Chapter 18: Concurrent Units in Ada

The Ada programming language provides an interesting case study both for the way it was designed and for the design itself. One of Ada's strongest features is the way it handles concurrency. In this chapter we examine concurrency in Ada in some detail.

18.1 Overview of Ada

18.1.1 History

In the mid 1970s, the United States Department of Defense was concerned over the proliferation of programming languages that were being used under Department of Defense contracts and by the lack of standardization found in those languages. In addition, none of these languages were really suitable for implementing so-called embedded systems, systems that run on hardware that is embedded within some other device or piece of equipment. For these reasons, the Department of Defense formed an initiative to establish suitable programming language requirements and evaluate existing or proposed languages against those requirements.

In 1977, a version of the requirements document was released along with a request for proposals for a language design that would meet those requirements. After two years, the final design was chosen, a design from a team led by Jean Ichbiah from Cii Honeywell/Bull in France. The language was named Ada, for Augusta Ada Byron, the daughter of Lord Byron. She was a colleague of Charles Babbage during his work on the Analytic Engine and wrote several programs for that device, and hence is considered by some to be the first computer programmer.

The design team completed their design of the language in 1980 and it received official approval as a standard by the American National Standards Institute in 1983. In order to enforce this standard, the Department of Defense established a compiler validation process to carry out an extensive series of tests on any compiler that was to bear the name “Ada.”

By 1988 it became obvious that Ada had a number of shortcomings, especially its lack of support for object-oriented programming. Therefore, a second initiative was begun called the Ada 9X project. After another lengthy process, this project resulted in the definition of a new standard, called Ada 95.

18.1.2 Philosophy and Characteristics

Although Ada was originally designed for embedded systems, it is a very powerful general-purpose programming language. The syntax of the language was based on that of Pascal. Its philosophy is to support good program design practices for large software development projects. It does this through strong typing and excellent data abstraction facilities. Data abstraction is implemented by packages, which encapsulate the definition of data and procedures and implement information hiding. Packages also may be defined as generics, allowing parameterized data abstractions. The Ada 95 revision provided additional features to support inheritance and polymorphism to these data abstraction features.

In support of the design goal of facilitating embedded systems, Ada also provides extensive exception handling features. It also provides comprehensive facilities to specify concurrent execution. These facilities are the subject of this chapter and will be examined in some detail.
The Ada programming language has many adherents who point out that it is a language that not only encourages, but practically requires good program design practices. It has been quite successfully used in the development of many large software projects. It has, however, failed to become a popular programming language for a variety of reasons. In its early stages, its large size was a drawback, delaying the availability of good, affordable, and efficient compilers for several years. And while its standardization is certainly a major asset in terms of reuse and reliable software development, it has also proved to be a liability in the sense that the language has not been able to evolve as programming technology has changed. For this reason, Ada was unable to adapt quickly enough to the object-oriented paradigm and by the time Ada 95 became available, other languages such as C++ had already established themselves as the object-oriented languages of choice.

18.2 Concurrent Units in Ada

In Ada, concurrent units are called tasks. We will examine how tasks are defined and how they are called into execution. We will also discover that tasks in Ada can be used as a type.

18.2.1 Task Definition

In Ada tasks are defined in two parts: the specification and the body. The specification gives the name of the task and specifies all points in the task body where interprocess communication can occur. These communication points, called entries, are declared along with their formal parameters. Entries are described later.

The task body is like any other Ada program unit body, consisting of a declarative part for locally defined objects and an executable part containing a sequence of statements. A task completes execution when it reaches its last statement.

In Ada, the definition of a task determines a task type of the specified name that can then be bound to variables in the containing program unit. When the word type is omitted from the task specification, a single task with the assigned name is created, and no variable declaration is required. Figure 18.1 gives an example of two sections of Ada code, yielding the same declarative results: two tasks, each with three entries. Thus Ada permits both the procedure and the type model for task definition.

