Safe and Secure Software

An Invitation to **Ada 2005**

10

Safe Concurrency

Courtesy of AdaCore
The GNAT Pro Company

John Barnes

In real life many activities happen in parallel. Human beings do thing in parallel with considerable ease. Females seem to do this better than males – perhaps because they have to rock the baby while cooking the food and keeping the tiger out of the cave. The male typically just concentrates on one thing at a time such as catching that rabbit for dinner – or trying to find a bigger cave or perhaps even inventing a wheel.

Computers traditionally only do one thing at a time, and the operating system makes it look as if several things are going on in parallel. This is not quite so true these days, since many computers do truly have multiple processors but it still does apply to the vast majority of small computers including those used in process control.

Operating systems and tasks

Operating systems vary enormously in the amount of parallel activity that they permit. Operating systems supporting POSIX provide the programmer with multiple threads of control. These various threads of control can flow through the program quite independently and so support parallel activities.

On some hardware there will only be one processor, which will be allocated to the different threads according to some scheduling algorithm. One approach is simply to give the processor to each thread in turn for a small amount of time; more sophisticated approaches are to use priorities or deadlines to ensure that the processor is used effectively.

Some hardware might have multiple processors in which case several threads can truly be active in parallel. Again a scheduler will allocate the processors in a hopefully effective way to the active threads of control.

In a programming language the parallel activities are generally called *threads* or *tasks*. Here we will use the latter which is the Ada term. Languages take very different approaches to tasking. Some languages have intrinsic facilities for tasking built into the language itself. Others provide simple access to the underlying primitives of the operating system. Yet others ignore the subject completely.

Ada and Java are languages with intrinsic tasking facilities. C and C++ have no built-in support for tasking, so programmers using these languages need to rely on third-party libraries and make direct calls to operating system services.

There are at least three advantages of having tasking within the language itself

- Built-in syntactic constructions make it much easier to write correct programs because the language can prevent a number of errors from

being made. It is essentially the old story about abstraction. By hiding low-level details certain errors are prevented.

- Portability is difficult if operating system facilities are used directly because they vary widely from system to system.
- General operating systems do not provide the range of timing and related facilities needed by many real-time applications.

The operations typically required in a tasking program are

- Tasks must be prevented from violating the integrity of data if several tasks need access to the data concurrently.
- Tasks need to communicate with each other in order to transfer data between them.
- Tasks need to be controlled in order to meet specific timing requirements.
- Tasks need to be scheduled in order to use resources efficiently and to meet their overall deadlines.

This chapter will briefly look at these topics and illustrate how Ada addresses them in a reliable manner. This is a design challenge, since programs with tasking are much harder to write correctly than ordinary sequential programs. But first we introduce the simple idea of an Ada task and the overall program structure.

An Ada program can have many tasks running in parallel. A task is written in two parts rather like a package. It has a specification which describes the interface it presents to other tasks and a body which contains the code saying what it actually does. In simple cases the specification simply names the task so we might have

```
task A;                          -- task specification

task body A is                   -- task body
begin
  ...                        -- statements saying what the task does
end A;
```

Sometimes it is convenient to have several similar tasks in which case we can introduce a task type

```
task type Worker;

task body Worker is ...
```

We can then declare several tasks by declaring objects in the usual way

```
Tom, Dick, Harry: Worker;
```

This creates three tasks called Tom, Dick and Harry. We can also declare arrays of tasks and have task components inside records and so on. Tasks can be declared wherever other objects can be declared such as in a package or in a subprogram or even within another task. Not surprisingly, task types are limited types, since assigning one task to another is not a meaningful operation.

The main subprogram of a complete program is invoked by the so-called environment task and it is this environment task that elaborates library packages, as described in the chapter on Safe Startup. An overall program with library packages A, B and C and main subprogram Main can therefore be thought of as

```
task Environment_Task;

task body Environment_Task is
  ...              -- declarations of library packages A, B, C
  ...              -- and main subprogram Main
begin
  ...              -- call of main subprogram Main
end;
```

A task becomes active simply by being declared. It finishes by reaching the end of the task body. An important rule is that a local task declared within a subprogram or another task must finish before the enclosing unit can itself be left and the enclosing unit will be suspended until the local task terminates. This rule prevents dangling references to data that no longer exists.

