'Unchecked_Access,
                              TPE.Pool(I)'Identity);
    end loop;
  end Initialize;

  task body Worker is separate;
  procedure Submit(Exec : in out Thread_Pool_Executor;
       Command : My_Callables.Any_Callable;
       Params : Parameter;
       My_Future : My_Futures.Any_Future) is separate;
end Thread_Pool_Executors;
```

A worker task first finds its position in the pool and uses this as its identifier. It then places this identifier in the `Controller` buffer, indicating that it is available for work. It then waits on a select statement for the work to arrive, after which it executes the item of work (a `Callable` object), places the result in the given future, and loops back around waiting for more work.

```
with Ada.Task_Identification; use Ada.Task_Identification;
separate(Thread_Pool_Executors)
task body Worker is
  use My_Futures;
  My_Function : My_Callables.Any_Callable;
  The_Result : Result;
  The_Params : Parameter;
  Result_Future : Any_Future;
  My_Id : Positive;
  My_TPE : Any_Thread_Pool_Executor;
  Found : Boolean := False;
begin
  -- find position in the pool
  My_TPE := My_Attributes.Value;
  for I in My_TPE.Pool'Range loop
    if My_TPE.Pool(I)'Identity = Current_Task then
      Found := True;
      My_Id := I;
      exit;
    end if;
  end loop;

  loop
    -- available for work
    My_TPE.Controller.Put(My_Id);
    select
      accept Fork(This : My_Callables.Any_Callable;
                  Params : Parameter; Result : Any_Future) do
        My_Function := This;
        Result_Future := Result;
        The_Params := Params;
      end Fork;
    or terminate;
    end select;
    The_Result :=  My_Function.Call(The_Params);
    Result_Future.Set(The_Result);
  end loop;
end Worker;
```

The allocation of workers to an item of work is performed by the `Submit` procedure.

The relationship between workers and clients means that workers without work are suspended on a select statement with a terminate alternative. This allows them to terminate appropriately if the utility is no longer needed.

```
separate(Thread_Pool_Executors)
procedure Submit(Exec : in out Thread_Pool_Executor;
          Command : My_Callables.Any_Callable;
          Params : Parameter;
          My_Future : My_Futures.Any_Future) is
   Worker_Id : Positive;
begin
  Exec.Controller.Get(Worker_Id);
  Exec.Pool(Worker_Id).Fork(Command, Params, My_Future);
end Submit;
```

11.13 Completion services

The `Executor` package assumes that the clients will manipulate their returned futures and at a later stage wait for the result. In some algorithms, the client is not concerned with the order that the results are returned, they simply want to get the next result. Completion services provide this functionality.

Completion services can be attached to an executor via the following package. The type extends `Executor` and, as well as overriding the `Submit` procedure, provides the `Take` procedure. This returns a reusable future whose value has already been set. If none of the previously submitted (unfinished) executions have terminated, the procedure blocks until a completed future can be returned.

```
with Executors;
with Futures;
with Callables;
generic
package Executors.Completion_Services is
  type Completion_Service is limited interface and Executor;

  overriding
  procedure Submit(Exec : in out Completion_Service;
           Command : My_Callables.Any_Callable;
           Params : Parameter;
           My_Future: My_Futures.Any_Future) is abstract;
  procedure Take(Exec : in out Completion_Service;
           My_Future : out  My_Futures.Any_Future) is abstract;
  type Any_Completion_Service is access all
       Completion_Service'Class;
end Executors.Completion_Services;
```

A thread pool executor with supporting completion services can now be shown.

```
with Ada.Finalization; use Ada.Finalization;
with Executors;
with Executors.Completion_Services;
with Buffers.Mailboxes;
with Buffers;
generic
  with package My_Executors is new Executors(<>);
  use My_Executors;
  with package My_Completion_Services is new
       My_Executors.Completion_Services(<>);
  use My_Completion_Services;
package Completion_Pool_Executors is
  type Completion_Pool_Executor (Pool_Size : Positive) is
       limited new Limited_Controlled and
       Completion_Service with private;

  overriding
  procedure Submit(Exec : in out Completion_Pool_Executor;
                   Command : My_Callables.Any_Callable;
                   Params : Parameter;
                   My_Future : My_Futures.Any_Future);
  overriding
  procedure Take(Exec : in out Completion_Pool_Executor;
                 My_Future : out My_Futures.Any_Future);
  type Any_Completion_Pool_Executor is access all
       Completion_Pool_Executor'Class;
private
  procedure Initialize(TPE : in out Completion_Pool_Executor);

  task type Worker is
     entry Fork(This : My_Callables.Any_Callable;
           Params : Parameter; Result : My_Futures.Any_Future);
  end Worker;

  type My_Worker_Pool is array (Positive range <>) of Worker;

  package My_Futures_Buffers is new Buffers(
          My_Futures.Any_Future);
  package Future_Mailboxes is new My_Futures_Buffers.Mailboxes;

  package My_Task_Buffers is new Buffers(Positive);
  package My_Task_Mailboxes is new My_Task_Buffers.Mailboxes;

  type Completion_Pool_Executor (Pool_Size : Positive) is
       limited new Limited_Controlled and Completion_Service with
  record
    Controller : My_Task_MailBoxes.Mailbox(Pool_Size);
    Pool : My_Worker_Pool(1 .. Pool_Size);
    Future_Buf : Future_Mailboxes.Mailbox(Pool_Size);
  end record;
end Completion_Pool_Executors;
```

Its structure is similar to the previously defined thread pool implementation but with the addition of an extra `Mailbox` implementation of a bounded buffer, in which to save the completed futures.

```
with Ada.Task_Attributes;
with Ada.Unchecked_Deallocation;
with Default_Reusable_Futures;
package body Completion_Pool_Executors is
  package My_Attributes is new Ada.Task_Attributes(
          Any_Completion_Pool_Executor, null);
  package My_Default_Futures is new Default_Reusable_Futures(
          My_Futures);

  procedure Initialize(TPE : in out Completion_Pool_Executor) is
    use My_Futures;
  begin
    for I in TPE.Pool'Range loop
      My_Attributes.Set_Value(TPE'Unchecked_Access,
                              TPE.Pool(I)'Identity);
    end loop;
  end Initialize;

  task body Worker is separate;
  overriding
  procedure Submit(Exec : in out Completion_Pool_Executor;
                   Command : My_Callables.Any_Callable;
                   Params : Parameter;
                   My_Future : My_Futures.Any_Future)
                        is separate;

  overriding
  procedure Take(Exec : in out Completion_Pool_Executor;
                 My_Future : out My_Futures.Any_Future) is
    Tmp : My_Futures.Any_Future;
  begin
    Exec.Future_Buf.Get(Tmp);
    My_Future := My_Futures.Any_Future(Tmp);
  end Take;
end Completion_Pool_Executors;
```

The `Submit` procedure simply submits the required concurrent activity to the thread pool, the future will be placed in a local buffer when the work is completed.

```
separate(Completion_Pool_Executors)
procedure Submit(Exec : in out Completion_Pool_Executor;
          Command : My_Callables.Any_Callable;
          Params : Parameter;
          My_Future : My_Futures.Any_Future) is
```

```
   Worker_Id : Positive := 1;
begin
  Put_Line("Submit calling get worker");
  Exec.Controller.Get(Worker_Id);
  Exec.Pool(Worker_Id).Fork(Command, Params, My_Future);
end Submit;
```

When the future is set by the worker task, the worker also places the future in the buffer. The `Take` procedure then can return futures from the buffer.

```
with Ada.Task_Identification; use Ada.Task_Identification;
separate(Completion_Pool_Executors)
task body Worker is
  use My_Futures;
  My_Function : My_Callables.Any_Callable;
  The_Result : Result;
  The_Params : Parameter;
  Result_Future : Any_Future;
  My_TPE : Any_Completion_Pool_Executor;
  My_Id : Positive;
  Found : Boolean := False;
  use My_Futures;
begin
  My_TPE := My_Attributes.Value;
  for I in My_TPE.Pool'Range loop
    if My_TPE.Pool(I)'Identity = Current_Task then
      Found := True;
      My_Id := I;
      exit;
    end if;
  end loop;

  if not Found then
    raise Program_Error;
  end if;

  loop
    My_TPE.Controller.Put(My_Id);
    select
      accept Fork(This : My_Callables.Any_Callable;
                  Params : Parameter; Result : Any_Future) do
        My_Function := This;
        Result_Future := Result;
        The_Params := Params;
      end Fork;
    or terminate;
    end select;
    The_Result :=  My_Function.Call(The_Params);
    Result_Future.Set(The_Result);
    My_TPE.Future_Buf.Put(Result_Future);
  end loop;
end Worker;
```

11.14 Image processing example revisited

In Subsection 10.2.1, an image processing example was given that used a farm of worker tasks. The facilities presented in Section 11.10 are a generalisation of those mechanisms. Here, the example is repeated using the new framework.

Recall from Subsection 10.2.1, the following packages are required:

- `Images` – this defines the basic types;
- `Images_Acq` – this takes the picture and stores the image for processing;
- `Calc_Seg` – this contains the routine that performs image processing; the routine is executed concurrently;
- `Manager` – this contains the manager that oversees the whole process.

It is mainly these last two packages that must be modified to fit the new framework. However, there are some new types that are needed and some that are not, so first the new `Images` package is presented.

```
package Images is
  type Seg_Index is range 1 .. 8;
  type Image_Index is range 1 .. 4;
  type Image_Seg is array(Seg_Index, Seg_Index) of Integer;
  type Image_Seg_Position is
  record
    I: Image_Index;
    J: Image_Index;
  end record;
  type Image is array (Image_Index, Image_Index) of Image_Seg;
  pragma Volatile_Components(Image);
end Images;
```

The `Image_Seg_Position` type is used to contain the indexes into the `Images` array of a segment that is to be processed concurrently.

The routine that performs the normal calculations must be represented as a `Callable` in the framework. This is a function that takes and returns the `Image_Seg_Position` and processes it. First, the generic `Callables` package must be instantiated with the `Image_Seg_Position` type,

```
with Images; use Images;
with Callables;
package Calc_Seg_Callables is new Callables(Image_Seg_Position,
                                           Image_Seg_Position);
```

and then the `Calc_Seg` rewritten as a callable.

```
with Images; use Images;
with Calc_Seg_Callables; use Calc_Seg_Callables;
package Calc_Seg is
  type Do_Normal_Callable is new Callable with null record;
  overriding
  function Call(C : Do_Normal_Callable; P : Image_Seg_Position)
           return Image_Seg_Position;
end Calc_Seg;
```

The structure of the function is very similar to the original `Do_Normal` procedure except that it no longer needs to communicate with the manager to obtain its segment.

```
with Images; use Images;
with Image_Acq;
package body Calc_Seg is
  function Call(C : Do_Normal_Callable;
           P : Image_Seg_Position) return Image_Seg_Position is
  begin
      -- Do calculation of normals for segment P.I, P.J,
      -- leaving result in Picture.
      -- Picture(P.I, P.J) := ...;
      return (P.I,P.J);
  end Call;
end Calc_Seg;
```

The manager is where the bulk of the changes occur. The interface now simply contains the specification of the `Controller` task:

```
package Manager is
  task Controller;
end Manager;
```

The body is where the decision is taken as to what asynchronous execution facility to use. Here, a `Completion_Pool_Executor` is appropriate as the manager is not concerned with the order in which the parallel executions terminate. It simply want to wait for them all.

First the required generic package must be instantiated and then the appropriate types declared (including an array of futures for storing the results).

```
with Futures, Default_Reusable_Futures;
with Executors, Executors.Completion_Services;
with Completion_Pool_Executors;
with Buffers, Buffers.Mailboxes, Image_Acq;
with Calc_Seg_Callables, Calc_Seg, Images;
use Calc_Seg_Callables, Calc_Seg, Images;
package body Manager is
  Default_Position : Image_Seg_Position := (1,1);
  Pool_Size : constant := ...;
  package Image_Seg_Futures is new
     Futures(Image_Seg_Position);
  package Default_Image_Seg_Futures is new
     Default_Reusable_Futures(Image_Seg_Futures);
  package Calc_Seg_Executors is new
     Executors (Image_Seg_Position, Image_Seg_Position,
              Image_Seg_Futures, Calc_Seg_Callables);
  package Calc_Seg_Executors_Completion_Services is new
     Calc_Seg_Executors.Completion_Services;
  package Calc_Seg_Completion_Pools is new
     Completion_Pool_Executors(Calc_Seg_Executors,
        Calc_Seg_Executors_Completion_Services);
  use Image_Seg_Futures, Default_Image_Seg_Futures,
      Calc_Seg_Executors, Calc_Seg_Completion_Pools;

  type Callable_Access is access Do_Normal_Callable;
  type Future_Array is array (Image_Index, Image_Index) of
       aliased Default_Future;
  My_Workers : Calc_Seg_Completion_Pools.
               Completion_Pool_Executor(Pool_Size);
  A_Future : aliased Future_Array;
  Future_Access : Image_Seg_Futures.Any_Future;
  A_Callable : aliased Do_Normal_Callable;

  task body Controller is separate;
end Manager;
```

Finally, the `Controller` task farms out the work. Unlike before, when it passes the callable to the worker, the parameters to the callable now contain the details of the segment to be processed. This avoids the worker having to call back in to the manager via a rendezvous.

```
separate(Manager)
task body Controller is
begin
  loop
    -- The image acquisition process is often of a repetitive
    -- nature, so some sort of loop structure would often be
    -- used here. The actual nature of the loop is not of
    -- significance to this example, so no details are given.
    -- Whatever computation and communication are required
    -- take place before image acquisition.

    Image_Acq.Get_The_Picture; -- this could delay the task

    -- Start the process of image acquisition. Initiate the
    -- worker tasks with the required segments.
    declare
    begin
      for Major_Index in Image_Index loop
        for Minor_Index in Image_Index loop
          My_Workers.Submit(
               A_Callable'Access, (Major_Index,Minor_Index),
               A_Future(Major_Index, Minor_Index)'Access);
        end loop;
      end loop;
      -- Whatever computation and communication take place
      -- during image processing.
      -- Now wait for completion
      for Major_Index in Image_Index loop
        for Minor_Index in Image_Index loop
          My_Workers.Take(Future_Access);
        end loop;
      end loop;

      -- Whatever computation and communication take place
      -- after image processing.
    end;
  end loop;
end Controller;
```

11.15 Summary

This chapter has served a dual role. It has illustrated how the new concurrent object-oriented programming facilities of Ada 2005 can be used. More importantly though, it has also shown how the mechanisms can be used to produce a library of concurrency utilities. Both synchronisation and concurrent execution abstractions have been developed. For synchronisation:

- Semaphores – counting, binary and quantity semaphores variations.
- Locks – mutual exclusion and read-write locking mechanisms.
- Signals – condition synchronisation without passing data.
- Event variables – bivalued state variables.
- Buffers – the standard bounded buffer abstraction.
- Blackboards – events with arbitrary data passing.
- Broadcasts – a standard broadcast paradigm.
- Barriers – a pure synchronisation mechanism to control task release.

In addition, a framework is presented that combines several abstractions to furnish a set of utilities for the asynchronous execution of concurrent work. This set of utilities comprises:

- Callable – an interface in support of concurrent execution.
- Futures – an interface used to represent a data item that will eventually contain a value.
- Default_Reusable_Future – a protected type that provides the default implementation of a reusable future.
- Executor – an interface that execution engines must support.
- Executors.Thread_Pool_Executors – an execution engine based on thread pools.
- Completion_Service – an interface for an executor that also manages the futures.
- Completion_Pool_Executors – an execution engine based on thread pools that also manages the completion of the futures.

12
Tasking and systems programming

Ada is a high-level programming language; it provides abstract constructs that allow programs to be constructed easily and safely. However, it is recognised that one of the intended application areas for Ada is the production of embedded systems. Often these (and other) systems require the programmer to become more concerned with the implementation, and efficient manipulation, of these abstract program entities. Ada resolves this conflict in two ways:

(1) by allowing the programmer to specify the representation of program abstractions on the underlying hardware, for example by specifying the layout of a record or the address of a variable; and
(2) by having extra facilities in the *Systems Programming Annex* for interrupt handling, controlling access to shared variables, unique identification of tasks, task attributes and the notification of task termination. As with all Ada annexes, these features need not be supported by all compilers.

The areas that are of concern in this book are those which relate directly to the tasking model. These are:

- device driving and interrupt handling – covered in this chapter;
- access to shared variables – previously covered in Section 7.12
- task identification – motivated in Section 4.4 and covered fully in this chapter;
- task attributes – covered in this chapter;
- task termination (notification thereof) – covered in Section 15.7.

Other relevant embedded systems issues such as access to intrinsic subprograms, control over storage pools and data streams, and the use of machine code inserts are not dealt with. Nevertheless, a useful place to start this discussion on low-level programming is the following predefined packages that provide key implementation details for the language:

- `System` – contains definitions of certain configuration-dependent characteristics; for example, the maximum and minimum values of the root types, the definition of an address, and the bit ordering of the machine.
- `System.Storage_Elements` – a child package of `System` that allows individual storage elements to be manipulated and gives further control over addresses.

```
package System is
   pragma Pure(System);

   type Name is <implementation-defined-enumeration-object>;
   System_Name : constant Name := <implementation-defined>;

   -- System-dependent named numbers:
   Min_Int               : constant := Root_Integer'First;
   Max_Int               : constant := Root_Integer'Last;
   Max_Binary_Modulus    : constant := <implementation-defined>;
   Max_Nonbinary_Modulus : constant := <implementation-defined>;
   Max_Base_Digits       : constant := Root_Real'Digits;
   Max_Digits            : constant := <implementation-defined>;
   Max_Mantissa          : constant := <implementation-defined>;
   Fine_Delta            : constant := <implementation-defined>;
   Tick                  : constant := <implementation-defined>;

   -- Storage-related declarations:
   type Address is <implementation-defined>;
   pragma Preelaborable_Initialization(Address);
   Null_Address : constant Address;
   Storage_Unit : constant := <implementation-defined>;
   Word_Size    : constant := <implementation-defined> *
                              Storage_Unit;
   Memory_Size  : constant := <implementation-defined>;

   -- Address Comparison:
   function "<" (Left, Right : Address) return Boolean;
   function "<="(Left, Right : Address) return Boolean;
   function ">" (Left, Right : Address) return Boolean;
   function ">="(Left, Right : Address) return Boolean;
   function "=" (Left, Right : Address) return Boolean;
   -- function "/=" (Left, Right : Address) return Boolean;
   -- "/=" is implicitly defined
   pragma Convention(Intrinsic, "<");
   -- ditto for all language-defined subprograms in this package

   -- Other system-dependent declarations:
   type Bit_Order is (High_Order_First, Low_Order_First);
   Default_Bit_Order : constant Bit_Order :=
                       <implementation-defined>;

   -- Priority-related declarations (see D.1):
   subtype Any_Priority is Integer range <implementation-defined>;
   subtype Priority is Any_Priority range Any_Priority'First ..
            <implementation-defined>;
   subtype Interrupt_Priority is Any_Priority range
           Priority'Last+1 .. Any_Priority'Last;
   Default_Priority : constant Priority :=
            (Priority'First + Priority'Last)/2;
private
   -- not specified by the language
end System;
```

```
package System.Storage_Elements is
  pragma Pure(System.Storage_Elements);

  type Storage_Offset is range <implementation-defined>;

  subtype Storage_Count is Storage_Offset range
          0..Storage_Offset'Last;
  type Storage_Element is mod <implementation-defined>;
  for Storage_Element'Size use Storage_Unit;

  type Storage_Array is array
      (Storage_Offset range <>) of aliased Storage_Element;
  for Storage_Array'Component_Size use Storage_Unit;

   -- Address Arithmetic:
  function "+"(Left : Address; Right : Storage_Offset)
                return Address;
  function "+"(Left : Storage_Offset; Right : Address)
                return Address;

  function "-"(Left : Address; Right : Storage_Offset)
                return Address;
  function "-"(Left, Right : Address) return Storage_Offset;

  function "mod"(Left : Address; Right : Storage_Offset)
               return Storage_Offset;

  -- Conversion to/from integers:

  type Integer_Address is <implementation-defined>;
  function To_Address(Value : Integer_Address) return Address;
  function To_Integer(Value : Address) return Integer_Address;

  pragma Convention(Intrinsic, "+"); -- built in to the compiler
  -- and so on for all language-defined subprograms
  -- declared in this package.
end System.Storage_Elements;
```

Examples of the use of some of these facilities will be given later in this chapter.

12.1 Device driving and interrupt handling

A major difficulty in developing embedded systems is the design and implementation of device drivers.

Definition: A device driver is a subsystem that has sole responsibility for controlling access to some external device. It must manipulate device registers and respond to interrupts.

A hardware device is an object that is operating in parallel with other elements of the system. It is, therefore, logical to consider the device as being a hardware 'task'. In Ada, there are three ways in which tasks can synchronise and communicate with each other:

- through the rendezvous,
- using protected units and
- via shared variables.

In general, Ada assumes that the device and the program have access to shared memory device registers that can be specified using its representation specification techniques. In Ada 83, interrupts were represented by hardware generated task entry calls. In the current version of Ada, this facility is considered obsolete. Consequently, it will not be discussed in this book.

The preferred method of device driving is to encapsulate the device operations in a protected object. An interrupt may be handled by mapping it to a protected procedure call.

This section first considers how device registers can be manipulated and specified using representation aspects, and then covers the model of interrupt handling that is supported with protected objects. Finally, an example device driver is presented.

12.1.1 Representation aspects

Representation aspects are a compromise between abstract and concrete structures. They are described in most Ada textbooks and are outlined below.

Definitions: Four main representational aspects are available:

(1) Attribute definition clause – allows various attributes of an object, task or subprogram to be set; for example, the size (in bits) of objects, the storage alignment, the maximum storage space for a task, and the address of an object.

(2) Enumeration representation clause – allows the literals of an enumeration type to be given specific internal values.
(3) Record representation clause – allows record components to be assigned offsets and lengths within single storage units.
(4) At clause – this was the main Ada 83 mechanism for positioning an object at a specific address; this facility has been maintained for compatibility with Ada 83 but it is now obsolete (attributes can be used) and will therefore not be discussed further.

If an implementation cannot obey a specification request then the compiler must either reject the program or raise an exception at run-time.

As an example of the use of a representation aspect, consider the control status register of a disk drive (for use on a 16-bit embedded computer system). This register is illustrated in Figure 12.1. It is assumed that all fields can be read without changing the status of the device.

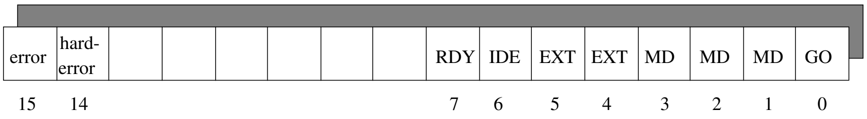


Fig. 12.1: Control status register layout

Five fields (**error**, **hard-error**, **RDY** (Ready), **IDE** (interrupt enable) and **GO**) are all two-state variables that are best thought of as flags:

```
type Flag is (Off,On);
```

with 0 representing `Off` and 1 representing `On` (this being the default representation).

The next field is **Mode of Operation (MD)**; this covers three bits but provides only four operations:

```
type Mode is (Reset, Write, Read, Seek);
```

The required internal codes for these four states are 0, 1, 2 and 4; this is specified using an enumeration clause as follows:

```
for Mode use (Reset => 0, Write => 1, Read => 2, Seek => 4);
```

As long as the sequence is still strictly increasing, any representation is allowed.

Finally the **Memory Extension Bits (EXT)** field can be mapped onto a type which is restricted to two bits by an attribute definition clause:

```
type MEB is range 0 .. 3;
for MEB'Size use 2;
```

Having constructed types for each individual component, the register itself can be represented as a record:

```
type Control_Status is record
  GO : Flag;
  MD : Mode;
  EXT : MEB;
  IDE : Flag;
  RDY : Flag;
  Hard_Error : Flag;
  Error : Flag;
end record;
```

The representation of the record type can then be specified:

```
Word : constant := 2; -- 2 bytes in a word
Bits_in_Word : constant := 16; -- bits in a word

for Control_Status use record
  GO  at 0*Word range 0 .. 0; -- field GO at word 0 position 0
  MD  at 0*Word range 1 .. 3;
  EXT at 0*Word range 4 .. 5;
  IDE at 0*Word range 6 .. 6;
  RDY at 0*Word range 7 .. 7;
  Hard_Error at 0*Word range 14 .. 14;
  Error at 0*Word range 15 .. 15;
end record;

for Control_Status'Size use Bits_In_Word;
for Control_Status'Alignment use Word;
for Control_Status'Bit_Order use Low_Order_First;
```

Each implementation has the notion of a ‘storage unit’ to control layout. The value of this unit is contained within the predefined library package `System`. Each component of the record is given a position within one or more storage units; for example, `MD` is to be placed at storage unit 0 using bits 1, 2 and 3. In this example the storage unit is deemed to have the value 8, and hence `Error` is to be found in the storage unit 1 at bit 7 (that is, at word 0 bit 15).

To constrain the system to using only 16 bits in total for an object of type `Control_Status`, an attribute definition clause is used. The alignment clause ensures that the record is placed at an even byte boundary, and the `Bit_Order` attribute indicates that bit 0 is the least significant bit.

Finally, a data object to represent the control register must be defined and placed at the correct memory location; this is the physical address of the register:

```
Control_Register : Control_Status;
for Control_Register'Address use
    System.Storage_Elements.To_Address(8#777404#);
```

Having now constructed the abstract data representation of the register, and placed an appropriately defined variable at the correct address, the hardware register can be manipulated by assignments to this variable:

```
Control_Register := (GO => On,
                     MD => Read,
                     EXT => 0,
                     IDE => On,
                     RDY => Off,
                     Hard_Error => Off,
                     Error => Off);
```

The use of this record aggregate assumes that the entire register will be assigned values at the same time. To ensure that GO is not set before the other fields of the record it may be necessary to use a temporary (shadow) control register:

```
Temp_Cr : Control_Status;
```

This temporary register is then assigned control values and copied into the real register variable:

```
Control_Register := Temp_Cr;
```

The code for this assignment will in most cases ensure that the entire control register is updated in a single action. If any doubt still remains, then the pragma `Atomic` can be used (see Section 7.13).

After the completion of the I/O operation, the device itself may alter the values on the register; this is recognised in the program as changes in the values of the record components:

```
if Control_Register.Error = On then
  raise Disk_Error;
end if;
```

The object `Control_Register` is therefore a collection of shared variables, which are shared between the device control task and the device itself. Mutual exclusion between these two concurrent (and parallel) processes is necessary to give reliability and performance. This is achieved in Ada by using a protected object. Condition synchronisations must ensure that the correct sequence of events takes place between the hardware and its driver. The two most common forms of interaction are:

- polling;
- interrupt driven.

Whichever is used on any particular device is a property of the hardware as well as the system software. Polling cannot, therefore, be eliminated if it is the only method available for examining the condition of the device. If polling is used, then the busy-wait loop must incorporate a delay statement so that the device task does not monopolise the processor. Alternatively, the whole device driver can be programmed as a periodic task (see next chapter).

12.2 Model of interrupts

The ARM defines the following model of an interrupt:

- An interrupt represents a class of events that are detected by the hardware or the system software.
- The *occurrence* of an interrupt consists of its *generation* and its *delivery*.
- The generation of an interrupt is the event in the underlying hardware or system which makes the interrupt available to the program.
- Delivery is the action that invokes a part of the program (called the interrupt *handler*) in response to the interrupt occurrence. In between the generation of the interrupt and its delivery, the interrupt is said to be *pending*. The handler is invoked once for each delivery of the interrupt.
- While an interrupt is being handled, further interrupts from the same source are *blocked*; all future occurrences of the interrupt are prevented from being generated. It is usually device dependent as to whether a blocked interrupt remains pending or is lost.
- Certain interrupts are *reserved*. The programmer is not allowed to provide a handler for a reserved interrupt. Usually, a reserved interrupt is handled directly by the run-time support system of Ada (for example, a clock interrupt used to implement the time packages and the delay statements).
- Each non-reserved interrupt has a default handler that is assigned by the run-time support system.

12.2.1 Handling interrupts using protected procedures

The main representation of an interrupt handler in Ada is a parameterless protected procedure. Each interrupt has a unique discrete identifier, which is supported by the system. How this unique identifier is represented is implementation defined; it might, for example, be the address of the hardware interrupt vector associated with the interrupt.

Identifying interrupt handling protected procedures is done using one of two pragmas:

```
pragma Attach_Handler(Handler_Name, Expression);
  -- This can appear in the specification or body of a
  -- library-level protected object and allows the
  -- static association of a named handler with the
  -- interrupt identified by the expression; the handler
  -- becomes attached when the protected object is created.
  -- Raises Program_Error:
  --   (a) when the protected object is created and
  --       the interrupt is reserved,
  --   (b) if the interrupt already has a
  --       user-defined handler, or
  --   (c) if any ceiling priority defined is
  --       not in the range Ada.Interrupt_Priority.

pragma Interrupt_Handler(Handler_Name);
  -- This can appear in the specification of a library-level
  -- protected object and allows the dynamic association of
  -- the named parameterless procedure as an interrupt
  -- handler for one or more interrupts. Objects created
  -- from such a protected type must be library level.
```

The following package defines the Systems Programming Annex's support for interrupt identification and the dynamic attachment of handlers:

```
package Ada.Interrupts is
  type Interrupt_Id is <implementation-defined>; -- is discrete
  type Parameterless_Handler is access protected procedure;

  function Is_Reserved(Interrupt : Interrupt_Id) return Boolean;
    -- Returns True if the interrupt is reserved,
    -- returns False otherwise.

  function Is_Attached(Interrupt : Interrupt_Id) return Boolean;
    -- Returns True if the interrupt is attached to a
    -- handler, returns False otherwise.
    -- Raises Program_Error if the interrupt is reserved.

  function Current_Handler(Interrupt : Interrupt_Id)
                           return Parameterless_Handler;
    -- Returns an access variable to the current handler for
    -- the interrupt. If no user handler has been attached, a
    -- value is returned which represents the default handler.
    -- Raises Program_Error if the interrupt is reserved.

  procedure Attach_Handler(New_Handler : Parameterless_Handler;
                           Interrupt : Interrupt_Id);
    -- Assigns New_Handler as the current handler
    -- for the Interrupt.
    -- If New_Handler is null, the default handler is restored.
    -- Raises Program_Error:
    --     (a) if the protected object associated with the
    --         New_Handler has not been identified with a
    --         pragma Interrupt_Handler,
    --     (b) if the interrupt is reserved,
    --     (c) if the current handler was attached statically
    --         using pragma Attach_Handler.

  procedure Exchange_Handler(
            Old_Handler : out Parameterless_Handler;
            New_Handler : Parameterless_Handler;
            Interrupt : Interrupt_Id);
    -- Assigns New_Handler as the current handler for the
    -- Interrupt and returns the previous handler in
    -- Old_Handler.
    -- If New_Handler is null, the default handler is restored.
    -- Raises Program_Error:
    --      (a) if the protected object associated with the
    --          New_Handler has not been identified with a
    --          pragma Interrupt_Handler,
    --      (b) if the interrupt is reserved,
    --      (c) if the current handler was attached statically
    --          using pragma Attach_Handler.

  procedure Detach_Handler(Interrupt : Interrupt_Id);
    -- Restores the default handler for the specified interrupt.
    -- Raises Program_Error:
    --      (a) if the interrupt is reserved,
    --      (b) if the current handler was attached statically
    --          using pragma Attach_Handler.

  function Reference(Interrupt : Interrupt_Id)
           return System.Address;
    -- Returns an Address which can be used to attach
    -- a task entry to an interrupt via an address
    -- clause on an entry.
    -- Raises Program_Error if the interrupt cannot be
    -- attached in this way.

private
  -- not specified by the language
end Ada.Interrupts;
```

In all cases where `Program_Error` is raised, the currently attached handler is not changed.

It should be noted that the `Reference` function provides the mechanisms by which interrupt task entries are supported. As mentioned earlier, this model of interrupt handling is considered obsolete and should therefore not be used.

It is possible that an implementation will also allow the association of names with interrupts via the following package:

```
package Ada.Interrupts.Names is
  <implementation-defined> : constant Interrupt_Id
                           := <implementation-defined>;
  ...

  <implementation-defined> : constant Interrupt_Id
                           := <implementation-defined>;
end Ada.Interrupts.Names;
```

This will be used in the following example.

12.2.2 A simple driver example

A common class of equipment to be attached to an embedded computer system is the analogue-to-digital converter (ADC). The converter samples some environmental factors such as temperature or pressure; it translates the measurements it receives, which are usually in millivolts, and provides scaled integer values on a hardware register. Consider a single converter that has a 16-bit result register at hardware address 8#150000# and a control register at 8#150002#. The computer is a 16-bit machine and the control register is structured as follows:

Bit	Name	Meaning
0	A/D Start	Set to 1 to start a conversion.
6	Interrupt Enable/Disable	Set to 1 to enable interrupts.
7	Done	Set to 1 by the converter when conversion is complete.
8-13	Channel	The converter has 64 analogue inputs, the particular one required is indicated by the value of the channel.
15	Error	Set to 1 by the converter if the device malfunctions.

The driver for this ADC will be defined using a protected type within a library package, so that the interrupt can be processed as a protected procedure call, and so that more than one ADC can be catered for:

```
package Adc_Device_Driver is
  Max_Measure : constant := (2**16)-1;
  type Channel is range 0..63;
  subtype Measurement is Integer range 0..Max_Measure;

  procedure Read(Ch : Channel; M : out Measurement);
    -- potentially blocking
  Conversion_Error : exception;
private
  for Channel'Size use 6;
  -- indicates that six
  -- bits only must be used
end Adc_Device_Driver;
```

For any request the driver will make three attempts before raising the exception. The package body follows:

```
with Ada.Interrupts.Names; use Ada.Interrupts;
with System; use System;
package body Adc_Device_Driver is
  Bits_In_Word : constant := 16;
  Word : constant := 2; -- bytes in word
  type Flag is (Down, Set);

  type Control_Register is record
    Ad_Start : Flag;
    IE       : Flag;
    Done     : Flag;
    Ch       : Channel;
    Error    : Flag;
  end record;

  for Control_Register use record
    -- specifies the layout of the control register
    Ad_Start at 0*Word range 0..0;
    IE       at 0*Word range 6..6;
    Done     at 0*Word range 7..7;
    Ch       at 0*Word range 8..13;
    Error    at 0*Word range 15..15;
  end record;

  for Control_Register'Size use Bits_In_Word;
    -- the register is 16 bits long
  for Control_Register'Alignment use Word;
    -- on a word boundary
  for Control_Register'Bit_Order use Low_Order_First;

  type Data_Register is range 0 .. Max_Measure;
  for Data_Register'Size use Bits_In_Word;
    -- the register is 16 bits long

  Contr_Reg_Addr : constant Address :=
           System.Storage_Elements.To_Address(8#150002#);
  Data_Reg_Addr : constant Address :=
           System.Storage_Elements.To_Address(8#150000\);
  Adc_Priority : constant Interrupt_Priority := 63;
  Control_Reg : aliased Control_Register;
  for Control_Reg'Address use Contr_Reg_Addr;
      -- specifies the address of the control register
  Data_Reg : aliased Data_Register;
  for Data_Reg'Address use Data_Reg_Addr;
      -- specifies the address of the data register

  protected type Interrupt_Interface(Int_Id : Interrupt_Id;
                Cr : access Control_Register;
                Dr : access Data_Register) is
    entry Read(Chan : Channel; M : out Measurement);
  private
    entry Done(Chan : Channel; M : out Measurement);
    procedure Handler;
    pragma Attach_Handler(Handler, Int_Id);
    pragma Interrupt_Priority(Adc_Priority);
      -- see Chapter 13 for discussion on priorities
    Interrupt_Occurred : Boolean := False;
    Next_Request : Boolean := True;
  end Interrupt_Interface;

  Adc_Interface : Interrupt_Interface(Names.Adc,
                  Control_Reg'Access, Data_Reg'Access);
    -- this assumes that 'Adc' is registered as an
    -- Interrupt_Id in Ada.Interrupts.Names

  protected body Interrupt_Interface is
    entry Read(Chan : Channel; M : out Measurement)
          when Next_Request is
      Shadow_Register : Control_Register;
    begin
      Shadow_Register := (Ad_Start => Set, IE => Set,
            Done => Down, Ch => Chan, Error => Down);
      Cr.all := Shadow_Register;
      Interrupt_Occurred := False;
      Next_Request := False;
      requeue Done;
    end Read;

    procedure Handler is
    begin
      Interrupt_Occurred := True;
    end Handler;

    entry Done(Chan : Channel; M : out Measurement)
                              when Interrupt_Occurred is
    begin
      Next_Request := True;
      if Cr.Done = Set and Cr. Error = Down then
            M := Measurement(Dr.all);
      else
        raise Conversion_Error;
      end if;
    end Done;
  end Interrupt_Interface;

  procedure Read(Ch : Channel; M : out Measurement) is
  begin
    for I in 1..3 loop
      begin
        Adc_Interface.Read(Ch,M);
        return;
      exception
        when Conversion_Error => null;
      end;
    end loop;
    raise Conversion_Error;
  end Read;
end Adc_Device_Driver;
```

The client tasks simply call the `Read` procedure indicating the channel number from which to read, and an output variable for the actual value read. Inside the procedure, an inner loop attempts three conversions by calling the `Read` entry in the protected object associated with the converter. Inside this entry, the control register, `Cr`, is set up with appropriate values. Once the control register has been assigned, the client task is requeued on a private entry to await the interrupt.

When the interrupt has arrived (as a parameterless protected procedure call), the barrier on the `Done` entry is set to true; this results in the `Done` entry being executed (as part of the interrupt handler), which checks that `Cr.Done` has been set and that the error flag has not been raised. If this is the case the out parameter `M` is constructed, again using a type conversion from the value on the buffer register. (Note that this value cannot be out of range for the subtype `Measurement`.) If conversion has not been successful, the exception `Conversion_Error` is raised; this is trapped by the `Read` procedure, which makes three attempts in total at a conversion before allowing the error to propagate.

The above example illustrates that it is often necessary when writing device drivers to convert objects from one type to another. In these circumstances the strong typing features of Ada can be an irritant. It is, however, possible to circumvent this difficulty by using a generic function that is provided as a predefined library unit:

```
generic
  type Source (<>) is limited private;
  type Target (<>) is limited private;
function Ada.Unchecked_Conversion(S : Source) return Target;
pragma Convention(Intrinsic, Ada.Unchecked_Conversion);
pragma Pure(Ada.Unchecked_Conversion);
```

The effect of unchecked conversion is to copy the bit pattern of the source over to the target. The programmer must make sure that the conversion is sensible and that all possible patterns are acceptable to the target. Note that conversions can also be made between `'Access` and an object of type `Address`.

12.2.3 Dynamic attachment of interrupt handlers

In the previous subsection, the `Adc_Device_Driver` interrupt interface was established when the associated protected object was created. It is possible, using the `Ada.Interrupts` package, to attach and detach handlers to and from interrupts dynamically. Suppose, for example, that a programmer under certain conditions wishes to change the interrupt handler for the ADC device. The original protected type is no longer suitable, and should be replaced with the following:

```
protected type New_Interrupt_Interface(
               Cr : access Control_Register;
               Dr : access Data_Register) is
  entry Read(Ch : Channel; M : out Measurement);
  procedure Handler;
  pragma Interrupt_Handler(Handler);
private
  entry Done(Ch : Channel; M : out Measurement);
  pragma Interrupt_Priority(Adc_Priority);
end New_Interrupt_Interface;

New_Adc_Interface : New_Interrupt_Interface(Control_Reg'Access,
                    Data_Reg'Access);
```

Here the pragma `Interrupt_Handler` indicates that the procedure `Handler` will be used as an interrupt handler. Now, to switch over to (and back from) the new handler requires

```
  Old : Parameterless_Handler := null;
  ...
  -- attach new handler
  if Is_Attached(Names.Adc) then
    -- handler attached
    Exchange_Handler(Old,New_Adc_Interface.Handler'Access,
                     Names.Adc);
  else
    -- no user handler attached
    Attach_Handler(New_Adc_Interface.Handler'Access,
                   Names.Adc);
  end if;
  ...
  -- change back the handlers
  if Old = null then
    Detach_Handler(Names.Adc);
  else
    Attach_Handler(Old, Names.Adc);
  end if;
```

Initially, of course the default handler will be attached. Note that, strictly speaking, changing back could be accomplished simply by

```
Attach_Handler(Old,Names.Adc);
```

as a null handler is taken to mean the default handler.

12.2.4 User-implemented timers

As a further example of interrupt handling, consider the implementation of a user-defined delay statement. The motivation for such a facility is to allow the programmer to access a particular interval timer that has a granularity different from that supplied with the standard implementation of Ada.

Tasks, wishing to be delayed, call the procedure `Set_Alarm` given below. The first parameter indicates the interval and the second is a reference to a protected object which implements a persistent signal (see Section 11.4):

```
with Signals.Persistent_Signals; use Signals.Persistent_Signals;
package User_Timers is
  type Fine_Duration is delta 0.000_004 range 0.0 .. 17_179.0;
         -- 32-bit timer with a resolution of 4 microseconds
  Max_Alarm_Events : constant Integer :=  <some appropriate value>;
  procedure Set_Alarm(For_Time : Fine_Duration;
                      Sync : access Persistent_Signal);
  -- potentially blocking

  Capacity_Exceeded : exception;
  -- raised by Set_Alarm if the number of currently
  -- delayed tasks exceeds some maximum
end User_Timers;
```

The body of the package keeps an ordered list of intervals and alarms. The first item in the list indicates the time to the next alarm, the second item indicates the time after that to the following alarm. So, for example, if there are three tasks delayed with values D1, D2, D3 where D1 < D2 < D3, then the list will be ordered with values (D1, D2 − D1, D3 − D2). Every time the clock interrupts, the first item in the list is decremented (the other values remain the same). When the first item reaches 0, it is removed from the list and its associated alarm is sent. When a task issues a `Set_Alarm` call, the list is searched and the task is inserted in the appropriate place. The algorithm is implemented below using a static array and a list of array elements ordered according to release position.