18.2.2 Task Invocation

The invocation of an Ada task is implicit. All task variables defined in the declarative part of a program unit (called the master) begin execution concurrently when the master begins execution. The master unit may be an Ada block, procedure, or function.
Figure 18.1 Examples of Different Task Declarations

(a) task type Bank_Task is
   entry deposit (acct: in accounts; amount: in real);
   entry withdrawl (acct: in accounts; amount: in real);
   entry balance (acct: in accounts; amount: out real);
end Bank_Task;

task body Bank_Task is
   -- task definition is skipped
end Bank_Task;

First_National, National_Bank_of_Detroit: Bank_Task;

(b) task First_National is
   entry deposit (acct: in accounts; amount: in real);
   entry withdrawl (acct: in accounts; amount: in real);
   entry balance (acct: in accounts; amount: out real);
end Bank_Task;

task body First_National is
   -- task definition is skipped
end Bank_Task;

task National_Bank_of_Detroit is
   entry deposit (acct: in accounts; amount: in real);
   entry withdrawl (acct: in accounts; amount: in real);
   entry balance (acct: in accounts; amount: out real);
end Bank_Task;

task body National_Bank_of_Detroit is
   -- task definition is skipped
end Bank_Task;

Ada tasks can be invoked explicitly, but only by relying on the explicit semantics of other language features and not of task invocation. Consider the code in Figure 18.1(a). The following declaration declares a pointer to an instance of Bank_Task:

Bank_of_Fargo: access Bank_Task;

At this point no task has been created for Bank_of_Fargo to reference, so no task instance yet exists. It now takes a statement such as:

Bank_of_Fargo := new Bank_Task;

to create a task and start its execution. Note that, while we appear to have explicitly controlled the invocation of a task, we have actually relied on the explicit semantics of pointer data creation to start the task.

Ada uses fork-join semantics for process invocation. The master will not terminate execution until all of its concurrent tasks have terminated. Therefore, if the master unit completes all its
statements, but some of its member tasks are still active, it waits until all of these tasks terminate before it proceeds with normal termination and returns to its invoking unit.

### 18.3 Data Sharing

The primary way of sharing data among tasks while avoiding a race condition is the protected type, an innovation of Ada 95. Types in Ada can be declared in a way similar to classes in Java, by specifying actions and data. The syntax but the result is the same. If a type is declared to be a protected type, that indicates that any functions and procedures defined by that type will execute in a mutually exclusive way. In other words, an object declared to be a protected type may not have more than one of its functions or procedures in execution simultaneously.

Figure 18.2 contains an illustration of a protected type definition in Ada. This protected type is Counter, which contains a single integer count that can be incremented or cleared by multiple tasks. Because Counter is a protected type, all of its procedures and functions are protected as well.

**Figure 18.2 Protected type Counter**

```ada
protected type Counter is
    procedure increment;
    function get return integer;
    procedure clear;
private
    count : integer := 0;
end IntBuffer;

protected body Counter is
    procedure increment is
        begin
            count := count + 1;
        end increment;

    function get return integer is
        begin
            return count;
        end get;

    procedure clear is
        begin
            count := 0;
        end clear;
end IntBuffer;

An object of protected type Counter can be declared by
aCounter : Counter;
Incrementing this aCounter is done by
aCounter.increment;
and clearing it by
aCounter.clear;
```
The current value of aCounter can be retrieved into an integer variable i by

   i := aCounter.get;

Notice that in Ada a parameterless function or procedure is called without empty parentheses.

When a protected procedure is called for an object and some other procedure or function for that object is already executing, the calling unit suspends execution until the procedure call can proceed. A protected function behaves a little differently. It will cause the calling unit to wait if another procedure for that object is executing, but will proceed in the case where no procedure but one or more function of the object are in execution. This is allowed because functions are not permitted to change their receiving object and since they only access the object, there is no potential conflict if more than one function is in execution simultaneously.