Protected objects

Suppose that the three tasks Tom, Dick and Harry are using a stack as some sort of temporary storage device. From time to time one of them pushes an item onto the stack and from time to time one of them (perhaps the same one, perhaps a different one) pops an item off the stack.

The three tasks run in parallel and the runtime system gives the processor to each in turn according to some algorithm. Perhaps they each get 10 ms in turn.

Suppose the stack they are using is as declared in the chapter on Safe Architecture. Suppose that Harry is calling Push when his time slot expires and control then passes to Tom who calls Pop. To be precise, suppose Harry loses the processor just after he has executed the statement to increment Top in

```
procedure Push(X: Float) is
begin
  Top := Top + 1;              -- Harry loses processor just after this
  A(Top) := X;
end Push;
```

At this point Top has been incremented but the new value X has not been assigned to the component of the array. When Tom calls Pop, he gets the old and possibly meaningless value in the array component that was about to be overwritten by the new value. When Harry gets the processor back (and assuming no other stack activity occurs meanwhile) he will write the value X into a component of the array that is a part of the stack that is not in use. In other words the value X is lost.

A worse situation can occur if the processor is switched part way through a statement. Thus Harry might lose the processor just after he has picked up Top into a register but before he replaces Top with the new value. Suppose Dick now comes along and also does a Push thereby adding 1 to the old value of Top. When Harry resumes he will replace the value that Dick computed by the same value. In other words the two calls of Push add just 1 to Top rather than 2 as expected.

This unwanted behavior is overcome in Ada by using a protected object for the stack. We write

```
protected Stack is
   procedure Clear;
   procedure Push(X: in Float);
   procedure Pop(X: out Float);
private
   Max: constant := 100;
   Top: Integer range 0 .. Max := 0;
   A: array (1 .. Max) of Float;
end Stack;

protected body Stack is

   procedure Clear is
   begin
      Top := 0;
   end Clear;

   procedure Push(X: in Float) is
   begin
      Top := Top + 1;
      A(Top) := X;
   end Push;

   procedure Pop(X: out Float) is
   begin
      X := A(Top);
      Top := Top – 1;
   end Pop;

end Stack;
```

Note that **package** has been changed to **protected**, the data which was in the body now appears in the private part of this new construct, and for reasons explained below the function Pop has been changed into a procedure Pop.

The three procedures Clear, Push and Pop are called *protected operations* and are invoked in the same way as procedures. Their behavior is that only one task can access the operations of the object at a time. If a task such as Tom attempts to call the procedure Pop while Harry is executing Push then Tom is forced to wait until Harry returns from Push. This is all done automatically with no effort on the part of the programmer. So any inconsistency problems are avoided.

Behind the scenes the protected object has a lock, and a task attempting to access an operation of the object has to acquire the lock first. If another task already has the lock then the first one has to wait until that other task has finished with the protected operation of the object that it was using and so relinquishes the lock.

We can modify this example to show how we might cope with an attempt to push an item on the stack when it is full. In the package formulation this would raise Constraint_Error on the attempt to assign the value Max+1 to Top. As it is written the same thing would happen and the lock would be automatically relinquished, because the exception terminates the call of the protected procedure.

But we can do much better. We can modify the protected object to use barriers as follows

```
protected Stack is
   procedure Clear;
   entry Push(X: in Float);
   entry Pop(X: out Float);
private
  Max: constant := 100;
   Top: Integer range 0 .. Max := 0;
   A: array (1 .. Max) of Float;
end Stack;

protected body Stack is

   procedure Clear is
   begin
      Top := 0;
   end Clear;

   entry Push(X: in Float) when Top < Max is
   begin
      Top := Top + 1;
```

```
    A(Top) := X;
  end Push;

  entry Pop(X: out Float) when Top > 0 is
  begin
    X := A(Top);
    Top := Top – 1;
  end Pop;

end Stack;
```

The operations Push and Pop are now *entries* rather than procedures, and they have Boolean barrier expressions such as Top < Max. The effect of a barrier is to prevent the body of the entry from being executed if the barrier is False. Note that his does not prevent the entry from being called. All that happens is that the calling task is suspended until the barrier becomes True. So if Harry tries to call Push when the stack is full then he has to wait until some other task (Tom or Dick) calls Pop and removes the top item. Harry will then automatically proceed. The user does not have to program anything special.