```
with System; use System;
with Ada.Interrupts; use Ada.Interrupts;
with Ada.Interrupts.Names;
package body User_Timers is
  subtype Index is Integer range 1 .. Max_Alarm_Events;
  subtype Next_Range is Integer range 0 .. Max_Alarm_Events;
  type Signal is access all Persistent_Signal;
  type Alarm_Event is record
    Interval : Fine_Duration;
    Tell : Signal;
    Next_Event : Next_Range; -- Index of next Alarm_Event,
                             -- 0 is terminator
  end record;

  type Waiting_Array is array(Index) of Alarm_Event;

  type Control_Register is ...;
  CR : constant Address :=
               System.Storage_Elements.To_Address(8#150000#);
  Int_Id : constant Interrupt_Id := Names.User_Clock;
  One_Tick : constant Fine_Duration := 0.000_004;
    -- granularity of the clock
  Clock_Priority : constant Interrupt_Priority := 63;
```

```
protected type Timer(Control_Reg : access Control_Register) is
  procedure Set_Alarm(For_Time : Fine_Duration;
                      Sync : access Persistent_Signal);
private
  procedure Handler;
  pragma Attach_Handler(Handler, Int_Id);
  pragma Interrupt_Priority(Clock_Priority);
  Waiting : Waiting_Array := (others => (0.0, null, 0));
  First : Next_Range := 0;
end Timer;

Creg : Control_Register := 8#010#;
  -- interrupt and device enabled
for Control_Reg'Address use Creg;
  -- specifies the address of the control register
My_Timer : Timer(Creg'Access);

procedure Set_Alarm(For_Time : Fine_Duration;
                    Sync : access Persistent_Signal) is
begin
  My_Timer.Set_Alarm(For_Time, Sync);
end Set_Alarm;

protected body Timer is
  procedure Empty_Slot(Indx : out Index) is
  begin
    for I in Index loop
      if Waiting(I).Tell = null then
        Waiting(I).Next_Event := 0;
        Indx := I;
        return;
      end if;
    end loop;
    raise Capacity_Exceeded;
  end Empty_Slot;

  procedure Set_Alarm(For_Time : Fine_Duration;
                      Sync : access Persistent_Signal) is
    Slot, Next : Index;
    Accumulated : Fine_Duration := 0.0;
  begin
    if First = 0 then
      -- empty list
      Waiting(1).Interval := For_Time;
      Waiting(1).Tell := Sync;
      Waiting(1).Next_Event := 0;
      First := 1;
      return;
    end if;
    Empty_Slot(Slot);
    if For_Time <= Waiting(First).Interval then
      -- needs to be placed at the front of the list
```

```
        Waiting(Slot).Interval := For_Time;
        Waiting(Slot).Tell := Sync;
        Waiting(Slot).Next_Event := First;
        Waiting(First).Interval :=
               Waiting(First).Interval - For_Time;
        First := Slot;
        return;
      end if;
      Accumulated := Accumulated + Waiting(First).Interval;
      Next := First;

      while Waiting(Next).Next_Event /= 0 loop
        if Accumulated + Waiting(Waiting(Next).Next_Event).
                                Interval > For_Time then
          -- place after next
          Waiting(Slot).Interval := For_Time - Accumulated;
          Waiting(Slot).Tell := Sync;
          Waiting(Slot).Next_Event := Waiting(Next).Next_Event;
          Waiting(Next).Next_Event := Slot;
          Next := Waiting(Slot).Next_Event;
          Waiting(Next).Interval := Waiting(Next).Interval -
                                    Waiting(Slot).Interval;
          return;
        end if;
        Next := Waiting(Next).Next_Event;
        Accumulated := Accumulated + Waiting(Next).Interval;
      end loop;

      -- place at end of list
      Waiting(Slot).Interval := For_Time - Accumulated;
      Waiting(Slot).Tell := Sync;
      Waiting(Slot).Next_Event := 0;
      Waiting(Next).Next_Event := Slot;
    end Set_Alarm;

    procedure Handler is
    begin
      if First > 0 then
        Waiting(First).Interval :=
                Waiting(First).Interval - One_Tick;
        while Waiting(First).Interval <= 0.0 and First > 0 loop
          Waiting(First).Tell.Send;
          Waiting(First).Tell := null;
          First := Waiting(First).Next_Event;
        end loop;
      end if;
    end Handler;
  end Timer;
end User_Timers;
```

Note the new `Containers` package in Ada 2005 could have been used to simplify this code.

12.3 Task identifiers

If the Systems Programming Annex is being supported, all tasks have a unique identifier which can be accessed and manipulated by the `Ada.Task_Identification` package:

```
package Ada.Task_Identification is
  type Task_Id is private;
  Null_Task_Id : constant Task_Id ;

  function "="(Left, Right : Task_Id) return Boolean;

  function Image(T : Task_Id) return String;
    -- Returns an implementation defined string representing
    -- the task, the null task returns the null string.

  function Current_Task return Task_Id;
    -- Returns the unique id of the calling task; it is
    -- a bounded error to call this function from an
    -- interrupt handler or a protected entry body: either
    -- Program_Error will be raised or an implementation
    -- defined value of type Task_Id will be returned.
    -- Instead, the attribute 'Caller should be used for
    -- protected entries (see below)

  procedure Abort_Task(T : in out Task_Id);
    -- Has the same effect as aborting a task using
    -- the abort statement.
    -- Raises Program_Error if a Null_Task_Id is passed.

  function Is_Terminated(T : Task_Id) return Boolean;
    -- Equivalent to the 'Terminated attribute.
    -- Raises Program_Error if a Null_Task_Id is passed.

  function Is_Callable(T : Task_Id) return Boolean;
    -- Equivalent to the 'Callable attribute.
    -- Raises Program_Error if a Null_Task_Id is passed.
private
  -- not specified by the language
end Ada.Task_Identification;
```

Warning: A program is deemed erroneous if an attempt is made to use the `Task_Id` of a task that no longer exists.

In addition to the above package, the Systems Programming Annex also supports the following two attributes:

- `T'Identity`:

  for a given task, yields a value of type `Task_Id` that identifies the task denoted by `T`.

- `E'Caller`:
  for a given entry, yields a value of type `Task_Id` that identifies the task whose call is now being serviced. The attribute can only be used inside the accept or entry body denoted by `E`.

12.3.1 Secure resource control

An example of using task identifiers was given in Section 11.8, where it was used to ensure that all tasks in a group received a multicast. In this subsection a simpler example is given: one of secure resource allocation.

Consider a simple resource that can be allocated and freed. It is difficult to ensure that the task which frees the resource is the one to which it was allocated without the use of task identifiers. For example, the following package unsuccessfully attempts to provide secure resource allocation. It uses a limited private key to attempt to ensure that the key cannot be assigned or copied. On release of the resource, the key is set to zero:

```
package Insecure_Resource_Allocation is
  type Key is limited private;
  Null_Key : constant Key;
  protected Controller is
    entry Allocate(K : out Key);
    procedure Free(K : in out Key);
  private
    Allocated : Boolean := False;
    Current_Key : Key;
  end Controller;
  Not_Allocated : exception;  -- raise by Free
private
  type Key is record
    Value : Natural := 0;
  end record;
  Null_Key : Key;
end Insecure_Resource_Allocation;

package body Insecure_Resource_Allocation is
  protected body Controller is
    entry Allocate(K : out Key) when not Allocated is
    begin
      if Current_Key.Value = Natural'Last then
        Current_Key.Value := 1;
      else
        Current_Key.Value := Current_Key.Value + 1;
      end if;
      Allocated := True;
      K := Current_Key;
    end Allocate;
    procedure Free(K : in out Key) is
    begin
      if K /= Current_Key then
```

```
        raise Not_Allocated;
      else
        K := Null_Key;
        Allocated := False;
      end if;
    end Free;
  end Controller;
end Insecure_Resource_Allocation;
```

Although this approach will ensure that the resource will only be released once, it is still possible for the wrong task to issue the ‘free’ call. This can occur by passing a parameter to `Allocate` which is in scope to more than one task. The following secure resource allocator solves this problem:

```
with Ada.Task_Identification; use Ada.Task_Identification;
package Secure_Resource_Allocation is
  protected Controller is
    entry Allocate;
    procedure Free;
  private
    Allocated : Boolean := False;
    Current_Owner : Task_Id := Null_Task_Id;
  end Controller;
  Not_Allocated : exception; -- raised by Free
end Secure_Resource_Allocation;

package body Secure_Resource_Allocation is
  protected body Controller is
    entry Allocate when not Allocated is
    begin
      Allocated := True;
      Current_Owner := Allocate'Caller;
    end Allocate;

    procedure Free is
    begin
      if Current_Task /= Current_Owner then raise Not_Allocated;
      else
        Allocated := False;
        Current_Owner := Null_Task_Id;
      end if;
    end Free;
  end Controller;
end Secure_Resource_Allocation;
```

The above ensures that the task allocated the resource is the one that returns it.

12.4 Task attributes

Not only is it useful to associate a unique identifier with a particular task, it can also be beneficial to assign other attributes. The Systems Programming Annex therefore provides a generic facility, `Task_Attributes`, for associating user-defined attributes with tasks:

```
with Ada.Task_Identification; use Ada.Task_Identification;
generic
  type Attribute is private;
  Initial_Value : Attribute;
package Ada.Task_Attributes is
  type Attribute_Handle is access all Attribute;

  function Value(T: Task_Id := Current_Task) return Attribute;
    -- returns the value of the corresponding attribute of T

  function Reference(T : Task_Id := Current_Task)
           return Attribute_Handle;
    -- returns an access value that designates
    -- the corresponding attribute of T

  procedure Set_Value(Val : Attribute;
                      T : Task_Id := Current_Task);
    -- performs any finalization on the old value of the
    -- attribute of T and assigns Val to that attribute

  procedure Reinitialize(T : Task_Id := Current_Task);
    -- as for Set_Value where the Val parameter
    -- is replaced with Initial_Value

end Ada.Task_Attributes;
```

12.4.1 Periodic scheduling – an example of task attributes

To illustrate the use of the task attribute facility, consider the periodic scheduling of tasks. Typically, a periodic task in Ada is structured as follows.

```
with Ada.Real_Time; use Ada.Real_Time;

...
  task Periodic_Task;

  task body Periodic_Task is
    Interval : Time_Span := Milliseconds(30);
    -- define the period of the task, 30 ms in this example
    Next : Time;
  begin
    Next := Clock;  -- start time
    loop
      -- undertake the work of the task
      Next := Next + Interval;
      delay until Next;
    end loop;
  end Periodic_Task;
```

In some applications (for example, high integrity systems), it is necessary to separate the details of the scheduling from the task itself. In these systems, the task simply wishes to execute a `Wait_Until_Next_Schedule` call.

The following package uses task attributes and task identifiers to control the temporal parameters for a set of periodic tasks:

```
with Ada.Task_Identification; use Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;
package Periodic_Scheduler is
  procedure Set_Characteristic(T : Task_Id; Period : Time_Span;
                               First_Schedule : Time);
  procedure Wait_Until_Next_Schedule; -- potentially blocking
end Periodic_Scheduler;

with Ada.Task_Attributes;
package body Periodic_Scheduler is
  Start_Up_Time : Time := Clock;
  type Task_Information is
    record
      Period : Time_Span := Time_Span_Zero;
      Next_Schedule_Time : Time :=
           Time_Of(100_000,Time_Span_Zero);
    end record;
  Default : Task_Information;
     -- a default object needs to be provided
     -- to the following package instantiation

  package Periodic_Attributes is new
          Ada.Task_Attributes(Task_Information, Default);
  use Periodic_Attributes;

  procedure Set_Characteristic(T : Task_Id; Period : Time_Span;
                               First_Schedule : Time) is
  begin
    Set_Value((Period,First_Schedule), T );
  end Set_Characteristic;

  procedure Wait_Until_Next_Schedule is
    Task_Info : Task_Information := Value;
    Next_Time : Time;
  begin
    Next_Time := Task_Info.Period +
                 Task_Info.Next_Schedule_Time;
    Set_Value((Task_Info.Period,Next_Time));
    delay until Next_Time;
  end Wait_Until_Next_Schedule;
end Periodic_Scheduler;
```

Periodic tasks can now be encoded as

```
task Periodic_Task;

task body Periodic_Task is
```

```
begin
  loop
    -- statements to be executed each period
    Periodic_Scheduler.Wait_Until_Next_Schedule;
  end loop;
end Periodic_Task;
```

and some other component of the application can allocate, and if necessary alter, the period.

12.5 Summary

This chapter has addressed the additional issues, for Ada tasking, that arise if an implementation supports the Systems Programming Annex. One set of topics addresses low level programming, the other is concerned with the extra facilities that the Annex provides.

Embedded systems must often include code for interacting with special-purpose input and output devices. Of particular interest to the topic of this book are those devices that generate interrupts. This chapter has presented a model for interrupt handling that maps an interrupt on to a parameterless protected procedure. A normal task, released by the action of the protected procedure, is used to code the response to the interrupt.

The Systems Programming Annex also defines a couple of useful packages. One provides access to a unique identifier for each task. This can be used to write general-purpose routines, such as secure resource controllers. The other package enables task attributes to be defined and used in an efficient way.

12.6 Further reading

A. Burns and A.J. Wellings, *Real-time Systems and Programming Languages*, 3rd Edition, Addison-Wesley, 2001.

13

Scheduling real-time systems – fixed priority dispatching

It has been mentioned several times already in this book that real-time programming represents a major application area for Ada, and particularly for Ada tasking. The Real-Time Systems Annex specifies additional characteristics of the language that facilitate the programming of embedded and real-time systems. If an implementation supports the Real-Time Systems Annex then it must also support the Systems Programming Annex (see previous chapter). All issues discussed in the Real-Time Systems Annex affect the tasking facilities of the language. They can be grouped together into the following topics.

- Time and clocks – introduced in Chapter 1.
- Scheduling – how to allocate system resources, in particular the processor.
- Resource control – how to monitor and manage the use of the processor by individual tasks or groups of tasks.
- Optimisations and restrictions – specifically the Ravenscar profile.

All of these topics are discussed in this and the next two chapters; starting with the important issue of scheduling.

13.1 Scheduling

The functional correctness of a concurrent program should not depend on the exact order in which tasks are executed. It may be necessary to prove that the non-determinism of such programs cannot lead to deadlock or livelock (that is, progress is always taking place), but it is not necessary to program the order in which all actions must occur explicitly.

Important note: Non-determinism in Ada programs comes from the behaviour of the run-time dispatcher (that is, the order in which tasks are released), the choice mechanism of the select statement (when more than one alternative could be taken) and the behaviour of protected objects.

Although functional correctness should not depend upon the execution order of these non-deterministic constructs, real-time programs do require control over this behaviour. A real-time program has temporal requirements that dictate the order in which events must occur and be handled.

In an Ada program, this means that the programmer must be able to control (implicitly or explicitly) the order in which tasks execute, and hence the order in which they complete their work. Once these completion patterns are known, and the resource requirements (for example, processor load) of each task have been estimated (or measured), it is possible to analyse a program and decide if it will meet all its timing requirements. This analysis determines the *schedulability* of the program. The timing requirements themselves usually take the form of deadlines on the completion of a set of actions of particular tasks.

Scheduling is therefore concerned with controlling the order of execution of tasks. Most real-time tasks are either *periodic* or *sporadic*. A periodic task typically has an infinite loop within which is a 'delay until' statement that ensures that the task executes regularly:

```
with Ada.Real_Time; use Ada.Real_Time;
...
  task Periodic_Task;

  task body Periodic_Task is
    Interval : Time_Span := Milliseconds(30);
    -- define the period of the task, 30 ms in this example
    Next : Time;
  begin
    Next := Clock;  -- start time
    loop
      -- undertake the work of the task
      Next := Next + Interval;
      delay until Next;
    end loop;
  end Periodic_Task;
```

Task attributes can be used to centralise control over the key timing characteristics such as period, as illustrated in the Subsection 12.4.1.

A sporadic task requires a protected object to control its invocation (its release for execution):

```
  task Sporadic_Task;
```

```
protected Sporadic_Controller is
  entry Wait_Next_Invocation;
  procedure Release_Sporadic;
private
  Barrier : Boolean := False;
end Sporadic_Controller;

task body Sporadic_Task is
begin
  loop
    Sporadic_Controller.Wait_Next_Invocation;
    -- undertake the work of the task
  end loop;
end Sporadic_Task;

protected body Sporadic_Controller is
  entry Wait_Next_Invocation when Barrier is
  begin
    Barrier := False;
  end;

  procedure Release_Sporadic is
  begin
    Barrier := True;
  end;
end Sporadic_Controller;
```

A complete real-time program will typically contain a known number of periodic and sporadic tasks and a number of *event handlers*. One example of an event is the standard interrupt discussed in the previous chapter; other forms of events and event handlers will be introduced in the Chapter 15.

Ada 2005 change: Earlier versions of Ada only supported a single form of scheduling known as *preemptive fixed priority dispatching*. The language now has a significantly greater set of provisions: *non-preemptive fixed priority dispatching*, *round robin dispatching* and *earliest deadline first (EDF) dispatching*. Moveover it allows an application to mix scheduling schemes within the same program.

The remainder of this chapter deals with the fixed priority dispatching that may be consider the *standard* policy. This policy also forms the basis from which the other policies are defined. These other policies are described in the next chapter.

13.2 Fixed priority dispatching

Where more than one task is runnable, the actual order of execution can be determined by assigning a unique priority to each task. If two or more tasks are contending for the same processor, then the one with the highest priority will be the one that is actually dispatched. Priority can also be used to order entry queues and determine the choices made by the select statement and protected objects.

Important note: The use of priority at the language level is merely a mechanism for ordering executions. The meaning, or semantics, of the notion of priority is an issue solely for the programmer.

In some applications, priority will be used to imply criticality: hence the most critical tasks will always execute, and finish, first. In the domain of real-time programming, priority is used to represent the urgency with which tasks must complete. Urgency is a reflection of the timing characteristics of each task.

Fixed priority scheduling has been the subject of considerable research over the last three decades. It now represents a mature engineering technique for real-time programming. Priorities are assigned by an appropriate algorithm (such as rate monotonic, or deadline monotonic†) and then tests are applied to the task set to see if the complete program is schedulable (i.e. will always meet all its deadlines). It is beyond the scope of this book to include details of the various forms of scheduling analysis. The references at the end of the chapter provide detailed treatments of these topics.

To select the required dispatching policy, a pragma is defined within the Real-Time Systems Annex:

```
pragma Task_Dispatching_Policy(Policy_Identifier);
```

The Annex defines a number of possible policies. The fixed priority scheme is requested as follows:

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
```

Important note: Where tasks share the same priority, they are queued in FIFO order. Hence, as tasks become runnable, they are placed at the back of a notional run queue for that priority level. One exception to this case is when a task is preempted; here the task is placed at the front of the notional run queue for that priority level.

Dispatching polices (and other such policies) are actually defined on a per-partition basis. Partitions are units of configuration used mainly in distributed programs.

† With these algorithms the priority of a task is derived from its period or deadline. For rate monotonic, the priority of each of a set of tasks is assigned in an inverse relation to task periods – so the shorter a task's period, the higher its priority. The other popular scheme, deadline monotonic, uses the relative deadline of each task to allocate priority – the shorter the relative deadline, the higher the priority. For periodic systems with the added restriction that relative deadline must equal period, the two algorithms result in the same priority assignments. The notion of deadline and the definition of relative deadline are covered in the next chapter.

Warning: On a multiprocessor platform, it is implementation defined whether this fixed priority dispatching policy is on a per-processor basis or across the entire processor cluster.

Base priorities

The provisions of the Real-Time Systems Annex allow priorities to be assigned to tasks (and protected objects). Recall that from package `System` there are the following declarations:

```
subtype Any_Priority is Integer range <implementation-defined>;
subtype Priority is Any_Priority range
                    Any_Priority'First .. <implementation-defined>;
subtype Interrupt_Priority is Any_Priority range
                    Priority'Last+1 .. Any_Priority'Last;

Default_Priority : constant Priority :=
                   (Priority'First + Priority'Last)/2;
```

An integer range is split between standard priorities and the (higher) interrupt priority range. An implementation must support a range for `System.Priority` of at least 30 values and at least 1 distinct `System.Interrupt_Priority` value. Most scheduling theories work optimally if each task has a distinct priority. Hence, ranges in excess of 30 are beneficial.

A task's initial priority is set by including a pragma in its specification:

```
task Periodic_Task is
  pragma Priority(System.Default_Priority + 1);
end Periodic_Task;
```

If a task-type definition contains such a pragma, then all tasks of that type will have the same priority unless a discriminant is used:

```
task type Servers(Task_Priority : System.Priority) is
   entry Service1(...);
   entry Service2(...);
   pragma Priority(Task_Priority);
end Servers;
```

If the pragma is omitted, the default is assigned.

For protected objects acting as interrupt handlers, a special pragma is defined:

```
pragma Interrupt_Priority(Expression);
```

or simply

```
pragma Interrupt_Priority;
```

The definition, and use, of a different pragma for interrupt levels improves the readability of programs and helps to remove errors that can occur if task and interrupt priority levels are confused. If the `Ceiling_Locking` policy is in effect (see next section) and the associated protected object is used as an interrupt handler (i.e. it also has an `Attach_Handler` or `Interrupt_Handler` pragma) then the expression used in `Interrupt_Priority` must evaluate down to an `Interrupt_Priority` value. If the expression is omitted, the highest possible priority is assigned.

Definition: A priority assigned using one of these priority-pragmas is called a *base priority*.

A task also has an *active* priority – this will be explained in due course.

The main program, which is executed by a notional environmental task, can have its priority set by placing the `Priority` pragma in the main subprogram. If this is not done, the default value, defined in package `System`, is used. Any other task that fails to use the pragma has a default base priority equal to the base priority of the task that created it.

13.3 Priority ceiling locking

In Chapter 7, it was explained that protected objects have the fundamental property of ensuring mutually exclusive access to the internal data of the object. It was not explained how an implementation should ensure this mutual exclusion. In this section, it is shown how an inappropriate implementation scheme (that is, one without inheritance) can undermine the priority model used for ensuring timely behaviour. The difficulties with this scheme motivate the use of the locking protocol defined in the Real-Time Systems Annex.

Consider, for illustration, a three-task system. The tasks have high, medium and low priority, and will be identified by the labels H, M and L. Assume that H and L share data that is encapsulated in a protected object, P. The following execution sequence is possible:

(1) L is released, executes and enters P.

(2) M is released and preempts L while it is executing in P.

(3) H is released and preempts M.

(4) H executes a call on P.

Now, H cannot gain access to P as L has the mutual exclusion lock. Hence H is suspended. The next highest priority task is M and hence M will continue. As a result, H must wait for M to finish before it can execute again (see Figure 13.1). This phenomenon is known as *priority inversion*.

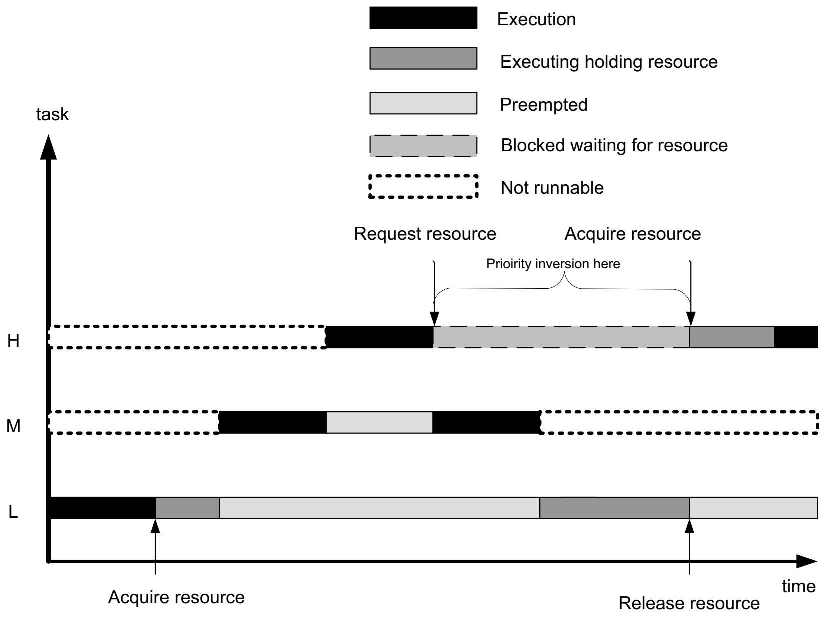


Fig. 13.1: Priority inversion

Definition: Priority inversion occurs when the priority ordering has been subverted – a high-priority task is kept from executing by a low-priority task that is running.

In the above scenario, priority inversion is occurring as M is holding up the higher-priority task H.

To limit the detrimental effect of priority inversion, some form of *priority inheritance* must be used.

Definition: Priority inheritance allows a task to execute with an enhanced priority if it is blocking (or could block) a higher-priority task.

There are a number of priority inheritance protocols; the one defined below corresponds to that supported by the Real-Time Systems Annex. It is known as the *Immediate Ceiling Priority Protocol* (ICPP) (often also called the *Ceiling Priority Emulation* or the *Priority Protect Protocol*). First, a ceiling priority is defined for each protected object.

Definition: The priority ceiling of a protected object represents the maximum priority of any task that calls the object.

Whenever a task executes within a protected object, it does so with the ceiling

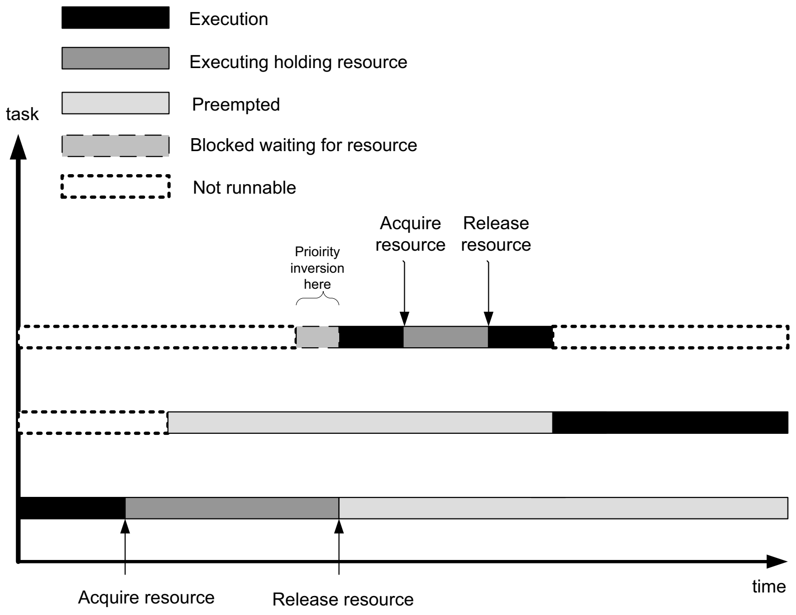


Fig. 13.2: Ceiling priority inheritance

priority (that is, its priority is raised – assuming it was lower before the call). The simple three-task system will now behave as follows (see Figure 13.2):

(1) L is released, executes and enters P; its priority is raised to that of H (at least).
(2) M is released but does not execute as the priority of M is less than the current priority of L.
(3) H is released but does not execute as its priority is not higher than L at this time.
(4) L exits P and has its priority lowered.
(5) H can now execute and will enter and leave P when required.

The result is that H is blocked for the minimum time.

Warning: Priority inversion cannot be entirely removed as the integrity of the protected data must be ensured.

Hence L must be able to block H for the time it is actually within P.

The use of the ICPP has a number of benefits in addition to minimising priority inversion. On a single-processor system, the protocol will itself ensure mutual exclusion. While a task is executing within the protected object, it must have a priority higher than any other task that may wish to call that object. Hence no other user of the object can call the object, and mutual exclusion is ensured.

Important note: There are two methods of implementing ceiling priorities: either the protected object has an assigned priority just greater than any calling task, or it has a priority equal to the highest priority caller. In the latter case, the task with priority equal to the ceiling cannot preempt another task with the same priority.

The ICPP also has three other useful properties:

- A task can only be blocked at the very beginning of its execution (for example, when it has just been made runnable).
- A task can only suffer a single block (that is, when it is released there is at most one lower priority task holding a resource that it requires).
- If all resources are accessed by protected objects, with appropriate ceiling priorities, then deadlocks cannot occur.

These properties only apply to single-processor systems. Their derivation is beyond the scope of this chapter.

In order to make use of the ICPP, an Ada program must include the following pragma:†

```
pragma Locking_Policy(Ceiling_Locking);
```

An implementation may define other locking policies; however, only `Ceiling_Locking` is required by the Real-Time Systems Annex. The default policy, if the pragma is missing, is implementation defined. To specify the ceiling priority for each object, the `Priority` and `Interrupt_Priority` pragmas defined in the previous section are used. If the pragma is missing, a ceiling of `System.Priority'Last` is assumed unless there is an `Attach_Handler` or `Interrupt_Handler` pragma within the object's specification. In the latter case, the ceiling value is implementation defined but within the range of `Interrupt_Priority`.

Warning: The exception `Program_Error` is raised if a task calls a protected object with a priority greater than the defined ceiling. If such a call were allowed, this could result in the mutually exclusive protection of the object being violated. If it is an interrupt handler that calls in with an inappropriate priority, then the program becomes erroneous. This must be prevented through adequate testing and/or static analysis of the program.

† If this locking policy is defined to apply to a partition then the implementation must also allow `FIFO_Within_Priorities` to be specified for the same partition.

With Ceiling_Locking, an effective implementation will use the thread of the calling task to execute not only the code of the protected call, but also the code of any other task that happens to have been released by the actions of the original call. For example, consider the following simple protected object:

```
protected Gate_Control is
  pragma Priority(28);
  entry Stop_And_Close;
  procedure Open;
private
  Gate: Boolean := False;
end Gate_Control;

protected body Gate_Control is
  entry Stop_And_Close when Gate is
  begin
    Gate := False;
  end Stop_And_Close;

  procedure Open is
  begin
    Gate := True;
  end Open;
end Gate_Control;
```

Assume a task T, priority 20, calls Stop_And_Close and is blocked. Later, task S (priority 27) calls Open. The thread that implements S will undertake the following actions:

(a) execute the code of Open for S;
(b) evaluate the barrier on the entry and note that T can now proceed;
(c) execute the code of Stop_And_Close on behalf of T;
(d) evaluate the barrier again;
(e) continue with the execution of S after its call on the protected object.

As a result, there has been no context switch. The alternative is for S to make T runnable at point (b) and leave the protected object; T now has a higher priority (28) than S (27) and hence the system must switch to T to complete its execution within Gate_Control. As T leaves, a switch back to S is required. This is much more expensive.

From what has already been said in this chapter, it should be clear that FIFO_Within_Priorities and Ceiling_Locking when used together form a very effective scheduling scheme. It should be noted however that *priority* is being used for two distinct purposes in this combination:

- it is used to order the execution of tasks, and

- it is used to control access to shared objects.

In other scheduling policies, priority is only used for the second property. This point will be returned to in the section on EDF scheduling in the next chapter.

13.4 Entry queue policies

A programmer may also choose a queuing policy for a task or protected object entry queue, and for the behaviour of the selection policy for open entries in a protected object and the select statement. Again, a pragma is used:

```
pragma Queuing_Policy (Policy_Identifier);
```

There are two predefined policies with this pragma: `FIFO_Queuing`, which is the default policy, and `Priority_Queuing`. An implementation may define other policies.

The priority policy behaves as expected: tasks are queued, and hence serviced, in priority order.

Important note: With the select statement or protected objects, an entry that is open and has the highest-priority task queued (across all open entries) is chosen. If two open entries have tasks at the head of their queues with equal priority, then the alternative which is textually first is selected.

Although the second rule seems a little contrived, it is a compromise between needing to define the exact semantics of the select and not wishing to add extra syntax. Note that, if FIFO queuing is chosen, the selection process is arbitrary (i.e., not defined).

13.5 Active priorities and dispatching policies

Before moving on, in the next chapter, to the other dispatching policies supported by Ada 2005, it is necessary to describe in a little more detail the dispatching model. This abstract model forms the basis for all the scheduling policies within the language.

As a task enters a protected object, its priority may rise above the base priority level defined by the `Priority` or `Interrupt_Priority` pragma. The priority used to determine the order of dispatching is the *active priority* of a task.

Definition: A task's active priority is a function of the task's base priority and any priority it has inherited. For fixed priority dispatching, the active priority is the maximum of the base priority and the inherited priorities.

The use of a protected object is one way in which a task can inherit a higher active priority. There are others, for example:

- During activation – a task will inherit the active priority of the parent task that created it (if it is higher than the child's base priority); remember the parent task is blocked waiting for its child task to complete, and this could be a source of priority inversion without this inheritance rule.
- During a rendezvous – the task executing the accept statement will inherit the active priority of the task making the entry call (if it is greater than its own priority).

Warning: Note that the last case does not necessarily remove all possible cases of priority inversion. Consider a server task, `S`, with entry `E` and base priority `L` (low). A high-priority task makes a call on `E`. Once the rendezvous has started, `S` will execute with the higher priority. But before `S` reaches the accept statement for `E`, it will execute with priority `L` (even though the high-priority task is blocked). This, and other candidates for priority inheritance, can be supported by an implementation, but there is no language requirement to do so. The assumption of the language designers is that to get true real-time behaviour, priority inversion must be appropriately bounded by using protected objects rather than direct inter-task rendezvous.