In the case of aCounter, increment and clear cannot be executed if another procedure or function of aCounter is already in execution. A call to function get cannot proceed if one of the procedures, increment or clear, is already in execution, but can be called if other instances of get are in execution. This is allowed because there is no danger of multiple get executions causing a race condition since they do not change any of the object’s data.

18.4 Interprocess Communication

Another form of data sharing in Ada is through the interprocess communication mechanism known as the entry. Ada uses the semantics of the remote procedure call to implement interprocess communication; entry behavior is like many-to-one communication under the phone model. Entries have parameters that are passed in the same way that procedure parameters are passed in Ada with the three modes of IN, OUT, and IN OUT. These correspond to parameters that are input only, output only, or both input and output. An entry is called by specifying the destination task, the entry name, and the actual parameters, much like calling a procedure. The destination task requests an entry by an accept statement, which receives the parameters that can then be used in the block of statements associated with the accept statement. As with procedures, the communication can be in either or both directions, depending on the mode of the parameters.

The synchronization associated with entries is the phone model approach described in the Section 9.4. The calling task places the call to a specific task. If that task is presently waiting at an accept statement for the called entry, the two tasks rendezvous, the called task executes the statements attached to the accept statement, and then both tasks proceed with their executions. If the called task is not presently waiting at a matching accept statement, the calling task is placed on a queue to wait for a rendezvous.

Consider the example of the two tasks shown in Figure 18.3. The tasks are called ONE and TWO, with ONE calling the entry MEET which is declared in task TWO. The action of task ONE is to calculate X using procedure CALCULATE, synchronize with TWO at MEET, sending X and getting back Y, and then use X and Y in procedure USE_IT. ONE will wait at its call to MEET until TWO reaches its accept statement. Task TWO will get a value of Z from procedure OBTAIN and then accept a call to MEET. If ONE has not yet called MEET, TWO will suspend until the call is placed. When the synchronization occurs, ONE is suspended and MEET takes in the value for formal parameter A. After calculating values for formal parameter B and local variable Z, the processing of MEET terminates. At this time, the value of formal parameter B is sent to actual parameter Y in ONE, ONE resumes execution at its call to USE_IT, and TWO resumes execution with its call to DISPLAY.
Figure 18.3 An Example of Ada Tasks

```adalog
task ONE;

task body ONE is
    X, Y : integer;
begin
    CALCULATE(X);
    TWO.MEET(X, Y);
    USE_IT(X, Y);
end ONE;

task TWO is
    entry MEET(A : in integer; B : out integer);
end TWO;

task body TWO is
    Z : integer;
begin
    OBTAIN(Z);
    accept MEET(A : in integer; B : out integer) do
        B := FN1(A, Z);
        Z := FN2(A, Z);
    end MEET;
    DISPLAY(Z);
end TWO;
```

Figure 18.4 An Example Ada Interaction of Task with Master Unit

```adalog
procedure TEST_TASKS is
    task TEST;

    task body TEST is
        COUNT : integer;
        begin
            for COUNT in 1..5 loop
                put(COUNT);
                new_line;
            end loop;
        end TEST;

        begin -- Main program
            for MAIN_COUNT in 6..10 loop
                put(MAIN_COUNT);
                new_line;
            end loop;
        end; --Main program
end TEST_TASKS;
```

Figure 18.4 illustrates the interaction of a task with its master unit. This program will result in the integer values from 1 to 10 being printed, but not necessarily in order. The integers 1 to 5 will appear in increasing order as will the integers 6 to 10, but the two lists will be merged in the output in an indeterminate manner. For example, one run of the program in Figure 18.4 produced the
18.5 Synchronization

The same mechanism that implements interprocess communication in Ada also implements synchronization. Because the accept statement can work with a code block or without a code block, it can implement both forms of the phone model in section 9.4, and it implements the gate model from section 9.5. Figure 18.5 contains an Ada program that illustrates the synchronization facilities of Ada entries.