Note that entries, like protected procedures, are also called in the same way as normal procedures, thus

```
Stack.Push(Z);
```

In summary, the protected object mechanism provided by Ada gives a structured mechanism for arranging mutually-exclusive access to a shared data object. A protected object declares its protected operations (procedures, functions, or entries) in the visible part of its specification, and the protected components in its private part. The body of the protected object contains the implementation of the protected operations. A protected procedure and a protected entry have "read/write" access to the protected components – that is, they can reference and/or assign to them – whereas a protected function only has read access. This restriction enables an optimization whereby multiple tasks may simultaneously read a protected object (through protected function calls) but only one task at a time is allowed to write to it. (This is sometimes called "Concurrent Read, Exclusive Write".) The prohibition against protected functions assigning to protected components is why we had to express Pop as a procedure rather than a function in the first protected object version of Stack above.

Note also that, just as we can declare a task type as a template for task objects, we can likewise declare a protected type as a template for protected objects. And like task types, protected types are limited.

It is instructive to consider how we might program this example using lower level primitives. The historic basic primitives are the operations *P (acquire)* and *V (release)* acting on objects called semaphores. The effect of P(sem) is to acquire the lock associated with sem, if the lock is available, and otherwise to

suspend the calling task on a queue associated with sem. The effect of V(sem) is to release the lock associated with sem and to awaken one of the tasks (if any) suspended on the queue of sem.

The idea is that we put pairs of calls of *P* and *V* around the operations for which we wish to ensure mutually exclusive access. Thus, using the same Ada syntax, Push would become

```
procedure Push(X: in Float) is
begin
  P(Stack_Lock);              -- secure the lock
  Top := Top + 1;
  A(Top) := X;
  V(Stack_Lock);              -- release the lock
end Push;
```

with similar pairs of calls around the body of Clear and Pop. This is essentially a Do-It-Yourself operation or assembly type coding for tasking. The opportunities for errors are many

- We might omit one of a *P* and *V* pair thus creating an imbalance.
- We might forget them altogether around one group of statements that should be protected.
- We might use the wrong semaphore name.
- We might inadvertently bypass a closing *V*.

The last problem would arise if, in the model without barriers, Push was called when the stack was full. This causes Constraint_Error to be raised. If we omit to provide a local exception handler to call *V* then the system will be permanently locked.

None of these difficulties can arise when using Ada protected objects because all this low-level mechanism is done automatically. Although, with care, semaphores can be used successfully in simple situations, it is very difficult to use them correctly in more complicated situations such as the example with barriers. Not only is it difficult to program correctly with semaphores but it is extremely difficult to prove that a program is correct.

Those familiar with Java will appreciate that the mechanisms of synchronized operations and wait/notify are rather low-level and error-prone. The programmer must be aware of the details of thread notification, which are handled automatically by Ada protected objects.

The rendezvous

The other important communication requirement between tasks is for one task to convey information (data) to another. This is done in Ada with a mechanism known as a rendezvous. The two tasks that communicate have a client–server relationship. The client that requests some service needs to know the identity of the server task, but the server task who provides it will accept a request from any client.

The general pattern of the server is

```
task Server is
  entry Some_Service(Formal: in out Data);
end;

task body Server is
begin
  ...
  accept Some_Service(Formal: in out Data) is
    ...              -- statements providing the service
  end Some_Service;
  ...
end Server;
```

The specification of the server indicates that it has an entry Some_Service. This is called by a client task in the same way as calling an entry of a protected object. The difference is that the code to be obeyed is given by an accept statement and that is only executed when the server task reaches the accept statement. Until that happens the calling task is suspended. When the server reaches the accept statement, it executes it using any parameters supplied by the client. The client remains suspended until the accept statement is finished and after any **out** or **in out** parameters have been updated.