If an implementation provides addition facilities it must provide a pragma that the user can employ to select the additional conditions explicitly.

Ada's dispatching policies are specified in terms of conceptual *ready queues*. A ready queue is an ordered list of ready tasks (i.e. tasks that are runnable but are not actually executing). For each priority value, there is a distinct ready queue. Tasks are placed on the queue for their active priority. The dispatching policy will determine where in the queue the task is placed.

Definition: With every dispatching policy there are defined *task dispatching points*. Whenever the running task reaches such a point, it is (conceptually) placed back on the ready queue for its active priority and the most eligible task is then chosen for execution – this is the task at the head of the highest-priority non-empty ready queue. Note this task will often be the task that was previously executing.

For `FIFO_Within_Priorities`, dispatching points occur whenever there is a runnable task with a higher priority (i.e. preemptive dispatching) or when the task executes a delay statement that does not result in blocking (e.g. **`delay`** `0.0`).

Obviously, whenever a task blocks or terminates, a new task is chosen for execution.

If the active priority changes while the task is suspended, then this has no effect. However, if the base priority of a runnable task changes (see next chapter) then the task is logically removed from its current ready queue and is subsequently replaced in a new queue that corresponds to its new priority.

The use of ready queues helps to define the required run-time behaviour of a system of tasks. It is described as being 'conceptual' to emphasise that an implementation may use other more efficient mechanisms as long as the behaviour is equivalent.

13.6 Summary

Many concurrent systems are required to execute in real-time (and almost all real-time systems are inherently concurrent). To support real-time applications, Ada defines a number of facilities in the Real-Time Systems Annex of the ARM.

In a non-real-time system, it is acceptable for any task that is executable to be given the available processor resources at any time. With real-time systems, control must be exercised over the allocation of system resources. The easiest way to do this is to give priorities to tasks and to use the notion of priority to obtain deterministic behaviour that can be analysed. One popular scheduling scheme uses fixed priorities for tasks and ceiling priorities for protected objects. This model is directly supported by the provisions of the Real-Time Systems Annex.

13.7 Further reading

A. Burns and A.J. Wellings, *Real-time Systems and Programming Languages*, 3rd Edition, Addison-Wesley, 2000.

G.C. Buttazzo, *Hard Real-Time Computing Systems*, 2nd Edition, Springer, 2005.

M.H. Klein, T. Ralya, B. Pollak, R. Obenza, M. Gonzalez Harbour, *A Practitioner's Handbook for Real-Time Analysis*, Kluwer, 1993.

H. Kopetz, *Real-Time Systems*, Kluwer, 1997.

J.W.S. Liu, *Real-Time Systems*, Prentice-Hall, 2000.

S.H. Son (editor), *Advances in Real-Time Systems*, Prentice-Hall, 1994.

14
Scheduling real-time systems – other dispatching facilities

This chapter considers three additional scheduling policies: non-preemptive priority based dispatching, round-robin dispatching and earliest deadline first dispatching. Consideration is also given as to how such policies can be used in tandem on the same processing resource.

As well as supporting explicit dispatching policies, Ada also provides a number of primitives that allow the programmer to construct and control their own dispatching behaviour. These features together with the above named policies are the subject of this chapter.

To support the use of other dispatching policies, Ada introduces a language-defined package that is used as the parent of further specifications that will described in the following pages:

```
package Ada.Dispatching is
  pragma Pure(Dispatching);
  Dispatching_Policy_Error : exception;
end Ada.Dispatching;
```

This package clearly introduces nothing really significant. It is used to define a parent for a number of child packages that are needed with the different policies. It also defines an exception that will be raised when an operation is applied in an inappropriate context (i.e. to the wrong dispatching policy). Examples of its use are also given in the following discussions.

14.1 Non-preemptive dispatching

The traditional way of implementing many high-integrity applications is with a cyclic executive. Here, a sequence of procedure calls are made within a defined time interval. Each procedure runs to completion; there is no concept of preemption. Data is passed from one procedure to another via shared variables, no synchronisation constraints are needed, as the procedures never run concurrently.

Whilst many system builders are prepared to move to use tasking (as defined by the Ravenscar profile – see Section 17.2) some are reluctant to embrace the preemptive dispatching policy. They prefer the reduced non-determinism of non-preemptive dispatching, which also increases the effectiveness of testing.

Definition: To support non-preemptive dispatching, Ada 2005 has defined a policy entitled `Non_Preemptive_FIFO_Within_Priorities`. The definition of this policy is identical to `FIFO_Within_Priorities` except for the important removal of the need to preempt when a higher priority task is runnable. This situation is *not* a task dispatching point for this policy.

It should be noted that non-preemption is not synonymous with non-interruptible. With non-preemption, a task may lose its processor while an interrupt is being handled (including the timer interrupt). But control will pass back to the original task when the handler has completed – and this will occur even if a higher priority task has been released by the action of the handler.

Warning: In general non-preemption reduces schedulability as a long low priority task will continue to execute even when a high priority task with a tight deadline is released.

To reduce the impact of this behaviour a task can periodically volunteer to be preempted. It can do this by executing '**`delay`** `0.0`', which is a task dispatching point for this non-preemption policy. This form of scheduling is called *cooperative* or *deferred preemption*.

As a task cannot suspend itself within a protected object, a non-preemptive scheme will never cause a task switch while an ordinary protected action is being executed. As a consequence, an implementation can optimise its execution by never changing the priority of a task as it enters (and leaves) a protected object. The exception to this rule is any protected object that is used as an interrupt handler – but these are easily recognised by the inclusion of one or more of the pragmas `Interrupt_Handler, Attach_Handler` and `Interrupt_Priority`.

14.2 Round-robin dispatching

As discussed earlier in this chapter, preemptive priority based dispatching is a natural choice for real-time applications. It can be implemented efficiently and leads to the development of applications that are amenable to effective analysis – especially when combined with ceiling locking of protected objects. There are, however, application requirements that cannot be fully accomplished with this policy alone.

For example, many applications have a mixture of real-time and non-real-time activities. The natural way of scheduling non-real-time tasks is by time sharing the processor, as in most general-purpose operating systems. With the standard policy (`FIFO_Within_Priorities`), some level of rotation can be achieved by giving the set of non-real-time tasks the same priority and requiring them to incorporate periodic yield operations (such as **`delay`** `0.0`). However, this is intrusive, and cannot easily be undertaken with legacy code or when using shared libraries. Real-time applications can also benefit from a round-robin approach. Although extra task switches increase run-time overheads, round-robin execution allows a set of tasks (with the same priority) to make progress at a similar rate.

Definition: Ada 2005 defines a round-robin dispatching policy using the identifier: `Round_Robin_Within_Priorities`.

As the name implies, the round-robin scheme operates within the priority model. All tasks that share a specific priority level share the same round-robin queue.

If the following pragma is defined for a partition then all tasks with priorities in the range defined by `System.Priority` are scheduled by the policy; all priorities within the range of `System.Interrupt_Priority` are allocated the `FIFO_Within_Priorities` policy.

```
pragma Task_Dispatching_Policy(Round_Robin_Within_Priority);
```

Although this policy can be used in this way and applied to all priority levels, it is more usual to combine it with other policies. This will be illustrated later (see Section 14.4); in the remainder of this section, it is assumed that it is the only policy being applied.

A general round-robin scheme defines a *quantum* of execution time. When a task executes it uses up its quantum; when no time is left, the task is preempted and moved to the back of the ready queue for its (active) priority. The particular scheme defined within Ada 2005 has the following support package:

```
with System, Ada.Real_Time;
use Ada;
package Ada.Dispatching.Round_Robin is
  Default_Quantum : constant Real_Time.Time_Span :=
             <implementation-defined>;
  procedure Set_Quantum(Pri : System.Priority;
             Quantum : Real_Time.Time_Span);
  procedure Set_Quantum(Low,High : System.Priority;
             Quantum : Real_Time.Time_Span);
  function Actual_Quantum
            (Pri : System.Priority) return Real_Time.Time_Span;
  function Is_Round_Robin
            (Pri : System.Priority) return Boolean;
end Ada.Dispatching.Round_Robin;
```

A call of either Set_Quantum procedure sets the required quantum value for the specific priority or the range of priorities (in the second procedure). If no quantum is set for a priority level, Default_Quantum is used. The function Actual_Quantum returns the actual quantum used by the implementation for the specified priority level. In some implementations it will not be possible to support all values that can be requested and hence the actual quantum used can be queried. If a program asks for this data for a priority level that is not subject to this policy, then the exception defined in the parent package will be raised. The function Is_Round_Robin can be used as a precondition to the call of Actual_Quantum.

Definition: The dispatching rules for the round-robin policy are as follows:

- The dispatching points are identical to those for FIFO_Within_Priorities with the addition of the expiry of a budget.
- When a task is added to the tail of the ready queue for its base priority, it has an execution-time budget set equal to the quantum for that priority level. This will also occur when a blocked task becomes executable again.
- When a task is preempted (by a higher priority task) and is added to the head of the ready queue for its priority level, it retains its remaining budget.
- When a task is executing, its budget is decreased by the amount of execution time it uses.
- When a task has exhausted its budget, and is without an inherited priority (and is not executing within a protected operation), it is moved to the tail of the ready queue for its priority level and is given a budget equal to the quantum for its priority level.

The last rule perhaps requires some explanation. It is crucial to the semantics of protected objects that a task cannot be preempted while executing a protected operation by any task that could also call the same protected object. The definition of the priority ceiling protocol ensures this as long as tasks do not block or otherwise cease executing while inheriting a ceiling priority. The round-robin scheme must, therefore, not allow the quantum to be exhausted while the affected task is inside a protected object – or in any other way has an active priority higher than its base priority. The scheme defined in Ada 2005 does not measure any overrun nor compensate for such an overrun, the task will get the same quantum for its next execution irrespective of whether it used more than its budget ‘last time’.

14.3 Earliest deadline first dispatching

Embedded systems are often subject to stringent timing requirements, which are usually expressed in terms of *deadlines*. They are also often subject to severe resource constraints. Hence optimal (or at least near optimal) resource allocation is required. Earliest deadline first (EDF) dispatching is a popular paradigm as it has been proven to be the most efficient scheme available in the following sense: if a set of tasks is schedulable by any dispatching policy then it will also be schedulable by EDF. Whilst fixed priority dispatching is the most popular scheduling scheme (and is certainly the one to be used in high-integrity application), for many applications EDF is the preferred approach. Ada 2005 supports both schemes, moreover it allows them to be used together in an integrated way. This mixed behaviour is consider in a later section (14.4) – here, a pure EDF approach is assumed.

To support EDF dispatching, three language features are required:

- a formal representation of the deadline of a task,
- use of deadlines to control dispatching and
- a means of sharing data via protected objects that is compatible with EDF dispatching.

Each of these issues is considered in turn.

Warning: Ada's support for deadline-based scheduling does not extend to any general notion of deadline inheritance. See Subsection 14.3.6.

14.3.1 Representing deadlines

A deadline is usually defined to be *absolute* if it refers to a specific (future) point in time; for example there was an absolute deadline of 31st December 2006 for the completion of this book (presumably at 23:59:59). A *relative* deadline is one that is anchored to the current time: 'There is six months to finish the book'.† Obviously, if the current time is known then a relative deadline can be translated into an absolute one. Repetitive (periodic) tasks often have a static relative deadline (for example 10 ms after release) but will have a different absolute deadline every time they are released.

Definition: The EDF scheme uses absolute deadline – it requires that the task with the shortest (earliest) absolute deadline is the next to be executed. The EDF scheme is perhaps more accurately expressed as *earliest absolute deadline first*.

† At the time of writing!

Although a program could employ a user-defined attribute to represent a task's deadline, a direct representation is provided in Ada 2005:

```
with Ada.Real_Time;
with Ada.Task_Identification;
use Ada;
package Ada.Dispatching.EDF is
  subtype Deadline is Real_Time.Time;
  Default_Deadline : constant Deadline :=
              Real_Time.Time_Last;
  procedure Set_Deadline(D : in Deadline;
            T : in Task_Identification.Task_ID :=
            Task_Identification.Current_Task);
  procedure Delay_Until_And_Set_Deadline(
            Delay_Until_Time : in Real_Time.Time;
            TS : in Real_Time.Time_Span);
  function Get_Deadline(T : in Task_Identification.Task_ID :=
           Task_Identification.Current_Task) return Deadline;
end Ada.Dispatching.EDF;
```

The identifier `Deadline` is explicitly introduced even though it is a direct subtype of the time type from `Ada.Real_Time`.

The `Set` and `Get` subprograms have obvious utility. A call of `Delay_Until_And_Set_Deadline` delays the calling task until time `Delay_Until_Time`. When the task becomes runnable again, it will have deadline `Delay_Until_Time + TS`. The inclusion of this procedure reflects a common task structure for periodic activity. Consider the earlier example of a periodic task; now assume it has a deadline at end of its execution (i.e. every time it executes it should finish before its next release):

```
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Dispatching.EDF; use Ada.Dispatching.EDF;

...

  task Periodic_Task;

  task body Periodic_Task is
    Interval : Time_Span := Milliseconds(30);
    -- define the period of the task, 30 ms in this example
    Next : Time;
  begin
    Next := Clock;  -- start time
    Set_Deadline(Clock+Interval);
    loop
      -- undertake the work of the task
      Next := Next + Interval;
      Delay_Until_And_Set_Deadline(Next,Interval);
    end loop;
  end Periodic_Task;
```

If, rather than use this language-defined delay procedure, the task had set a deadline and then used a delay until statement, this would most likely have resulted in an extra unwanted task switch (first the deadline is extended and hence a more urgent task will preempt, later the task will execute again just to put itself on the delay queue).

With EDF, all dispatching decisions are based on deadlines, and hence it is necessary for a task to always have a deadline.† However, a task must progress through activation before it can get to a position to call `Set_Deadline` and hence a default deadline value is given to all tasks (`Default_Deadline` defined in `Ada.Dispatching.EDF`). But this default value is well into the future and hence activation will take place with very low urgency (all other task executions will occur before this task's activation). If more urgency is required, the following language defined pragma is available for inclusion in a task's specification (only):

```
pragma Relative_Deadline(Relative_Deadline_Expression);
```

where the type of the parameter is `Ada.Real_Time.Time_Span`. The initial absolute deadline of a task containing this pragma is the value of `Ada.Real_Time.Clock + Relative_Deadline_Expression`. The call of the clock being made at sometime between task creation and the start of its activation. The actual timing of the call is implementation defined.

So the example above should use this pragma rather than include the first call of `Set_Deadline`:‡

```
task Periodic_Task is
  pragma Relative_Deadline(Milliseconds(30));
end Periodic_Task;
```

Important note: A final point to note with the deadline assignment routine concerns when it will take effect. The usual result of a call to `Set_Deadline` is for the associated deadline to be increased into the future and that a task switch is then likely (if EDF dispatching is in force). As this would not be appropriate if the task is currently executing within a protected object, the setting of a task's deadline to the new value takes place as soon as is practical but not while the task is performing a protected action. This is similar to the rule that applies to changes to the base priority of a task using `Set_Priority` – see later in Section 14.5.

† This is one of the criticisms of EDF – even a task that had no actual deadline must be given an artificial one so that it will be scheduled. For example, a general server task must be given a deadline even though its actual deadline should depend on the deadline of its current client.

‡ This assumes that the parent task has no deadline or a longer deadline. If this is not the case, the child task's activation deadline must reflect that of the parent (as the parent is blocked until the child finishes its activation).

Warning: The explicit use of deadlines in real-time programs is often necessary, but this does not mean that EDF dispatching must be used. The standard fixed priority scheme can be employed with no reference to the deadlines of the tasks, but the deadline may still be needed if deadline misses are to be caught and handled (see example below). Note Ada requires the use of package `Ada.Dispatching.EDF` to gain access to its deadline abstraction even if EDF dispatching is not used. This can lead to some confusion, as the name of the package seems to imply EDF dispatching.When in effect, it is the use of the dispatching pragma that determines the dispatching policy.

The following illustrates how a program can catch a deadline overrun via the asynchronous transfer of control (select-then-abort) feature:

```
loop
  select
    delay until Ada.Dispatching.EDF.Get_Deadline;
    -- action to be taken when deadline missed
  then abort
    -- code
  end select;
end loop;
```

Note that the `Get_Deadline` routine is called just once and fixes the point in time of the deadline – if the deadline of the tasks is altered in its ‘code’ this will have no impact on the asynchronous trigger. If the programmer wishes to cater for changing deadlines it would need to use a protected object and program the necessary trigger using a timing event – this is considered again in Section 15.3.

Important note: If deadlines are used with dispatching policies other than EDF then changes to deadlines are *not* task dispatching points.

14.3.2 Dispatching

To request EDF dispatching, the following use of the policy pragma is supported:

```
pragma Task_Dispatching_Policy(EDF_Across_Priorities);
```

The main result of employing this policy is that the ready queues are now ordered according to deadline (not FIFO). Each ready queue is still associated with a single priority but at the head of any ready queue is the runnable task with the earliest deadline. Whenever a task is added to a ready queue for its priority, it is placed in the position dictated by its current absolute deadline.

The active priority of a task, under EDF dispatching, is no longer directly linked to the base priority of the task. The rules for computing the active priority of a task are somewhat complex and are covered in the next subsection – they are derived from consideration of each task's use of protected objects. For a simple program with no protected objects, the following straightforward rules apply:

- any priorities set by the tasks are ignored;
- all tasks are always placed on the ready queue for the priority value `System.-Priority'First`.

Hence only one ready queue is used (and that queue is ordered by deadline).

Of course real programs require task interactions, and for real-time programs this usually means the use of protected objects. To complete the definition of the EDF dispatching policy, the rules for using protected objects must be considered in some detail. An understanding of these rules will enable the reader to understand why the above simple semantics apply to programs without protected objects.

14.3.3 EDF dispatching and the priority ceiling protocol

One of the major improvements that Ada 95 provided was the introduction of the protected object. Although the rendezvous is a powerful synchronisation and communication primitive, it does not easily lead to tight scheduling analysis. Rather, a more asynchronous scheme is desirable for real-time applications. It has already been shown that the locking policy `Ceiling_Locking` is well suited for use with `FIFO_Within_Priorities`. Ada 2005 has defined `EDF_Across_Priori-ties` to also work with `Ceiling_Locking`. Indeed if EDF dispatching is requested, `Ceiling_Locking` must also be specified for that partition.

As explain in the previous chapter, when fixed priorities and ceiling priorities are employed together, the notion of 'priority' is used to control dispatching and access to protected objects. Baker (1991) showed that it is possible to abstract away from this particular model with a single use of 'priority' to one that has two measures: *urgency* and *preemption level*. He showed that if preemption levels are related to urgency in a specific way then the concurrent tasks can access shared objects (protected objects) in a way that has the important (ceiling protocol) properties, namely

- a task only suffers at most a single block per invocation from a lower priority task,
- deadlocks are prevented and
- mutual exclusion is provided by the protocol itself.

This protocol is called the Preemption Level Control Protocol (PLCP) in the following discussion.

Entity	Period	Usage	Task PL	Ceiling PL
T1	10	P1	6	
T2	12	P2	5	
T3	15	P3	4	
T4	20	P1, P3	3	
T5	30	P2	2	
P1		T1, T4		6
P2		T2, T5		5
P3		T3, T4		4

Table 14.1: An example task set

Important note:

Ada 2005 supports a version of PLCP by using:

- deadline to represent urgency,
- base priority to represent each task's preemption level,
- ceiling priorities to represent preemption levels for protected objects and
- standard `Ceiling_Locking` for access to protected objects.

So each task has a base priority but this is not used directly to control dispatching, EDF controls dispatching.

Before explaining in detail the dispatching policy, the following illustrates a straightforward application of the PCLP. Assume an application consists of five tasks, T1, ..., T5, and three protected objects, P1, P2, P3. The tasks are all periodic with periods given in Table 14.1 and deadlines equal to their associated periods (i.e. every task must always complete before it should next be released). The optimal way to assign preemption levels is *deadline monotonic* – so the shorter the relative deadline, the higher the preemption level. This is the identical (optimal) method of assigning priorities in a fixed priority scheme. Table 14.1 includes the preemption levels (PLs) of the tasks and the ceiling preemption levels for the protected objects. To execute this program with EDF dispatching requires the use of 'priority' to assign preemption levels to each task and protected object, and then the usual behaviour when a task calls a protected object – it will run with the ceiling value whilst inside the object. As its initial priority is `Priority'First` (because of EDF rules, and assumed to be 1 in this example) then the preemption levels are the same as priorities in terms of the behaviour required by `Ceiling_Locking`.

To return to the dispatching algorithm. The PCLP states that a newly released task, T1 say, preempts the currently running task, T2 say, if and only if:

- the deadline of T1 is earlier than the deadline of T2, and

- the preemption level of T1 is higher than the preemption level of any locked protected object (i.e. protected objects that are currently in use by any task in the system).

To keep track of all locked protected objects is an implementation overhead and hence the rules defined for Ada 2005 have the following form. Remember that if `EDF_Across_Priorities` is defined then all ready queues within the range `Priority'First .. Priority'Last` are ordered by deadline. Now rather than always place tasks in the queue for `Priority'First` the following rules apply:

- Whenever a task T is added to a ready queue, other than when it is preempted, it is placed on the ready queue with the highest priority R, if one exists, such that:
  - another task, S, is executing within a protected object with ceiling priority R; and
  - task T has an earlier deadline than task S; and
  - the base priority of task T (its preemption level) is greater than R.

  If no such ready queue exists, the task is added to the ready queue for `Priority'First`.
- When a task is chosen for execution, it runs with the active priority of the ready queue from which the task was taken. If it inherits a higher active priority, it will return to its original active priority when it no longer inherits the higher level.

It follows that if no protected objects are in use at the time of the release of T then T will be placed in the ready queue at level `Priority'First` at the position dictated by its deadline.

Important note: A task dispatching point occurs for the currently running task T whenever:

- a change to the deadline of T takes effect; or
- a decrease to the deadline of any task on a ready queue for that processor takes effect and the new deadline is earlier than that of the running task; or
- there is a non-empty ready queue for that processor with a higher priority than the priority of the running task.

So dispatching is preemptive, but it may not be clear that the above rules implement the PCLP. Consider three scenarios. Remember in all of these behaviours, the running task is always returned to its ready queue whenever a task arrives. A task (possibly the same task) is then chosen to become the running task following the rules defined above.

The system contains four tasks T1, T2, T3 and T4, and three resources that are implemented as protected objects: R1, R2 and R3. Table 14.2 defines the parameters of these entities.

Task	Relative deadline D	Preemption level L	Uses resources	Arrives at time	Absolute deadline A
T1	100	1	R1,R3	0	100
T2	80	2	R2,R3	2	82
T3	60	3	R2	4	64
T4	40	4	R1	6	46

Table 14.2: A task set (time attributes in milliseconds)

Consider just a single invocation of each task. The arrival times have been chosen so that the tasks arrive in order of the lowest-preemption-level task first etc. It is assumed that all computation times are sufficient to cause the executions to overlap.

The resources are all used by more than one task, but only one at a time and hence the ceiling values of the resources are straightforward to calculate. For R1, it is used by T1 and T4; hence the ceiling preemption level is 4. For R2, it is used by T2 and T3; hence the ceiling value is 3. Finally R3, it is used by T1 and T2; the ceiling equals 2 (see Table 14.3).

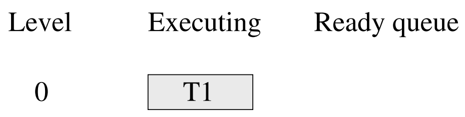
Protected object	Ceiling value
R1	4
R2	3
R3	2

Table 14.3: Ceiling values

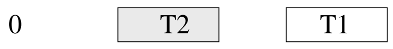
To implement this set of tasks and resources will require ready queues at level 0 (value of `Priority'First` in this example) and values 2, 3 and 4.

Scenario 1

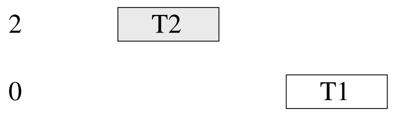
At time 0, T1 arrives. All ready queues are empty and all resources are free so T1 is placed in queue 0. It becomes the running task. This is illustrated in the following where 'Level' is the priority level, 'Executing' is the name of the task that is currently executing, and 'Ready queue' shows the other non-blocked tasks in the system.



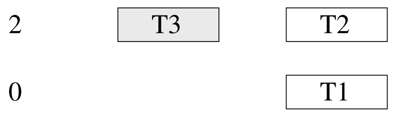
At time 2, T2 arrives and is added to ready queue 0 in front of T1 as it has a shorter absolute deadline. Now T2 is chosen for execution.



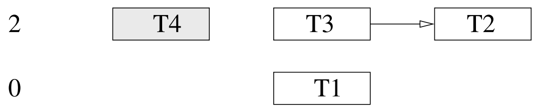
Assume at time 3, T2 calls R3. Its active priority will rise to 2.



At time 4, T3 arrives. Task T2 is joined by T3 on queue 2, as T3 has an earlier deadline and a higher preemption level; T3 is at the head of this queue and becomes the running task.



At time 6, T4 arrives. Tasks T3 and T2 are now joined by T4 as it has a deadline earlier than T2 and a higher preemption level (than 2). Task T4 now becomes the running task, and will execute until it completes; any calls it makes on resource R1 will be allowed immediately as this resource is free.

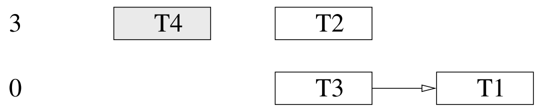


At some time later, T4 will complete, then T3 will execute (at priority 2, or 4 if it locks R2), then when it completes, T2 will execute (also at priority 2) until it releases resource R3, at which point its priority will drop to 0 but it will continue to execute. Eventually when T2 completes, T1 will resume (initially at priority 0 – but this will rise if it accesses either of the resources it uses).

Scenario 2

Here a simple change is made: at time 3, T2 calls R2 instead of R3. Its active priority will, therefore, rise to 3. Now when T3 arrives at time 4, it will not have a high enough preemption level to join ready queue 3 and will be placed on the lowest queue at level 0 (but ahead of T1). Task T2 continues to execute.

At time 6, T4 arrives. It passes both elements of the test and is placed on queue at level 3 ahead of T2 and therefore preempts it.



At some time later, T4 will complete, then T2 will execute (at priority 3) until it releases resource R2, at which point its priority will drop to 0. Now T3 will preempt and becomes the running task.

Scenario 3

For a final example, return to the first scenario but assume T3 makes use of resource R2 before T4 arrives:

At time 0, T1 arrives. All ready queues are empty and all resources are free so T1 is placed in queue 0. It becomes the running task.

At time 2, T2 arrives and is added to ready queue 0 in front of T1.

Assume at time 3, T2 calls R3. Its active priority will rise to 2.

At time 4, T3 arrives and becomes the running task at priority level 2.

At time 5, T3 calls R2 (note all resource requests are always to resources that are currently free) and thus its active priority rises to 3.

At time 6, T4 arrives. There is now one task on queue 0; one on queue 2 (T2 holding resource R3), and one task on queue 3 (T3 holding R2). The highest ready queue that the dispatch rules determine is that at level 3, and hence T4 joins T3 on this queue – but at the head and hence becomes the running task.

14.3.4 An important constraint

The above rules and descriptions are, unfortunately, not quite complete. The ready queue for `Priority'First` plays a central role in the model as it is, in some senses, the default queue. If a task is not entitled to be put in a higher queue, or if no protected objects are in use, then it is placed in this base queue. Indeed during the execution of a typical program most runnable tasks will be in the `Priority'First` ready queue most of the time. However, the protocol only works if there are no protected objects with a ceiling at the `Priority'First` level. Such ceiling values must be prohibited but this is not a significant constraint. Ceilings at this level are rare (all user tasks would need to have priority `Priority'First`) and the use of `Priority'First + 1` will not have a major impact. Indeed a common implementation practice is to set any ceiling at maximum usage + 1.

14.3.5 Example task set

To illustrate the minimal changes that have to be made to the application code to move from one scheduling paradigm to another, consider a simple periodic task scheduled according to the standard fixed priority method. This task has a period of 10 ms.

```
task A is
  pragma Priority(5);
end A;

task body A is
  Next_Release: Real_Time.Time;
begin
  Next_Release := Real_Time.Clock;
  loop
    -- code
    Next_Release := Next_Release + Real_Time.Milliseconds(10);
    delay until Next_Release;
  end loop;
end A;
```

If EDF dispatching is required (perhaps to make the system schedulable) then very little change is needed. The priority levels of the task and protected objects remain exactly the same. The task's code must, however, now explicitly refer to deadlines:

```
task A is
  pragma Priority(5);
  pragma Relative_Deadline(10); -- gives an initial relative
                                -- deadline of 10 milliseconds
end A;

task body A is
  Next_Release: Real_Time.Time;
begin
  Next_Release := Real_Time.Clock;
  loop
    -- code
    Next_Release := Next_Release + Real_Time.Milliseconds(10);
    Delay_Until_And_Set_Deadline(Next_Release,
                    Real_Time.Milliseconds(10);
  end loop;
end A;
```

Finally the dispatching policy must be changed from

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
```

to

```
pragma Task_Dispatching_Policy(EDF_Across_Priorities);
```

14.3.6 Rendezvous

The above descriptions have concentrated on the use of protected objects for task communications. This is because their use is more appropriate for real-time systems. A new dispatching policy must nevertheless be complete in the sense of being well defined for other language features. There are a number of places where priority inheritance is defined in the tasking model of Ada.

Warning: However, as noted earlier, with EDF dispatching there is no notion of deadline inheritance. Consequently, a server task engaged in a rendezvous with a client task will not inherit the deadline of the client. Similarly, an activating child task will not inherit the deadline of its parent. Also note, task entry queues are not ordered according to the client tasks' deadlines.

To understand the impact of a rendezvous on the PCLP protocol consider the following example of two tasks scheduled according to the EDF scheme. A rendezvous is defined to take place at the maximum active priority of the two tasks involved. If the highest priority task is the second one to 'arrive' at the rendezvous then this is straightforward. However if the highest priority task must block waiting for its rendezvous partner then this could compromise the EDF model. Consider task T executing at priority P. If it now blocks at a rendezvous entry call, then the PCLP protocol requires its active priority to be re-calculated when it next executes. For this reason any task, when it blocks, must have its active priority assigned to `Priority'First`. A similar argument applies to task activation.

14.3.7 Other considerations

Although the rules for EDF dispatching may not be immediately intuitive it will rarely be the case that tasks are released at the same time as a string of protected actions being executed. The key property for EDF remains that tasks are usually executed in deadline order with the use of shared objects occasionally imposing a more static ordering.

The use of preemption levels is a means of controlling the temporal interactions of tasks. If they are assigned correctly then the significant properties mentioned earlier (e.g. simple block and no deadlocks) will be delivered. There is not however, within the definition of Ada, any way of ensuring that the programmer does indeed assign the correct preemption levels. This is the same issue as with fixed priority scheduling – are the base priorities correctly assigned? Only program inspection and verification can tackle these problems.

14.4 Mixed scheduling

All the above descriptions have assumed that a single dispatching policy is in effect for the whole program. The current language however goes further and allows mixed systems to be defined. To accomplish this, the system's priority range can be split into a number of distinct non-overlapping bands. In each band, a specified dispatching policy can be defined. So, for example, there could be a band of fixed priorities on top of a band of EDF with a single round robin level for non-real-time tasks at the bottom. To illustrate this, assume a priority range of 1..16:

```
pragma Priority_Specific_Dispatching
           (FIFO_Within_Priorities, 10, 16);

pragma Priority_Specific_Dispatching
           (EDF_Across_Priorities, 2, 9);

pragma Priority_Specific_Dispatching
           (Round_Robin_Within_Priorities, 1, 1);
```

Any task is assigned a dispatching policy by virtue of its base priority. If a task has base priority 12 it will be dispatched according to the fixed priority rules, if it has base priority 8 then it is under the control of the EDF rules. In a mixed system, all tasks have a preemption level (it is just their base priority) and all tasks have a deadline (it will be `Default_Deadline` if none is assigned in the program). But this deadline will have no effect on dispatching if the task in not in an EDF band.

To achieve any mixture including one or more EDF bands, the properties assigned in the definition of EDF dispatching to level `Priority'First` need to be redefined to use the minimum value of whatever range is used for the EDF tasks (in the above case, priority level 2). Also note that two adjacent EDF bands are not equivalent to one complete band. Runnable tasks in the upper bands will always take precedence over runnable tasks in the lower bands, even if the latter ones have earlier deadlines.

The use of pragma `Priority_Specific_Dispatching` precludes any use of pragma `Task_Dispatching_Policy` – the compiler will catch any attempt to use both pragmas in the same partition. For completeness, any priority value not included in the pragma is assumed to be `FIFO_Within_Priorities`. Any of the predefined policies can be mixed, apart from the non-preemptive one. It is deemed incompatible to mix non-preemption with any other scheme as non-preemption is a system-wide property, a low priority non-preemptive task would impact on a higher priority task in another band. It might have been possible to define 'non-preemption within a band', but Ada 2005 does not go that far.

Tasks within different bands can communicate using protected objects (and rendezvous if required). The use of `Ceiling_Locking` for the entire partition ensures that protected objects behave as required. For example an ‘EDF task’ (preemption level 7) could share data with a ‘round-robin’ task and a ‘fixed priority’ task (priority 12). The protected object would have a ceiling value of (at least) 12. When the EDF task accesses the object its active priority will rise from 7 to 12, and while executing this protected action it will prevent any other task executing from within the EDF band. The ‘round-robin’ task will similarly execute with priority 12 – if its quantum is exhausted inside the object it will continue to execute until it has completed the protected action.

Warning: Perhaps one minor weaknesses of Ada’s support for mixed dispatching rules is that a task cannot directly ask under what policy it is being scheduled (other than asking if it is under round-robin control). However, a task can always find out its own priority and from that use program constants to ascertain under which policy it is executing.

This ability to mix dispatching policies is unique to Ada. Experience will show whether this level of support for real-time programs proves to be useful, or if implementations are able to deliver this flexibility in an efficient manner.

14.5 Dynamic priorities

All the above discussions have assumed that the base priority of a task remains constant during the entire existence of the task. This is an adequate model for many scheduling approaches. There are, however, situations in which it is necessary to alter base priorities. For example:

- to implement mode changes;
- to implement application specific scheduling schemes.

In the first example, base priority changes are infrequent and correspond to changes in the relative temporal properties of the tasks after a mode change (for example, a task running more frequently in the new mode). The second use of dynamic priorities allows programmers to construct their own schedulers. An example of this will be given in Section 15.3 after a further language feature, `Timing_Events`, has been introduced.

In EDF, where base priority is interpreted to mean preemption level, then the ability of the programmer to change priority also facilitates changes to these preemption levels.

To support dynamic priorities, the language defines the following library package:

```
with Ada.Task_Identification;
with System;
use Ada;
package Ada.Dynamic_Priorities is
  procedure Set_Priority(Priority : System.Any_Priority;
                T : Task_Identification.Task_Id :=
                Task_Identification.Current_Task);
     -- raises Program_Error if T is the Null_Task_Id
     -- has no effect if the task has terminated

   function Get_Priority(T : Task_Identification.Task_Id :=
            Task_Identification.Current_Task)
            return System.Any_Priority;
     -- raises Tasking_Error if the task has terminated
     -- or Program_Error if T is the Null_Task_Id
private
  -- not specified by the language
end Ada.Dynamic_Priorities;
```

The function `Get_Priority` returns the current base priority of the task; this can be changed by the use of `Set_Priority`.

Note that the package, which was introduced in Ada 95, uses task identifiers to identify the designated task.†

A change of base priority takes effect immediately the task is outside a protected object.‡

Ada 2005 change: Ada 2005 has tightened the definition of `Set_Priority` so that it must now occur immediately unless the task is on an entry queue

Where a task is queued on an entry, the implementation is allowed to postpone the effect of the priority change – it should be as soon as is practical. This might seem a little vague; however, it is difficult to be more precise because the implementation of Ada might be running on top of a standard operating system, and task priorities might be mapped to operating system priority levels, and the lock on the queue to an operating system lock. It is however, a requirement that any implementation of Ada must document when priority changes take effect.

Base priority changes can affect the active priority of the task and hence have an impact on dispatching and queuing. With the predefined scheduling policies, the effect of calling `Set_Priority` with the existing priority level is to first extract the task from the queue and then replace it in the appropriate place for the policy – for all but `EDF_Within_Priorities` this means putting the task at the back of the ready queue for that priority. With a mixed dispatching scheme, a change of base priority could move the task from one dispatching policy to another (e.g. from

† If this functionality had been added only in Ada 2005, it is likely that a task interface would have been used (i.e. all tasks would have been assumed to implement an interface with get and set routines).
‡ Strictly, at the first point the task is outside the execution of an abort-deferred operation (see Subsection 9.2.1).

EDF dispatching to fixed priority. If this occurs, the task is immediately subject to the new policy.

14.5.1 Dynamic ceiling priorities for protected objects

Ada 2005 change: Ada 95 allows tasks to change priority but not protected objects. This inconsistency led to some difficulties with certain application requirements (see following mode change example). Ada 2005 has removed this inconsistency by adding a new attribute (`'Priority`) for all protected objects.

For any protected object, `P`, the attribute `P'Priority` represents a component of `P` of type `System.Any_Priority`.

Important note: References to the `Priority` attribute can only occur from within the body of the associated protected body. Such references can be read or write. If the locking policy `Ceiling_Locking` is in effect then a change to the `Priority` attribute results in the ceiling value changing to this new value – but only at the end of the protected action that resulted in the change.

The above description perhaps needs further explanation. The rules governing ceiling locking require all calling tasks to have priorities no higher than the ceiling of the protected objects they are calling. If a task has its base priority changed while executing a protected action then it has already been noted that the change will not take effect until the action is completed. A similar rule must apply for changes to a ceiling priority. Consider as an example a task with priority 12 calling a protected object with ceiling priority 16. If the reason for the call is to lower the priority to 10, say, then an immediate change would lead to ceiling violation. The rules, therefore, postpone the change until the task with priority 12 leaves the object. Note also that references to the priority attribute can only occur from within the protected object. Hence all changes to ceiling are requested by tasks with a mutual exclusive hold on the object – it is not possible to change the priority externally as this would again lead to violation problems.

Warning: The use of both dynamic task priorities and dynamic ceiling values opens up the possibility of a task queued on a protected object entry having its priority raised above that of the ceiling of the object – or the ceiling being lowered to be below that of the queued task. This is defined to be a bounded error with a number of possible outcomes. The most likely one being that `Program_Error` is raised in the calling task at the point of the call.

Care must be taken when altering the priority of a protected object that is used as an interrupt handler. Remember from the previous chapter that if the `Ceiling_Locking` policy is in effect then the expression used in `Interrupt_Priority` must be in the range of `Interrupt_Priority` if the associated protected object is actually to be used as an interrupt handler (i.e. it has an `Attach_Handler` or `Interrupt_Handler` pragma). So if such an interrupt handler has its ceiling priority level lowered to a value below the interrupt range then the exception `Program_Error` is raised at the point of the assignment to `'Priority` (and the priority level is unchanged).

14.5.2 Mode change example

With the mode change problem, a number of tasks need to have their base priorities changed, and as a consequence the ceiling of the protected objects they use may also need to be altered. To illustrate this consider a simple example of four avionics tasks (T1, ..., T4), two protected objects (P1 & P2) and two modes of operation: *climbing* and *cruising*. Scheduling could be by fixed priority or EDF; both require changes in task base priorities to be coupled with changes to protected object ceiling priorities. Table 14.4 gives the task priority parameters for this example.

Task	Period in mode Climbing	Priority in mode Climbing	Period in mode Cruising	Priority in mode Cruising
T1	20	10	100	7
T2	70	9	70	9
T3	80	8	80	8
T4	100	7	50	10

Table 14.4: A mode change example

So during a change between the two modes defined, two tasks exchange their frequency of execution and hence their period and priority (if optimal priority assignment is to be maintained).

In this simple example, protected object P1 is used by T1 and T3 (in both modes) and P2 is used by T2 and T3 (again in both modes of operation). As a result of a mode change, the ceiling of the two protected object should ideally change: in *climbing* the ceiling of P1 is 10 and P2 is 9; and in *cruising* P1 is 8 and P2 is 10. In Ada 95 (before the inclusion of dynamic ceilings), both objects would have had to have the ceiling of 10 in both modes.

To organise the change of priority parameters following a mode change, it is important to follow a protocol that will prevent a task calling a protected object with an inappropriate (lower) ceiling priority. In the above example, if the ceiling of P1 is lowered to 8 as the system goes into the *cruising* mode before a final call from T1 in the old mode is made, this would lead to failure.

In general, a mode change must deal with four sets of objects.

- Tasks whose priority must increase.
- Tasks whose priority must decrease.
- Protected objects whose priority ceiling must increase.
- Protected objects whose priority ceiling must decrease.

Important note: To ensure an orderly mode change, it is recommended that the following sequence is followed over the four sets of objects that have their priorities changed.

(1) First change the priority of those tasks whose priority must be lowered.
(2) Next change the ceiling priorities of those protected objects that must have their ceiling raised.
(3) Then change the ceiling priorities of those protected objects that must have their ceiling lowered.
(4) Finally, change the priority of those tasks whose priority must be raised.

Importantly, the entity (task) that manages these changes must modify its own priority so that it can make the ceiling changing calls on the protected objects (this is illustrated below).

In the example, to move from *climbing* to *cruising* the following steps should be followed:

(1) T1's priority lowered from 10 to 7;
(2) P2's ceiling raised from 9 to 10;
(3) P1's ceiling lowered from 10 to 8;
(4) T4's priority raised from 7 to 10.

To program the priority changes, it is necessary to focus on the differences between how task priorities are altered and how protected object ceilings are modified.

- A task's priority can be changed from outside the task. The task calling `Set_Priority` can itself be of any priority. All tasks can be identified by their `Task_ID`.

- A protected object can only have its ceiling changed from inside. The object must have an operation defined for changing ceiling. The task calling this operation must have an active priority no higher than the current ceiling of the object. No standard operation is defined for all protected objects.

To counter this last point, the use of a protected interface that has a `Set_Ceiling` routine (and a `Get_Ceiling` routine for completeness) can be defined:

```
with System;
package Dynamic_Ceilings is
  type Dynamic_Ceiling is protected interface;
  procedure Set_Ceiling(P : in out Dynamic_Ceiling;
      Level : System.Any_Priority) is abstract;
  procedure Get_Ceiling(P : in out Dynamic_Ceiling;
      Level : out System.Any_Priority) is abstract;
end Dynamic_Ceilings;
```

In the above example, P1 and P2 implement this interface. The required code is straightforward:

```
protected type P1_Type is new Dynamic_Ceiling with
  overriding procedure Set_Ceiling(Level : System.Any_Priority);
  overriding
  procedure Get_Ceiling(Level : out System.Any_Priority);
  -- real entries and procedures
  -- local state variables
end P1_Type;

P1 : P1_Type;
-- similar definition of P2's type and P2 itself

protected body P1_Type is
  procedure Set_Ceiling(Level : System.Any_Priority) is
  begin
    P1_Type'Priority := Level;
  end Set_Ceiling;
  procedure Get_Ceiling(Level : out System.Any_Priority) is
  begin
    Level := P1_Type'Priority;
  end Get_Ceiling;
  -- other subprograms and entries
end P1_Type;
```

To give a little more detail on this example, assume that the mode change manager is a task with entries that are called to change mode:

```
task Mode_Controller is
  pragma Priority(20); -- higher value than T1's or T2's
  entry Climbing_To_Cruising;
  entry Cruising_To_Climbing;
  ...
end Mode_Controller;
```

The body of the task would be structured as

```
task body Mode_Controller is
begin
  loop
    select
      accept Climbing_To_Cruising;
      Set_Priority(7, T1_ID);
      Set_Priority(9); -- own priority
      P2.Set_Ceiling(10);
      Set_Priority(10);
      P1.Set_Ceiling(8);
      Set_Priority(20);
      Set_Priority(10, T2_ID);
    or
      accept Cruising_To_Climbing;
      ...
    end select;
  end loop;
end Mode_Controller;
```

Once the accept statement has executed, the mode control task will take precedence over T1 and T2; they will not execute again until the mode change is complete (on a single-processor implementation).

Of course in many applications, it will be necessary for the tasks themselves to be informed of the mode change. Particularly if they access different protected objects in different modes.

14.6 Synchronous and asynchronous task control

The final two subsections in this chapter describe two further primitives that are provided by Ada to program fine-level control of task dispatching and, in particular, task synchronisation.

During the development of Ada 95 there was a long running debate as to the right level of abstraction for the language synchronisation primitives. Low-level primitives are efficient but do not lend themselves to effective use (from a software engineering viewpoint). But Ada 83's support of only a very high-level rendezvous structure was strongly criticised for not facilitating the production of efficient concurrent or real-time applications. It also leads to what is called abstraction inversion. Recall, that this is when a programmer takes a high-level abstraction (such as the Ada rendezvous) and uses it to construct lower-level services (such as semaphores). These services are then used in the rest of the program.

This debate over language synchronisation abstractions is further complicated by the realisation that the simple provision of both high- and low-level abstractions is not appropriate, as the interactions between the levels of abstraction have to be defined. And this can prove to be very difficult in practice.

The compromise that has emerged in Ada is the definition of two packages in the Real-Time Systems Annex. One package (synchronous task control) allows a task to suspend itself. The other package (asynchronous task control) allows a task to suspend other tasks. The semantics for this more controversial feature are expressed in terms of priorities. These two packages will now be described.

14.6.1 Synchronous task control

The predefined package provides a simple binary-semaphore-like construct:

```
package Ada.Synchronous_Task_Control is
  type Suspension_Object is limited private;
  procedure Set_True(S : in out Suspension_Object);
  procedure Set_False(S : in out Suspension_Object);
  function Current_State(S : Suspension_Object) return Boolean;
  procedure Suspend_Until_True(S: in out Suspension_Object);
    -- raises Program_Error if more than one task tries
    -- to suspend on S at once.
    -- Potentially blocking.
private
  -- not specified by the language
end Ada.Synchronous_Task_Control;
```

An object of this type has two notional values: true and false. The first three subprograms are non-blocking and atomic with respect to one another. They simply allow a `Suspension_Object` to be set or interrogated.

The procedure `Suspend_Until_True` will suspend a calling task until the value of `S` is true. On return from this procedure, the value of `S` will be false.

The intention is that suspension objects be used with simple protected objects (ones without entries) to provide an efficient queuing mechanism. Consider the case of a bounded buffer which is accessed by a single reader and a single writer:

```
with Ada.Finalization; use Ada.Finalization;
with Ada.Synchronous_Task_Control;
use Ada.Synchronous_Task_Control;
package Buffers is
  type Buffer is limited private;
  type Data_Item is ...
  procedure Get(BU : in out Buffer; Item : out Data_Item);
  procedure Put(BU : in out Buffer; Item : in Data_Item);
private
  Buffer_Size : constant Integer := ...
  type Index is mod Buffer_Size;
  subtype Count is Natural range 0 .. Buffer_Size;
  type Data_Buffer is array(Index) of Data_Item;

  protected type Bounded_Buffer is
    procedure Get(Item: out Data_Item;
                  Reader_Ok, Writer_Ok : in out Suspension_Object);
    procedure Put(Item: in Data_Item;
                  Reader_Ok, Writer_Ok : in out Suspension_Object);
  private
    First : Index := Index'First;
    Last : Index := Index'Last;
    Number_In_Buffer : Count := 0;
    Buf :  Data_Buffer;
  end Bounded_Buffer;

  type Buffer is new Limited_Controlled with
    record
      Reader_Ok : Suspension_Object;
      Writer_Ok : Suspension_Object;
      B : Bounded_Buffer;
    end record;

  procedure Initialize(BU : in out Buffer);
end Buffers;
```

The `Buffer` type is limited private and is represented by a record which contains an instance of the protected type used to provide mutually exclusive access to the buffer and two suspension objects. Two suspension objects are required because there are two condition synchronisations needed: when the buffer is full and when it is empty. The predefined initial value for each suspension object is false; to set `Buffer.Writer_Ok` to true (as it is correct to write to an empty buffer) requires the use of initialisation. Hence `Buffer` is derived from `Limited_Controlled` – so that the initialisation routine is called automatically when a buffer is created, this sets the `Suspension_Object` to true.

In the body of the package, a call to the protected object is blocked until the conditions are appropriate for entry. Every call to `Put` can release a reader and vice versa. Note that as `Suspend_Until_True` resets the suspension object to false it is necessary, for example, for `Get` to return `Reader_Ok` to true if the buffer is not empty.

```
package body Buffers is
  procedure Initialize(BU : in out Buffer) is
  begin
    Set_True(BU.Writer_Ok);
  end Initialize;

  protected body Bounded_Buffer is
    procedure Get(Item: out Data_Item;
               Reader_Ok, Writer_Ok : in out Suspension_Object) is
    begin
      Item := Buf(First);
      First := First + 1;
      Number_In_Buffer := Number_In_Buffer - 1;
      Set_True(Writer_Ok);
      if Number_In_Buffer /= 0 then
```

```
        Set_True(Reader_Ok);
      end if;
    end Get;

    procedure Put(Item: in Data_Item;
                 Reader_Ok, Writer_Ok : in out Suspension_Object) is
    begin
      Last := Last + 1;
      Buf(Last) := Item;
      Number_In_Buffer := Number_In_Buffer + 1;
      Set_True(Reader_Ok);
      if Number_In_Buffer /= Buffer_Size then
        Set_True(Writer_Ok);
      end if;
    end Put;
  end Bounded_Buffer;

  procedure Get(BU : in out Buffer;
                Item : out Data_Item) is
  begin
    Suspend_Until_True(BU.Reader_Ok);
    BU.B.Get(Item, BU.Reader_Ok, BU.Writer_Ok);
  end Get;

  procedure Put(BU : in out Buffer;
                Item : in Data_Item) is
  begin
    Suspend_Until_True(BU.Writer_Ok);
    BU.B.Put(Item, BU.Reader_Ok, BU.Writer_Ok);
  end Put;
begin
  null;
end Buffers;
```

Note, generalising this algorithm to the multi-reader and multi-writer case is not trivial due to the restriction on suspension objects.

14.6.2 Asynchronous task control

The ability to suspend another task is fraught with difficulties. For example, what should happen if the designated task is currently executing inside a protected object? Arguably, if 'suspend others' is to be allowed, then it must be supported in such a way as to be consistent with the other parts of the language. One interpretation of 'suspend task T' is to lower its priority to below that of other tasks in the system. Hence, a potentially consistent way of obtaining asynchronous task control is to express the required semantics in terms of priorities. This is the approach taken in the Real-Time Systems Annex by the provision of the following package:

```
with Ada.Task_Identification;
use Ada;
package Ada.Asynchronous_Task_Control is
  procedure Hold(T : Task_Identification.Task_Id);
  procedure Continue(T : Task_Identification.Task_Id);
  function Is_Held(T : Task_Identification.Task_Id)
            return Boolean;
end Ada.Asynchronous_Task_Control;
```

For each processor, there is a conceptual idle task which can always run (but has a priority below any application task). A call of `Hold` lowers the base priority of the designated task to below that of the idle task. It is said to be *held.* If the designated task is not executing with an inherited priority, it will be suspended immediately. A call of `Continue` restores the task's priority. Note that this explanation is valid whatever the dispatching policy of the task. If, for example, it was under EDF control then the call of `Hold` would conceptually move the task out of the EDF range. It would no longer be subject to the EDF dispatching rules. When its priority is restored (by another task calling `Continue`) then it would be subject to the rule of EDF for a task returning from a blocked state (see Section 14.3).

As only the base priority of the held task is affected, it is possible to give clear semantics to the consequences of calling `Hold` on a task which may be in one of a number of states. For example, if the designated task is executing inside a protected object (that is, it has inherited the ceiling priority) then it will continue executing until it leaves the object – it will then become held. Similarly, a held caller of a rendezvous will not affect the rendezvous taking place (although the rendezvous's priority will now be only that of the task that does the accepting).

Two further rules clarify the behaviour of a held task:

- If the held task is currently suspended on an accept statement (or a select), and a call comes in on an (open) entry, then the accept statement is executed. This rule follows from the accepting task inheriting the priority of the caller and is slightly counter-intuitive (given that the task is meant to be suspended!).
- If the held task is currently suspended on a protected object's barrier, then it will execute the entry when the barrier comes true (open) and it is the only task on the queue. This rule follows from the use of ceiling priorities and also the implementation scheme that allows the entry to be executed by the task that lowered the barrier (on behalf of the suspended task).

The use of `Hold` and `Continue`, although it does not give 'suspend NOW under all circumstances', does provide a safe abstraction for what some real-time programmers feel is a critically important feature.

14.7 Summary

In this chapter a number of dispatching policies (in addition to the standard fixed priority policy) have been described. Specifically, non-preemptive priority based, round-robin and EDF are covered. These are all new within Ada 2005. Other scheduling approaches can be programmed with the help of low level synchronisation primitives and routines that allow dynamic priorities to be used.

The other significant feature of Ada, in terms of its support for scheduling, is the ability of the programmer to specify mixed dispatching paradigms. With this support, a concurrent real-time program can be produced that dispatches some of its tasks by fixed priority, others by EDF and the rest by round-robin. Moreover these tasks can share data via the use of protected objects and can even move between policies at run time.

14.8 Further reading

A. Burns and A.J. Wellings, *Real-time Systems and Programming Languages*, 3rd Edition, Addison-Wesley, 2000.
G.C. Buttazzo, *Hard Real-Time Computing Systems*, 2nd Edition Springer, 2005.
M.H. Klein, T. Ralya, B. Pollak, R. Obenza, M. Gonzalez Harbour, *A Practitioner's Handbook for Real-Time Analysis*, Kluwer, 1993.
H. Kopetz, *Real-Time Systems*, Kluwer, 1997.
J.W.S. Liu, *Real-Time Systems*, Prentice-Hall, 2000.
S.H. Son (editor), *Advances in Real-Time Systems*, Prentice-Hall, 1994.

15
Timing events and execution-time control

This chapter introduces a number of language features that are new in Ada 2005. The main focus is on execution time and how tasks can monitor and control the amount of processor time they are using. For real-time systems this is of crucial importance as the processor is usually the resource in least supply. It needs to be

- used in a manner that is sympathetic to the scheduling policy of the system,
- not over-used by failing components (tasks), but
- fairly reallocated dynamically if spare capacity becomes available.

However before considering these topics, a more general view of Ada's model of event handling is warranted. The facilities for execution-time control all make use of events and event handling, and hence this chapter will start by examining events and a particular kind of event termed a *timing event*.

15.1 Events and event handling

It is useful in concurrent systems to distinguish between two forms of computation that occur at run-time: tasks and events. A task (or process or thread) is a long-lived entity with state and periods of activity and inactivity. While active, it competes with other tasks for the available resources – the rules of this competition are captured in a scheduling or dispatching policy (for example, fixed priority or EDF). By comparison, an event is a short-lived, stateless, one-shot computation. Its handler's execution is, at least conceptually, immediate; and having completed it has no lasting direct effect other than by means of changes it has made to the permanent state of the system. When an event occurs we say it is *triggered*; other terms used are *fired*, *invoked* and *delivered*.

The code to be executed by a task is defined in the *task body* within which will be synchronisation points that control activation. These are usually delay statements, or entry calls on protected objects or other tasks. The code associated with an event is termed the *event handler*, it will normally not contain any synchronisation calls that could lead to it becoming suspended. The handler will run to completion.

In Chapter 12, one form of event, the *interrupt*, was described. Initially defined in Ada 95, an interrupt is represented by `Interrupt_ID` which is an implementation defined discrete type. The event is triggered by the occurrence of an external interrupt. The handler is represented by an access type to a protected procedure:

```
type Parameterless_Handler is access protected procedure;
```

So, when the interrupt occurs, the associated procedure is executed; this procedure being a component of a protected object. The priority of the protected object determines how immediately the handler is executed. Typically, the handler will execute at a high priority level as it is executed directly by the run-time system supporting the program.

In this chapter, four event types are introduced; two are explicit `Timing_Event` and `Timer` and two are implicit: one is linked to the termination of a task, the other is associated with a `Group_Budget` – which is defined later in this chapter. Each of the explicit events is represented as a **`tagged limited private`** type. All have handlers of the form just described for interrupts. Although there are some differences between these events due to their particular application context, they all have essentially the same behaviour. Any particular event instance is either *set* or *cleared*, if set it has a handler attached, when the event is triggered the handler is executed. Examples of such handlers (and protected objects) will be given throughout this chapter, starting with timing events.

15.2 Timing events

One of the fundamental requirements for any real-time program is to coordinate the execution of the program with the environment's time base. Indeed that is precisely what the term *real-time* means. If a central heating system must come on at 7.00am then the control system needs a clock and a way of postponing execution until that clock says 7.00am. In Ada 95, and most real-time programming languages, the only way to deliver this coordination is to have a task that delays until 7.00am and then turns the heating on. In some situations this concurrency overhead is unnecessary and inefficient. Ada 2005 has introduced a lower level facility that maps a handler to a specific time without the need to use a task at all. The handler is associated with a *timing event*.

Important note: A timing event is similar to an interrupt but it is triggered not by an external action but by the passage of time.

When the event's time is due (as determined by a reference clock), the handler code is executed – and the central heating burner is ignited.

A further motivation for timing events comes from an exploration of various flexible scheduling schemes. For example, *anytime algorithms*,† illustrate the need to

- asynchronously change the priority of a task at a particular future time, and
- allow tasks to come off the delay queue at a different priority from that in effect when the task was delayed.

This functionality can only be achieved in Ada 95 by the use of a 'minder' high-priority task that makes the necessary priority changes to its client (using `Set_Priority`). This is an inefficient and inelegant solution.

Ada 2005 change: Ada 2005 has introduced a new abstraction of a timing event to allow code to be executed at specified times without the need to employ a task. The key constraint that allows efficient implementation of timing events is that the handling code is not allowed to block. The code can, therefore, be encapsulated within a protected procedure.

Timing events are supported by a child package of `Ada.Real_Time`:

```
package Ada.Real_Time.Timing_Events is
  type Timing_Event is tagged limited private;
  type Timing_Event_Handler is access protected
        procedure(Event : in out Timing_Event);
  procedure Set_Handler(Event : in out Timing_Event;
        At_Time : Time; Handler: Timing_Event_Handler);
  procedure Set_Handler(Event : in out Timing_Event;
        In_Time: Time_Span; Handler: Timing_Event_Handler);
  function Is_Handler_Set(Event : Timing_Event)
        return Boolean;
  function Current_Handler(Event : Timing_Event)
           return Timing_Event_Handler;
  procedure Cancel_Handler(Event : in out Timing_Event;
           Cancelled : out Boolean);
  function Time_Of_Event(Event : Timing_Event) return Time;
private
  -- Not specified by the language.
end Ada.Real_Time.Timing_Events;
```

The advantages of the event type being tagged will be illustrated later, but note the operations defined with this private type are concrete (not abstract). This means that the package can be used without run-time dispatching; a useful property for many real-time and high-integrity applications.

The handler type is an access to a protected procedure with the timing event

† An anytime algorithm is able to return a valid result whenever it is stopped, but if it is given more execution time it will improve on this result. These algorithms are also known as *imprecise computations*.

itself being passed back to the handler when the event is triggered. The event is *set* by the attachment of a handler. Two Set_Handler procedures are defined, one using absolute time, the other relative; both use the clock within package Ada.Real_Time as the reference time base. If a **null** handler is passed the event is *cleared*, this can also be achieved by calling Cancel_Handler. With this routine the boolean flag indicates if the event was actually set before it was cleared. If Set_Handler is called on an event that is already set then a new time is posted for the event with the original time and handler being lost.

Important note: As a handler is called by a clock, it must be accessible for the lifetime of the program. Consequently, only library-level handlers can be used.

As soon as possible after the time defined for the event has passed (i.e. as soon as it is detected), the handler is executed; this clears the event. It will not be triggered again unless it is reset. The most effective way for an implementation to support timing events is to execute the handlers directly from the interrupt handler of the clock. A typical clock routine will interrupt every 10 ms or less. On each occurrence, the clock handler will check the system's delay queue to see if any tasks need to be made runnable or if any timing events need to be triggered. Tasks are moved to ready queues, handlers are executed directly.

Important note: As the clock interrupt is typically the highest priority interrupt in the system, the application code's protected object (that embodies the handler procedure) must have a ceiling of Interrupt_Priority'Last.

Two further subprograms are defined in the support package. One allows the current handler to be obtained, while the other allows the current time of the event to be requested. Both return with sensible values if the event is not set (**null** and Ada.Real_Time.Time_First respectively).

Two small examples of use will now be given. In the first, a watchdog timer is implemented. Here a condition is tested every 50 milliseconds. If the condition has not been reset during this time, an alarm handling task is released. Consider the following specification that has the above package and System and Ada.Real_Time visible:

```
protected Watchdog is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  entry Alarm_Control;
     -- Called by alarm handling task.
  procedure Call_In;
     -- Called by application code every 50 ms if alive.
  procedure Timer(Event : in out Timing_Event);
     -- Timer event code, ie the handler.
private
  Alarm : Boolean := False;
end Watchdog;

Fifty_Mil_Event : aliased Timing_Event;
TS : Time_Span := Milliseconds(50);

Set_Handler(Fifty_Mil_Event, TS, Timer'Access);
```

This watchdog object has a common structure. An entry with an initially closed barrier holds back a task that will be released by the handler if the handler executes. In this example the handler is actually never executed unless there is a problem. Each time the active code calls `Call_In` the timing event is reset to a point in the future. Only if another call does not occur before that time, will the handler be executed and the barrier be set to true:

```
protected body Watchdog is
  entry Alarm_Control when Alarm is
  begin
    Alarm := False;
  end Alarm_Control;

  procedure Timer(Event : in out Timing_Event) is
  begin
    Alarm := True;
    -- Note no use is made of the parameter in this example
  end Timer;

  procedure Call_in is
  begin
    Set_Handler(Fifty_Mil_Event, TS, Timer'Access);
    -- This call to Set_Handler cancels the previous call
  end Call_in;
end Watchdog;
```

In situations where it is necessary to undertake a small computation periodically (and with minimum jitter), the repetitive use of timing events is an effective solution. In the following example a periodic pulse is turned on and off under control of the application:

```
Not_Started : exception;
protected Pulser is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  procedure Start;
  procedure Stop;
  procedure Timer(Event : in out Timing_Event);
private
  Next_Time : Time;
end Pulser;
```

```
Pulse : Timing_Event;
Pulse_Interval : Time_Span := Milliseconds(15);


protected body Pulser is
  procedure Start is
  begin
    Pulse_Hardware;
    Next_Time := Clock + Pulse_Interval;
    Set_Handler(Pulse, Next_Time, Timer'Access);
  end Start;

  procedure Stop is
    Cancelled : Boolean;
  begin
    Cancel_Handler(Pulse, Cancelled);
    if not Cancelled then raise Not_Started; end if;
  end Stop;

  procedure Timer(Event : in out Timing_Event) is
  begin
    Pulse_Hardware;
    Next_Time := Next_Time + Pulse_Interval;
    Set_Handler(Event, Next_Time, Timer'Access);
  end Timer;
end Pulser;
```

Important note: This way of representing a periodic activity (i.e. without a task) is appropriate for small execution times or when minimum jitter on the periodic activity is required as long as the code does not access any shared resource that could lead to blocking. But it has the disadvantage that the code is always executed at interrupt priority level and hence has a temporal interference on the rest of the program.

From a scheduling point of view, if the computation time of the handler is non-trivial it is better to place the code in a periodic task with an appropriate priority (or deadline if EDF dispatching is in use). With Ada 2005, the programmer now has this choice.

15.3 Dual priority scheduling

In this section a further example of the use of timing events is given that illustrates a method of scheduling that combines EDF and fixed priority. Within the framework of fixed priority scheduling, much research has focused on the problem of maximising the completion of soft tasks whilst retaining the 100% guarantees for the hard task set. Soft tasks have clear deadlines but do not have to be guaranteed.

They can occasionally overrun their deadlines, or miss them completely. Soft tasks have many uses in real-time programming.

Dual priority scheduling is one means of integrating the execution of hard and soft tasks. This minimally dynamic approach retains the predictability afforded to hard tasks by fixed priority scheduling, whilst facilitating the responsive scheduling of soft tasks. The form of dual priority scheduling described here has all tasks first executing together using EDF (as this is the most efficient means of scheduling tasks). For each hard task there will be a time defined that will signal a change of scheduling for that task. It will have its priority raised away from EDF to a band of fixed priorities above the EDF band. Assume that there are ten priority levels in each band:

```
pragma Priority_Specific_Dispatching
               (FIFO_Within_Priorities, 11, 20);

pragma Priority_Specific_Dispatching
               (EDF_Across_Priorities, 1, 10);
```

A soft task will have a base priority in the range 1..10 and be subject to EDF dispatching at all times. A hard task will have an initial base priority in the range 1..10 but this will rise (by adding 10 to the base value) at a defined time called the *promotion* point after the release of the task. A timing event will be used to make this change.

It would be quite possible for each task to have its own handler that alters its base priority when required. However, using the ‘tagged’ feature of the event type it is possible to have just a single handler that uses the event parameter to reference the correct task. To do this, the type `Timing_Event` is first extended to include a task ID field:

```
type Dual_Event is new Timing_Event with
record
  TaskID : Ada.Task_Identification.Task_ID;
end record;
```

The single handler has a straightforward form (note the assumption that the `Ada.Task_Identification`, `System`, `Ada.Dynamic_Priorities` and `Ada.Dispatching_EDF` packages are visible):

```
protected Dual_Event_Handler is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  procedure Change_Band(Event : in out Timing_Event);
end Dual_Event_Handler;

protected body Dual_Event_Handler is
  procedure Change_Band(Event : in out Timing_Event) is
    The_Task : Task_ID;
    P : Priority;
  begin
    The_Task := Dual_Event(Timing_Event'Class(Event)).TaskID;
    P := Get_Priority(The_Task);
    Set_Priority(P+10, The_Task);
  end Change_Band;
end Dual_Event_Handler;
```

The specific task ID is obtained by two view conversions, the parameter `Event` is converted first to a class-wide type and then to the specific type `Dual_Event`. The run-time system does not know about the extended type but the underlying type is not changed or lost, and is retrieved using the view conversions.

Now consider a hard task that has a base priority 4 in the lower band and hence 14 in the upper. Initially it will run with priority 14 to make sure all its initialisation is complete. It has a period of 50 ms and a relative deadline set to the end of its period (i.e. also 50 ms). Its promotion point is 30 ms after its release.

```
with Ada_Dispatching_EDF; use Ada_Dispatching_EDF;
...
task Example_Hard is
  pragma Priority(14);
end Example_Hard;

task body Example_Hard is
  Dual_E : Dual_Event := (Timing_Event with
                            TaskID => Current_Task);
  Start_Time : Time := Clock;
  Period : Time_Span := Milliseconds(50);
  Promotion : Time_Span := Milliseconds(30);
begin
  Dual_E.Set_Handler(Start_Time + Promotion,
                     Dual_Event_Handler.Change_Band'Access);
  Set_Deadline(Start_Time + Period);
  Set_Priority(4);
  -- now dispatched according to EDF
  loop
    -- code of the task
    Start_Time := Start_Time + Period;
    Set_Priority(14);
    Dual_E.Set_Handler(Start_Time + Promotion,Dual_Event_Handler.
                                  Change_Band'Access);
    Set_Priority(4);
    Delay_Until_And_Set_Deadline(Start_Time,Period);
  end loop;
end Example_Hard;
```

If the event triggers then the task's priority will be raised to 14 and it will be subject to fixed priority dispatching and will compete its code by its deadline (guaranteed by the scheduling analysis). It will then set its new release time (`Start_Time`), set its event handler again and lower its priority back to the value within the EDF range. Its deadline will be quite soon and so it is likely to continue executing into its delay statement.

If the system is not heavily loaded, the hard task will complete its invocation before the promotion point. The second call of `Set_Handler` will then cancel the previous call. Note the priority is raised to 14 prior to this call to remove any possible race condition.

A final point to note with this example concerns the use of protected objects by the hard and soft tasks. If such an object is used by a hard task then it must have a ceiling in the 11..20 range, otherwise an error would occur if the task with its promoted priority calls the object. Using the event handler to also change the ceiling priorities of such protected objects is unlikely to be justified – although it is an interesting exercise that is left to the reader. Protected objects used solely by soft tasks can have ceiling in the range 2..10 (remember level 1 is not available by the definition of EDF).

15.4 Execution-time clocks

Performance analysis techniques are usually based on the assumption that the application developer can accurately measure/estimate the execution time of each task. Measurement is always very difficult, because, with effects like cache misses, pipelines, branch prediction, superscalar processor architectures etc., the execution time is highly variable. There are models that allow calculation of worst-case execution time (WCET) for some architectures, but they are generally very complex and not widely available for all architectures. A language defined means of measuring execution time is desirable.

In hard real-time systems, it is essential to monitor the execution times of all tasks and detect situations in which the estimated WCET is exceeded. This detection was usually available in systems scheduled with cyclic executives, because the periodic nature of their cycles makes it easy to check that all initiated work had been completed by the end of each cycle. In event-driven concurrent systems, the same capability should be available, and this can be accomplished with execution-time clocks and timers. In addition, many flexible real-time scheduling algorithms require the capability to measure execution time and be able to perform scheduling actions when a certain amount of execution time has been consumed.

Ada 2005 change: Ada 2005 directly supports execution-time clocks for tasks, and supports timers that can be fired when a task has used a defined amount of execution time.

This section examines Ada 2005's support for execution-time clocks; in the next section the extension of this provision to timers is described. Ada 83 had a single clock mechanism (`Calendar`) that approximated the external time base of the computer's environment with its seconds, days, months, years, leap ticks etc. Ada 95 added a `Real_Time` clock that is more accurate and monotonic in nature. Now Ada 2005 has added an execution-time clock; indeed it has added a clock per task that measures the task's execution time. A new package is defined that is similar in structure to `Calendar` and `Real_Time`:

```
with Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;
package Ada.Execution_Time is
   type CPU_Time is private;
   CPU_Time_First : constant CPU_Time;
   CPU_Time_Last  : constant CPU_Time;
   CPU_Time_Unit  : constant :=
         <implementation-defined-real-number>;
   CPU_Tick : constant Time_Span;

   function Clock
     (T : Ada.Task_Identification.Task_ID
          := Ada.Task_Identification.Current_Task)
                return CPU_Time;

   function "+"(Left : CPU_Time; Right : Time_Span)
               return CPU_Time;
   function "+"(Left : Time_Span; Right : CPU_Time)
               return CPU_Time;
   function "-"(Left : CPU_Time; Right : Time_Span)
               return CPU_Time;
   function "-"(Left : CPU_Time; Right : CPU_Time)
               return Time_Span;

   function "<"  (Left, Right : CPU_Time) return Boolean;
   function "<=" (Left, Right : CPU_Time) return Boolean;
   function ">"  (Left, Right : CPU_Time) return Boolean;
   function ">=" (Left, Right : CPU_Time) return Boolean;

   procedure Split
      (T : CPU_Time; SC : out Seconds_Count; TS : out Time_Span);

   function Time_Of (SC : Seconds_Count; TS : Time_Span)
            return CPU_Time;
private
   -- Not specified by the language.
end Ada.Execution_Time;
```

Important note: The execution time of a task, or CPU time as it is commonly called, is the time spent by the system executing the task and services on its behalf. The accuracy of the measurement of execution time cannot be dictated by the language specification, it is heavily dependent on the run-time support software.

On some implementation, (perhaps on non-real-time operating systems) it may not even be possible to support this package. But if the package is supported, the range of `CPU_Time` must be at least 50 years and `CPU_Tick` should be no greater than 1 ms.

When a task is created, so is an execution-time clock. This clock registers zero at creation and starts recording the task execution time from the point at which the task starts its *activation*. To read the value of any task's clock, a `Clock` function is defined. So in the following, a loop is allocated at least 7 ms of execution time before exiting at the end of its current iteration:

```
Start : CPU_Time;
Interval : Time_Span := Milliseconds(7);

...

Start := Ada.Execution_Time.Clock;

while Ada.Execution_Time.Clock - Start < Interval loop
  -- code of loop
end loop;
```

Note the '`-`' operator returns a value of type `Time_Span` which can then be compared with `Interval`.

To monitor the execution time of each invocation of a periodic task, for example, is simple:

```
Last : CPU_Time;
Exe_Time : Duration;
...
Last := Ada.Execution_Time.Clock;
loop
  -- code of task
  Exe_Time := To_Duration(Ada.Execution_Time.Clock - Last);
  -- print out or store Exe_Time
  Last := Execution_Time.Clock;
  delay until ...
end loop;
```

Of course, this is not completely accurate as there is code that is executed between reading and assigning to `Last`.

15.5 Execution-time timers

As well as monitoring a task's execution-time profile, it is also useful to trigger an event if the task's execution-time clock gets to some specified value. Often this is an error condition – where the task has executed for longer than was anticipated. A child package of `Execution_Time` provides support for this type of event:

```
package Ada.Execution_Time.Timers is
  type Timer(T : not null access constant
     Ada.Task_Identification.Task_ID) is tagged
     limited private;

  type Timer_Handler is access protected
     procedure(TM : in out Timer);

  Min_Handler_Ceiling : constant System.Any_Priority :=
   <implementation defined>;

  procedure Set_Handler(TM : in out Timer;
         In_Time : Time_Span; Handler : Timer_Handler);
  procedure Set_Handler(TM : in out Timer;
         At_Time : CPU_Time;  Handler : Timer_Handler);
  procedure Cancel_Handler(TM : in out Timer;
         Cancelled : in out Boolean);

  function Current_Handler(TM : Timer) return Timer_Handler;
  function Time_Remaining(TM : Timer) return Time_Span;

  Timer_Resource_Error : exception;
private
   -- Not specified by the language.
end Ada.Execution_Time.Timers;
```

Each `Timer` event is strongly linked to the task that will trigger it. This static linkage is ensured by the access discriminant for the type that is required to be **constant** and **not null**.

The handler type is standard, but there is now a need to specify the minimum ceiling priority the associated protected object must have if ceiling violation is to be avoided. With timing events, this was the top interrupt priority level but for timer events a lower level is possible. This priority will be set by the supporting implementation.

The `Set_Handler` procedures and the other routines all have the same properties as those defined with timing events. However, in recognition that an implementation may have a limited capacity for timers, or that only one timer per task is possible, the exception `Timer_Resource_Error` may be raised when a `Timer` is defined or when a `Set_Handler` procedure is called for the first time with a new `Timer` parameter.

To illustrate the use of a timer event, assume that a hard real-time task has a worst-case execution time (WCET) of 1.25 ms per invocation. If it executed for more than this value, its priority should be lower from its correct value of 14 to a minimum value of 2. If it is still executing after a further 0.25 ms then that invocation of the task must be terminated; this implies the use of an ATC construct. First, the overrun handler protected type is defined.