Figure 18.5 An Example Ada Program Synchronizing Facilities of Entries

```ada
procedure TASKS3 is
    -- Two tasks, each with one entry
    task T1 is
        entry t1_entry;
    end T1;
    task T2 is
        entry t2_entry;
    end T2;
    task body T1 is
        begin
            put_line("In T1: before first accept.");
            accept t1_entry do
                put_line("In T1: in first accept.");
            end t1_entry;
            accept t1_entry do
                put_line("In T1: in second accept.");
            end t1_entry;
            put_line("In T1: terminating.");
        end T1;
    task body T2 is
        begin
            put_line("In T2: before first accept.");
            accept t2_entry do
                put_line("In T2: accepting the entry.");
            end t2_entry;
        end T2;
    begin
        -- Main program. At this point, the main program
```
and its two tasks begin execution concurrently.
put_line("In main: before calling T2.");
T2.t2_entry;
put_line("In main: after calling T2.");

put_line("In main: before calling T1.");
T1.t1_entry;
put_line("In main: after calling T2.");

-- Terminate, waiting for tasks to terminate.
put_line("In main: terminating.");
end TASKS3;

Output:
In T2: before first accept.
In T1: before first accept.
In main: before calling T2.
In T2: accepting the entry.
In T1: in first accept.
In Main: after calling T2.
In main: after calling T1.
In T1: in second accept.
In T1: terminating.
In T2: terminating.
In main: after calling T1.
In main: terminating.

In fact, Ada extends the gate model to include the monitoring of multiple gates in its implementation of mutual exclusion. This powerful feature in Ada is implemented through the use of selective waits to accept one from a number of pending synchronizations. This feature is rather complex, and we do not attempt to describe all of its capabilities here. We will instead examine the basic structure of the selective wait and its use of guarded commands.

First we examine the unguarded selective wait. In this case, a task is waiting for any of several accept statements to be called. This is similar to a receptionist waiting for any of several phones to ring. When one does ring, the receptionist responds to that phone. If two or more ring simultaneously, the receptionist chooses one at random to answer. Similarly, the select statement permits a task to wait on all of several different accepts to be called. When one is called, its associated handler is executed, and the task continues executing beyond the select. If more than one is called simultaneously, one of the called accepts is chosen in an indeterminate manner. The form of the select is

```ada
select
    accept MEET_1 do
        HANDLE_1;
    end MEET_1;
    or
    accept MEET_2 do
        HANDLE_2;
    end MEET_2;
    or
    accept MEET_3 do
        HANDLE_3;
```
end MEET_3;
end select;

This construct will wait until at least one of the accept alternatives is called. If none of the alternatives is called, it will wait forever. Ada provides a way to avoid this situation by expressing one of the alternatives as a \texttt{DELAY}. Consider the following example:

\begin{verbatim}
select
  accept MEET_1 do
    HANDLE_1;
  end MEET_1;
  or
  delay 1.0;
  HANDLE_2;
end select;
\end{verbatim}

This construct will wait up to one second for \texttt{MEET_1} to be called. If it has not been called by that time, \texttt{HANDLE_2} will be executed and the task will proceed beyond the select. A special case of \texttt{DELAY} is \texttt{ELSE} which has the same effect as \texttt{DELAY 0.0}. This is chosen by the select construct if none of the accepts have a call pending at the time the select begins execution.

Ada also permits \texttt{guards} to be placed on alternative conditions in a selective wait. To understand the role of guards, let us return to our example of the receptionist handling multiple phones. Some of the phones may have conditions placed on them specifying when they are to be connected and when they are to be disconnected. For example, Phone 1 may only need to be connected on Mondays, Phone 2 is connected only on the first day of the month, and Phone 3 is connected only when the outside temperature at 7:00 a.m. is greater than 25 degrees Celsius. When the receptionist comes to work, he or she first determines which phones to connect by evaluating each of the guard conditions. The receptionist then operates as above, handling the first phone that rings, choosing a phone at random if more than one rings simultaneously. Of course, the disconnected phones can by ignored since they will never ring. If it ever happens that all of the phones are disconnected, we consider that an exceptional condition since the receptionist can then go home.

