Chapter 4
Concurrent programming

Concurrent programming is the name given to the programming notations and techniques for expressing potential parallelism and solving the resulting synchronization and communication problems.

There are three main motivations for wanting to write concurrent programs.

(1) 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.
(2) To fully utilize the processor – modern processors run at speeds far in excess of the input and output devices with which they must interact. A sequential program that is waiting for I/O is unable to perform any other operation.
(3) 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). Modern hardware platforms consist of multiple processors to obtain more powerful execution environments. A concurrent program is able to exploit this true parallelism and obtain faster execution.

From a concurrency perspective, the implementation platform can be considered irrelevant. However, in some application areas the amount or presence of parallelism is critical to the success of the program. For example, in high-performance computing there is a requirement to get maximum performance from the platform. Inevitably, creating many tasks adds overhead. Hence, there may be some advantages from limiting the number of concurrent activities to the numbers of CPUs, or rather writing the application so that the number of actvities created at run-time matches the number of processors available.

From a theoretical point of view, Amdahl's law gives the relationship between the expected speedup of parallelizing implementations of an algorithm. If P is the proportion of that algorithm's code that can benefit from parallelization and N is the number of processors, the maximum speedup that can be achieved by using these N processors is

$$\frac{1}{(1-P)+\frac{P}{N}} \tag{4.1}$$

In the limit, as N tends to infinity, the maximum speedup tends to $1/(1-P)$. Consequently, if only 50% of the algorithm is amenable to parallel execution, then the maximum possible speedup is only 2. Consequently, there is no point (for efficiency considerations) in having 100 processors to perform the algorithm.

Of course, speedup is only one reason for the use of concurrent activities. Hence, even though the number of processors might be limited, there are still advantages to having more concurrent activities than processors. This, and the following two chapters, concentrate on the issues associated with general concurrent programming.

4.1 Processes and tasks/threads

Any language, natural or computer, has the dual property of enabling expression while 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, FORTRAN and COBOL share the common property of being sequential programming 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. This is not adequate for the programming of real-time systems.

Following the pioneering work of Dijkstra (1968), a concurrent program is conventionally viewed as consisting of a collection of autonomous sequential processes, executing (logically) in parallel. Concurrent programming languages all incorporate, either explicitly or implicitly, the notion of process; each process itself has a single thread of control. All operating systems provide facilities for creating concurrent processes. Usually, each process executes in its own virtual machine to avoid interference from other, unrelated, processes. Each process is, in effect, a single program. However, in recent years there has been a tendency to provide the facilities for processes to be created within programs.

Modern operating systems allow processes created within the same program to have unrestricted access to shared memory (such processes are called **threads** or **tasks**). Hence, in operating systems like those conforming to the POSIX API, it is necessary to distinguish the concurrency between programs (processes) from the concurrency within a program (threads/tasks). Often, there is also a distinction between threads which are visible to the operating system (often called *kernel-level* threads) and those that are supported solely by library routines (*library-level* threads). For example, Windows XP/2000 supports threads and **fibers**, the latter being invisible to the kernel. Hybrid models are also possible where a process can be allocated a maximum number of kernel-level threads and a thread library (or sometimes the kernel itself) is responsible for mapping the process's threads to the appropriate library or kernel-level threads.

> *In this book the term **task** or **thread** will be used (interchangeably) to represent a single thread of control. The term* process *will be used to indicate one or more threads/tasks executing within its own shared memory context. With most concurrent programming languages, it is the thread/task abstraction that is supported rather than the process abstraction.*

Where a concurrent program is being executed on top of an operating system, its Run-Time Support System (RTSS) can either map the program's notion of a task onto the underlying operating system's notion of a thread, or it can make the program's notion of task invisible to the operating system (and in effect implement its own fibers library). Of course, in the latter case, the RTSS must ensure that all input/output operations are asynchronous, otherwise the whole concurrent program will block every time one of its tasks performs an operation requiring operating system support.

The actual implementation (that is, execution) of a collection of tasks usually takes one of three forms. Tasks can either:

(1) multiplex their executions on a single processor;
(2) multiplex their executions on a multiprocessor system where all processors have access to common shared memory (for example, a symmetric multiprocessor (SMP) system);
(3) multiplex their executions on several processors where there is no common shared memory (for example, a distributed system).

Hybrids of these three methods are also possible, for example, Non-Uniform Memory Architectures (NUMA) where there may be a mixture of shared and non-shared memory.