```
protected Overrun is
  pragma Priority(Min_Handler_Ceiling);
  entry Stop_Task;
  procedure Handler(TM : in out Timer);
  procedure Reset(C1, C2 : Time_Span);
private
  Abandon : Boolean := False;
  First_Occurrence : Boolean := True;
  WCET : Time_Span;
  WCET_Overrun : Time_Span;
end Overrun;

protected body Overrun is
  entry Stop_Task when Abandon is
  begin
    Abandon := False;
    First_Occurrence := True;
  end Stop_Task;

  procedure Reset(C1, C2 : Time_Span) is
  begin
    Abandon := False;
    First_Occurrence := True;
    WCET := C1;
    WCET_Overrun := C2;
  end Reset;

  procedure Handler(TM : in out Timer) is
  begin
    if First_Occurrence then
      Set_Handler(TM,WCET_Overrun,Handler'Access);
      Set_Priority(2, TM.T.all);
      First_Occurrence := False;
    else
      Abandon := True;
    end if;
  end Handler;
end Overrun;
```

It may not be immediately clear why a `Reset` routine is required. But without it a race condition may lead to incorrect execution. Consider the code of the task:

```
task Hard_Example;

task body Hard_Example is
  ID : aliased Task_ID := Current_Task;
  WCET_Error : Timer(ID'Access);
  WCET : Time_Span := Microseconds(1250);
  WCET_Overrun : Time_Span := Microseconds(250);
  Bool : Boolean := False;
  ...
begin
  -- initialisation
  loop
    Overrun.Reset(WCET, WCET_Overrun);
    Set_Handler(WCET_Error,WCET,Overrun.Handler'Access);
    select
      Overrun.Stop_Task;
      -- handle the error if possible at priority level 2
    then abort
      -- code of the application
    end select;
    Cancel_Handler(WCET_Error, Bool);
    Set_Priority(14);
   delay until ...
  end loop;
 ...
end Hard_Example;
```

It is possible for the timer to trigger (or *expire*) after completion of the select statement but before it can be cancelled. This would leave the state of the boolean variable `Abandon` with the incorrect value of `True` for the next invocation – hence the call to `Reset`. Similarly, it is necessary to cancel the timer before changing the priority back to 14 – otherwise the event could trigger just before executing the delay statement and the task would be stuck with the wrong low priority for its next invocation.

15.6 Group budgets

The support for execution-time clocks allows the CPU resource usage of individual tasks to be monitored, and the use of timers allows certain control algorithms to be programmed but again for single tasks. There are, however, situations in which the resource usage of groups of tasks needs to be managed. Typically this occurs when distinct subsystems are programmed together but need to be protected from one another – no failures in one subsystem should lead to failures in the others. So even if a high priority task gets into an infinite loop, tasks in other subsystems should still meet their deadlines.

Another application need is to program *execution-time servers*. These allow a set of tasks to share a budget so that they execute immediately if there is budget available but must wait if the budget is exhausted. The tasks do not need to be statically analysed to understand their maximum load, their impact is contained by the budget. Other tasks, perhaps hard ones, will suffer a bounded impact from the behaviour of the server and its tasks. There are a number of different execution-time servers described in the literature, for example the *sporadic server* and *deferrable server* – the former being specified within the POSIX standard.

Ada 95 has no mechanisms to allow the implementation of execution-time servers. This severely limited the language's ability to handle aperiodic activities at anything other than a background priority. The fundamental problem that prohibits the implementation of server algorithms is that tasks cannot share CPU budgets. Although Ada 2005 does not directly support these servers, it does provide the primitives from which servers can be programmed.

Group budgets allow different execution-time servers to be implemented, consequently the language itself does not have to provide a small number of predefined server types. Note, servers can be used with fixed priority or EDF scheduling.

A typical execution-time server has a budget and a replenishment period. At the start of each period, the available budget is restored to its maximum amount. Unused budget at this time is discarded. To program an execution-time server requires timing events to trigger replenishment and a means of grouping tasks together and allocating them an amount of CPU resource. A standard package (a child of `Ada.Execution_Time`) is defined to accomplish this:

```
package Ada.Execution_Time.Group_Budgets is
  type Group_Budget is tagged limited private;

  type Group_Budget_Handler is access
       protected procedure(GB : in out Group_Budget);

  type Task_Array is array(Positive range <>) of
                                  Ada.Task_Identification.Task_ID;

  Min_Handler_Ceiling : constant System.Any_Priority :=
    <Implementation Defined>;

  procedure Add_Task(GB : in out Group_Budget;
                     T : Ada.Task_Identification.Task_ID);
  procedure Remove_Task(GB : in out Group_Budget;
                     T : Ada.Task_Identification.Task_ID);
  function Is_Member(GB : Group_Budget;
           T : Ada.Task_Identification.Task_ID) return Boolean;
  function Is_A_Group_Member(
           T : Ada.Task_Identification.Task_ID) return Boolean;
  function Members(GB : Group_Budget) return Task_Array;

  procedure Replenish(GB : in out Group_Budget; To : Time_Span);
  procedure Add(GB : in out Group_Budget; Interval : Time_Span);
  function Budget_Has_Expired(GB : Group_Budget) return Boolean;
  function Budget_Remaining(GB : Group_Budget) return Time_Span;

  procedure Set_Handler(GB : in out Group_Budget;
                      Handler : Group_Budget_Handler);
  function Current_Handler(GB : Group_Budget)
                          return Group_Budget_Handler;
  procedure Cancel_Handler(GB : in out Group_Budget;
             Cancelled : out Boolean);

  Group_Budget_Error : exception;
```

```
private
    --  not specified by the language
end Ada.Execution_Time.Group_Budgets;
```

The type Group_Budget represents a CPU budget to be used by a group of tasks.

There are a number of routines defined in this package, consider first those concerned with the grouping of tasks. Each Group_Budget has a set of tasks associated with it. Tasks are added to the set by calls of Add_Task, and removed using Remove_Task. Functions are defined to test if a task is a member of any group budget, or one specific group budget. A further function returns the collection of tasks associated with a group budget by returning an unconstrained array type of task IDs.

An important property of these facilities is that a task can be a member of at most one group budget. Attempting to add it to a second group will cause Group_Budget_Error to be raised. This exception is also raised if an attempt is made to remove a task from a set that it is not a member of.

Important note: When a task terminates, if it is still a member of a group budget, it is automatically removed.

The budget decreases whenever a task from the associated set executes. The accuracy of this accounting is again implementation defined. To increase the amount of budget available, two routines are provided. The Replenish procedure sets the budget to the amount of 'real-time' given in the To parameter. It replaces the current value of the budget. By comparison, the Add procedure increases the budget by the Interval amount. But, as this parameter can be negative, it can also be used to, in effect, reduce the budget.

The minimal budget that can be held in a Group_Budget is zero – represented by Time_Span_Zero. If a budget is exhausted, or if Add is used to reduce the budget by an amount greater than its current value, then the lowest the budget can get is zero. To inquire about the state of the budget, two functions are provided. Note that when Budget_Has_Expired returns True then Budget_Remaining will return Time_Span_Zero.

A handler is associated with a group budget by use of the Set_Handler procedure.

Important note: There is an implicit event associated with a Group_Budget that occurs whenever the budget goes to zero. If at that time there is a non-null handler set for the budget, the handler will be executed. This will also occur if the budget goes to zero as the result of a call to Add with a large negative parameter.

As with timers, an implementation must define the minimum ceiling priority level for the protected object linked to any group budget handler. Also note there are Current_Handler and Cancel_Handler subprograms defined.

By comparison with timers and timing events, which are triggered when a certain clock value is reached (but will then never be reached again for monotonic clocks), the group budget event can occur many times – whenever the budget goes to zero. So the handler is permanently associated with the group budget, it is executed every time the budget is exhausted (obviously following replenishment and further usage). The handler can be changed by a further call to Set_Handler or removed by using a null parameter to this routine (or by calling Cancel_Handler), but for normal execution the same handler is called each time. The better analogy for a group budget event is an interrupt, its handler is called each time the interrupt occurs.

Important note: When the budget is zero, the associated tasks *continue* to execute. If action should be taken when there is no budget, this has to be programmed (it must be instigated by the handler). So group budgets are not in themselves an execution-time server abstraction – but they allow these abstractions to be constructed.

As a first example, consider four aperiodic tasks that should share a budget of 2 ms that is replenished every 10 ms. The tasks first register with a Controller1 protected object that will manage the budget. They then loop around waiting for the next invocation event. In all of the examples in this section, fixed priority scheduling is assumed.

```
task Aperiodic_Task is
  pragma Priority(Some_Value);
end Aperiodic_Task;


task body Aperiodic_Task is
  ...
begin
  Controller1.Register;
  loop
    -- wait for next invocation
    -- undertake the work of the task
  end loop;
end Aperiodic_Task;
```

The Controller1 will use a timer event and a group budget, and hence defines handlers for both.

```
protected Controller1 is
  pragma Interrupt_Priority(Interrupt_Priority'Last);
  entry Register;
  procedure Timer_Handler(E : in out Timing_Event);
  procedure Group_Handler(G : in out Group_Budget);
private
    T_Event : Timing_Event;
    G_Budget : Group_Budget;
    For_All : Boolean := False;
end Controller1;

with Ada.Asynchronous_Task_Control;
use Ada;
protected body Controller1 is
  entry Register when Register'Count = 4 or For_All is
  begin
    if not For_All then
      For_All := True;
      G_Budget.Add(Milliseconds(2));
      G_Budget.Add_Task(Register'Caller);
      T_Event.Set_Handler(Milliseconds(10),Timer_Handler'Access);
      G_Budget.Set_Handler(Group_Handler'Access);
    else
      G_Budget.Add_Task(Register'Caller);
    end if;
  end Register;

  procedure Timer_Handler(E : in out Timing_Event) is
    T_Array : Task_Array := G_Budget.Members;
  begin
    G_Budget.Replenish(Milliseconds(2));
    for ID in T_Array'range loop
      Asynchronous_Task_Control.Continue(T_Array(ID));
    end loop;
    E.Set_Handler(Milliseconds(10),Timer_Handler'Access);
  end Timer_Handler;

  procedure Group_Handler(G : in out Group_Budget) is
    T_Array : Task_Array := G.Members;
  begin
    for ID in T_Array'range loop
      Asynchronous_Task_Control.Hold(T_Array(ID));
    end loop;
  end Group_Handler;
end Controller1;
```

The `Register` entry blocks all calls until each of the four 'clients' has called in. The final task to register (which becomes the first task to enter) sets up the group budget and the timing event, adds itself to the group and alters the boolean flag so that the other three tasks will also complete their registration. For these tasks it is straightforward to add themselves to the group budget. Note the tasks in this example may have different priorities.

The two handlers work together to control the tasks. Whenever the group budget handler executes, it stops the tasks from executing by using the `Hold` routine (from the Asynchronous_Task_Control package). It always gets a new list of members in case any have terminated. The Timer_Handler releases all the tasks using Continue, it replenishes the budget and then sets up another timing event for the next period (10 ms).

In a less stringent application it may be sufficient to just prevent new invocations of each task if the budget is exhausted. The current execution is allowed to compete and hence tasks are not suspended. The following example implements this simpler scheme, and additionally allows tasks to register dynamically (rather than all together at the beginning). The protected object is made more general purpose by representing it as a type with discriminants for its main parameters (replenishment in terms of milliseconds and budget measured in microseconds):

```
protected type Controller2(Period, Bud : Positive) is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  procedure Register;
  entry Proceed;
  procedure Timer_Handler(E : in out Timing_Event);
  procedure Group_Handler(G : in out Group_Budget);
private
  T_Event : Timing_Event;
  G_Budget : Group_Budget;
  First : Boolean := True;
  Allowed : Boolean := False;
  Req_Budget : Time_Span := Microseconds(Bud);
  Req_Period : Time_Span := Milliseconds(Period);
end Controller2;

Con : Controller2(10, 2000);
```

The client task would now have the structure:

```
task body Aperiodic_Task is
  --
begin
  Con.Register;
  loop
    Con.Proceed;
    -- wait for next invocation
    -- undertake the work of the task
  end loop;
end Aperiodic_Task;
```

The body of the controller is as follows:

```
with Ada.Task_Identification; use Ada.Task_Identification;
protected body Controller2 is
  entry Proceed when Allowed is
  begin
    null;
  end Proceed;
  procedure Register is
  begin
    if First then
      First := False;
      Add(G_Budget,Req_Budget);
      T_Event.Set_Handler(Req_Period,Timer_Handler'Access);
      G_Budget.Set_Handler(Group_Handler'Access);
      Allowed := True;
    end if;
    G_Budget.Add_Task(Current_Task);
  end Register;

  procedure Timer_Handler(E : in out Timing_Event) is
  begin
    Allowed := True;
    G_Budget.Replenish(Req_Budget);
    E.Set_Handler(Req_Period,Timer_Handler'Access);
  end Timer_Handler;

  procedure Group_Handler(G : in out Group_Budget) is
  begin
    Allowed := False;
  end Group_Handler;
end Controller2;
```

The next example illustrates the mechanism known as a **deferrable server**. Here, the server has a fixed priority, and when the budget is exhausted, the tasks are moved to a background priority `Priority'First`. This is closer to the first example, but retains some of the properties of the second approach:

```
protected type Deferrable_Controller(Period, Bud : Positive;
                           Pri : Priority) is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  procedure Register;
  procedure Timer_Handler(E : in out Timing_Event);
  procedure Group_Handler(G : in out Group_Budget);
private
  T_Event : Timing_Event;
  G_Budget : Group_Budget;
  First : Boolean := True;
  Req_Budget : Time_Span := Microseconds(Bud);
  Req_Period : Time_Span := Milliseconds(Period);
end Deferrable_Controller;

Con : Deferrable_Controller(10, 2000, 12);
-- assume this server has priority 12


protected body Deferrable_Controller is
  procedure Register is
  begin
    if First then
```

```
        First := False;
        G_Budget.Add(Req_Budget);
        T_Event.Set_Handler(Req_Period,Timer_Handler'Access);
        G_Budget.Set_Handler(Group_Handler'Access);
      end if;
      G_Budget.Add_Task(Current_Task);
      if G_Budget.Budget_Has_Expired then
        Set_Priority(Priority'First);
        -- sets client task to background priority
      else
        Set_Priority(Pri);
        -- sets client task to server's 'priority'
      end if;
    end Register;

    procedure Timer_Handler(E : in out Timing_Event) is
      T_Array : Task_Array := G_Budget.Members;
    begin
      G_Budget.Replenish(Req_Budget);
      for ID in T_Array'Range loop
        Set_Priority(Pri,T_Array(ID));
      end loop;
      E.Set_Handler(Req_Period,Timer_Handler'Access);
    end Timer_Handler;

    procedure Group_Handler(G : in out Group_Budget) is
      T_Array : Task_Array := G_Budget.Members;
    begin
      for ID in T_Array'Range loop
        Set_Priority(Priority'First,T_Array(ID));
      end loop;
    end Group_Handler;
  end Deferrable_Controller;
```

When a task registers, it is running outside the budget so it is necessary to check if the budget is actually exhausted during registration. If it is then the priority of the task must be set to the low value. Other properties of this algorithm should be clear to the reader from the previous discussions.

As a final example, a simple form of **sporadic server** will be programmed. This has a different replenishment algorithm from the previously presented algorithms. The budget is replenished not periodically but at a fixed time after it is used. The motivation for the sporadic server is that a sporadic task must not have a higher impact on lower priority tasks than an equivalent periodic task would have. So if the sporadic task is released at time t and executes for C then at time t+T the value C should be added to the budget; where T is the 'period' of the sporadic server (the supposed minimum inter-arrival time of the events releasing the sporadic task).

To program the sporadic server requires a number of changes to be made to the `Sporadic_Controller` given at the beginning of Chapter 13. Again, a timing event and a group budget are required. In the following there is a single controller for each sporadic task, so strictly speaking it is not a group budget and a timer could be used to program the server. However, the example can be expanded to support more than one task (see Section 16.5), and even for one task the server is easier to construct with a group budget. The task registers and then has the usual structure however, the release mechanism is now incorporated into the sporadic controller:

```
task body Sporadic_Task is
begin
  Sporadic_Controller.Register;
  -- any necessary initialisations etc
  loop
    Sporadic_Controller.Wait_For_Next_Invocation;
    -- undertake the work of the task
  end loop;
end Sporadic_Task;
```

Each time the task calls its controller, the amount of computation time it used last time must be noted and returned to the budget at the appropriate time in the future. To do this and still block the task until its release event occurs means that a requeue operation is required. Although there is only one task, it may execute a number of times (using less than the budget each time) and hence there can be more than one timing event outstanding. To enable a single handler to deal with all of these requires the timing event to be extended to include the amount of budget that must be returned. A dynamic algorithm is used that defines a new timing event every time a replenish event should occur. This requires an access type:

```
type Budget_Event is new Timing_Event with
record
  Bud : Time_Span;
end record;

type Bud_Event is access Budget_Event;
```

The full code for the server is as follows. Its behaviour and examples of its execution will be given later.

```
protected Sporadic_Controller is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  entry Wait_For_Next_Invocation;
  procedure Register;
  procedure Timer_Handler(E : in out Timing_Event);
  procedure Group_Handler(G : in out Group_Budget);
  procedure Release_Sporadic;
private
  entry Wait_For;
  TB_Event : Bud_Event;
  G_Budget : Group_Budget;
  Req_Budget : Time_Span := Milliseconds(4);
  Req_Period : Time_Span := Milliseconds(10);
  Start_Budget : Time_Span;
  Pri : Priority;
  Release_Time : Real_Time.Time;
  ID : Task_ID;
  Barrier : Boolean := False;
  Task_Executing : Boolean := True;
end Sporadic_Controller;

protected body Sporadic_Controller is
  procedure Register is
  begin
    ID := Current_Task;
    G_Budget.Add_Task(ID);
    G_Budget.Add(Req_Budget);
    G_Budget.Set_Handler(Group_Handler'Access);
    Release_Time := Real_Time.Clock;
    Start_Budget := Req_Budget;
    Pri := Get_Priority;
  end Register;

  entry Wait_For_Next_Invocation when True is
  begin
    -- work out how much budget used, construct timing event and
    -- set the handler
    Start_Budget := Start_Budget - G_Budget.Budget_Remaining;
    TB_Event := new Budget_Event;
    TB_Event.Bud := Start_Budget;
    TB_Event.Set_Handler(Release_Time+Req_Period,
                         Timer_Handler'Access);
    Task_Executing := False;
    requeue Wait_For with abort;
  end Wait_For_Next_Invocation;

  entry Wait_For when Barrier is
  begin
    if not G_Budget.Budget_Has_Expired then
      Release_Time := Real_Time.Clock;
      Start_Budget := G_Budget.Budget_Remaining;
    end if;
    Barrier := False;
    Task_Executing := True;
  end Wait_For;

  procedure Release_Sporadic is
  begin
    Barrier := True;
  end Release_Sporadic;

  procedure Timer_Handler(E : in out Timing_Event) is
  begin
    if G_Budget.Budget_Has_Expired and Task_Executing then
      -- the task is ready and able to execute
```

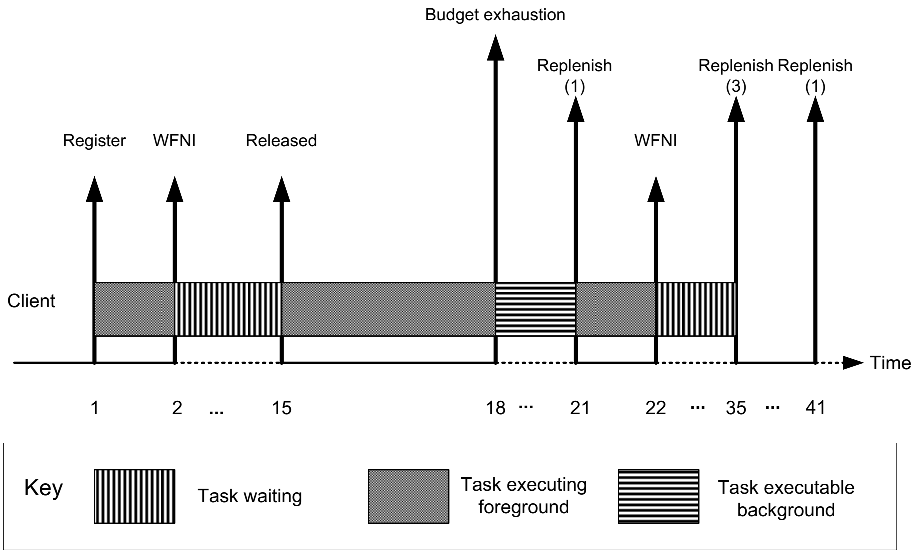


Fig. 15.1: Sporadic server illustration 1

```
      Release_Time := Real_Time.Clock;
      Start_Budget := Budget_Remaining(G_Budget);
    end if;
    G_Budget.Add(Budget_Event(Timing_Event'Class(E)).Bud);
    Set_Priority(Pri,ID);
  end Timer_Handler;


  procedure Group_Handler(G : in out Group_Budget) is
  begin
    -- a replenish event required for the budget used so far
    TB_Event := new Budget_Event;
    TB_Event.Bud := Start_Budget;
    TB_Event.Set_Handler(Release_Time+Req_Period,
                         Timer_Handler'Access);
    Set_Priority(Priority'First,ID);
    Start_Budget := Time_Span_Zero;
  end Group_Handler;
end Sporadic_Controller;
```

To understand how this algorithm works consider a sporadic server with budget of 4 ms and a replenishment interval of 20 ms. Figure 15.1 shows the execution of an example task.

The task will first call `Register`, this will set up the group budget, add the task to this group budget and note the time of the registration (for illustrative purposed assume the clock ticks at 1 ms intervals and that the registration took place at time 1). It also notes that its starting budget is 4 ms. After the execution of other

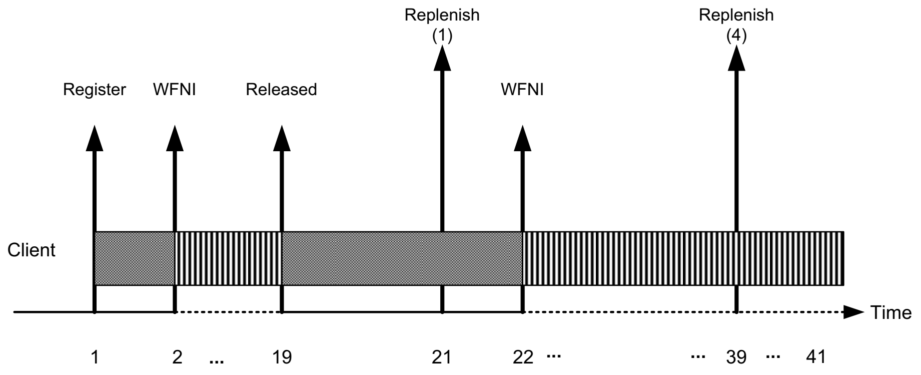


Fig. 15.2: Sporadic server illustration 2

initialisation activities the task will call in to await its release event. Assume this initial phase of execution takes 1 ms. Within `Wait_For_Next_Invocation`, a timing event is constructed that will trigger at time 21 with the budget parameter set at 1 ms.

The external call to `Release_Sproadic` now occurs at, say, time 15. The task is released and, as there is budget available, the release time of the task is noted (15) as is the current capacity of the budget (which is 3). If the task can execute immediately (i.e no high priority tasks are runnable) then it will start to use the budget. If its CPU requirement is, say, 4 ms, then it will execute for 3 ms and then, at time 18, the budget handler will be triggered, as the budget has been exhausted. In this handler, a timing event is again constructed; its trigger time is 35 and its budget parameter is 3 ms. The task is given a minimum priority, where it may or may not be able to execute. Assume first that it does not execute (i.e there are other runnable tasks with priorities above the minimum. The next event will be the triggering of the first timing event at time 21. This will add 1 ms to the budget and will allow the task to continue to execute at its correct priority level. The time of release is noted (21) and the budget outstanding (1 ms). If the task terminates within this 1 ms capacity then it will call `Wait_For_Next_Invocation` again and a timing event will be declared for triggering at time 41 (with parameter 1 ms).

If the task, while at the background priority, was able to execute then this would not impact on the budget (which is zero). If it gets as far as completing its execution then when it calls `Wait_For_Next_Invocation`, a timing event with parameter 0 ms will be constructed – this is inefficient, but a rare event and hence probably better to allow rather than test for non-zero parameter.

To illustrate just one further feature of the algorithm (see Figure 15.2). Now assume the call of `Release_Sporadic` occurred at time 19 (rather than 15). Now the task will start executing at time 19 and run until it completed that invocation at time 23. During that interval, the triggering of the timing event will occur at time 21. This will add 1 ms to the budget but have no further effects as the budget is non-zero at that time. When the task calls `Wait_For_Next_Invocation`, a timing event with a parameter of 4 ms and a triggering time of 39 will be constructed. It may appear that this replenishment time is too soon (should it not be 40?) but the analysis of the sporadic server algorithm does allow this optimisation – see the book by Liu in the reading list at the end of this chapter.

As CPU-time monitoring (and hence budget monitoring) is not exact, the above algorithm is likely to suffer from drift in the sense that the budget returned is unlikely to be exactly the equivalent of the amount used. To counter this, some form of re-asserting the maximum budget is required. For example, the `Timer_Handler` routine could be of the form:

```
procedure Timer_Handler(E : in out Timing_Event) is
begin
  G_Budget.Add(Budget_Event(Timing_Event'Class(E)).Bud);
  if Budget_Remaining(G_Budget) > Req_Budget then
    G_Budget.Replenish(Req_Budget);
  end if;
  ...
end Timer_Handler;
```

This will prevent too much extra budget being created. To counter budget leakage, it would be necessary to identify when the task has not run for time `Period` and then make sure the budget was at its maximum level (using `Replenish`). This could be achieved with a further timing event that is set when the task is blocked on `Wait_For` and triggered at this time plus `Period` unless it is cancelled in `Release_Sporadic`.

A final consideration with the above structure is the use of a new timing event every time a replenishment action is required. If this approach is too dynamic then a fixed number of timing events could be defined and reused (see Section 16.5). If at any time a timing event was not available then the last used timing event is reused in the following manner. Assume the last timing event used is expecting to return capacity C1 at time t. Now we have a new request to return C2 at time s (where s is after t). We combine the two actions by returning C1+C2 at time s. This is a safe action as the capacity C1 is being returned later than is necessary. The restriction on the maximum number of outstanding replenishment actions is also part of the POSIX sporadic server facility.

Eliminating a race condition

In all the server examples given so far in this chapter, there is a potential race condition that requires attention if resilient algorithms are to be used. The following scenario will generate such a race condition:

- a group budget is exhausted, and hence a group budget event is due to be triggered, but at the same time
- a timing event is also due that will replenish the same group budget.

If the timing event is delivered to the program first it will set all the tasks (in the group) to the foreground priority after replenishing the budget. Unfortunately, the group budget handler will now run and set the tasks' priorities to background, even though there is budget available.

To remedy this situation, it is necessary to check that there is no budget remaining before lowering the priorities in the budget handling protected procedure.

For example, the budget group handler of the deferrable server, given in the previous section, becomes:

```
procedure Group_Handler(G : in out Group_Budget) is
  T_Array : Task_Array := G_Budget.Members;
begin
  if G.Budget_Remaining = Time_Span_Zero then
    for ID in T_Array'Range loop
      Set_Priority(Priority'First,T_Array(ID));
    end loop;
  end if;
end Group_Handler;
```

15.7 Task termination events

Finally, this chapter covers the fourth event type in Ada 2005.

Ada 2005 change: Ada 2005 supports an implicit event type that is fired whenever a task terminates

This facility for handling task termination is a compromise between the needs to assign a specific handler to a specific task and to assign a general handler to a set of tasks (possible the whole program). As groups of tasks are already linked by the master/dependants relationship, Ada 2005 uses this means of grouping tasks together rather than introducing a new abstraction. The support package for the notification of task termination is as follows:

```
with Ada.Task_Identification;
with Ada.Exceptions;
package Ada.Task_Termination is
  pragma Preelaborate(Task_Termination);

  type Cause_Of_Termination is (Normal,Abnormal,
                                 Unhandled_Exception);
  type Termination_Handler is access protected procedure
    (Cause : in Cause_Of_Termination;
     T     : in Ada.Task_Identification.Task_Id;
     X     : in Ada.Exceptions.Exception_Occurrence);
  procedure Set_Dependents_Fallback_Handler
    (Handler : in Termination_Handler);

  function Current_Task_Fallback_Handler
           return Termination_Handler;

  procedure Set_Specific_Handler
    (T       : in Ada.Task_Identification.Task_Id;
     Handler : in Termination_Handler);

  function Specific_Handler (T : Ada.Task_Identification.Task_Id)
       return Termination_Handler;
end Ada.Task_Termination;
```

To understand the difference between the specific handler and the fall-back handler, it is best to consider first the required behaviour whenever a task terminates. As part of finalisation of the task, a handler is sought for execution; if there is a specific handler set for the task then that handler is executed. If there is no specific handler (i.e. there has never been one set or a null handler has been set) then a search is made for a fall-back handler. First the master task of the terminating task is checked to see if it set a fall-back handler, if not then the master of the master is checked and so on. Either a non-null fall-back handler is found, in which case it is executed, or no handler is found and hence none is executed.

So at maximum one handler is executed, and specific handlers have precedence over fall-back handlers. It follows that if the environment task sets a fall-back handler (in some library routine) then all task terminations will result in the handler being executed. Note that a task that sets a fall-back handler for its dependants is not covered by that handler – it would need to use `Set_Specific_Handler` on itself.

The handler is, as always, an access to a protected procedure. But it is not parameterless on this occasion. The task involved, the cause of termination and the ID of the unhandled exception (if that is what caused termination) are all passed to the handler. The null exception ID is passed if appropriate.

Warning: Particular care must be taken when setting the ceiling of any protected object that contains a handler. It must be higher than the priority of any task that could terminate and invoke the handler. If it is not then an exception will be raised in the handler. This will have no effect so the task's termination will again be silent.

The motivation for these language features is to deal with task termination in a uniform way that is external to the task. No changes to the body of the task are required. One use will be during debugging to notice 'silent' task terminations (although a decent debugger should allow the programmer to trace such behaviour).

In a live system, the scope for program recovery following partial failure is usually limited, but graceful program termination is one possible response to unanticipated task termination. Of course, for many situations, it is more acceptable to put code within the task body to deal with these issues. The use of a catch-all **`when others`** exception handler or a local object of a controlled type will allow different causes of termination to be identified and acted upon.

15.8 Summary

Five important new features of Ada 2005 have been introduced in this chapter; timing events, execution-time monitoring, timers, group budgets and task termination events. Each is useful in its own right, but together they allow a range of high-level abstractions to be programmed efficiently. As an example of this expressive power, a number of execution-time server mechanisms have been illustrated. The sporadic server for example uses timing events, execution-time control and group budgets to enforce the constrained behaviour required of sporadic tasks – they cannot have an impact on the system greater than a period task with period equal to the minimum arrival interval of the sporadic task.

Although all four event types introduced in this chapter have similar characteristics, there are a number of clear differences. Firstly, the event itself is explicit for timing events and timers but implicit for group budgets and task terminations. Secondly, the handler is permanently linked to the event in the case of group budgets, but has to be reset when timing events or timers are reused. For task termination, the event can be fired at most once and hence the handler cannot be re-executed, although the same handler can be associated with many tasks.

15.9 Further reading

J.W.S. Liu, *Real-Time Systems*, Prentice-Hall, 2000.

16
Real-time utilities

Ada 2005 provides a very flexible framework for real-time object-oriented concurrent programming. In Chapter 11, a set of generic concurrency utilities were illustrated. This chapter adopts a similar approach to define a set of reusable real-time programming abstractions based on real-time tasks.

Warning: It is beyond the scope of this book to provide a definitive set of reusable real-time utilities (that might be suitable for public distribution), rather the goal is to initiate discussion within the Ada community towards the development of such a library.

In the field of real-time programming, real-time tasks are often classified as being periodic, sporadic or aperiodic. Chapter 15 has shown how the Ada primitives can be combined to implement each of these task types. Simple real-time tasks are easy to program but once more complicated ones are needed (such as those that detect deadline misses and execution time overruns), the paradigms become more complex. Hence, there is a need to package up some of these and provide them as real-time tasking utilities.

A programmer wanting to use a real-time tasking abstraction will want to indicate:

- whether the abstraction is periodic, sporadic or aperiodic;
- either to terminate the current release of the task in the event of a deadline miss or to simply inform the program that this event has occurred (in which case, the program can choose to react or ignore the event);
- either to terminate the current release of the task in the event of an execution-time overrun or to simply inform the program that this event has occurred (in which case, the program can choose to react or ignore the event);
- whether it is associated with an execution-time server that can limit the amount of CPU-time it receives.

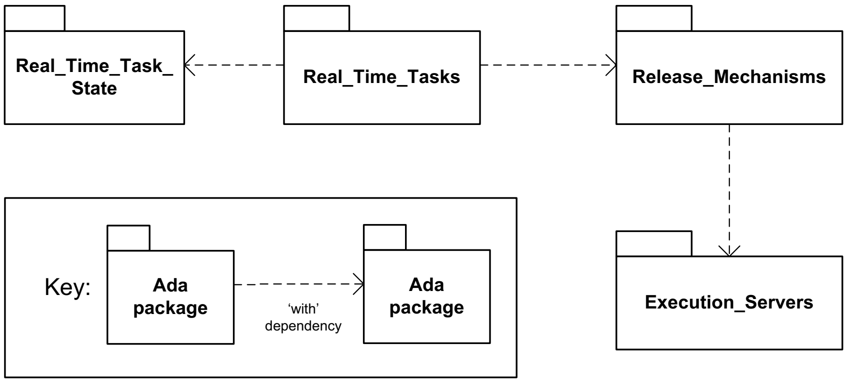


Fig. 16.1: Top-level packages

This chapter illustrates how real-time task abstractions, supporting these variations, can be developed in Ada. Section 16.7 shows how the abstractions can be used to construct an application (in this case, a simple cruise control system).

The approach that has been taken is to divide the support for the tasks into four components.

(1) The functionality of the task – this is encapsulated by the `Real_Time_Task_State` package. Its goal is to define a structure for the application code of the tasks. It is here that code is provided: to execute on each release of the task, to execute when deadline misses occur and when execution-time overruns occur.

(2) The various idioms of how the task should respond to deadline misses and execution-time overruns – this is encapsulated in the `Real_Time_Task` package. It is here that the actual Ada tasks are provided.

(3) The mechanisms needed to control the release of the real-time tasks and to detect the deadline misses and execution-time overruns – this is encapsulated in the `Release_Mechanisms` package. Each mechanism is implemented using a combination of protected objects and the new Ada 2005 timer and execution-time control features.

(4) The mechanisms needed to encapsulate subsystems and ensure that they are only given a fixed amount of the CPU resource (often called temporal firewalling) – this is the responsibility of the `Execution_Servers` package.

Figure 16.1 illustrates the top-level packages that make up the abstractions. The full details are discussed in the following sections.

Warning: This chapter assumes fixed priority scheduling and that the deadlines of all periodic tasks are less than or equal to their associated periods.

16.1 Real-time task state

First, it is necessary to provide a framework within which the programmer can express the code that the real-time task wishes to execute along with its associated state variables. This is achieved, in the usual object-oriented fashion, by defining the state within a tagged type, and providing operations to execute on the state. The following package shows the state and operations that all real-time tasks need.

```
with Ada.Real_Time, System;
use Ada.Real_Time, System;
package Real_Time_Task_State is
  type Task_State is abstract tagged record
    Relative_Deadline : Time_Span;
    Execution_Time : Time_Span := Time_Span_Last;
    Pri : Priority := Default_Priority;
  end record;

  procedure Initialize(S: in out Task_State) is abstract;
  procedure Code(S: in out Task_State) is abstract;
  procedure Deadline_Miss(S: in out Task_State) is null;
  procedure Overrun(S: in out Task_State) is null;

  type Any_Task_State is access all Task_State'Class;
end Real_Time_Task_State;
```

Every real-time task has a deadline, an execution time and a priority. Here, these fields are made public, but they could have just as well been made private and procedures to 'get' and 'set' them provided. No assumptions have been made about the values of these attributes. For example, the execution time could be worst-case or average.

The operations to be performed on a task's state are represented by the four abstract procedures:

- `Initialize` – this code is used to initialise the real-time task's state when the task is created;
- `Code` – this is the code that is executed on each release of the task;
- `Deadline_Miss` – this is the code that is executed if a deadline is missed.
- `Overrun` – this is the code that is executed if an execution-time overrun occurs.

Note, all real-time code must provide the `Initialize` and the `Code` procedures. There are default null actions for missed deadlines and overruns.

Child packages of `Real_Time_Task_State` provide support for periodic, aperiodic and sporadic task execution (as illustrated in Figure 16.2).

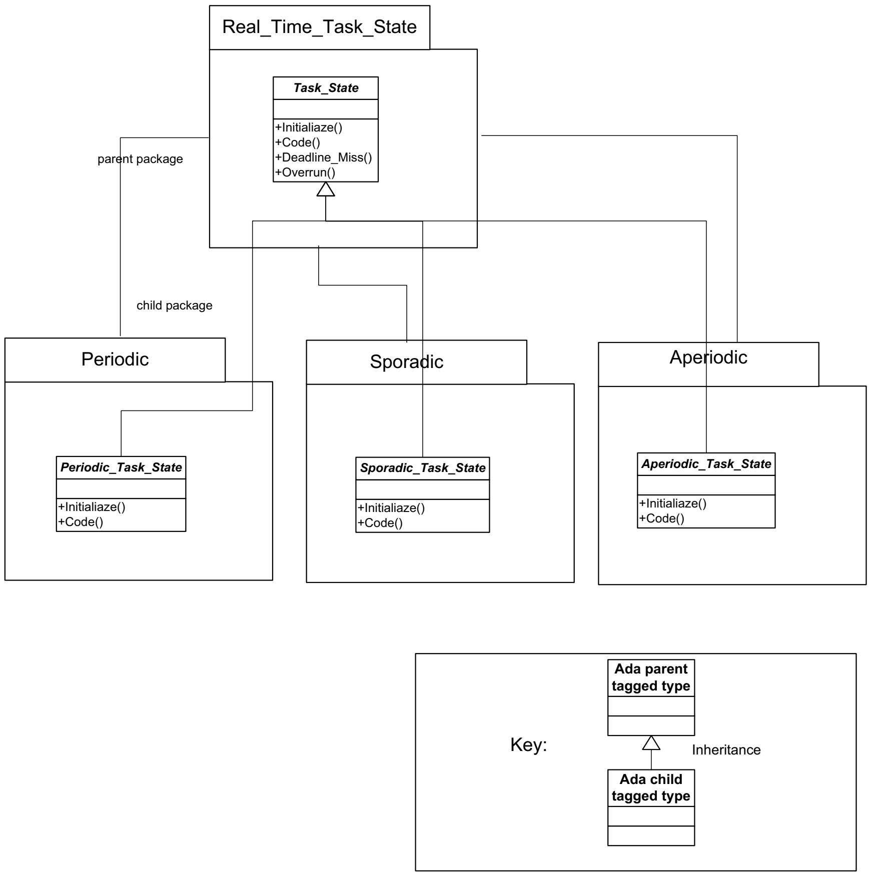


Fig. 16.2: Task states

A periodic task's state includes that of a real-time task with the addition of its period of execution. In other words, it has regular time-triggered releases.