The Ada representation of the preceding situation is expressed by the following guarded select statement:

\begin{verbatim}
select
  when day_of_week = MONDAY =>
    accept PHONE_1 do
      HANDLE_1;
    end PHONE_1;
  or
  when day_of_month = 1 =>
    accept PHONE_2 do
      HANDLE_2;
    end PHONE_2;
  or
  when OUTSIDE_TEMPERATURE > 25 =>
    accept PHONE_3 do
      HANDLE_3;
    end PHONE_3;
end select;
\end{verbatim}
The guard conditions are evaluated just once, when the select statement begins. These evaluations determine the open alternatives. The first alternative called that has an open guard is then handled, with the choice being made at random if more than one is called simultaneously. If all guards are false and an else alternative has not been specified, a PROGRAM_ERROR exception is raised.

Entry calls may be timed in Ada as well. This construct also utilizes the keyword select. For example,

```
select
  TASK_1.MEET_1;
or
  delay 5.0;
  NO_ANSWER_ACTION;
end select;
```

will wait 5 seconds for TASK_1 to execute an accept for entry MEET_1. If this does not occur within 5 seconds, procedure NO_ANSWER_ACTION will be invoked, and the task will proceed to the statements following the select. This is analogous to placing a phone call and hanging up if nobody answers in 5 seconds.

A delay of zero seconds can be expressed by

```
select
  TASK_1.MEET_1;
else
  NO_ANSWER_ACTION;
end select;
```

This will invoke NO_ANSWER_ACTION if TASK_1 is not waiting at an accept for MEET_1 at the time the entry call is made.

### 18.6 Ada Examples

#### 18.6.1 ATM Management

In this section we will develop the simple concurrent Ada program shown in Figure 18.6 to simulate the actions of an Automated Teller type application. Our system supports six accounts, which are accessible to all users. The task `data_manager`, accepts two entry calls, lock and unlock. The entry lock is a request from a user for exclusive access to an account and has an in parameter, the name of the account, and an out parameter, a Boolean done that indicates whether the account was accessible and therefore able to be locked by this entry. If the account is already locked, done is returned as false.

The task `data_manager` first initializes all locks to be unlocked. It then proceeds to a non-terminating loop, where it awaits calls to either lock or unlock.

Each user has an associated task of type `user`. These tasks are structured in an array of tasks named `users`. Each user task first accepts `identify` which establishes its `id_number` and then attempts to lock `Acct1` and `Acct2` through entry calls to `data_manager`. When these accounts have been successfully locked by the user, the user task `t` transfers $100 from `Acct1` to `Acct2`. After the transfer, the two accounts are unlocked using the `unlock` entry call.
procedure ATM is
  package INT_IO is new INTEGER_IO(integer);
  use INT_IO;
  package FLT_IO is new FLOAT_IO(float);

  -- Accounts are composed of a name and an amount of money.
  type account_name is (Acct1, Acct2, Acct3, Acct4, Acct5, Acct6);
  amount : array (account_name) of float;

  -- Declare the manager of all data.
  task data_manager is
    entry lock(account: in account_name);
    entry unlock(account: in account_name);
  end data_manager;

  -- Declare a task type for the user so we can declare many users.
  task type user is
    entry identify (id: in integer);
  end user;

  users: array (1..5) of user;