Only in cases (2) and (3) is there the possibility of true parallel execution of more than one task. The term **concurrent** indicates potential parallelism. Concurrent programming languages thus enable the programmer to express logically parallel activities without regard to their implementation.

The life of a task is illustrated, simply, in Figure 4.1. A task is created, moves into the state of initialization, proceeds to execution and termination. Note that some tasks may never terminate and that others, which fail during initialization, pass directly to termination without ever executing. After termination, a task goes to non-existing when it can no longer be accessed (as it has gone out of scope). Clearly, the most important state for a task is executing; however, as processors are limited not all tasks can be executing

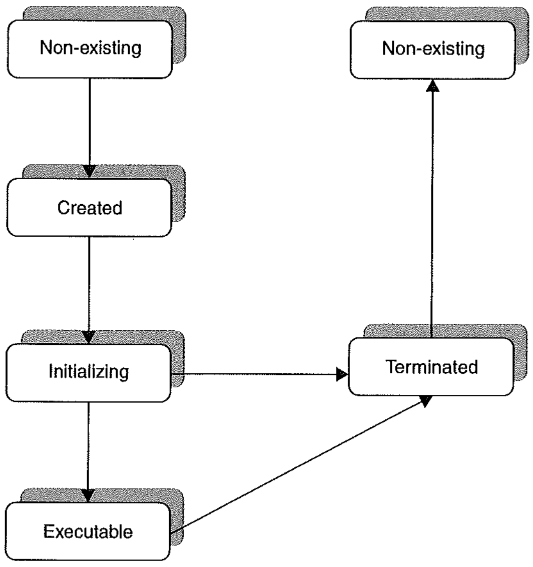

Figure 4.1 Simple state diagram for a task.

at once. Consequently, the term **executable** is used to indicate that the task could execute if there is a processor available.

From this consideration of a task, it is clear that the execution of a concurrent program is not as straightforward as the execution of a sequential program. Tasks must be created and terminated, and dispatched to and from the available processors. These activities are undertaken by the RTSS or Run-Time Kernel. The RTSS sits logically between the hardware (along with any provided operating system) and the application software. In reality, it may take one of a number of forms.

(1) A software structure programmed as part of the application (that is, as one component of the concurrent program). C and C++ programs often provide their own thread libraries that are tailored to their applications' requirements.

(2) A standard software system generated with the program object code by the compiler. This is normally the structure with Ada and Java programs.

(3) A hardware structure microcoded into the processor for efficiency. For example, the aJile System's aJ100 processor directly supports the execution of Java byte code.

The algorithm used for scheduling by an RTSS (or an operating system) to decide which task to execute next if there is more than one executable will affect the time behaviour of the program. However, for well-constructed programs, the logical behaviour of a program will not be dependent on the scheduling. From the program's point of view, the RTSS (or operating system) is assumed to schedule tasks non-deterministically. For real-time systems, the characteristics of the scheduling are significant. They are considered further in Chapter 11.

4.1.1 Concurrent programming constructs

Although constructs for concurrent programming vary from one language (and operating system) to another, there are three fundamental facilities that must be provided. These allow the following:

(1) the expression of concurrent activities (threads, tasks or processes);
(2) the provision of synchronization mechanisms between concurrent activities;
(3) primitives that support of communication between concurrent activities.

In considering the interaction of tasks, it is useful to distinguish between three types of behaviour:

- independent
- cooperating
- competing.

Independent tasks do not communicate or synchronize with each other. Cooperating tasks, by comparison, regularly communicate and synchronize their activities in order to perform some common operation. For example, a component of an embedded computer system may have several tasks involved in keeping the temperature and humidity of a gas in a vessel within certain defined limits. This may require frequent interactions.

A computer system has a finite number of resources which may be shared between processes and tasks; for example, peripheral devices, memory and processor power. In order for tasks to obtain their fair share of these resources, they must compete with each other. The act of resource allocation inevitably requires communication and synchronization between the tasks in the system. However, although these tasks communicate and synchronize in order to obtain resources, they are, essentially, independent.

Discussion of the facilities which support task creation and task interaction is the focus of this and the next two chapters.

4.2 Concurrent execution

Although the notion of tasks or threads is common to all concurrent programming languages, there are considerable variations in the models of concurrency adopted. These variations appertain to:

- structure
- level
- granularity
- initialization
- termination
- representation.