```
package Real_Time_Task_State.Periodic is
  type Periodic_Task_State is abstract new Task_State with
  record
    Period : Time_Span;
  end record;

  procedure Initialize(S: in out Periodic_Task_State)
            is abstract;
  procedure Code(S: in out Periodic_Task_State)is abstract;

  type Any_Periodic_Task_State is access all
       Periodic_Task_State'Class;
end Real_Time_Task_State.Periodic;
```

There is more than one model of a sporadic task; here, it is assumed that the task must have an enforced minimum inter-arrival time between releases. Hence, the state includes this value.

```
package Real_Time_Task_State.Sporadic is
  type Sporadic_Task_State is abstract new Task_State with
  record
    MIT : Time_Span;
  end record;
  procedure Initialize(S: in out Sporadic_Task_State)
            is abstract;
  procedure Code(S: in out Sporadic_Task_State) is abstract;
  type Any_Sporadic_Task_State is access all
       Sporadic_Task_State'Class;
end Real_Time_Task_State.Sporadic;
```

The state for aperiodic tasks has no new fields over the normal `Task_State`, but for uniformity, a new type is created.

```
package Real_Time_Task_State.Aperiodic is
  type Aperiodic_Task_State is abstract new Task_State
       with null record;
  procedure Initialize(S: in out Aperiodic_Task_State)
            is abstract;
  procedure Code(S: in out Aperiodic_Task_State)
            is abstract;
  type Any_Aperiodic_Task_State is access all
      Aperiodic_Task_State'Class;
end Real_Time_Task_State.Aperiodic;
```

Application real-time tasks choose the appropriate real-time state to extend, and add their own state variables. Examples of periodic and sporadic application tasks are given in Section 16.7.

16.2 Real-time task release mechanisms

Real-time tasks can be released by the passage of time or via a software or hardware event. The following package (`Release_Mechanisms`) provides the common interfaces for all mechanisms (illustrated in Figure 16.3).

The root of the interface hierarchy (`Release_Mechanism`) simply supports the facility for a real-time task to wait for notification of its next release to occur (be that a time-triggered or an event-triggered release). The `Release_Mechanism_With_Deadline_Miss` interface is provided for the case where the real-time task wishes to be informed when it has missed a deadline. Similarly, the `Release_Mechanism_With_Overrun` interface is provided for the case where the real-time task wishes to be informed when it has overrun its execution time. Finally,

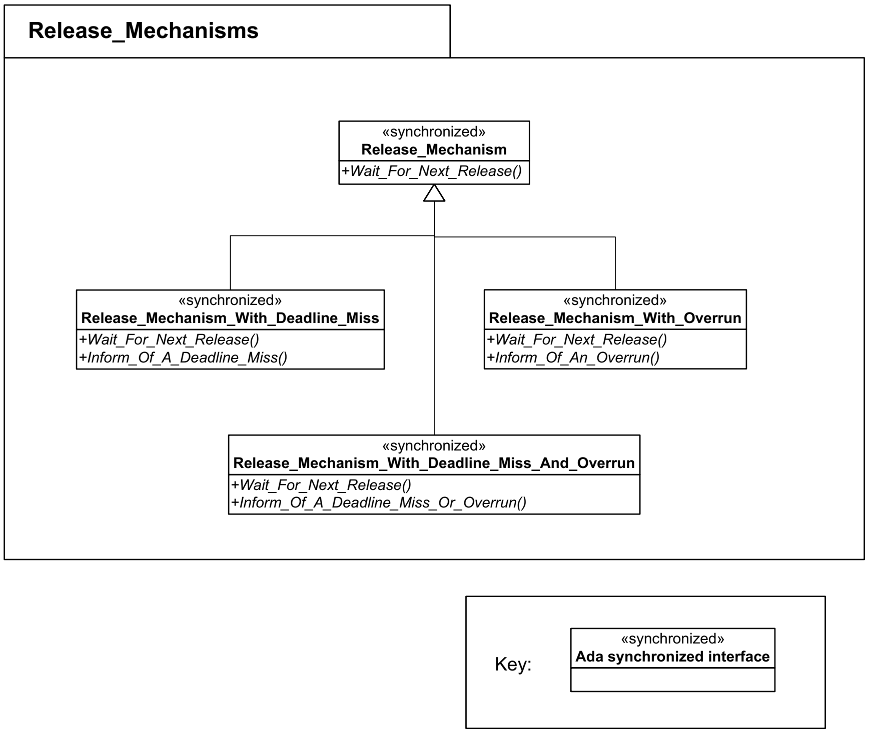


Fig. 16.3: Release mechanism interfaces

`Release_Mechanism_With_Deadline_Miss_And_Overrun` allows detection of both deadline misses and execution-time overruns. The Ada code is shown below.

```
package Release_Mechanisms is
  type Release_Mechanism is synchronized interface;
  procedure Wait_For_Next_Release(R : in out Release_Mechanism)
            is abstract;
  type Any_Release_Mechanism is access all Release_Mechanism'Class;

  type Release_Mechanism_With_Deadline_Miss is
      synchronized interface and Release_Mechanism;
  procedure Wait_For_Next_Release(R : in out
      Release_Mechanism_With_Deadline_Miss) is abstract;
  procedure Inform_Of_A_Deadline_Miss(R : in out
      Release_Mechanism_With_Deadline_Miss) is abstract;
  type Any_Release_Mechanism_With_Deadline_Miss is access all
      Release_Mechanism_With_Deadline_Miss'Class;

  type Release_Mechanism_With_Overrun is
      synchronized interface and Release_Mechanism;
  procedure Wait_For_Next_Release(R : in out
      Release_Mechanism_With_Overrun) is abstract;
  procedure Inform_Of_An_Overrun(R : in out
       Release_Mechanism_With_Overrun) is abstract;
  type Any_Release_Mechanism_With_Overrun is access all
       Release_Mechanism_With_Overrun'Class;

  type Release_Mechanism_With_Deadline_Miss_And_Overrun is
       synchronized interface and Release_Mechanism;
  procedure Wait_For_Next_Release(R : in out
       Release_Mechanism_With_Deadline_Miss_And_Overrun)
       is abstract;
  procedure Inform_Of_A_Deadline_Miss_Or_Overrun(R : in out
       Release_Mechanism_With_Deadline_Miss_And_Overrun;
       Missed : out Boolean) is abstract;
  type Any_Release_Mechanism_With_Deadline_Miss_And_Overrun
       is access all
       Release_Mechanism_With_Deadline_Miss_And_Overrun'Class;
end Release_Mechanisms;
```

Note that the final interface supports a single notification for either deadline miss or overrun detection. Hence, it does not inherit from the Release_Mechanism_With_Deadline_Miss and the Release_Mechanism_With_Overrun interfaces, but is a direct descendant of the Release_Mechanism interface.

Child packages provide the actual release mechanisms. For example, Figure 16.4 shows the mechanisms for periodic real-time tasks.

16.3 Periodic release mechanisms

The Periodic child package of Release_Mechanism implements time triggered releases. First, the package specification is given:

```
with System, Ada.Real_Time, Ada.Real_Time.Timing_Events,
     Real_Time_Task_State.Periodic;
use  System, Ada.Real_Time, Ada.Real_Time.Timing_Events,
     Real_Time_Task_State.Periodic;
package Release_Mechanisms.Periodic is
  protected type Periodic_Release(S: Any_Periodic_Task_State) is
          new Release_Mechanism with
    entry Wait_For_Next_Release;
    pragma Priority(System.Interrupt_Priority'Last);
  private
    procedure Release(TE : in out Timing_Event);
    Event : Timing_Event;
    Next : Time;
    New_Release : Boolean := True;
    First: Boolean := True;
  end Periodic_Release;
end Release_Mechanisms.Periodic;
```

In the above package, a protected type is defined that implements the synchronized Release_Mechanism interface. The type takes as a discriminant an ac-

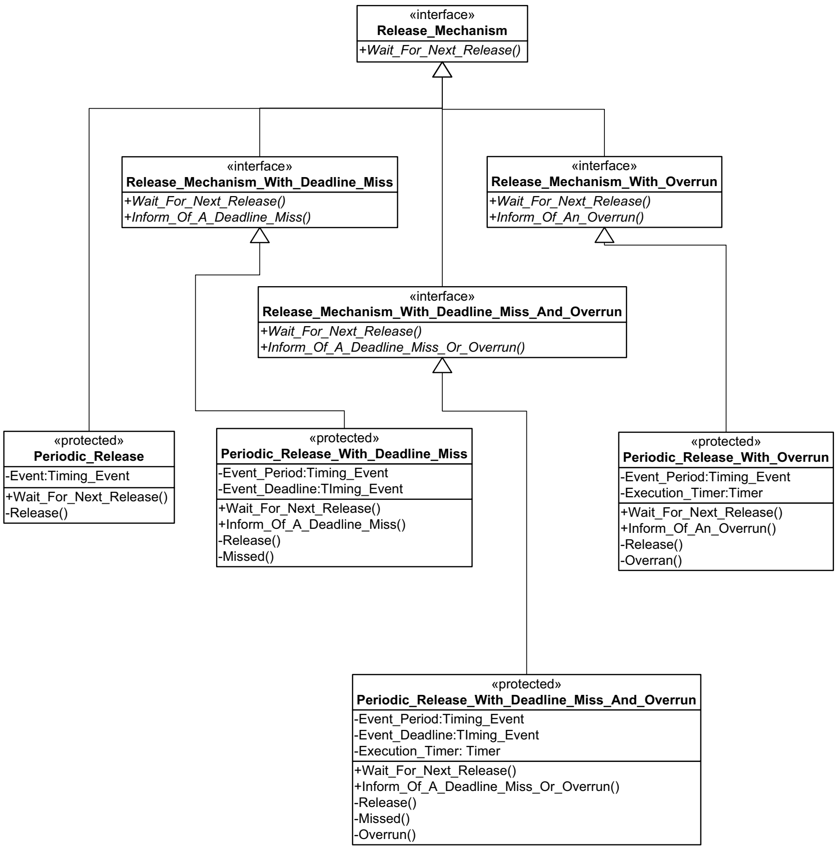


Fig. 16.4: Release mechanism classes

cess value to any periodic task state. It defines two protected actions: one that implements the required abstract operation of the `Release_Mechanism` interface (the `Wait_For_Next_Release` entry); the other (`Release`) that implements the operation that will be called to provide the releases at the appropriate times. Note, this has been made private as it should not be called by the real-time tasks. Also note that as this mechanism provides no support for deadline miss or execution-time overrun detection, the `Relative_Deadline` and `Execution_Time` state attributes are not used.

The periodic release mechanism uses Ada's new timing events facility to generate the calls to the `Release` procedure. Hence, the private part of the protected type defines the `Event` object as part of the state (and the ceiling priority is set to `Interrupt_Priority'Last`). The other variables are used to control the actual release of the real-time threads, as shown in the body below.

```
with Epoch_Support; use Epoch_Support;
package body Release_Mechanisms.Periodic is
  protected body Periodic_Release is
```

```
    entry Wait_For_Next_Release when New_Release is
    begin
      if First then
        First := False;
        Epoch_Support.Epoch.Get_Start_Time(Next);
        Next := Next + S.Period;
        Event.Set_Handler(Next, Release'Access);
        New_Release := False;
        requeue Periodic_Release.Wait_For_Next_Release;
      else
        New_Release := False;
      end if;
    end Wait_For_Next_Release;

    procedure Release(TE : in out Timing_Event) is
    begin
      Next := Next + S.Period;
      New_Release := True;
      TE.Set_Handler(Next, Release'Access);
    end Release;
  end Periodic_Release;
end Release_Mechanisms.Periodic;
```

The release mechanisms for all periodic activities assume that the first release should be relative to some common program start time. This is maintained by an `Epoch_Support` package (see Section 17.4) that allows this time to be acquired.

The first time the `Wait_For_Next_Release` entry is called, it sets up the timing event to expire at one period from the epoch, and then requeues the calling task. When the timer expires, the system automatically calls the `Release` procedure, which sets up the next timing event and opens the barrier on the entry.

The following simple example shows how the above release mechanism is used in conjunction with a real-time task's state.

```
with Real_Time_Task_State, Real_Time_Task_State.Periodic,
     Release_Mechanisms, Release_Mechanisms.Periodic;
use  Real_Time_Task_State, Real_Time_Task_State.Periodic,
     Release_Mechanisms, Release_Mechanisms.Periodic;
...
  task type Example_Periodic(S : Any_Task_State;
                             R : Any_Release_Mechanism) is
    pragma Priority(S.Pri);
  end Example_Periodic ;

  task body Example_Periodic is
  begin
    S.Initialize;
    loop
      R.Wait_For_Next_Release;
      S.Code;
    end loop;
  end Example_Periodic;
```

Examples of the actual state and release mechanism declarations are given as follows:

```
with Release_Mechanisms, Real_Time_Task_State,
     Real_Time_Task_State.Periodic,
     Release_Mechanisms.Periodic;
use  Release_Mechanisms, Real_Time_Task_State,
     Real_Time_Task_State.Periodic,
     Release_Mechanisms.Periodic;
...
  type My_State is new Periodic_Task_State with record
    I : Integer;
  end record;

  procedure Initialize(S: in out My_State);
  procedure Code(S: in out My_State);

  Example_State: aliased My_State :=
                 (Pri=> System.Default_Priority + 1);
  Releaser : aliased Periodic_Release(Example_State'Access);
```

In the above example, the only added state is an integer variable. An instance of the `Periodic_Release` protected object is declared and given an access value to the task's state.

```
...
with System;
...
  procedure Initialize(S: in out My_State) is
  begin
    S.I := 2;
    S.Pri := System.Default_Priority;
    -- set up the deadline, execution time if needed
  end Initialize;

  procedure Code(S: in out My_State) is
  begin
     S.I := S.I * S.I;
  end Code;
```

The body contains the actual code of the task. It initialises its state variable to the value 2, and then squares the value on each periodic release. The initialisation procedure also sets the task's priority.

Deadline miss detection

To support deadline miss detection requires a different release mechanism, which is provided by the following package:

```
with System, Ada.Real_Time.Timing_Events,
     Real_Time_Task_State.Periodic;
use  System, Ada.Real_Time.Timing_Events,
     Real_Time_Task_State.Periodic;
package Release_Mechanisms.Periodic_And_Deadline_Miss_Detection is
  protected type
    Periodic_Release_With_Deadline_Miss(
       S: Any_Periodic_Task_State; Termination : Boolean) is
            new Release_Mechanism_With_Deadline_Miss with
    entry Wait_For_Next_Release;
    entry Inform_Of_A_Deadline_Miss;
    pragma Priority(System.Interrupt_Priority'Last);
  private
    procedure Release(TE : in out Timing_Event);
    procedure Missed(TE : in out Timing_Event);
    Event_Period : Timing_Event;
    Event_Deadline : Timing_Event;
    Next : Time;
    New_Release : Boolean := True;
    First: Boolean := True;
    Missed_Deadline : Boolean := False;
    Completed : Boolean := True;
  end Periodic_Release_With_Deadline_Miss;
end Release_Mechanisms.Periodic_And_Deadline_Miss_Detection;
```

The main difference between this and the previous package is that an additional timing event is required to track the deadline of the real-time task. Furthermore, it is necessary to know if the task has completed its last release.

For convenience, here the mechanisms support both notification and termination semantics on a deadline miss. The required semantics can be set via the discriminant. An additional entry and an additional procedure are provided to support the deadline timing event. Note, the `Inform_Of_A_Deadline_Miss` entry is provided for the termination case – the intention is for it be called within a 'select-then-abort' statement (see Section 16.6).

The body of the above package is shown below:

```
with Epoch_Support; use Epoch_Support;
package body Release_Mechanisms.
             Periodic_And_Deadline_Miss_Detection is
  protected body Periodic_Release_With_Deadline_Miss is
    entry Wait_For_Next_Release when New_Release or
                                                 not Completed is
     Cancelled : Boolean; -- for cancelling a timing event
    begin
      if First then
        First := False;
        Epoch_Support.Epoch.Get_Start_Time(Next);
        Next := Next + S.Period;
        Event_Period.Set_Handler(Next, Release'Access);
        New_Release := False;
        requeue Periodic_Release_With_Deadline_Miss.
                Wait_For_Next_Release;
      elsif New_Release then
        New_Release := False;
        Completed := False;
      else
        Completed := True;
        Event_Deadline.Cancel_Handler(Cancelled);
        if not Cancelled then ... end if;
        requeue Periodic_Release_With_Deadline_Miss.
                Wait_For_Next_Release;
      end if;
    end Wait_For_Next_Release;

    entry Inform_Of_A_Deadline_Miss when Missed_Deadline is
    begin
      Missed_Deadline := False;
    end Inform_Of_A_Deadline_Miss;

    procedure Release(TE : in out Timing_Event) is
    begin
      Event_Deadline.Set_Handler(Next+S.Relative_Deadline,
                                 Missed'Access);
      Next := Next + S.Period;
      New_Release := True;
      Event_Period.Set_Handler(Next, Release'Access);
    end Release;

    procedure Missed(TE : in out Timing_Event) is
    begin
      if Termination then
        Missed_Deadline := True;
      else
        S.Deadline_Miss;
      end if;
    end Missed;
  end Periodic_Release_With_Deadline_Miss;
end Release_Mechanisms.Periodic_And_Deadline_Miss_Detection;
```

In the above, the timer to detect the deadline miss is set when the release event occurs. The next release event is also set at this time. When the task completes, it calls the `Wait_For_Next_Release` entry, which cancels the deadline timer and requeues the task to wait for its next release event to occur.

Execution-time overrun

To support execution-time overrun detection again requires a different release mechanism, as illustrated by the following package specification.

```
with System, Ada.Real_Time.Timing_Events,
     Real_Time_Task_State.Periodic,
     Ada.Task_Identification, Ada.Execution_Time,
     Ada.Execution_Time.Timers, Ada.Real_Time;
```

```
use  System, Ada.Real_Time.Timing_Events,
     Real_Time_Task_State.Periodic,
     Ada.Task_Identification, Ada.Execution_Time,
     Ada.Execution_Time.Timers, Ada.Real_Time;
package Release_Mechanisms.Periodic_And_Overrun_Detection is
  protected type Periodic_Release_With_Overrun(
    S: Any_Periodic_Task_State; Termination : Boolean) is
            new Release_Mechanism_With_Overrun with
    entry Wait_For_Next_Release;
    entry Inform_Of_An_Overrun;
    pragma Priority(System.Interrupt_Priority'Last);
  private
    procedure Release(TE : in out Timing_Event);
    procedure Overran(TE : in out Timer);
    Event_Period : Timing_Event;
    Tid : aliased Task_Id;
    Execution_Timer : access Timer;
    Next : Time;
    New_Release : Boolean := True;
    First: Boolean := True;
    Overrun : Boolean := False;
  end Periodic_Release_With_Overrun;
end Release_Mechanisms.Periodic_And_Overrun_Detection;
```

The package uses the new Ada 2005 facilities for managing CPU-time. The execution timer must be tied to a particular task. Unfortunately, there is a circular dependency between the release mechanism (which needs task identifiers) and the real-time task (which needs the release mechanism). Below, the conflict is resolved by creating the timer when the release mechanism is first called using the task identifier of the task calling the `Wait_For_Next_Release` entry.

```
with Epoch_Support; use Epoch_Support;
package body Release_Mechanisms.Periodic_And_Overrun_Detection is
  protected body Periodic_Release_With_Overrun is
    entry Wait_For_Next_Release when New_Release is
    begin
      if First then
        First := False;
        Epoch_Support.Epoch.Get_Start_Time(Next);
        Next := Next + S.Period;
        Event_Period.Set_Handler(Next, Release'Access);
        Tid := Periodic_Release_With_Overrun.
                        Wait_For_Next_Release'Caller;
        Execution_Timer := new Timer(Tid'Access);
        New_Release := False;
        requeue Periodic_Release_With_Overrun.
                Wait_For_Next_Release;
      else
        Execution_Timer.Set_Handler(S.Execution_Time,
                        Overran'Access);
        New_Release := False;
      end if;
    end Wait_For_Next_Release;
    entry Inform_Of_An_Overrun when Overrun is
    begin
      Overrun := False;
    end Inform_Of_An_Overrun;

    procedure Release(TE : in out Timing_Event) is
    begin
      Next := Next + S.Period;
      New_Release := True;
      Event_Period.Set_Handler(Next, Release'Access);
    end Release;

    procedure Overran(TE : in out Timer) is
    begin
      if Termination then
        Overrun := True;
      else
        S.Overrun;
      end if;
    end Overran;
  end Periodic_Release_With_Overrun;
end Release_Mechanisms.Periodic_And_Overrun_Detection;
```

Deadline miss and execution-time overrun

Finally, a package is given that supported both deadline miss and overrun detection. The package combines the above two approaches. Here, only the package specification is given, the body can be found on the book's web site.

```
with System, Ada.Real_Time.Timing_Events,
     Real_Time_Task_State.Periodic, Ada.Execution_Time,
     Ada.Task_Identification, Ada.Execution_Time.Timers;
use  System, Ada.Real_Time.Timing_Events,
     Real_Time_Task_State.Periodic, Ada.Execution_Time,
     Ada.Task_Identification, Ada.Execution_Time.Timers;
package Release_Mechanisms.
        Periodic_And_Deadline_Miss_And_Overrun_Detection is

  protected type
    Periodic_Release_With_Deadline_Miss_And_Overrun(
      S: Any_Periodic_Task_State; Termination : Boolean) is new
      Release_Mechanism_With_Deadline_Miss_And_Overrun with
    entry Wait_For_Next_Release;
    entry Inform_Of_A_Deadline_Miss_Or_Overrun(
          Missed : out Boolean);
    pragma Priority(System.Interrupt_Priority'Last);
  private
    procedure Release(TE : in out Timing_Event);
    procedure Overran(TE : in out Timer);
    procedure Missed(TE : in out Timing_Event);
    Event_Period : Timing_Event;
    Event_Deadline : Timing_Event;
    Tid : aliased Task_Id;
    Execution_Timer : access Timer;
    Next : Time;
    New_Release : Boolean := True;
    First: Boolean := True;
    Missed_Deadline : Boolean := False;
    Overrun : Boolean := False;
    Completed : Boolean := True;
  end Periodic_Release_With_Deadline_Miss_And_Overrun;
end Release_Mechanisms.
    Periodic_And_Deadline_Miss_And_Overrun_Detection;
```

16.4 Sporadic release mechanisms

The sporadic release mechanisms mirror the periodic ones and are illustrated in Figure 16.5.

One of the main issues when handling sporadic releases is the handling of minimum inter-arrival time (MIT) violations. Here, the approach adopted is not to allow the release event to occur until the MIT has passed. To implement this requires a timing event. The specification of the simple sporadic release mechanism is given below.

```
with System, Ada.Real_Time.Timing_Events,
     Real_Time_Task_State.Sporadic;
use  System, Ada.Real_Time.Timing_Events,
     Real_Time_Task_State.Sporadic;
package Release_Mechanisms.Sporadic is
  protected type Sporadic_Release(S: Any_Sporadic_Task_State)
            is new Release_Mechanism with
    entry Wait_For_Next_Release;
    procedure Release;
    pragma Priority(System.Interrupt_Priority'Last);
  private
    procedure Release_Allowed(TE : in out Timing_Event);
    Event_MIT : Timing_Event;
    Last : Time;
    New_Release : Natural := 0;
    Minimum_Separation: Boolean := True;
  end Sporadic_Release;
end Release_Mechanisms.Sporadic;
```

Note that the `Release` procedure is now public as it will be called by the application. The body of the package is given below.

```
package body Release_Mechanisms.Sporadic is
  protected body Sporadic_Release is
    entry Wait_For_Next_Release when Minimum_Separation and
         New_Release > 0 is
    begin
      Last := Clock;
```

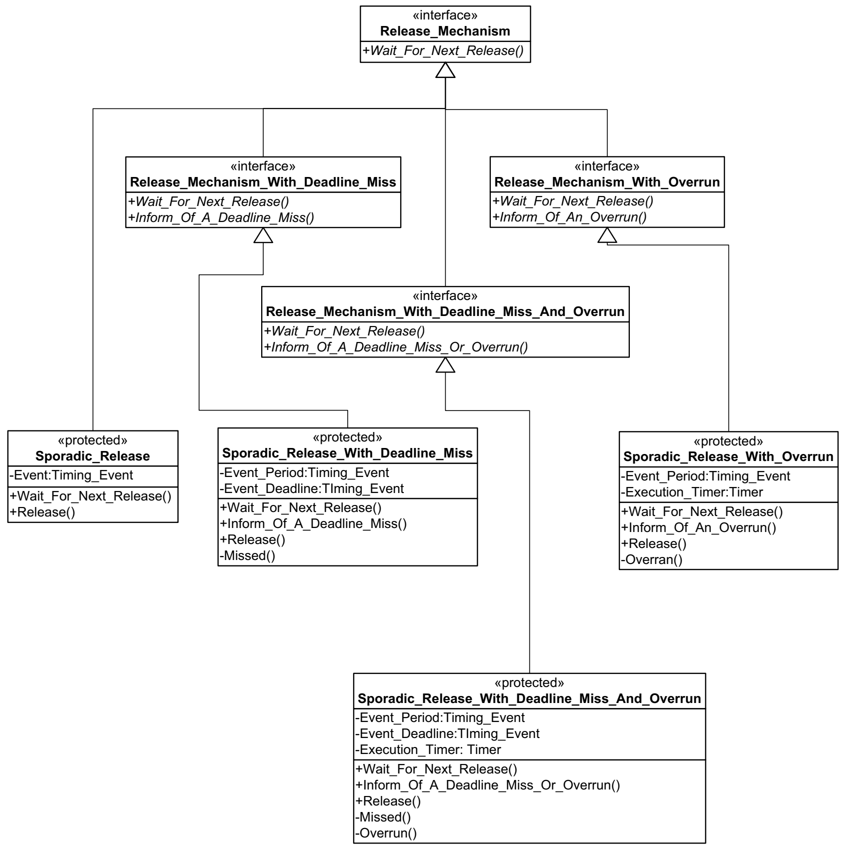


Fig. 16.5: Release mechanisms: sporadics

```
      Event_MIT.Set_Handler(Last + S.MIT, Release_Allowed'Access);
      New_Release := New_Release - 1;
      Minimum_Separation := False;
    end Wait_For_Next_Release;

    procedure Release is
    begin
      New_Release := New_Release + 1;
    end Release;

    procedure Release_Allowed(TE : in out Timing_Event) is
    begin
      Minimum_Separation := True;
    end Release_Allowed;
  end Sporadic_Release;
end Release_Mechanisms.Sporadic;
```

In the above code, only when the `Release` procedure has been called and the MIT has passed, is the barrier on the `Wait_For_Next_Release` entry opened.

To model release events generated by interrupts requires a different approach. In this case, the `Release` would be private and would be associated with an interrupt identifier, as illustrated below:

```
with System, Ada.Real_Time.Timing_Events,
     Real_Time_Task_State.Sporadic, Ada.Interrupts;
use  System, Ada.Real_Time.Timing_Events,
     Real_Time_Task_State.Sporadic, Ada.Interrupts;
package Release_Mechanisms.Sporadic_Interrupt is
  protected type Interrupt_Release(
        S: Any_Sporadic_Task_State; Interrupt: Interrupt_Id)
            is new Release_Mechanism with
    entry Wait_For_Next_Release;
    pragma Priority(System.Interrupt_Priority'Last);
  private
    procedure Release;
    pragma Attach_Handler(Release, Interrupt);
    procedure Release_Allowed(TE : in out Timing_Event);
    Event_MIT : Timing_Event;
    Last : Time;
    New_Release : Natural := 0;
    Minimum_Separation: Boolean := True;
  end Interrupt_Release;
end Release_Mechanisms.Sporadic_Interrupt;
```

The code for the body of this package and the other sporadic release mechanisms can be found on the web site accompanying this book.

16.5 Aperiodic release mechanisms and execution-time servers

The final type of release mechanism is that for handling aperiodic releases. Typically, the CPU time allocated to aperiodic tasks must be constrained as they potentially can have unbounded resource requirements. In Section 15.6 various approaches to managing the execution time of aperiodic tasks were presented. These can be used to program execution-time servers, which ensure that only a defined amount of CPU time is made available.

Before the release mechanism can be programmed, it is necessary to consider how it interacts with the execution servers. This chapter will only consider periodic execution servers. The following package specification defines the common interface.

```
with Ada.Real_Time, Ada.Task_Identification, System;
use  Ada.Real_Time, Ada.Task_Identification, System;
package Execution_Servers is
  type Server_Parameters is tagged record
    Period : Time_Span;
    Budget : Time_Span;
  end record;
  type Execution_Server is synchronized interface;
  procedure Register(ES: in out Execution_Server;
                     T : Task_Id) is abstract;
  procedure Start_Session(ES: in out Execution_Server;
                     T : Task_Id) is null;
  procedure Complete_Session(ES: in out Execution_Server;
                     T : Task_Id) is null;
  type Any_Execution_Server is access all Execution_Server'Class;
end Execution_Servers;
```

All servers have parameters that determine the servers' characteristics. They include:

- the budget – how much CPU time has been allocated to the server;
- the period – this relates to how often the server's budget is replenished.

The two servers illustrated in this book (the deferrable and sporadic servers) also require their clients to have foreground and background priorities, but in the general case this may not be the situation. Some servers suspend their clients when their execution time expires, and other servers allow their clients to have different priorities.

All execution servers require their clients to register. Here, any task can register any other tasks. Some types of execution servers will also want to know when the client tasks are executable. These periods of execution are called *sessions*. The associated procedures have default null values.

To facilitate the use of execution-time servers, it is necessary to modify the release mechanism. The approach is illustrated by considering aperiodic releases (although the same approach could be applied to the periodic or sporadic release mechanisms).

The simplest form of an aperiodic release mechanism does not support deadline miss detection as, in general, an aperiodic task does not have a strict deadline but requires a fast response time. The following package defines such a mechanism.

```
with System, Ada.Real_Time, Real_Time_Task_State.Aperiodic,
     Execution_Servers;
use  System, Ada.Real_Time, Real_Time_Task_State.Aperiodic,
     Execution_Servers;
package Release_Mechanisms.Aperiodic is
  protected type Aperiodic_Release(S: Any_Aperiodic_Task_State;
      ES: Any_Execution_Server) is new Release_Mechanism with
    entry Wait_For_Next_Release;
    procedure Release;
    pragma Priority(System.Interrupt_Priority'Last);
  private
    New_Release : Natural := 0;
    First : Boolean := True;
    Completed: Boolean := True;
  end Aperiodic_Release;
end Release_Mechanisms.Aperiodic;
```

The `Aperiodic_Release` protected type is similar in structure to the others presented in this chapter except for the inclusion of a reference to an execution server within the discriminants. This is used to register the real-time task when it first waits for a release, as shown below.

```
with Ada.Asynchronous_Task_Control; use Asynchronous_Task_Control;
with Ada.Dynamic_Priorities; use Ada.Task_Prioritites;
package body Release_Mechanisms.Aperiodic is

  protected body Aperiodic_Release is
    entry Wait_For_Next_Release when New_Release > 0 or
                First or not Completed is
    begin
      if First then
        First := False;
        ES.Register(Aperiodic_Release.
                    Wait_For_Next_Release'Caller);
        requeue Aperiodic_Release.Wait_For_Next_Release;
      elsif not Completed then
        Completed := True;
        ES.Complete_Session(Aperiodic_Release.
                            Wait_For_Next_Release'Caller);
        requeue Aperiodic_Release.Wait_For_Next_Release;
      end if;
      New_Release := New_Release - 1;
      Completed := False;
      ES.Start_Session(Aperiodic_Release.
                       Wait_For_Next_Release'Caller);
    end Wait_For_Next_Release;

    procedure Release is
    begin
      New_Release := New_Release + 1;
    end Release;
  end Aperiodic_Release;
end Release_Mechanisms.Aperiodic;
```

Notice, that there is no attempt to regulate the releases of the associated real-time task, as this will be done via the server – examples of which are now given.

Deferrable servers

The approach to implementing a deferrable server has already been outlined in Section 15.6. The same approach is adopted here, only modified to link together with the aperiodic release mechanisms.

```
with System, Ada.Real_Time, Ada.Real_Time.Timing_Events,
     Ada.Execution_Time.Group_Budgets, System;
use  System, Ada.Real_Time, Ada.Real_Time.Timing_Events,
     Ada.Execution_Time.Group_Budgets, System;
package Execution_Servers.Deferrable is
  type Deferrable_Server_Parameters is new Server_Parameters with
  record
    Foreground_Pri : Priority;
    Background_Pri : Priority;
  end record;

  protected type Deferrable_Server(
            Params : access Deferrable_Server_Parameters)
       is new Execution_Server with
    overriding procedure Register(T : Task_Id := Current_Task);
    pragma Priority(System.Priority'Last);
  private
    procedure Timer_Handler(E : in out Timing_Event);
    procedure Group_Handler(G : in out Group_Budget);
    T_Event : Timing_Event;
    G_Budget : Group_Budget;
    First : Boolean := True;
  end Deferrable_Server;
end Execution_Servers.Deferrable;
```

A deferrable server does not need to know when its clients are executing; hence it does not override the procedures that support sessions. Consequently, these remain as null operations.

The package body makes use of the Ada 2005 group budget mechanisms.

```
with Ada.Dynamic_Priorities;
use Ada.Dynamic_Priorities;
package body Execution_Servers.Deferrable is
  protected body Deferrable_Server is
    procedure Register(T : Task_Id := Current_Task) is
    begin
      if First then
        First := False;
        G_Budget.Add(Params.Budget);
        T_Event.Set_Handler(Params.Period,Timer_Handler'Access);
        G_Budget.Set_Handler(Group_Handler'Access);
      end if;
      Add_Task(G_Budget, T);
      if G_Budget.Budget_Has_Expired then
        Set_Priority(Params.Background_Pri);
        -- sets client task to background priority
      else
        Set_Priority(Params.Foreground_Pri);
        -- sets client task to servers 'priority'
      end if;
    end Register;
    procedure Timer_Handler(E : in out Timing_Event) is
      T_Array : Task_Array := G_Budget.Members;
    begin
      G_Budget.Replenish(Params.Budget);
      for ID in T_Array'Range loop
        Set_Priority(Params.Foreground_Pri,T_Array(ID));
      end loop;
     E.Set_Handler(Req_Period,Timer_Handler'Access);
    end Timer_Handler;

    procedure Group_Handler(G : in out Group_Budget) is
      T_Array : Task_Array := G_Budget.Members;
    begin
      if G_Budget.Budget_Remaining = Time_Span_Zero then
        for ID in T_Array'Range loop
          Set_Priority(Params.Background_Pri,T_Array(ID));
        end loop;
      end if;
    end Group_Handler;
  end Deferrable_Server;
end Execution_Servers.Deferrable;
```

The Ada group budget facility is used to keep track of the CPU-time used by all the registered tasks. When the budget expires, the `Group_Handler` procedure is called. This first checks that there is no budget remaining and then sets the priority of all the registered tasks to background. (An alternative approach would be to use the Ada asynchronous task control mechanisms to suspend the task – as is done for the banded sporadic server below.) The replenishment period is triggered by a timing event; the resulting call to the `Timer_Handler` procedures replenishes the group budget and resets the registered tasks' priorities.

Banded sporadic servers

A simple sporadic server was discussed in Section 15.6. That approach is generalised to support more than one real-time task, each controlled by its own release mechanism. Furthermore, each task is allowed to have its own priority as long as it falls within a priority band. When the banded sporadic server's capacity is exhausted, the tasks are suspended.

Warning: The banded sporadic server here assumes that the client tasks do not suspend during their execution.

The specification for the banded sporadic server is given below:

```
with System, Ada.Real_Time.Timing_Events,
     Ada.Execution_Time.Group_Budgets,
     Ada.Dynamic_Priorities, Ada.Task_Identification;
```

```
use  System, Ada.Real_Time.Timing_Events,
     Ada.Execution_Time.Group_Budgets,
     Ada.Dynamic_Priorities, Ada.Task_Identification;
package Execution_Servers.Banded_Sporadic is
  Priority_Out_Of_Range : exception;
  Already_Member_Of_A_Group_Budget : exception;

  type Banded_Sporadic_Server_Parameters is
       new Server_Parameters with record
    Low_Priority : Priority;
    High_Priority : Priority;
  end record;

  type Budget_Event is new Timing_Event with record
    Bud : Time_Span;
  end record;
  type Bud_Event is access Budget_Event;
  type Bud_Events is array(Natural range <>) of Budget_Event;

  protected type Banded_Sporadic_Server
     (Params : access Banded_Sporadic_Server_Parameters;
      No_Timing_Events : Positive) is new Execution_Server with
    pragma Interrupt_Priority (Interrupt_Priority'Last);
    overriding procedure Start_Session(T : Task_Id);
    overriding procedure Complete_Session(T : Task_Id);
    overriding procedure Register(T : Task_Id := Current_Task);
  private
    procedure Timer_Handler(E : in out Timing_Event);
    procedure Group_Handler(G : in out Group_Budget);
    G_Budget : Group_Budget;
    B_Events : Bud_Events(1 .. No_Timing_Events);
    Next : Natural := No_Timing_Events;
    Number : Natural := 0;
    Start_Budget : Time_Span;
    Release_Time : Time;
    Tasks_Executing : Natural := 0;
    First : Boolean := True;
  end Banded_Sporadic_Server;
end Execution_Servers.Banded_Sporadic;
```

The server's parameters consist of the priority band within which all the client tasks must reside. If a task tries to register and it has a priority outside this range, the `Priority_Out_Of_Range` exception is raised. As the server uses the Ada 2005 group budget facility, the clients must also not already be members of other groups. Again, an exception is raised if the register task is already a member of a group. These tests are performed when the client tasks register.

Recall from Section 15.6 that the replenishment policy for the sporadic server requires that replenishment events must be generated at times relative to when budget is consumed. Hence, there may be more than one replenishment event at any particular time. Each event is represented in the program by the `Budget_Event` type. The maximum number of allowable outstanding events is passed as a discriminant to the Banded_Sporadic_Server type when an object of that type is created.

The overall approach taken by the banded sporadic server is to take a note of the amount of available budget (in Start_Budget) when the first task begins to execute a session. It also keeps track of the number of currently active sessions (in Tasks_Executing). When the number of active sessions goes to 0, the server generates a replenishment event for the amount of budget consumed by all the tasks in their associated sessions. This is shown below in the Start_Session and Complete_Session protected actions.