The body of a client might look like

```
task body Client is
  Actual: Data;
begin
  ...
  Server.Some_Service(Actual);
  ...
end Client;
```

Each entry has an associated queue. If a task calls an entry of a server and the server is not waiting at an accept statement for that entry, then the caller is queued. On the other hand, if the server reaches an accept statement and there are no tasks waiting on the associated entry queue, then the server is suspended. An accept statement can appear anywhere, for example within a branch of a

conditional (if) statement, or within a loop, and so the mechanism is very flexible.

The rendezvous is a high level abstract mechanism (like the protected object) and as such is relatively easy to use correctly. The corresponding queuing mechanisms programmed at a low level are hard to write correctly.

Here is an example of how the rendezvous can be used to enable a service to be provided without the client waiting. The idea is that the client gives the server an entry to be called when a job is done. First we declare a mailbox type

```
task type Mailbox is
  entry Deposit(X: in Item);
  entry Collect(X: out Item);
end;

task body Mailbox is
  Local: Item;
begin
  accept Deposit(X: in Item) do
    Local := X;
  end;
  accept Collect(X: out Item) do
    X := Local;
  end;
end Mailbox;
```

A task of this type acts as a simple mailbox. An item can be deposited and collected later. The client passes the identity of a mailbox to the server so that the server can deposit the item in the mailbox from which the user can collect it later. We need an access type

```
type Mailbox_Ref is access Mailbox;
```

The tasks Server and Client now take the following form

```
task Server is
  entry Request(Ref: Mailbox_Ref; X: Item);
end;

task body Server is
  Reply: Mailbox_Ref;
  Job: Item;
begin
  loop
    accept Request(Ref: Mailbox_Ref; X: Item) do
      Reply := Ref;
      Job := X;
    end;
```

```
      ...                          -- work on job
      Reply.Deposit(Job);
    end loop;
  end Server;

  task Client;

  task body Client is
    My_Box: Mailbox_Ref := new Mailbox;      -- create mailbox task
    My_Item: Item;
  begin
    Server.Request(My_Box, My_Item);
    ...                          -- do something whilst waiting
    My_Box.Collect(My_Item);
  end Client;
```

In practice the client might poll the mailbox from time to time to see if the item is ready. This is easily done using a conditional entry call which takes the form

```
select
  My_Box.Collect(My_Item);
  -- item collected successfully
else
  -- not ready yet
end select;
```

It is important to realize that the mailbox agent task serves several purposes. It decouples the deposit and collect operations so that the server can get on with the next job. Moreover, it means that the server need know nothing about the client; calling the client directly would require the client to be of a particular task type and this would be most impractical. The mailbox agent task enables us to factor out the only property required of the client, namely the existence of the entry Deposit.

Restrictions

The pragma Restrictions which can be used to ensure that we do not use certain features of the language in a particular program was mentioned in the chapters on Safe Object-Oriented Programming and Safe Memory Management.

Many of the restrictions in Ada 2005 relate to tasking. The tasking features in Ada are very comprehensive and provide a whole range of facilities necessary to meet the programming needs of a variety of real-time applications. But some applications are quite simple and do not need many of these facilities. Here are some samples of the sort of restrictions that can be applied.

```
No_Task_Hierarchy
No_Task_Termination
Max_Entry_Queue_Length => n
```

The restriction No_Task_Hierarchy prevents tasks from being declared inside other tasks or inside subprograms – all tasks are therefore inside library-level packages. No_Task_Termination means that all tasks run for ever – this is common in many control applications where each task essentially has an endless loop doing some repetitive action. And the restriction on entry queues places a limit on the number of tasks that can be queued on a single entry at any time.

The advantage of giving appropriate restrictions are twofold

- It might enable a somewhat simpler runtime system to be used. This could be smaller and faster and thus more appropriate for some time- and space-critical embedded applications.
- It might enable various properties of the application to be proved correct, concerning matters such as determinism, absence of deadlock, and ability to meet deadlines. This might be vital for certain safety-critical applications.

There are many other tasking restrictions and most of these concern tasking facilities that we have not described.

Ravenscar

A particularly important group of restrictions is imposed by the Ravenscar profile. In order to ensure that a program conforms to this profile we write

```
pragma Profile(Ravenscar);
```

in the program. Use of any of the excluded features (summarized below) would then cause a compile-time error.