Language	Structure	Level
Concurrent Pascal	static	flat
occam2	static	nested
Modula-1	dynamic	flat
C/POSIX	dynamic	flat
Ada	dynamic	nested
Java	dynamic	nested
C#	dynamic	nested

Table 4.1 The structure and level characteristics of a number of concurrent programming languages.

The *structure* of a task may be classified as follows.

- **Static** – the number of tasks is fixed and known at compile-time.
- **Dynamic** – tasks are created at any time. The number of extant tasks is determined only at run-time.

Another distinction between languages comes from the *level* of parallelism supported. Again, two distinct cases can be identified.

(1) **Nested** – tasks are defined at any level of the program text; in particular, tasks are allowed to be defined within other tasks.

(2) **Flat** – tasks are defined only at the outermost level of the program text.

Table 4.1 gives the structure and level characteristics for a number of concurrent programming languages. In this table, the language C is considered to incorporate the Real-Time POSIX pthread mechanisms.

Within languages that support nested constructs, there is also an interesting distinction between what may be called **coarse** and **fine grain** parallelism. A coarse grain concurrent program contains relatively few tasks, each with a significant life history. By comparison, programs with a fine grain of parallelism will have a large number of simple tasks, some of which will exist for only a single action. Most concurrent programming languages, typified by Ada, display coarse grain parallelism. Occam2 is a good example of a concurrent language with fine grain parallelism.

When a task is created, it may need to be supplied with information pertinent to its execution (much as a procedure may need to be supplied with information when it is called). There are two ways of performing this **initialization**. The first is to pass the information in the form of parameters to the task; the second is to communicate explicitly with the task after it has commenced its execution. Most modern concurrent languages allow parameters to be passed at task creation time.

Task **termination** can be accomplished in a variety of ways. The circumstances under which tasks are allowed to terminate can be summarized as follows:

(1) completion of execution of the task's body;

(2) suicide, by execution of a 'self-terminate' statement;

(3) abortion, through the explicit action of another task;

(4) occurrence of an untrapped error condition;

(5) never: tasks are assumed to execute non-terminating loops;

(6) when no longer needed.

With nested levels, hierarchies of tasks can be created and intertask relationships formed. For any task, it is useful to distinguish between the task (or block) that is responsible for its creation and the task (or block) which is affected by its termination. The former relationship is known as **parent/child** and has the attribute that the parent may be delayed while the child is being created and initialized. The latter relationship is termed **guardian/dependant**. A task may be dependent on the guardian task itself or on an inner block of the guardian. The guardian is not allowed to exit from a block until all dependent tasks of that block have terminated (that is, a task cannot exist outside its scope). It follows that a guardian cannot terminate until all its dependants have also terminated. This rule has the particular consequence that a program itself will not be able to terminate until all tasks created within it have also terminated.

In some situations, the parent of a task will also be its guardian. This will be the case when using languages which allow only static task structures. With dynamic task structures (that are also nested), the parent and guardian may or may not be identical. This will be illustrated later in the discussion of Ada. Figure 4.2 includes the new states that have been introduced in the above discussion.

One of the ways a task may terminate (point (3) in the above list) is by the application of an abort statement. The existence of abort in a concurrent programming language is a question of some contention and is considered in Chapter 8 within the context of resource control. For a hierarchy of tasks, it is usually necessary for the abort of a guardian to imply the abort of all dependants (and their dependants and so on).