  -- This task handles the locking and unlocking of accounts.
  task body data_manager is
    type lock_states is (locked, unlocked);
    lock_status: array (account_name) of lock_states;

    begin
      for a in account_name loop
        lock_status(a) := unlocked;
      end loop;
      loop
        select
          accept lock(account: in account_name; done: out boolean) do
            if lock_status(account) = unlocked then
              lock_status(account) := locked;
              done := true;
            else
              done := false;
            end if;
          end lock;
        or
          accept unlock(account: in account_name) do
            lock_status(account) := unlocked;
          end unlock;
        else
          delay 5.0;
        end select;
      end loop;
    end data_manager;
--This simple user process handles one type of transaction:
--the transfer of $100 from Acct1 to Acct2.
task body user is

  ok: boolean;
  id_number: integer;

begin
  --Get the user number from the main program.
  accept identify(id: in integer) do
    id_number := id;
  end identify;

  --Lock Acct1
  loop
    data_manager.lock(Acct1,ok);
    if ok then exit; end if;
  end loop;
  put("User "); put(id_number); put_line("has grabbed Acct1.");

  --Lock Acct2
  loop
    data_manager.lock(Acct2,ok);
    if ok then exit; end if;
  end loop;
  put("User ");put(id_number); put_line("has grabbed Acct2.");

  --Make the transfer.
  amount(Acct1) := amount(Acct1) - 100.0;
  amount(Acct2) := amount(Acct2) + 100.0;
  put("User ");put(id_number); put_line("has made transfer");

  --Release the locks.
  data_manager.unlock(Acct1);
  put("User ");put(id_number); put_line("has unlocked Acct1");
  data_manager.unlock(Acct2);
  put("User ");put(id_number); put_line("has unlocked Acct2");
end user;

begin -- Main program
  --Initialize
  amount(Acct1) := 1000.0;
  amount(Acct2) := 1000.0;
  put_line("Initialized accounts, starting users.");
  --Start users by identifying them.
  for i in 1..5 loop
    users(i).identify(i);
  end loop;
end ATM;

Initialized accounts, starting users.
User # 1 has grabbed Acct1.
18.6.2 Sieve of Eratosthenes

One approach to generating prime numbers is known as the **Sieve of Eratosthenes**. The basic idea is that the positive integers are, in increasing order, sent through a sequence of filters that remove some integers and pass the others on to the next filter in the sequence. Whenever an integer \( n \) is tested, there already exists a filter for each prime that is smaller than \( n \). Each filter passes on only those integers that are not divisible by that filter’s associated prime number. When an integer has passed through all of the filters, it is known to be prime, because it is not divisible by any prime smaller than itself. Such an integer will then have a filter established in its honor so that all larger integers can pass through it.

This method is an excellent candidate for concurrent processing, because all of the filters can operate concurrently. An Ada program for the Sieve of Eratosthenes is given in Figure 18.7. Each filter is defined as a task. Since the number of tasks needed is not known at compile time, we introduce dynamic tasks through the creation of type `filter_task`, which is a pointer to a task of type `filter`. Each `filter` task then creates a new `filter` task to which it will pass integers that are not divisible by its prime. Every `filter` task assumes the first integer passed to it has already passed all previous filters and is hence the prime associated with the present filter.
Figure 18.7 Sieve of Eratosthenes in Ada

procedure SIEVE is
  package INT_IO is new INTEGER_IO(integer); use INT_IO;

  -- First, define the filter type.
  task type filter is
    entry number_stream(number : in integer);
  end filter;

  type_filter_task is access filter; -- point to a filter task
  first_filter : filter_task;
  numbers : integer;

  function new_filter return filter_task is
    begin
      return new filter;
    end new_filter;

  task body filter is
    next_filter : filter_task;
    prime, num : integer;
    begin
      accept number_stream (number : in integer) do
        prime := number;
      end number_stream;
      if prime > -1 then -- NOTE: “-1” is an end-of-numbers flag.
        put(prime); new_line;
        next_filter := new_filter; -- Spawn a new filter
        loop
          accept number_stream(number : in integer) do
            num := number;
          end number_stream;
          if num = -1 then
            next_filter.number_stream(num);
          end if;
          if num mod prime /= 0 then
            next_filter.number_stream(num); -- Pass the number on
          end if;
        end loop;
      end if;
    end -- Main program. Generate the initial list of integers.
    begin
      put("Enter end of integer range: ");
      get(numbers);
      first_filter := new filter;
      put_line("All primes in the range are: ");
      for i in 2..numbers loop
        first_filter.number_stream(i);
      end loop;
      first_filter.number_stream(-1); -- Flag the end-of-numbers.
    end;
Terms