```
with Ada.Asynchronous_Task_Control;
use Ada.Asynchronous_Task_Control;
package body Execution_Servers.Banded_Sporadic is
  protected body Banded_Sporadic_Server is
    procedure Set_Timing_Events(B : Time_Span; T : Time) is
    begin
      if Number < No_Timing_Events then
        Next := Next mod No_Timing_Events + 1;
        B_Events(Next).Bud := B;
        B_Events(Next).Set_Handler(T, Timer_Handler'Access);
        Number := Number + 1;
      else
        B_Events(Next).Bud := B_Events(Next).Bud + B;
        B_Events(Next).Set_Handler(T, Timer_Handler'Access);
      end if;
    end Set_Timing_Events;

    procedure Register(T : Task_Id := Current_Task) is
    begin
      if First then
        First := False;
        G_Budget.Add(Params.Budget);
        G_Budget.Set_Handler(Group_Handler'Access);
      end if;
      G_Budget.Add_Task(T);
      if not Get_Priority(T) in
             Params.Low_Priority .. Params.High_Priority then
        raise Priority_Out_Of_Range;
      end if;
    exception
      when Group_Budget_Error =>
        raise Already_Member_Of_A_Group_Budget;
    end Register;

    procedure Start_Session(T : Task_Id)  is
    begin
      if Tasks_Executing = 0 then
        Release_Time := Clock;
        Start_Budget := G_Budget.Budget_Remaining;
      end if;
      Tasks_Executing := Tasks_Executing + 1;
      if G_Budget.Budget_Has_Expired then
        Hold(T);
      end if;
    end Start_Session;

    procedure Complete_Session(T : Task_Id) is
    begin
      -- work out how much budget used, construct timing
      -- event and set the handler
      Tasks_Executing := Tasks_Executing - 1;
      if Tasks_Executing = 0 then
        Set_Timing_Events(
            Start_Budget - G_Budget.Budget_Remaining,
            Release_Time + Params.Period);
      end if;
    end Complete_Session;

    procedure Timer_Handler(E : in out Timing_Event) is
      Bud : Time_Span;
      T_Array : Task_Array := G_Budget.Members;
    begin
      Number := Number - 1;
      Bud := Budget_Event(Timing_Event'Class(E)).Bud;
      if G_Budget.Budget_Has_Expired then
        G_Budget.Replenish(Bud);
        for I in T_Array'range loop
          Continue(T_Array(I));
        end loop;
        Release_Time := Clock;
        Start_Budget := Bud;
      else
        G_Budget.Add(Bud);
        Start_Budget := Start_Budget+Bud;
      end if;
    end Timer_Handler;

    procedure Group_Handler(G : in out Group_Budget) is
      T_Array : Task_Array := G_Budget.Members;
    begin
      if G_Budget.Budget_Remaining = Time_Span_Zero then
        -- a replenish event required for the budget used so far
        Set_Timing_Events(Start_Budget, Release_Time +
             Params.Period);
        for I in T_Array'range loop
          Hold(T_Array(I));
        end loop;
      end if;
    end Group_Handler;
  end Banded_Sporadic_Server;
end Execution_Servers.Banded_Sporadic;
```

When the group budget expires, the handler Group_Handler suspends all the registered tasks using Ada's Asynchronous_Task_Control mechanisms (using the Hold facility). When the replenishment events occur, the associated protected action (Timer_Handler) continues the held tasks.

16.6 Real-time tasks

The Real_Time_Tasks package provides the code that integrates the task states with the release mechanisms to provide the required application abstraction. Several task types are shown in the following package specification. The Init_Prio discriminant defines the priority at which the task will perform its initialisation. The Initialize subprogram associated with the task's state can set the priority for the main execution of the task.

```
with System, Ada.Real_Time, Release_Mechanisms,
     Real_Time_Task_State;
use  System, Ada.Real_Time, Release_Mechanisms,
     Real_Time_Task_State;
package Real_Time_Tasks is
  task type Simple_Real_Time_Task(S : Any_Task_State;
       R : Any_Release_Mechanism; Init_Prio : Priority) is
    pragma Priority(Init_Prio);
  end Simple_Real_Time_Task;

  task type Real_Time_Task_With_Deadline_Termination(
       S : Any_Task_State;
       R : Any_Release_Mechanism_With_Deadline_Miss;
       Init_Prio : Priority) is
    pragma Priority(Init_Prio);
  end Real_Time_Task_With_Deadline_Termination;

  task type Real_Time_Task_With_Overrun_Termination(
       S : Any_Task_State;
       R : Any_Release_Mechanism_With_Overrun;
       Init_Prio : Priority) is
    pragma Priority(Init_Prio);
  end Real_Time_Task_With_Overrun_Termination;

  task type Real_Time_Task_With_Deadline_And_Overrun_Termination(
       S : Any_Task_State;
       R : Any_Release_Mechanism_With_Deadline_Miss_And_Overrun;
       Init_Prio : Priority) is
    pragma Priority(Init_Prio);
  end Real_Time_Task_With_Deadline_And_Overrun_Termination;
end Real_Time_Tasks;
```

- `Simple_Real_Time_Task` – This can be used for the cases where there is no required termination of the current release in the advent of a missed deadline or execution-time overrun.
- `Real_Time_Task_With_Deadline_Termination` – This can be used for the cases where the current release of the task must be immediately terminated on the occurrence of a deadline miss.
- `Real_Time_Task_With_Overrun_Termination` – This can be used for the cases where the current release of the task must be immediately terminated if the execution time is exceeded.
- `Real_Time_Task_With_Deadline_And_Overrun_Termination` – This can be used for the cases where the current release of the task must be immediately terminated on the occurrence of a deadline miss or an execution-time overrun.

The body of this package shows the various structures. Note the use of the 'select-then-abort' statement to achieve the termination semantics.

```
package body Real_Time_Tasks is
  task body Simple_Real_Time_Task is
  begin
    S.Initialize;
    loop
      R.Wait_For_Next_Release;
      S.code;
    end loop;
  end Simple_Real_Time_Task;

  task body Real_Time_Task_With_Deadline_Termination is
  begin
    S.Initialize;
    loop
      R.Wait_For_Next_Release;
      select
        R.Inform_Of_A_Deadline_Miss;
        S.Deadline_Miss;
      then abort
        S.code;
      end select;
    end loop;
  end Real_Time_Task_With_Deadline_Termination;

  task body Real_Time_Task_With_Overrun_Termination is
  begin
    S.Initialize;
    loop
      R.Wait_For_Next_Release;
      select
        R.Inform_Of_An_Overrun;
        S.Overrun;
      then abort
```

```
        S.code;
      end select;
    end loop;
  end Real_Time_Task_With_Overrun_Termination;

  task body
    Real_Time_Task_With_Deadline_And_Overrun_Termination is
    Missed : Boolean;
  begin
    S.Initialize;
    loop
      R.Wait_For_Next_Release;
      select
        R.Inform_Of_A_Deadline_Miss_Or_Overrun(Missed);
        if Missed then S.Deadline_Miss;
        else S.Overrun; end if;
      then abort
        S.code;
      end select;
    end loop;
  end Real_Time_Task_With_Deadline_And_Overrun_Termination;
end Real_Time_Tasks;
```

Note that if the `Simple_Real_Time_Task` is used with one of the release mechanisms that support deadline miss or execution-time overrun detection, it should set the `Termination` discriminant to false. This will allow the task to be notified using a direct call to the `Deadline_Miss` and `Overrun` operations via the `Task_State`.

A simple example

Consider again the simple example given in Section 16.3. This is now extended to show termination and notification of deadline miss:

```
with Release_Mechanisms,
     Release_Mechanisms.Periodic_And_Deadline_Miss_Detection,
     Real_Time_Task_State, Real_Time_Task_State.Periodic;
use  Release_Mechanisms,
     Release_Mechanisms.Periodic_And_Deadline_Miss_Detection,
     Real_Time_Task_State, Real_Time_Task_State.Periodic;
...
  type My_State is new Periodic_Task_State with record
    I : Integer;
  end record;

  procedure Initialize(S : in out My_State);
  procedure Code(S : in out My_State);
  procedure Deadline_Miss(S : in out My_State);

  Example_State1 : aliased My_State;
  Example_State2 : aliased My_State;

  Releaser1 : aliased Periodic_Release_With_Deadline_Miss(
              Example_State1'Access, Termination => True);
  Releaser2 : aliased Periodic_Release_With_Deadline_Miss(
              Example_State2'Access,
              Termination => False);
```

In the above, two instances of the states are now created, one for each real-time task. There are two release mechanisms protected objects: `Release1` supports the termination model, and `Release2` supports the notification model.

Now, the real-time tasks can be declared:

```
with Release_Mechanisms, Real_Time_Task_State,
     Real_Time_Task_State.Periodic,
     Release_Mechanisms.Periodic, Real_Time_Tasks, System;
use  Release_Mechanisms, Real_Time_Task_State,
     Real_Time_Task_State.Periodic,
     Release_Mechanisms.Periodic, Real_Time_Tasks, System;
...
  T1 : Real_Time_Task_With_Deadline_Termination(
         Example_State1'Access,
         Releaser1'Access, Default_Priority);
  T2 : Simple_Real_Time_Task (
         Example_State2'Access,
         Releaser2'Access, Default_Priority);
```

Here, `T1` uses the real-time task type that supports the termination semantics, and `T2` uses the simple real-time task type.

For completeness, the codes of the tasks are given (they are identical, in this example).

```
with System, Ada.Real_Time;
use  System, Ada.Real_Time;
...
  procedure Initialize(S : in out My_State) is
  begin
    S.I := 2;
    S.Pri := Default_Priority + 2;
    S.Relative_Deadline := To_Time_Span(0.1);
    -- set up execution time if needed
  end Initialize;

  procedure Code(S : in out My_State) is
  begin
     S.I := S.I * 2;
  end Code;

  procedure Deadline_Miss(S : in out My_State) is
  begin
     S.I := 2;
  end Deadline_Miss;
```

The body initialises the state, sets the priority and the deadline; it then squares the state value on each periodic release. On a deadline miss, the value is reset to 2.

16.7 The cruise control system example

The automobile cruise control system (ACCS) case study has often been used by the real-time community over the years as it illustrates many of the differences between real-time embedded systems and traditional information processing systems. The requirements for the ACCS are presented first, followed by a discussion of the interactions between the system and its environment. The software design is then presented along with the Ada 2005 implementation that uses the real-time utilities introduced in this chapter.

16.7.1 ACCS requirements

The overall goal of an automobile cruise control system is to maintain the speed of a vehicle automatically to a value set by the driver. Commands to the system may be given explicitly or implicitly. The explicit commands are given by the driver's interface, which is a lever on the side of the steering wheel. The command lever has several positions that represent the following instructions to the system (the commands are only valid if the engine is turned on):

- activate – turn on the cruise control system if the car is in top gear, and maintain (and remember) the current speed;
- deactivate – turn off the cruise control system;
- start accelerating – accelerate the vehicle at a comfortable rate;
- stop accelerating – stop accelerating and maintain (and remember) the current speed;
- resume – return the car to the last remembered speed and maintain that speed.

Implicit commands are issued when the driver

- changes gear – the cruise control system is deactivated when the driver changes out of top gear;
- presses the brake pedal – the cruise control system is deactivated whenever the driver brakes.

The speed of the car is measured by the ACCS via the rotation of the shaft that drives the back wheels. The shaft will generate an interrupt for each rotation. The system has a default setting for the number of interrupts that should be generated for each kilometre travelled.

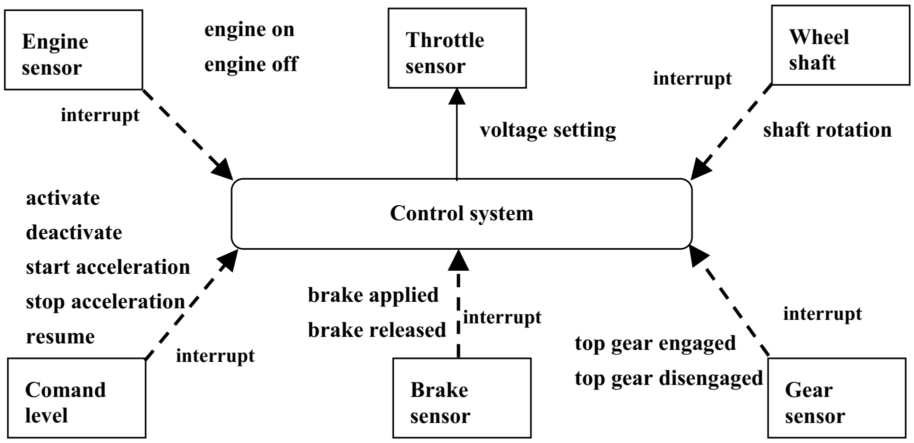


Fig. 16.6: Cruise control: system boundaries

The speed of the car depends on the throttle position. The throttle position is determined by two factors:

- the amount of depression of the accelerator pedal and
- the value supplied by the cruise control system.

The throttle's cruise control component is controlled by varying the voltage to the throttle actuator. The values range from 0 (throttle fully closed) to 8 (throttle fully open). Voltages can be set in units of 0.1 volts. A voltage setting holds its value for 3 seconds. If it is not reset, then a default of 0 volts is used. It is assumed that the combining of the two throttle component values is performed outside the cruise control system.

Cruise control throttle settings are defined for 'maintaining the current speed' and for 'comfortable acceleration'. The required settings are as follows:

- Maintaining speed – to maintain the desired speed, the following algorithm is used:
  (1) if the desired speed minus the actual speed is greater than 2 kph, the throttle is set to fully open;
  (2) if the actual speed minus the desired speed is greater than 2 kph, the throttle is set to fully closed;
  (3) otherwise, the value of the throttle is set to $2(S_D - S_A + 2)$ volts where S_D is the desired speed and S_A is the actual speed.

  More formally, if V is the voltage, then

$$V = \begin{cases} 8 & : \quad (S_D - S_A) > 2 \\ 2(S_D - S_A + 2) & : \quad -2 \leq (S_D - S_A) \leq 2 \\ 0 & : \quad -2 > (S_D - S_A) \end{cases}$$

- Comfortable acceleration – in order to avoid rapid changes in speed, the voltage to the throttle should not be changed by more than 0.1 in any one-second period.

16.7.2 System interactions

All devices are memory mapped and have associated control and data registers. Sensors in the automobile detect state changes and generate appropriate interrupts to the control system, as illustrated in Figure 16.6. With each interrupt, the data register contains the event that caused the state change. The throttle actuator's data register takes a value between 0 and 80. This represents the requested voltage in units of 0.1 volts.

16.7.3 Software design and implementation

The system consists of the following major components:

- Interrupt handling tasks for the command lever, brake, engine, wheel shaft and gear interrupts;
- A periodic real-time thread to control the throttle;
- A periodic thread to monitor the speed;
- A cruise controller to coordinate the system.

The interaction between these components is represented by the object collaboration diagram given in Figure 16.7.

Interrupt handling tasks

First-level interrupt handlers in Ada 2005 are represented as protected procedures. Second-level handlers are sporadic real-time tasks. Using the real-time utilities presented earlier in this chapter, each of the interrupt handlers in the cruise control system is modelled using a simple real-time task in conjunction with the sporadic interrupt release mechanism. First, however, it is necessary to provide the functionality of the interrupt processing itself. Consider, for example, the command lever interrupt processing. The following package specification illustrates how the code is constructed.

```
with Real_Time_Task_State.Sporadic, Cruise_Controller;
use Real_Time_Task_State.Sporadic, Cruise_Controller;
package Lever_Control_System is
  type Lever_Control is new Sporadic_Task_State with private;
  procedure Initialize(S : in out Lever_Control);
```

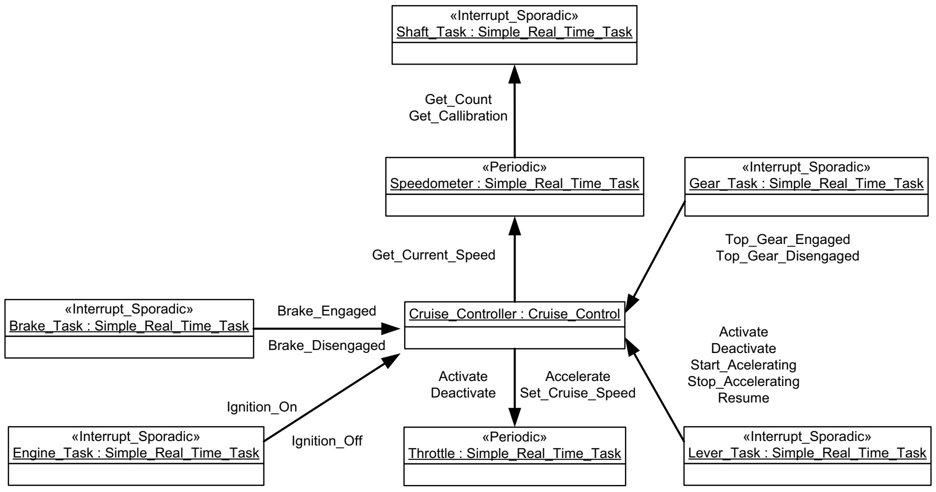


Fig. 16.7: Interaction between tasks

```
  procedure Code(S : in out Lever_Control);
private
  type Lever_Control is new Sporadic_Task_State with record
    Cruise : access Cruise_Control;
  end record;
end Lever_Control_System;
```

The Lever_Control_System package contains an Ada class that encapsulates the required functionality. The type Lever_Control extends the Sporadic_Task_State, thereby indicating that it is destined for a sporadic task. It overrides the Initialize and the Code procedures. This version assumes that missed deadlines are of no importance. The state of the real-time tasks is extended to include a reference to the cruise controller software.

The control of the command lever interrupts requires access to their associated device registers. A Device_Registers package contains the necessary data types. The ones relevant to the command lever are shown below:

```
with System, System.Storage_Elements;
use System, System.Storage_Elements;
package Device_Registers is
  Word : constant := 2; -- bytes in word
  type Control_Register is record
         Enable : Boolean;
         Done : Boolean;
  end record;

  for Control_Register use record
         Enable at 0 range 6 .. 6;
         Done at 0 range 15 .. 15;
  end record;
  for Control_Register'Size use Word_Size;
  for Control_Register'Alignment use Word;
  for Control_Register'Bit_Order use Default_Bit_Order;

  -- Lever Device Registers
  Lever_CSR : Control_Register;
  for Lever_CSR use at To_Address(8#170004#);

  type Lever_Command is (Activate_C, Deactivate_C,
       Start_Acceleration_C, Stop_Acceleration_C, Resume_C);
  for Lever_Command use  (Activate_C => 0, Deactivate_C => 1,
                         Start_Acceleration_C => 2,
                         Stop_Acceleration_C => 4, Resume_C => 8);
  Lever_Data : Lever_Command;
  for Lever_CSR use at To_Address(8#170006#);
  pragma Atomic(Lever_Data);
end Device_Registers;
```

The body of `Lever_Control_System` uses these to determine the cause of the interrupt and to inform the cruise controller of the event.

```
with Device_Registers; use Device_Registers;
with Cruise;
with Ada.Dynamic_Priorities; use Ada.Dynamic_Priorities;
package body Lever_Control_System is
  procedure Initialize(S : in out Lever_Control) is
  begin
    S.Cruise := Cruise.Cruise_Controller'Access;
    S.MIT := Cruise.Lever_MIT;
    S.Pri := Cruise.Lever_Priority;
    S.Execution_Time := Cruise.Lever_Budget;
    S.Relative_Deadline := Cruise.Lever_Deadline;
    Set_Priority(Cruise.Lever_Priority);
    Lever_CSR.Enable := True;
  end Initialize;

  procedure Code(S : in out Lever_Control) is
  begin
    case Lever_Data is
      when Activate_C =>
        Activate(S.Cruise.all);
      when Deactivate_C =>
        Deactivate(S.Cruise.all);
      when Start_Acceleration_C =>
        Start_Acceleration(S.Cruise.all);
      when Stop_Acceleration_C =>
        Stop_Acceleration(S.Cruise.all);
      when Resume_C =>
        Resume(S.Cruise.all);
    end case;
  end Code;
end Lever_Control_System;
```

The initialisation code first establishes the real-time attributes of the sporadic task. Here, it is assumed that they are defined in the `Cruise` package. It then stores a reference to the cruise controller and initialises the device register to enable interrupts.

The code to be executed on each interrupt is encapsulated in the `Code` procedure. This is simply a case statement on the device's data register. The cruise controller is then called to indicate the occurrence of the event.

Periodic real-time tasks

The system consists of two periodic activities: the speedometer and the throttle controller. Here, the `Speed_Control_System` package is presented.

```
with Real_Time_Task_State.Periodic, Shaft_Control_System;
use Real_Time_Task_State.Periodic, Shaft_Control_System;
package Speed_Control_System is
  subtype KPH is Integer range 0 .. 200;
  type Speedometer is new Periodic_Task_State with private;
  procedure Initialize(S : in out Speedometer);
  procedure Code(S : in out  Speedometer);
  procedure Deadline_Miss(S : in out Speedometer);
  function Get_Current_Speed(S : in Speedometer) return KPH;
private
  type Speedometer is new Periodic_Task_State with record
    Number_Rotations : Natural;
    Last_Number_Rotations : Natural;
    Current_Speed : KPH;
    pragma Atomic(Current_Speed);
    Calibration : Natural;
    Iterations_In_One_Hour : Natural;
    Shaft : access Shaft_Control;
  end record;
end Speed_Control_System;
```

The functionality of the speedometer is provided in a similar manner to that of the command lever control system. However, here the task will be periodic, so the state is an extension of `Periodic_Task_State`.

The initialisation procedure saves the last count from the shaft controller and determines its calibration. It also calculates the number of iterations that the task will make in one hour. On every period of the task, the current speed is approximated from the difference in the number of shaft rotations since its last period. The value is an approximation because of the variability of the task's execution due to the nature of fixed priority scheduling.

```
with Device_Registers, Cruise;
use Device_Registers;
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Dynamic_Priorities; use Ada.Dynamic_Priorities;
package body Speed_Control_System is
```

```
  Cm_In_Kilometre : constant := 100_000;

  procedure Initialize(S : in out Speedometer) is
  begin
    S.Shaft := Cruise.Shaft_Controller'Access;
    S.Period := Cruise.Speedometer_Period;
    S.Relative_Deadline := Cruise.Speedometer_Deadline;
    S.Pri := Cruise.Speedometer_Priority;
    Set_Priority(Cruise.Speedometer_Priority);
    S.Execution_Time := Cruise.Speedometer_Budget;
    S.Current_Speed := 0;
    S.Last_Number_Rotations := S.Shaft.Get_Count;
    S.Calibration := S.Shaft.Get_Calibration;
    S.Iterations_In_One_Hour := (3_600_000 / S.Period);
  end Initialize;

  procedure Code(S : in out Speedometer) is
    Difference : Integer;
    Tmp : Float;
  begin
    S.Number_Rotations := S.Shaft.Get_Count;
    Difference := S.Number_Rotations -
                    S.Last_Number_Rotations;
    Tmp := (float(Difference) * float(S.Calibration) *
            float(S.Iterations_In_One_Hour)) /
                              float(Cm_In_Kilometre);
    S.Current_Speed := Integer(Tmp);
    -- display speed
    S.Last_Number_Rotations := S.Number_Rotations;
  end Code;

  procedure Deadline_Miss(S : in out Speedometer) is
  begin
    -- record deadline miss for maintenance use
  end Deadline_Miss;

  function Get_Current_Speed(S : in Speedometer) return KPH is
  begin
    return S.Current_Speed;
  end Get_Current_Speed;
end Speed_Control_System;
```

Missing the deadline for the speedometer is, perhaps, more serious than for the interrupt handling tasks. It indicates that the calculated speed has a greater level of error. Here, the fact is simply logged and made available at the next car maintenance.

The throttle control system is similarly structured.

```
with Real_Time_Task_State.Periodic, Speed_Control_System;
use Real_Time_Task_State.Periodic, Speed_Control_System;
package Throttle_Control_System is
  type Throttle_Control is new Periodic_Task_State with private;
  procedure Initialize(S : in out Throttle_Control);
  procedure Code(S : in out  Throttle_Control);
  procedure Deadline_Miss(S : in out Throttle_Control);
  procedure Set_Cruise_Speed(S : in out Throttle_Control;
                             Speed : KPH);
  procedure Accelerate(S : in out Throttle_Control);
  procedure Activate(S : in out Throttle_Control);
  procedure Deactivate(S : in out Throttle_Control);
private
  type Throttle_Control is new Periodic_Task_State with record
    Active : Boolean := False;
    Cruise_Speed : KPH;
    Maintain_Speed : Boolean;
    Accelerating : Boolean;
    Voltage : Integer := 0;
    Speed : access Speedometer;
  end record;
end Throttle_Control_System;


with Device_Registers; use Device_Registers;
with Cruise; use Cruise;
with Ada.Dynamic_Priorities; use Ada.Dynamic_Priorities;
package body Throttle_Control_System is
  -- Throttle Control
  procedure Initialize(S : in out Throttle_Control) is
  begin
    S.Speed := Cruise.Speedometer_Controller'Access;
    S.Relative_Deadline := Throttle_Deadline;
    S.Period := Throttle_Period;
    S.Execution_Time := Throttle_Budget;
    S.Pri := Throttle_Priority;
    Set_Priority(Throttle_Priority);
  end Initialize;

  procedure Increase_Voltage(S : in out Throttle_Control) is
  begin
  -- Increase the voltage (to 8 max) but only if
  -- hasn't been increased within the last second.
    -- write voltage out
  end Increase_Voltage;

  procedure Code(S : in out Throttle_Control) is
  begin
    if not S.Active then return; end if;
    if S.Accelerating then
      Increase_Voltage(S);
    elsif S.Maintain_Speed then
      if (S.Cruise_Speed - S.Speed.Get_Current_Speed) > 2 then
        -- Move towards throttle fully open
        Increase_Voltage(S);
      elsif (S.Cruise_Speed - S.Speed.Get_Current_Speed) < -2 then
        -- throttle fully close
        S.Voltage := 0;
      elsif 2*(S.Cruise_Speed - S.Speed.Get_Current_Speed + 2) >
            S.Voltage then
        Increase_Voltage(S);
      end if;
    end if;
  end Code;

  procedure Deadline_Miss(S : in out Throttle_Control) is
  begin
    -- Log details for maintenance
  end Deadline_Miss;

  procedure Set_Cruise_Speed(S : in out Throttle_Control;
                             Speed : KPH) is
  begin
    S.Cruise_Speed := Speed;
    S.Maintain_Speed := True;
    S.Accelerating := False;
  end Set_Cruise_Speed;

  procedure Accelerate(S : in out Throttle_Control) is
  begin
    S.Accelerating := True;
  end Accelerate;

  procedure Activate(S : in out Throttle_Control) is
  begin
    S.Active := True;
    S.Voltage := 0;
  end Activate;

  procedure Deactivate(S : in out Throttle_Control) is
  begin
    S.Active := False;
  end Deactivate;
end Throttle_Control_System;
```

The cruise controller

The cruise controller system activities are coordinated by a state machine. Figure 16.8 summarises the states and the transitions.

The states are defined by a collection of boolean variables defined in the package specification. The interrupt handling tasks call the interface subprograms. The body of the package maintains the states and calls the throttle controllers and the speedometer as needed.

```
with Throttle_Control_System; use Throttle_Control_System;
with Speed_Control_System; use Speed_Control_System;
package Cruise_Controller is
  type Cruise_Control is private;
  procedure Initialize(CC : in out Cruise_Control);
```

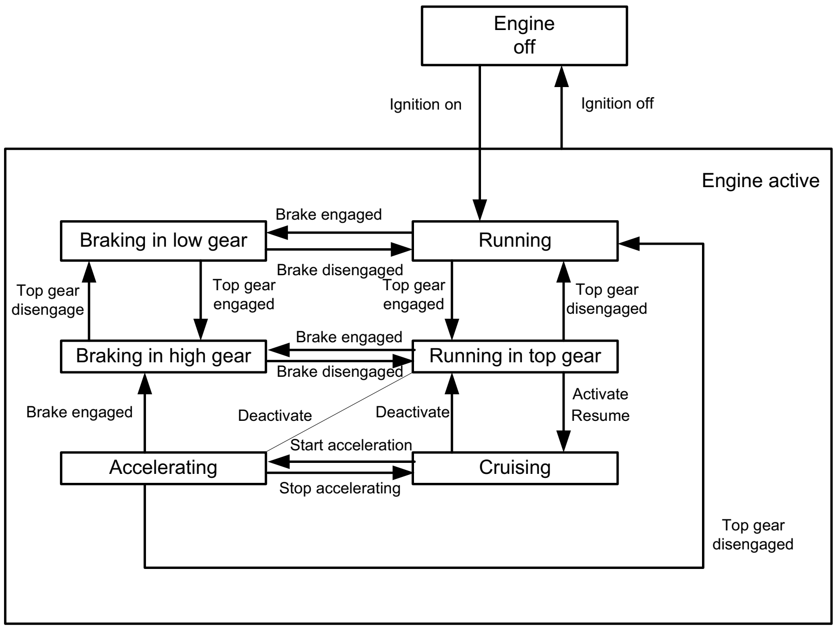


Fig. 16.8: Cruise controller state machine

```
  procedure Activate(CC : in out Cruise_Control);
  procedure Deactivate(CC : in out Cruise_Control);
  procedure Resume(CC : in out Cruise_Control);
  procedure Start_Acceleration(CC : in out Cruise_Control);
  procedure Stop_Acceleration(CC : in out Cruise_Control);
  procedure Ignition_On(CC : in out Cruise_Control);
  procedure Ignition_Off(CC : in out Cruise_Control);
  procedure Top_Gear_Engaged(CC : in out Cruise_Control);
  procedure Top_Gear_Disengaged(CC : in out Cruise_Control);
  procedure Brake_Engaged(CC : in out Cruise_Control);
  procedure Brake_Disengaged(CC : in out Cruise_Control);
private
  type Cruise_Control is record
    Engine_Active : Boolean := False;
    Top_Gear : Boolean := False;
    Braking : Boolean := False;
    Accelerating : Boolean := False;
    Cruising : Boolean := False;
    Throttle : access Throttle_Control;
    Speed : access Speedometer;
  end record;
end Cruise_Controller;
```

The package body is as follows:

```
with Cruise; use Cruise;
package body Cruise_Controller is
  procedure Initialize(CC : in out Cruise_Control) is
```

```
begin
  CC.Throttle := Cruise.Throttle_Controller'Access;
  CC.Speed := Cruise.Speedometer_Controller'Access;
end Initialize;

procedure Activate(CC : in out Cruise_Control) is
begin
  if CC.Engine_Active and CC.Top_Gear and not CC.Braking then
    CC.Cruising := True;
    Throttle_Control_System.Set_Cruise_Speed(
                  CC.Throttle.all, CC.Speed.Get_Current_Speed);
    Throttle_Control_System.Activate(CC.Throttle.all);
  end if;
end Activate;

procedure Deactivate(CC : in out Cruise_Control) is
begin
  if CC.Cruising then
    CC.Cruising := False;
    Throttle_Control_System.Deactivate(CC.Throttle.all);
  end if;
end Deactivate;

procedure Resume(CC : in out Cruise_Control) is
begin
  if CC.Top_Gear and not CC.Braking then
    CC.Cruising := True;
    Throttle_Control_System.Activate(CC.Throttle.all);
  end if;
end Resume;

procedure Start_Acceleration(CC : in out Cruise_Control) is
begin
  if CC.Engine_Active and CC.Top_Gear and not CC.Braking then
    CC.Accelerating := True;
    Throttle_Control_System.Activate(CC.Throttle.all);
    Throttle_Control_System.Accelerate(CC.Throttle.all);
  end if;
end Start_Acceleration;

procedure Stop_Acceleration(CC : in out Cruise_Control) is
begin
  if CC.Engine_Active and CC.Top_Gear and not CC.Braking and
     CC.Accelerating then
    CC.Accelerating := False;
    CC.Cruising := True;
    Throttle_Control_System.Set_Cruise_Speed(CC.Throttle.all,
         Speed_Control_System.Get_Current_Speed(CC.Speed.all));
  end if;
end Stop_Acceleration;

procedure Ignition_On(CC : in out Cruise_Control) is
begin
```

```
    CC.Engine_Active := True;
    CC.Braking := False;
    CC.Top_Gear := False;
    CC.Cruising := False;
  end Ignition_On;

  procedure Ignition_Off(CC : in out Cruise_Control) is
  begin
    CC.Engine_Active := False;
    CC.Braking := False;
    CC.Top_Gear := False;
    if CC.Cruising then
      CC.Cruising := False;
      Throttle_Control_System.Deactivate(CC.Throttle.all);
    end if;
    CC.Shaft_Simulator.Off;
  end Ignition_Off;

  procedure Top_Gear_Engaged(CC : in out Cruise_Control) is
  begin
    if CC.Engine_Active then CC.Top_Gear := True; end if;
  end Top_Gear_Engaged;

  procedure Top_Gear_Disengaged(CC : in out Cruise_Control) is
  begin
    if CC.Engine_Active then
      CC.Top_Gear := False;
      if CC.Cruising then
        CC.Cruising := False;
        CC.Throttle.Deactivate;
      end if;
    end if;
  end Top_Gear_Disengaged;

  procedure Brake_Engaged(CC : in out Cruise_Control) is
  begin
    if CC.Engine_Active then
      if CC.Cruising then
        CC.Cruising := False;
        CC.Throttle.Deactivate;
      end if;
      CC.Braking := True;
    end if;
  end Brake_Engaged;

  procedure Brake_Disengaged(CC : in out Cruise_Control) is
  begin
    if CC.Engine_Active then CC.Braking := False; end if;
  end Brake_Disengaged;

begin
  Initialize(Cruise.Cruise_Controller);
end Cruise_Controller;
```

16.7.4 Putting it all together

To configure the system requires the application code and the real-time utilities to be linked together. For example, consider the command lever. The following code is needed:

```
...
with Lever_Control_System; use Lever_Control_System;
with Cruise_Controller; use Cruise_Controller;
package Cruise is
  -- Assumes the following interrupt names
  --   Brake_Interrupt
  --   Command_Lever_Interrupt
  --   Engine_Interrupt
  --   Gear_Interrupt
  --   Shaft_Interrupt
  ...
  Lever_Controller : aliased Lever_Control;
  Lever : aliased Interrupt_Release(Lever_Controller'Access,
           Command_Lever_Interrupt);
  Lever_Task : Simple_Real_Time_Task(Lever_Controller'Access,
           Lever'Access, Priority'Last);

  Lever_Deadline : Time_Span := Milliseconds(70);
  Lever_MIT : Time_Span :=  Milliseconds(100);
  Lever_Budget : Time_Span := Milliseconds(2);
  Lever_Priority : Any_Priority := Interrupt_Priority'First + 4;
end Cruise;
```

The `Lever_Controller` object declares the state for the real-time task. The `Lever` object controls the release of the real-time task, it is passed an access to the task's state and the associated interrupt identifier. The task itself is created by the `Lever_Task` object declaration. A reference to the state of the task is passed along with a reference to the release mechanism. Finally, the real-time attributes of the command lever task need to be declared.

The speedometer software is similarly structured:

```
with Lever_Control_System; use Lever_Control_System;
with Speed_Control_System; use Speed_Control_System;
with Throttle_Control_System; use Throttle_Control_System;
with Cruise_Controller; use Cruise_Controller;
package Cruise is
  -- continuing from above

  Speedometer_Controller : aliased Speedometer;
  Speed : aliased Periodic_Release_With_Deadline_Miss(
      Speedometer_Controller'Access, Termination => False);
  Speedometer_Task : Simple_Real_Time_Task(
      Speedometer_Controller'Access, Speed'Access, Priority'Last);
  Speedometer_Deadline : Time_Span := Milliseconds(95);
  Speedometer_Period : Time_Span :=  Milliseconds(100);
  Speedometer_Budget : Time_Span := Milliseconds(1);
  Speedometer_Priority : Any_Priority := Priority'First + 6;
end Cruise;
```

Note here, that the release mechanism provides time-triggered periodic releases and detects deadline misses.

16.8 Summary

The chapter has shown how a variety of real-time programming abstractions can be programmed in Ada 2005. The goal has been to illustrate that the primitive mechanisms provided by the language allow utilities to be constructed (and potentially standardised) that can address common real-time problems. The chapter focuses on the provision of real-time task abstractions. The supporting packages allow the programmer

- to express whether the abstraction is periodic, sporadic or aperiodic;
- either to terminate the current release of the task in the event of a deadline miss or to simply inform the program that this event has occurred;
- either to terminate the current release of the task in the event of an execution-time overrun or to simply inform the program that this event has occurred;
- to express whether it is associated with an execution-time server that can limit the amount of CPU time it receives.

The common components of the utilities make use of the following Ada mechanisms:

- tasks to provide the basis of real-time tasks;
- protected types to interface between tasks and the environment;
- timing events to initiate releases of time-triggered activities;
- execution-time clocks to ensure the CPU resource is appropriately partitioned;

The use of some of the utilities has been illustrated by the development of a cruise control system.

17
Restrictions, metrics and the Ravenscar profile

The last few chapters have demonstrated the extensive set of facilities that Ada 2005 provides for the support of real-time programming. The expressive power of the full language is clearly more comprehensive than any other mainstream engineering language. There are, however, situations in which a restricted set of features are desirable. This chapter looks at ways in which certain restrictions can be identified in an Ada program. It then describes in detail the Ravenscar profile, which is a collection of restrictions aimed at applications that require very efficient implementations, or have high-integrity requirements, or both. This chapter also includes a discussion of the metrics identified in the Real-Time Systems Annex.

17.1 Restricted tasking and other language features

Where it is necessary to produce very efficient programs, it is useful to have run-time systems (kernels) that are tailored to the particular needs of the program actually executing. As this would be impossible to do in general, the Ada language defines a set of restrictions that a run-time system should recognise and 'reward' by giving more effective support. The following restrictions are identified by the pragma called `Restrictions`, and are checked and enforced before run-time. Note however, that there is no requirement on the run-time to tailor itself to the restrictions specified.

- **No_Task_Hierarchy**
  All (non-environment) tasks only depend directly on the environment task.
- **No_Nested_Finalization**
  Objects with controlled parts, and access types that designate such objects, are declared only at library level.
- **No_Abort_Statement**
  There are no abort statements.

- **No_Terminate_Alternatives**
  There are no select statements with terminate alternatives.
- **No_Task_Allocators**
  There are no allocators for task types or types containing task subcomponents.
- **No_Implicit_Heap_Allocation**
  There are no operations that implicitly require heap storage allocation to be performed by the implementation. For example, the concatenation of two strings usually requires space to be allocated on the heap to contain the result.
- **No_Dynamic_Priorities**
  There is no use of dynamic priorities.
- **No_Dynamic_Attachments**
  There are no calls to any of the operations defined in package `Interrupts`, e.g. `Attach_Handler`.
- **No_Local_Protected_Objects**
  Protected objects are only declared at the library level.
- **No_Local_Timing_Events**
  Timing events are only declared at the library level.
- **No_Protected_Type_Allocators**
  There are no allocators for protected types or types containing protected subcomponents.
- **No_Relative_Delay**
  There are no relative delay statements (i.e. **delay**).
- **No_Requeue_Statements**
  There are no requeue statements.
- **No_Select_Statements**
  There are no select statements.
- **No_Specific_Termination_Handlers**
  There are no calls to the specific handler routines in the task termination package.
- **Simple_Barriers**
  The boolean expression in an entry barrier is either a static boolean expression or a boolean component of the enclosing protected object (e.g. a simple boolean variable).

The following restrictions are also defined:

- **Max_Select_Alternatives**
  Specifies the maximum number of alternatives in a select statement.

- **Max_Task_Entries**
  Specifies the maximum number of entries per task. The maximum number of entries for each task type (including those with entry families) must be determinable at compile-time. A value of zero indicates that no rendezvous is possible.
- **Max_Protected_Entries**
  Specifies the maximum number of entries per protected type. The maximum number of entries for each protected type (including those with entry families) must be determinable at compile-time.

The following restrictions are checked at run-time.

- **No_Task_Termination**
  All tasks are non-terminating. It is implementation defined what happens if a task terminates – but any fall-back handler must be executed as the first task terminates.
- **Max_Storage_At_Blocking**
  Specifies the maximum portion (in storage elements) of a task's storage size that can be retained by a blocked task. If a check fails, `Storage_Error` is raised at the point where the respective construct is elaborated.
- **Max_Asynchronous_Select_Nesting**
  Specifies the maximum dynamic nesting level of asynchronous select statements. A value of zero prevents the use of any such statement. If a check fails, `Storage_Error` is raised as above.
- **Max_Tasks**
  Specifies the maximum number of tasks, excluding the environment task, that are allowed to exist over the lifetime of a partition. A zero value prevents tasks from being created. If a check fails, `Storage_Error` is raised as above.
- **Max_Entry_Queue_Length**
  This defines the maximum number of calls queued on an entry. Violation will cause `Program_Error` to be raised at the point of call.

So to restrict a program to have a flat task structure and not use the select statement would require:

```
pragma Restrictions(No_Task_Hierarchy, No_Select_Statements);
```

In addition to these specific restrictions a program may indicate that it does not wish to make use of a predefined language package by using the restriction identifier `No_Dependence`. So for example to state that the calendar package will not be used:

```
pragma Restrictions(No_Dependence => Ada.Calendar};
```

The following section will show how the above restrictions can be used together to define an execution *profile*.

17.2 The Ravenscar profile

Ada 2005 change: The Ravenscar profile was initially defined as a *de facto* standard associated with Ada 95. It is now fully incorporated into Ada 2005.

There is increasing recognition that the software components of critical real-time applications must be provably predictable. This is particularly so for a hard real-time system, in which the failure of a component of the system to meet its timing deadline can result in an unacceptable failure of the whole system. The choice of a suitable design and development method, in conjunction with supporting tools that enable the real-time performance of a system to be analysed and simulated, can lead to a high level of confidence that the final system meets its real-time constraints.

Traditional methods used for the design and development of complex applications, which concentrate primarily on functionality, are increasingly inadequate for hard real-time systems. This is because non-functional requirements such as dependability (e.g. safety and reliability), timeliness, memory usage, power consumption and dynamic change management are left until too late in the development cycle.

The traditional approach to formal verification and certification of critical real-time systems has been to dispense entirely with separate processes, each with their own independent thread of control, and to use a cyclic executive that calls a series of procedures in a fully deterministic manner. Such a system becomes easy to analyse, but is difficult to design for systems of more than moderate complexity, inflexible to change, and is not well suited to applications where sporadic activity may occur and where error recovery is important. Moreover, it can lead to poor software engineering if small procedures have to be artificially constructed to fit the cyclic schedule.

The use of Ada has proven to be of great value within high-integrity and real-time applications, albeit via language subsets of deterministic constructs, to ensure full analysability of the code. Such subsets have been defined for Ada 83 and Ada 95, but these have excluded tasking on the grounds of its non-determinism and inefficiency.† Advances in the area of schedulability analysis currently allow hard deadlines to be checked, even in the presence of a run-time system that enforces preemptive task scheduling based on multiple priorities. This has opened the way

† A program can easily exclude tasking features using the restrictions defined in the previous section.

for these tasking constructs to be used in high-integrity subsets whilst retaining the core elements of predictability and reliability.

The Ravenscar profile is a subset of the tasking model, restricted to meet the real-time community requirements for determinism, schedulability analysis and memory-boundedness, as well as being suitable for mapping to a small and efficient run-time system that supports task synchronisation and communication, and which could be certifiable to the highest integrity levels. The concurrency model promoted by the Ravenscar profile is consistent with the use of tools that allow the static properties of programs to be verified. Potential verification techniques include information flow analysis, schedulability analysis, execution-order analysis and model checking. These techniques allow analysis of a system to be performed throughout its development life cycle, thus avoiding the common problem of finding, only during system integration and testing, that the design fails to meet its non-functional requirements.

Important note: The Ravenscar profile is silent on the non-tasking (i.e. sequential) aspects of the language.

For example it does not dictate how exceptions should, or should not, be used. For any particular application, it is likely that constraints on the sequential part of the language will be required. These may be due to other forms of static analysis to be applied to the code, or to enable worst-case execution-time information to be derived for the sequential code.

The Ravenscar profile is identified by a new Ada 2005 pragma called `Profile`:

```
pragma Profile(Ravenscar);
```

A program may include other profiles, but only `Ravenscar` is defined by the language itself.† A profile is a collection of restrictions and other pragmas. But before the definition of the Ravenscar profile can be given, which concentrates on the negative aspects of the definition (i.e. what it does not contain), a list of those features that it does allow is given. Together these provide an adequate form of tasking that can be used even for software that needs to be verified to the very highest integrity levels. Ravenscar-compliant programs can include:

- Task types and objects, defined at the library level.
- Protected types and objects, defined at the library level.
- One entry per protected object with a maximum of one task queued at any time on that entry.
- Entry barriers but restricted to a single boolean variable (or a Boolean literal).

† The name 'Ravenscar' comes from a small Yorkshire village in the UK at which the 8th International Real-Time Ada Workshop (IRTAW) was held in April 1997. This workshop and others in the series of IRTAW produced the definition of the Ravenscar profile that has now been incorporated into the language definition.

- The `Atomic` and `Volatile` pragmas.
- Any number of **delay until** statements.
- Use of `Ceiling_Locking` policy and `FIFO_Within_Priorities` dispatching policy.
- The `E'Count` attribute for protected entries except within entry barriers.
- The `Ada.Task_Identification` package plus task attributes `T'Identity` and `E'Caller`.
- Synchronous task control.
- Task type and protected type discriminants.
- The `Ada.Real_Time` package.
- Protected procedures as statically bound interrupt handlers.
- Execution-time monitoring.
- Library-level timing events.
- Fall-back task termination handlers.

The profile is defined as follows:

```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy(Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions(
   No_Abort_Statements,
   No_Dynamic_Attachment,
   No_Dynamic_Priorities,
   No_Implicit_Heap_Allocations,
   No_Local_Protected_Objects,
   No_Local_Timing_Events,
   No_Protected_Type_Allocators,
   No_Relative_Delay,
   No_Requeue_Statements,
   No_Select_Statements,
   No_Specific_Termination_Handlers,
   No_Task_Allocators,
   No_Task_Hierarchy,
   No_Task_Termination,
   Simple_Barriers,
   Max_Entry_Queue_Length => 1,
   Max_Protected_Entries => 1,
   Max_Task_Entries => 0,
   No_Dependence => Ada.Asynchronous_Task_Control,
   No_Dependence => Ada.Calendar,
   No_Dependence => Ada.Execution_Time.Group_Budget,
   No_Dependence => Ada.Execution_Time.Timers,
   No_Dependence => Ada.Task_Attributes);
```

Most of these restrictions should be easy to understand. They come from a need to have a static program with a fixed set of tasks and protected objects. As communication via protected objects is amenable to analysis, the rendezvous and the consequential use of select statements are prohibited. Other restrictions are motivated by the need to have a small and efficient run-time support system – hence the use of abort statements and complex barriers is prevented. The protocol required for implementing general protected objects is significantly simplified by the restriction of one entry per object and one queued task per entry. Non-static features such as the dynamic attachment of interrupt handlers and task specific termination handlers are excluded, as is the use of package `Calendar` (the real-time clock package is the preferred time base to use). Note that two of the restrictions `No_Task_Termination` and `Max_Entry_Queue_Length => 1` result in run-time checks (see the definitions of these restrictions).

Important note: A point to emphasise about the Ravenscar profile is that it is not a substitute for the full tasking model. It has significantly less expressive power, and many applications needs cannot be programmed with this profile.

Later in this chapter, a series of examples will be given that can be implemented in the Ravenscar profile; this should convince the reader that the profile is useful, but it is a much simplified language model.

To complete the definition, the pragma `Detect_Blocking` needs to be defined. Within a protected operation it is illegal to block, for example to call an external procedure that has a delay statement within it. Although illegal it is not possible for the compiler to check that this cannot occur, so it becomes a run-time issue. The program is said to experience a bounded error that can result in a number of possible behaviours, one of which is for the exception `Program_Error` to be raised. What the pragma `Detect_Blocking` does is ensure that the error condition is recognised and `Program_Error` raised. Although this checking will add to the overheads, i.e. detract from real-time performance, it is deemed necessary for high-integrity applications – indeed this pragma is defined in the High Integrity Systems Annex (Annex H).

Before giving some examples of the use of the profile, another feature which is also defined in Annex H will be covered. This pragma is likely to be used in tandem with Ravenscar but it is not a required feature and hence is not part of the profile definition.

17.3 Partition elaboration control

One of the consequences of the Ravenscar restrictions is that most if not all of the application code resides in library units (typically packages). In standard Ada, program library units are elaborated in a somewhat non-deterministic order. The tasks they contain may thus become activated and runnable in a sequence that could undermine the integrity of the program. Ideally a programmer should be able to easily define two distinct phases of execution: *initialisation* when all library objects are elaborated, tasks are created but none have started activation, and *execution* in which the tasks become active. To allow a program to construct these phases, a pragma is defined:

```
pragma Partition_Elaboration_Policy (policy_identifier);
```

The policy can be either `Sequential` or `Concurrent`. If `Concurrent` is chosen, or if the pragma is omitted, then the normal language semantics apply. But if `Sequential` is specified then the following order is followed:

(1) The activation of all library-level tasks and the attachment of interrupt handlers are deferred until all library units are elaborated.
(2) The interrupt handlers are attached by the environment task.
(3) The environment task is suspended while the library-level tasks are activated.
(4) The environment task executes the main subprogram (if any) concurrently with these executing tasks.

Remember that the environmental task is the notional task that is master of all the library-level tasks and which actually executes the main procedure of the program. To prevent deadlocks occurring as a result of the sequential policy, the restriction `No_Task_Hierarchy` must also be specified with this policy. If any task fails during its activation or if the environmental task becomes blocked for any reason during the first two actions above then it is recommended that the implementation causes the program to be abandoned.

Warning: As the use of this pragma with the sequential policy clearly changes the language rules, the reference manual uses the following phrase to legalise its own inconsistencies: *Notwithstanding what this International Standard says elsewhere*. Fortunately this phase is not used extensively in the ARM.

17.4 Examples of the use of the Ravenscar profile

In this section a number of Ravenscar-compliant examples will be given. As emphasised throughout this book, many real-time tasks have one of two simple structures that relate to periodic or sporadic patterns of release. A simple periodic structure is already in the Ravenscar subset and hence requires no change:

```
task type Cyclic(Pri : System.Priority; Cycle_Time : Positive) is
  pragma Priority(Pri);
end Cyclic;
```

```
task body Cyclic is
  Next_Period : Ada.Real_Time.Time;
  Period : constant Ada.Real_Time.Time_Span :=
                    Ada.Real_Time.Microseconds(Cycle_Time);
  -- Other declarations

begin
  -- Initialisation code
  Next_Period := Ada.Real_Time.Clock + Period;
  loop
    delay until Next_Period;
    -- Non-suspending periodic response code
    -- May include calls to protected procedures
    Next_Period := Next_Period + Period;
  end loop;
end Cyclic;

-- now declare two or more task objects of this type
C1 : Cyclic(20,200); -- priority 20, period 200 microseconds
C2 : Cyclic(15,100);
```

This example includes a number of cyclic tasks that initially read the clock and then suspend for a ‘Period’ of time. It can, however, be useful for all such tasks to coordinate their start times so that they share a common epoch. This can help to enforce precedence relations across tasks. To achieve this, a protected object is used which reads the clock on creation and then makes this clock value available to all cyclic tasks.

```
protected Epoch is
  function Start_Time return Ada.Real_Time.Time;
private
  pragma Priority(System.Priority'Last);
  Start : Ada.Real_Time.Time := Ada.Real_Time.Clock;
end Epoch;

protected body Epoch is
  function Start_Time return Ada.Real_Time.Time is
  begin
    return Start;
  end Start_Time;
end Epoch;
```

Note, a protected object is not strictly needed as a shared variable appropriately initialised will suffice. A more robust scheme and one that only reads the epoch time once a task actually needs it is as follows:

```
protected Epoch is
  procedure Get_Start_Time(T : out Ada.Real_Time.Time);
private
  pragma Priority(System.Priority'Last);
  Start : Ada.Real_Time.Time;
  First : Boolean := True;
end Epoch;

protected body Epoch is
  procedure Get_Start_Time(T : out Ada.Real_Time.Time) is
  begin
    if First then
      First := False;
      Start := Ada.Real_Time.Clock;
    end if;
    T := Start;
  end Get_Start_Time;
end Epoch;
```

This leads to the following:

```
task type Cyclic(Pri : System.Priority; Cycle_Time : Positive) is
  pragma Priority(Pri);
end Cyclic;

task body Cyclic is
  Next_Period : Ada.Real_Time.Time;
  Period : constant Ada.Real_Time.Time_Span :=
                    Ada.Real_Time.Microseconds(Cycle_Time);
  -- Other declarations
begin
  -- Initialisation code
  Epoch.Get_Start_Time(Next_Period);
  Next_Period := Next_Period + Period;
  loop
    delay until Next_Period; -- wait until next period after epoch
    -- Non-suspending periodic response code
    -- May include calls to protected procedures
    Next_Period := Next_Period + Period;
  end loop;
end Cyclic;
```

The use of priorities and a shared epoch can be used to enforce precedence, between tasks with the same period, if the application can be restricted so that the tasks do not block during execution. An alternative scheme is to use an offset in time. Here, scheduling analysis is used to ensure that each task has completed before the next is released.

```
task type Cyclic(Pri : System.Priority;
                 Cycle_Time, Offset : Natural) is
  pragma Priority(Pri);
end Cyclic;

task body Cyclic is
  Next_Period : Ada.Real_Time.Time;
  Period : constant Ada.Real_Time.Time_Span :=
                    Ada.Real_Time.Microseconds(Cycle_Time);
  -- Other declarations
```

```
begin
  -- Initialisation code
  Next_Period := Epoch.Start_Time +
                 Ada.Real_Time.Microseconds(Offset);
  loop
    delay until Next_Period; -- wait until next period after offset
    -- Non-suspending periodic response code
    -- May include calls to protected procedures
    Next_Period := Next_Period + Period;
  end loop;
end Cyclic;

First : Cyclic(20,200,0); -- required to complete with deadline 70
Second : Cyclic(20,200,70);
```

To program sporadic tasks is again straightforward within the Ravenscar profile. Unlike the comprehensive code given in the previous chapter, here the application is not informed of deadlines misses etc. In the following a suspension object is used and a minimum separation between invocations is enforced. Remember, for scheduling analysis sporadic tasks must not execute 'too early'. The use of a suspension object rather than a protected object is sufficient for the required synchronisation but, of course, does not enable data to be passed to the sporadic task when it is released.

```
 -- A suspension object, SO, is declared in a visible library
 -- unit and is set to True in another (releasing) task

task type Sporadic_Task(Pri : System.Priority; MIT : Natural) is
  pragma Priority(Pri);
end Sporadic_Task;

task body Sporadic_Task is
  -- Declarations
  Minimum_Separation : constant Ada.Real_Time.Time_Span :=
                       Ada.Real_Time.Microseconds(MIT);
  Next : Ada.Real_Time.Time;
begin
  -- Initialisation code
  loop
    Ada.Synchronous_Task_Control.Suspend_Until_True(SO);
    Next := Ada.Real_Time.Clock + Minimum_Separation;
    -- Non-suspending code
    delay until Next; -- this ensure minimum temporal separation
  end loop;
end Sporadic_Task;
```

Note that in this simple approach, if the task is preempted between returning from the suspension object and reading the clock, the delay could be significantly more than the minimum.

The next example will show how a requirement that seems to be beyond the Ravenscar profile (the need for a protected object with two entries) can actually be accommodated quite easily. To illustrate the way a two-entry protected object can be transformed, consider the standard buffer with one task calling the buffer to extract an item and another task calling it to place items in the buffer. Usually both of these calls would be made via entries in a protected object as the extract item call must block if the buffer is empty, and the place item call must block if the buffer is full. To comply with the Ravenscar restriction of only one entry in any protected object, a protected object is used for mutual exclusion only and two suspension objects are introduced for the necessary conditional synchronisation.

```
package Buffer is
  procedure Place_Item(Item : Some_Type);
  procedure Extract_Item(Item : out Some_Type);
end Buffer;

package body Buffer is
  protected Buff is
    procedure Place(Item : in Some_Type; Success : out Boolean);
    procedure Extract(Item : out Some_Type; Success : out Boolean);
  private
    Buffer_Full : Boolean := False;
    Buffer_Empty : Boolean := True;
    -- other declarations for the buffer itself
  end Buff;

  Non_Full : Ada.Synchronous_Task_Control.Suspension_Object;
  Non_Empty : Ada.Synchronous_Task_Control.Suspension_Object;

  procedure Place_Item(Item : Some_Type) is
    OK : Boolean;
  begin
    Buff.Place(Item, OK);
    if not OK then
      Ada.Synchronous_Task_Control.Suspend_Until_True(Non_Full);
      Buff.Place(Item, OK); -- OK must be true
    end if;
    Ada.Synchronous_Task_Control.Set_True(Non_Empty);
  end Place_Item;

  procedure Extract_Item(Item : out Some_Type) is
    OK : Boolean;
  begin
    Buff.Extract(Item, OK);
    if not OK then
      Ada.Synchronous_Task_Control.Suspend_Until_True(Non_Empty);
      Buff.Extract(Item, OK); -- OK must be true
    end if;
    Ada.Synchronous_Task_Control.Set_True(Non_Full);
  end Extract_Item;
  protected body Buff is
    procedure Place(Item : in Some_Type; Success : out Boolean) is
    begin
      Success := not Buffer_Full;
      if not Buffer_Full then
        -- put Item into Buffer
        Buffer_Empty := False;
        -- set Buffer_Full if appropriate
      end if;
    end Place;

    procedure Extract(Item: out Some_Type; Success: out Boolean) is
    begin
      Success := not Buffer_Empty;
      if not Buffer_Empty then
        -- extract Item from Buffer
        Buffer_Full := False;
        -- set Buffer_Empty if appropriate
      end if;
    end Extract;
  end Buff;
end Buffer;
```

Now consider the situation of having, in full Ada, a need for two or more tasks to queue up on the same entry within a protected object. In Ravenscar-compliant programs the number of protected objects needed is increased. To illustrate, consider a requirement to release two tasks but only if both of them are ready to be released. So in full Ada this would be:

```
protected Controller is  -- Full Ada, not Ravenscar compliant
  pragma Priority(X);
  entry Wait_For_Release;
  -- called by two tasks
  procedure Release;  -- called by external task
private
  Released : Boolean := False;
end Controller;

protected body Controller is
  entry Wait_For_Release when Released is
  begin
    if Wait_For_Release'Count = 0 then
      Released := False;
    end if;
  end Wait_For_Release;

  procedure Release is
  begin
    Released := Wait_For_Release'Count = 2;
  end Release;
end Controller;
```

In a Ravenscar-compliant program this becomes:

```
protected Controller1 is
  pragma Priority(X);
  entry Wait_For_Release; -- called by one client task
  procedure Release;  -- called by Controller2
  function Called_In return Boolean;
private
  Released : Boolean := False;
end Controller1;

protected Controller2 is
  pragma Priority(X);
  entry Wait_For_Release; -- called by the other client task
  procedure Release; -- called by the external releasing task
private
  Released : Boolean := False;
end Controller2;

protected body Controller1 is
  entry Wait_For_Release when Released is
  begin
    Released := False;
  end Wait_For_Release;

  function Called_In is
  begin
    return Wait_For_Release'Count = 1;
  end Called_In;

  procedure Release is
  begin
    Released := True;
  end Release;
end Controller1;

protected body Controller2 is
  entry Wait_For_Release when Released is
  begin
    Released := False;
  end Wait_For_Release;

  procedure Release is
  begin
    if Wait_For_Release'Count = 1 and then
            Controller1.Called_In then
      Controller1.Release;
      Released := True;
    end if;
  end Release;
end Controller2;
```

Note that as the two protected objects have the same priority then their interactions will be atomic with respect to the other tasks involved (on a single processor).

One of the restrictions of Ravenscar is that inter-task interactions are never asynchronous in the sense of using an ATC to interrupt another task's execution. Neither is **abort** allowed. Hence tasks must check for the condition that may require them to change their mode of operation. So for example, if execution-time clocks are supported (they are not disallowed by the Ravenscar profile – although timers and group budgets are disallowed) and a maximum execution time per invocation is required then the following could be an appropriate structure for a cyclic task:

```
task type Cyclic(Pri : System.Priority;
                 Cycle_Time : Positive;
                 Bound : Positive) is
  pragma Priority(Pri);
end Cyclic;

task body Cyclic is
  Next_Period : Ada.Real_Time.Time;
  Period : constant Ada.Real_Time.Time_Span :=
                    Ada.Real_Time.Microseconds(Cycle_Time);
  Last_Reading : Ada.Execution_Time.CPU_Time;
  WCET : Time_Span := Ada.Real_Time.Microseconds(Bound);
  Timing_Failure : Boolean := False;
  -- Other declarations
begin
  -- Initialisation code
  Epoch.Get_Start_Time(Next_Period);
  Next_Period := Next_Period + Period;
  Last_Reading := Ada.Execution_Time.Clock;
  loop
    delay until Next_Period; -- wait until next period after epoch
    if Timing_Failure then
      -- failure in last interaction
      -- take some remedial action
      Timing_Failure := False;
    end if;
    -- Non-suspending periodic response code,
    -- perhaps including a loop:
    while <some_condition> loop
      -- code of application
      if Ada.Execution_Time.Clock - Last_Reading > WCET then
        Timing_Failure := True;
        exit; -- stops the work for this interaction
      end if;
    end loop;
    Next_Period := Next_Period + Period;
    Last_Reading := Ada.Execution_Time.Clock;
  end loop;
end Cyclic;
```

The test need not be for execution-time overrun but could be for any significant state change including time (e.g. deadline miss or mode change). The watchdog timer example in Section 15.2 is legal Ravenscar as long as the timing event is declared at the library level. So the task could check that a timeout event had not occurred.

The Ravenscar profile is an important aspect of Ada 2005. It was defined as an extension to Ada 95 and has already been supported by slimmed down run-time support systems and used in a number of application. The fact that it is now sanctioned by the language definition should add to its usage. To read more examples see the Ravenscar technical report listed at the end of this chapter. To see coverage of the scheduling analysis that accompanies Ravenscar see the Burns and Wellings book also referenced at the end of this chapter.

17.5 Metrics and optimisations

In addition to the features described so far in this and the previous chapters, the Real-Time Systems Annex defines a number of implementation requirements, documentation requirements, optimisations and metrics. The metrics allow the cost (in processor cycles) of the run-time system to be obtained. Examples of the defined metrics are the execution times of a delay statement which does not lead to a delay (e.g. **delay** 0.0), gaining access to an entry-less simple protected object, an abort statement that leads to the completion of one aborted task, a simple ATC statement and a `Set_Priority` call.

The timing features (that is, the real-time clock, execution-time clocks and the delay primitives) are all defined precisely. An implementation is required to document an upper bound on the size of each clock's tick, an upper bound on the size of every clock's jump, an upper bound on the drift rate of the real-time clock (when compared with an external clock that keeps perfect IAT – International Atomic Time) and upper bounds on the overrun of the delay and delay until operations. Metrics must also be provided for the cost of a call to a clock (either the real-time clock or an execution-time clock) and the cost of all the operations defined in the clock packages. It is recommended that `Calendar.Clock` and `Real_Time.Clock` be implemented as transformations of the same time source.

The Real-Time Systems Annex also defines a number of optimisations and constraints. There is a requirement for the abort action to be immediate on a single-processor system. This is a stronger statement than that in the main definition of the language (ARM Chapter 9) where the only requirement is for the aborted construct to complete no later than its next *abort completion point*. Of course, an abort action cannot be immediate if the task is currently executing within an *abort-deferred region*, such as a protected object. In real-time systems, it is not necessary for all things to be immediate. Their latency must, however, be bounded and under programmer control.

There are two optimisations explicitly noted in the ARM. First, an implementation should recognise an entry-less protected type and produce code that is consequently minimised (e.g. no code needed for entry evaluation). Second, the call of `Unchecked_Deallocation` on an access object referring to a terminated task should release all storage (memory) associated with the task – this includes run-time and heap storage. If an implementation satisfies this requirement then it is possible to construct a long-lived program that allocates and deallocates tasks without the storage leakage that would eventually lead to memory exhaustion and failure. The entry-less optimisation is one of a number that an implementation could recognise. One of the motivations of the Ravenscar profile is to bring a number of restrictions together so that a significantly simplified, and consequently more efficient, run-time system can be supported by implementations.

To undertake scheduling analysis of a complete application requires knowledge of all computational activities including interrupts. For those interrupts under direct user control this is feasible. But if the implementation blocks interrupts (for non-interruptible sections of the run-time system) then the Real-Time Systems Annex requires this block to be bounded in time and an upper bound on its duration to be documented.

Most of the metrics and documentation requirements defined above are primarily aimed at single-processor systems. For multiprocessor implementations, it is recognised that certain metrics may be difficult to obtain, and some constraints impossible to satisfy. For example, the immediacy of abort may not be possible if the task to be aborted is actually running on a different processor. In all cases, the particular issues concerning multiprocessor execution must be documented.

17.6 Summary

This chapter has concentrated on the restrictions and optimisations that a programmer can exploit and expect. Many combinations of restrictions are possible and, hence, the need to collect common sets of restrictions into language profiles. The main, indeed only, profile defined for real-time systems is Ravenscar. This enables high-integrity, high-performance, concurrent real-time systems to be constructed and, where necessary, analysed and certified. Linked to the Ravenscar profile is the need to control partition elaboration so that it is easy to separate the initialisation of a system from the execution phase in which tasks are active and interrupts are enabled.

Outside the Ravenscar profile other restrictions are useful. For example the run-time system may be able to make use of the fact that the maximum number of tasks is less than 16 (or 32) to exploit bit-wise operations to move any number of tasks from the delay queue to a representation of the ready queues as a single processor instruction.

To fully analyse a complete system for its temporal behaviour requires data about the run-time system. This must either be measured in a systematic way or be included in the documentation provided with the system – or both, with measurements being taken to validate the documentation. To help implementors identify the data they should provide, a number of metrics are defined in the Real-Time System Annex. The quality of the data provided is, however, beyond the scope of the language definition; programmers must choose their implementations carefully.

17.7 Further reading

A. Burns and A.J. Wellings, *Real-Time Systems and Programming Languages,*, 3rd Edition, Addison-Wesley, 2001.

Information Technology – Programming languages – Guide for the use of the Ada programming language in high integrity systems, ISO/IEC Technical Report, 15942, 2000.

Information Technology – Programming languages – Guide for the use of the Ada Ravenscar Profile in high integrity systems, ISO/IEC Technical Report, 24718, 2005.

18
Conclusion

Computer languages, like natural languages, evolve. For regulated languages, changes are consolidated into distinct versions at well-defined points in time. For Ada, the main versions have been Ada 83, Ada 95 and now Ada 2005. Whereas Ada 95 was a significantly different language from its ancestor, Ada 2005 is more an upgrade. It brings Ada up to date with respect to current practice in other languages, operating systems and theory – especially in the real-time domain.

Although Ada is a general purpose programming language, much of its development has been driven by the requirements of particular application areas. Specifically, the needs of high-integrity and safety-critical systems, real-time systems, embedded systems and large complex long-life systems. To support this wide range of applications, Ada has a large number of language features and primitives that can be grouped into the following:

- strong typing with safe pointer operations,
- object-oriented programming support via tagged types and interfaces,
- hierarchical libraries and separate compilation,
- exception handling,
- annexes to give support to particular application domains,
- low-level programming features that enable device drivers and interrupt handlers to be written,
- an expressive concurrency model and
- an extensive collection of entities that support real-time systems programming.

This book has concentrated on the last three items in this list to provide a comprehensive description of real-time and concurrent programming. These are two of the unquestionable strengths of the Ada language.

18.1 Support for concurrency

Over the first half of the book the range of concurrency features of Ada were described. Briefly, these amount to:

- tasks being first class language entities that are dynamically created and that can have arbitrarily complex parent/child relations,
- a synchronous communications paradigm incorporating the use of the rendezvous and the select statement,
- an asynchronous communications paradigm incorporating the use of the protected objects and
- synchronisation via the use of guards/barriers, enhanced in expressive power by the provisions of the requeue statement.

The combined use of these features allows a wide range of concurrency patterns to be programmed, a number of which were illustrated in Chapter 11.

18.2 Support for real-time

Linked closely to the concurrency features are a collection of primitives and standard libraries that makes Ada the most powerful programming language in the real-time domain. Again, a simple list is adequate to provide a summary of these features:

- fixed priority dispatching with full support for the priority ceiling protocol (but allowing priorities and ceiling to be modified at run-time),
- EDF, round robin and other dispatching policies that can be used either in isolation or together in a controlled way,
- execution-time monitoring and control, including the use of execution-time budgets shared between groups of tasks,
- delay statements and timing events to enable computation to be synchronised with external time and
- profiles by which a reduced set of language features and primitives can be defined, one of which is the Ravenscar profile.

As with the concurrency features, common application paradigms, such as periodic tasks with deadline overrun detection, can be constructed and standardised. A number of these are included in Chapter 16.

18.3 New to Ada 2005

If one is familiar with Ada 95 it may be useful to see, in one place, what is actually new in Ada 2005. From the context of concurrent and real-time programming, the following are the main additions to the language.

- Synchronized, protected and task interfaces.
- EDF, round robin, non-preemption and other dispatching facilities.
- CPU-time monitoring
- Budget control for individual tasks and groups of tasks
- Timing events – events executed when a defined time is reached.
- Task termination events – events executed when a task terminates.
- Dynamic ceiling priorities for protected objects.
- The Ravenscar profile for high integrity applications.

18.4 Outstanding issues and the future

Ada 2005 is an updated and comprehensive language; however, it inevitably has some limitations and outstanding concerns. A number of these have been noted in previous chapters. In particular, there is a need for secondary standards to be developed in the areas of concurrency and real-time utilities, for both the full language and the Ravenscar profile.

A more major issue is Ada's support for programming multiprocessor and distributed systems. Ada 2005 has added little to what was available in Ada 95. And yet distribution platforms and the construction of distributed systems has been the focus of considerable attention over the last decade. Although there has been some experience with Ada's distribution facilities little new has emerged by way of proposed language changes or extensions. Perhaps one reason for this is the view that middleware issues are not the province of the programming language; and hence Ada's designers have made the sensible decision not to provide comprehensive support in this area. The next decade may help clarify the role of programming languages in the construction of software for non-uniprocessor hardware such as distributed systems, multiprocessor systems and SoCs (Systems on Chips).

The passage of time will inevitably impact on other aspects of Ada. The language will continue to grow and new application requirements will test the expressive power and usability of Ada-based technologies. Some people will continue to view Ada as a general purpose programming language in competition with C++ and Java. Others will see it as the 'technology of choice' in the high-integrity, safety-critical and real-time domains.

References

Baker, 1991 – T.P. Baker, ‘A Stack-Based Resource Allocation Policy for Real-Time Processes’, *Proceedings of the IEEE Real-Time Systems Symposium*, 191–200 (1991).

Ben-Ari, 1982 – M. Ben-Ari, *Principles of Concurrent Programming*, Prentice-Hall (1982).

Bloom, 1979 – T. Bloom, ‘Evaluating Synchronisation Mechanisms’, *Proceedings of the Seventh ACM Symposium on Operating System Principles*, Pacific Grove, 24–32 (December 1979).

Campbell and Habermann, 1974 – R.H. Campbell and A.N. Habermann, ‘The Specification of Process Synchronisation by Path Expressions’, *Lecture Notes in Computer Science*, **16**, 89–102, Springer-Verlag (1974).

Dijkstra, 1968 – E.W. Dijkstra, ‘Cooperating Sequential’ Processes, in *Programming Languages*, ed. F. Genuys, Academic Press, 43–112 (1968).

Dijkstra, 1975 – E.W. Dijkstra, ‘Guarded Commands, Nondeterminacy, and Formal Derivation of Programs’, *CACM*, **18**(8), 453–7 (August 1975).

Francez and Pnueli, 1978 – N. Francez and A. Pnueli, ‘A Proof Method for Cyclic Programs’, *ACTA*, **9**, 133–57 (1978).

Hoare, 1974 – C.A.R. Hoare, ‘Monitors – An Operating System Structuring Concept’, *CACM*, **17**(10), 549–57 (October 1974).

Hoare, 1979 – C.A.R. Hoare, *Subsetting of Ada*, Draft Document (June 1979).

Simpson, 1990 – H. Simpson, ‘Four-Slot Fully Asynchronous Communication Mechanism’, *IEE Proceedings*, **137** (Pt.E.1), 17–30 (January 1990).

Index