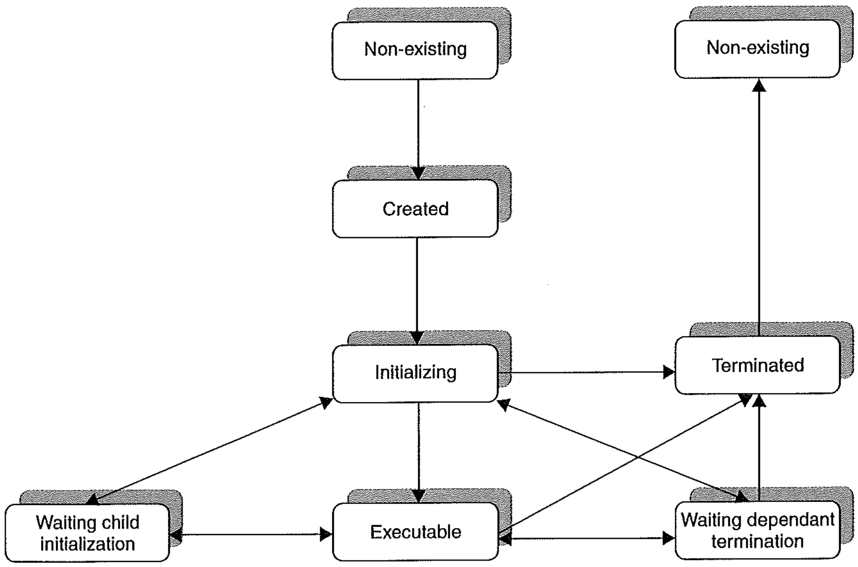

Figure 4.2 State diagram for a task.

The final circumstance for termination, in the above list, is considered in more detail when task communication methods are described. In essence, it allows a server task to terminate if all other tasks that could communicate with it have already terminated.

4.2.1 Tasks and objects

The object-oriented programming (OOP) paradigm encourages system (and program) builders to consider the artifact under construction as a collection of cooperating objects (or, to use a more neutral term, *entities*). Within this paradigm, it is constructive to consider two kinds of object – active and reactive. **Active** objects undertake spontaneous actions (with the help of a processor): they enable the computation to proceed. **Reactive** objects, by comparison, only perform actions when 'invoked' by an active object. Other programming paradigms, such as data-flow or real-time networks, also identify active agents and passive data.

Only active entities give rise to spontaneous actions. 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. 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** (or **synchronized**). Alternatively, if an active agent is required to program the correct level of control, then the resource is in some sense active. The term **server** will be used to identify this type of entity, and the term **protected resource** to indicate the passive kind. These, together with **active** and **passive**, are the four abstract program entities used in this book.

In a concurrent programming language, active entities are represented by tasks/threads. Passive entities can be represented either directly as data variables or they can be encapsulated by some module/package/class 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 synchronization facility. Servers, because they need to program the control agent, require a task.

From an object-oriented programming perspective:

- **passive object** – a reactive entity with no synchronization constraints, it needs an external thread of control for its methods to be executed;
- **protected object** – a reactive entity with synchronization constraints, it is typically shared between many threads and it needs an external thread of control for its methods to be executed;
- **active object** – an object with an explicit or implicit internal thread;
- **server object** – an active object with synchronization constraints, it is typically shared between many threads.

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 (for example, a semaphore), are normally implemented efficiently (at least on single-processor systems). However, they can be inflexible and lead to poor program structures for some classes of problems (this is discussed further in Chapter 5). Servers, because the control agent is programmed using a task, are eminently flexible. The drawback of this approach is that it can lead to a proliferation of tasks, 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. As will be illustrated in this chapter and Chapters 5 and 6, Ada, Java and C/Real-Time POSIX support the full range of entities.

4.3 Task representation

There are three basic mechanisms for representing concurrent execution: fork and join, cobegin and explicit task declaration.

4.3.1 Fork and join

This simple approach does not provide a visible entity for a task but merely supports two statements. The fork statement specifies that a designated routine should start executing concurrently with the invoker of the fork. The join statement allows the invoker to synchronize with the completion of the invoked routine. For example:

```
function F return ... is
begin
  ...
end F;

procedure P is
begin
  ...
  C:= fork F;
    .
    .
    .
  J:= join C;
  ...
end P;
```

Between the execution of the fork and the join, procedure `P` and function `F` will be executing concurrently. At the point of the join, the procedure will wait until the function has finished (if it has not already done so). Figure 4.3 illustrates the execution of fork and join.

The use of fork and join primitives is most prevalent in parallel programming languages (and they are often called spawn and join). The Linux operating system provides the `clone` system call that allows a new process to be created that shares the same address space as the parent process. The `wait` and `waitpid` system calls provide the join mechanism.

A version of fork and join can also be found in the C/Real-Time POSIX; here `fork` and `vfork` are used to create a copy of the invoker, and are used with the `wait` and `waitpid` system calls.