Task | Pragma | Selective Wait
---|---|---
Master Task | Rendezvous | Guards
Child Task | Entries | Sieve of Eratosthenes
Fork-join Semantics | Accept Statement

Discussion Questions
1. Why did the designers of Ada elect to describe a task in two parts like a package instead of one part like a procedure?
2. Can you devise a situation where implicit invocation can cause problems? When is explicit invocation desired?
3. Why did Ada’s designers allow several different alternative accept statements in a select statement, but only one entry call in a select statement? Can you devise a situation where multiple entry calls would cause problems?
4. Why did Ada’s designers “allow” implicit invocation to take place at all? Since they designed tasks to be implicitly invoked, they could have denied the possibility of implicit invocation altogether. Why did they not design Ada this way?

Exercises
1. Consider how a quicksort might be implemented concurrently. Write the task specification for such an implementation.
2. Write the necessary task bodies for the implementation of a concurrent quicksort.
3. Consider the example in Figure 18.4. Construct two other sequences that are possible, given the constraints of the Ada semantics.
4. Given the Ada program TASKING_MANIA below, which of the output sequences on the next page is legal?

```ada
with TEXT_IO; use TEXT_IO;
procedure TASKING_MANIA is
  task t1;
  task t2 is
    entry entry1;
  end t2;
  task t3 is
    entry entry1;
  end t3;
  task body t1 is
  begin
    select
    ```
t2.entry1;
else
  put_line("T1: dying");
  terminate;
end select;
put_line("T1: entry call accepted");
end t1;

task body t2 is
  begin
    accept entry1;
    put_line("T2: accepted entry1");
    t3.entry1;
    accept entry1;
    put_line("T2: accepted entry1 again");
  end t2;

task body t3 is
  begin
    t2.entry1;
    accept entry1;
    put_line("T3: accepted entry1");
  end t3;

begin
  null; -- main program needs at least a null statement
end TASKING_MANIA;

(a) T2: accepted entry1       (b) T2: accepted entry1
  T1: dying                   T1: entry call accepted
  T3: accepted entry1
  T2: accepted entry1 again

(c) T2: accepted entry1       (d) T1: dying
  T3: accepted entry1
  T1: entry call accepted
  T3: accepted entry1
  T2: accepted entry1 again

(e) T1: entry call accepted
  T2: accepted entry1
  T3: accepted entry1
  T2: accepted entry1 again

5. Modify the Sieve of Eratosthenes example given in the text so that all printing is done by a print-server, i.e., a task that loops through a select statement waiting to service centralized printing requests. You will need numeric and string printing routines in the server.

6. Ada abides by “remote procedure call semantics” with respect to interprocess communication. You are to determine whether the other forms can be “simulated” in Ada, i.e., whether Ada IPC semantics can look like “no-wait send” or “wait-for-notification send”. In both cases, if simulation is possible, write a program that demonstrates the required semantics; if simulation is impossible, write a paragraph (in English) that justifies your decision.
7. Write a program that would demonstrate how semaphores do not protect “critical sections” completely.

Laboratory Exercises
1. We stated that the IPC mechanism in Ada is the remote procedure call. Write a program to demonstrate the semantics.
2. Can you “simulate” other IPC mechanisms in Ada? If you can demonstrate how to do this, write some demonstration programs.
3. Implement a solution to the Dining Philosophers problem in Ada.
4. Implement a solution to both producer-consumer problems given in the last chapter.
5. Write a program that demonstrates the corruption of data through poor implementation of data sharing protection. Then correct your program and demonstrate that your corrected version protects the data properly.