The key purpose of the Ravenscar profile is to restrict the use of tasking facilities so that the effect of the program is predictable. (The profile was defined by the International Real-Time Ada Workshops which met twice at the remote village of Ravenscar on the coast of Yorkshire in North-East England.)

The profile is simply defined to be equivalent to a number of restrictions plus a few other related pragmas concerning matters such as scheduling. The restrictions include those mentioned earlier so there are no task hierarchies, all tasks run for ever, and entry queues have a limit size of one (that is, there can be only one task blocked at a time on a given entry).

The combined effect of the restrictions is that it is possible to make statements about the ability of a particular program to meet stringent requirements for the purposes of certification.

No other programming language offers the reliability of Ada as constrained by the Ravenscar profile. A description of the principles and use of the profile in high integrity systems will be found in an ISO/IEC Technical Report [3].

Timing and scheduling

No survey of Ada tasking, however brief, would be complete without a few words about timing and scheduling.

There are statements to enable a program to be synchronized with a clock. We can delay a program for a specific amount of time (this is referred to as a *relative delay*) or until a specific time thus

```
delay 2*Minutes;
delay until Next_Time;
```

assuming suitable declarations for Minutes and for Next_Time. Small relative delays might be useful for interactive use, whereas a delay until a particular time can be used to program periodic events. Time itself can be measured either by a real-time clock (which is guaranteed to have a certain accuracy) or by the local wall clock which might be subjected to changes such as occur because of Daylight Savings. In Ada, it is even possible to take account of time zones and leap seconds.

Ada also provides a number of standard timers whose expiry can be used to trigger actions defined by a protected procedure (a handler). There are three kinds of timers, one enables the monitoring of the CPU time used by an individual task, one concerns the CPU budget for a group of tasks, and the third concerns time as measured by the real-time clock. The handler is attached to a timing event by a call of a procedure such as Set_Handler.

This is illustrated by the following amusing example concerning the boiling of an egg. We declare a protected object Egg thus

```
protected Egg is
  procedure Boil(For_Time: in Time_Span);
private
  procedure Is_Done(Event: in out Timing_Event);
  Egg_Done: Timing_Event;
end Egg;

protected body Egg is
```

```
    procedure Boil(For_Time: in Time_Span) is
    begin
      Put_Egg_In_Water;
      Set_Handler(Egg_Done, For_Time, Is_Done'Access);
    end Boil;

    procedure Is_Done(Event: in out Timing_Event) is
    begin
      Ring_The_Pinger;
    end Is_Done;

  end Egg;
```

The consumer can then write

```
Egg.Boil(Minutes(4));
-- now read newspaper whilst waiting for egg
```

and the pinger will ring when the egg is ready.

A number of different scheduling policies are provided in Ada 2005. These can be applied to all tasks in a program or just to those in certain priority ranges by the use of pragmas. The policies are

FIFO_Within_Priorities – Within each priority level to which it applies tasks are dealt with on a first-in–first-out basis. Moreover, a task may preempt a task of a lower priority.

Non_Preemptive_FIFO_Within_Priorities – Within each priority level to which it applies tasks run to completion or until they are blocked or execute a delay statement. A task cannot be preempted by one of higher priority. This sort of policy is widely used in high integrity applications.

Round_Robin_Within_Priorities – Within each priority level to which it applies tasks are timesliced with an interval that can be specified. This is a very traditional policy widely used since the earliest days of concurrent programming.

EDF_Across_Priorities – This provides Earliest Deadline First dispatching. The general idea is that within a range of priority levels, each task has a deadline and that with the earliest deadline is processed. This is a new policy and has mathematically provable advantages with respect to processor utilization.

Ada also has comprehensive facilities concerning the setting and changing of task priorities and the so-called ceiling priorities of protected objects. These avoid problems of priority inversion as described in [4].

North American Headquarters
104 Fifth Avenue, 15th floor
New York, NY 10011-6901, USA
tel +1 212 620 7300
fax +1 212 807 0162
sales@adacore.com
www.adacore.com

European Headquarters
46 rue d'Amsterdam
75009 Paris, France
tel +33 1 49 70 67 16
fax +33 1 49 70 05 52
sales@adacore.com
www.adacore.com