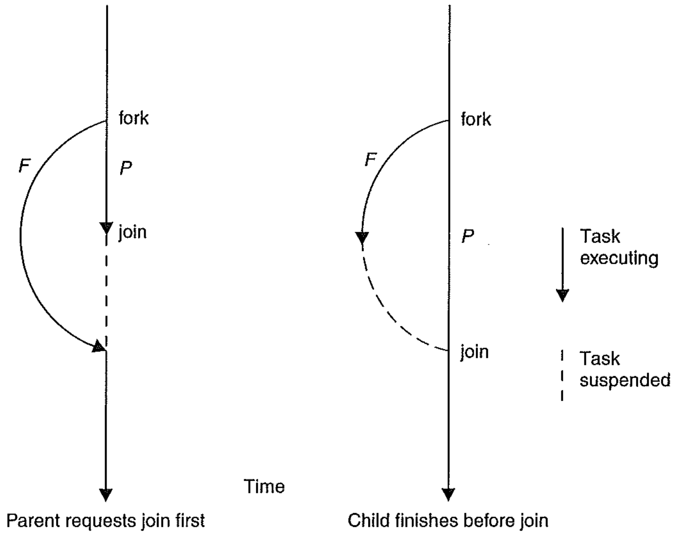

Figure 4.3 Fork and join.

Fork and join allow for dynamic process/task creation and provide a means of passing information to the child via parameters. Usually only a single value is returned by the child on its termination. Although flexible, fork and join do not provide a structured approach to process/task creation and can be error prone in use. For example, in some systems a guardian must explicitly 'rejoin' all dependants rather than merely wait for their completion.

4.3.2 Cobegin

The cobegin (or parbegin or par) is a structured way of denoting the concurrent execution of a collection of statements.

```
cobegin
  S1;
  S2;
  S3;
  .
  .
  .
  Sn
coend
```

This code causes the statements `S1`, `S2` and so on to be executed concurrently. The `cobegin` statement terminates when all the concurrent statements have terminated. Each of the `Si` statements may be any construct allowed within the language, including simple assignments or procedure calls. If procedure calls are used, data can be passed to the invoked task via the parameters of the call. A `cobegin` statement could even include a sequence of statements that itself has a `cobegin` within it. In this way, a hierarchy of tasks can be supported. Figure 4.4 illustrates the execution of the `cobegin` statement.

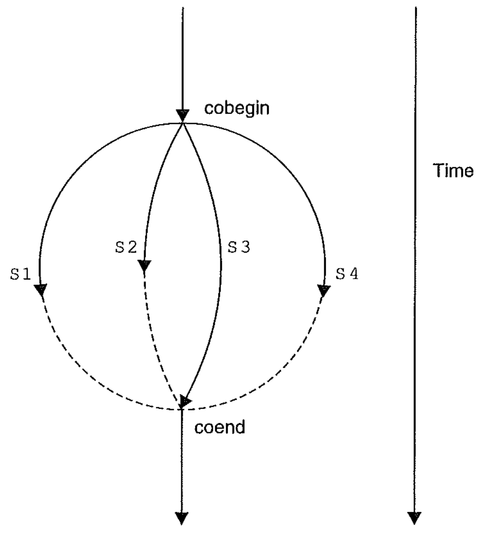

Figure 4.4 Cobegin.

Cobegin can be found in early concurrent programming languages (like Concurrent Pascal and occam2) but like 'fork and join' it has fallen from favour in modern real-time languages.

4.3.3 Explicit task declaration

Although sequential routines may be executed concurrently by means of the cobegin or fork, the structure of a concurrent program can be made much clearer if the routines themselves state whether they will be executed concurrently. Explicit task declaration provides such a facility, and has become the standard way of expressing concurrency in concurrent real-time languages. The following sections consider the mechanisms provided by Ada, Java and C/Real-Time POSIX.

4.4 Concurrent execution in Ada

The conventional unit of parallelism is called a **task** in Ada. Tasks may be declared at any program level; they are created implicitly upon entry to the scope of their declaration. The following example illustrates a procedure containing two tasks (A and B).

```
procedure Example1 is
  task A;
  task B;

  task body A is
  -- local declarations for task A
  begin
  -- sequence of statement for task A
  end A;
```

```
   task body B is
   -- local declarations for task B
   begin
   -- sequence of statements for task B
   end B;

begin
   -- tasks A and B start their executions before
   -- the first statement of the sequence of
   -- statements belonging to the procedure
   .
   .
   .
end Example1;   -- the procedure does not terminate
                -- until tasks A and B have terminated.
```

Tasks consist of a specification and a body. They can be passed initialization data upon creation.

In the above, tasks A and B (which are created when the procedure is called) are said to have anonymous types, as they do not have types declared for them. Types could easily have been given for A and B:

```
task type A_Type;
task type B_Type;
A : A_Type;
B : B_Type;
task body A_Type is
  -- as before for task body A
task body B_Type is
  -- as before for task body B
```

With task types, a number of instances of the same task can easily be declared.

```
task type T;
A,B : T;

type Long is array (1..100) of T;

type Mixture is record
  Index : Integer;
  Action : T;
end record;

L : Long;
M : Mixture;

task body T is ...
```

To give a more concrete example of the use of tasks in an Ada program consider the implementation of a robot arm system. The arm can move in three dimensions. The movement in each dimension is powered by a separate motor that is controlled by a distinct Ada task. These tasks loop around, each reading a new relative setting for its dimension and then calling a low-level procedure Move_Arm to cause the arm to move.

First a package is declared that provides the necessary support types and subprograms.

```
package Arm_Support is
  type Dimension is (Xplane, Yplane, Zplane);
  type Coordinate is new Integer range ...;

  procedure Move_Arm(D: Dimension; C: Coordinate);
    -- Moves the arm to C

  function New_Setting(D: Dimension) return Coordinate;
    -- Returns a new required relative position

end Arm_Support;
```

Now the main program can be presented.

```
with Arm_Support; use Arm_Support;
procedure Main is

  task type Control(Dim : Dimension);

  C1 : Control(Xplane);
  C2 : Control(Yplane);
  C3 : Control(Zplane);

  task body Control is
    Position : Coordinate;      -- current position
    Setting  : Coordinate;      -- required relative movement
  begin
    Position := Coordinate'First;   -- rest position
    loop
      Move_Arm(Dim, Position);
      Setting := New_Setting(Dim);
      Position := Position + Setting;
    end loop;
  end Control;
begin
  null;
end Main;
```

It is, perhaps, worth noting that Ada only allows discrete types and access (pointer) types to be passed as initialization parameters to a task.

By giving non-static values to the bounds of an array (of tasks), a dynamic number of tasks is created. Dynamic task creation can also be obtained explicitly using the 'new' operator on an access type (of a task type):

```
procedure Example2 is
  task type T;
  type A is access T;
  P : A;
  Q : A:= new T;
begin
  ...
  P := new T;
  Q := new T;
  ...
end Example2;
```

Q is declared to be of type A and is given a 'value' of a new allocation of T. This creates a task that immediately starts its initialization and execution; the task is designated `Q.all` (`all` is an Ada naming convention used to indicate the task itself, not the access pointer). During execution of the procedure, P is allocated a task (`P.all`) followed by a further allocation to Q. There are now three tasks active within the procedure; `P.all`, `Q.all` and the task that was created first. This first task is now anonymous as Q is no longer pointing to it. In addition to these three tasks, there is the task or main program executing the procedure code itself; in total, therefore, there are four distinct threads of control.

Tasks created by the operation of an allocator ('new') have the important property that the block that acts as its guardian (or **master** as Ada calls it) is not the block in which it is created but the one that contains the declaration of the access type. To illustrate this point, consider the following:

```
declare
  task type T;
  type A is access T;
begin
  .
  .
  .
  declare            -- inner block
    X : T;
    Y : A:= new T;
  begin
    -- sequence of statements
  end;  -- must wait for X to terminate but not Y.all
  .
  .     -- Y.all could still be active although the name Y is
        -- out of scope
  .
end;    -- must wait for Y.all to terminate
```

Although both X and `Y.all` are created within the inner block, only X has this block as its master. Task `Y.all` is considered to be a dependent of the outer block, and it therefore does not affect the termination of the inner block.

If a task fails while it is being initialized (an exercise called **activation** in Ada) then the parent of that task has the exception Tasking_Error raised. This could occur, for example, if an inappropriate initial value is given to a variable. Once a task has started its true execution, it can catch any raised exceptions itself.

4.4.1 Task identification

One of the main uses of access variables is in providing 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. Care must be exercised here, as duplicated names can cause confusion and lead to programs that are difficult to understand. Furthermore, tasks may be left without any access pointers; such tasks are said to be **anonymous**. For example, if Robot_Arm pointed to a task, when it was overwritten with New_Arm, the previous task will have become anonymous if there were no other pointers to it.

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 synchronization 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. An abridged version of the associated package is shown in Program 4.1.

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

(1) 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.

(2) 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 6).

Program 4.1 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 Current_Task return Task_Id;
    -- returns unique id of calling task

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

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.

4.4.2 Task termination

Having considered creation and representation, one is left with task termination. Ada provides a range of options; a task will terminate if:

(1) it completes execution of its body (either normally or as the result of an unhandled exception);

(2) it executes a 'terminate' alternative of a select statement (this is explained in Section 6.5) thereby implying that it is no longer required;

(3) it is aborted.

If an unhandled exception has caused the task's demise then the effect of the error is isolated to just that task.

Another task can enquire (by the use of an attribute) if a task has terminated:

```
if T'Terminated then   -- for some task T
  -- error recovery action
end if;
```

However using this mechanism, the enquiring task cannot differentiate between normal or error termination of the other task, and, of course, the task could terminate just after the test has been performed.

In Ada 2005, extra support has been added to help the program manage task termination, in particular unexpected termination due to error conditions. An abridged version of the package is shown in Program 4.2. Essentially, a task can have an associated access variable to a protected procedure.[1] The Ada run-time support system will call this procedure when the task terminates. A parameter to the call gives the cause of the termination.

4.4.3 Task abortion

Any task can abort any other task whose name is in scope. When a task is aborted, all its dependants are also aborted. The abort facility allows wayward tasks to be removed. It is, of course, a vary dangerous mechanism and should only be used when there is no alternative course of action (see Section 7.6.1).

4.4.4 OOP and concurrency

Ada 95 did not 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. Since Ada 95, object-oriented programming techniques have advanced and the notion of an *interface* has emerged as the preferred mechanism for acquiring many of the benefits of multiple inheritance without most of the problems. The introduction of this mechanism

[1]A protected procedure in Ada is a procedure that is declared within a monitor-like object, see Section 5.8.

Program 4.2 The Ada 2005 support for task termination.

```
with Ada.Task_Identification; with Ada.Exceptions;

package Ada.Task_Termination is
  ...
  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_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;
```

into Ada 2005 allows tasks to support interfaces. Whilst this does not give the full power of extensible tasks, it does give much of the functionality.

The interface facility provided by Ada 2005 is related to inter-task communication, and will be considered in detail in Sections 5.8.2 and 6.3.3.

4.5 Concurrent execution in Java

Java has a predefined class, `java.lang.Thread` which provides the mechanism by which threads (tasks) are created. However, to avoid all threads having to be child classes of `Thread`, Java also has a standard interface, called `Runnable`.

```
public interface Runnable {
  public void run();
}
```

Hence any class which wishes to express concurrent execution must implement this interface and provide the `run` method. The `Thread` class given in Program 4.3 does just this.

`Thread` is a subclass of `Object`. It provides several constructor methods and a `run` method. Using these constructors, threads can be created in two ways.

The first is to declare a class to be a subclass of `Thread` and override the `run` method. An instance of the subclass can then be allocated and started. For instance, in the robot arm example assume that the following classes and objects are available:

```
public class UserInterface {
  public int newSetting (int Dim) { ... }
  ...
}
```

Program 4.3 An abridged version of the Java `Thread` class.

```
package java.lang;
public class Thread extends Object implements Runnable {

  //nested classes
  public static final enum State {BLOCKED, NEW, RUNNABLE,
                 TERMINATED, TIMED_WAITING, WAITING};

  public static interface UncaughtExceptionHandler{
    public void uncaughtException(Thread t, Throwable e);
  };

  // constructors
  public Thread();
  public Thread(String name);
  public Thread(Runnable target);
  public Thread(Runnable target, String name);

  // methods associated with thread execution
  public void run();
  public void start();
  public final void join() throws InterruptedException;

  // methods associated with thread naming and identification
  public static Thread currentThread();
  public String getName();
  public void setName(String name);

  // methods associated with thread states and termination
  public static UncaughtExceptionHandler
          getDefaultUncaughtExceptionHandler();
  public UncaughtExceptionHandler
          getUncaughtExceptionHandler();
  public static setDefaultUncaughtExceptionHandler1(
          UncaughtExceptionHandler eh);
  public void setUncaughtExceptionHandler(
          UncaughtExceptionHandler);
  public final boolean isAlive();
  public final boolean isDaemon();
  public final void setDaemon();
  public State getState();

  // other methods etc will be introduced throughout this
  // book

  // Note, RuntimeExceptions are not listed as part of the
  // method specification.
}
```

```
public class Arm {
  public void move(int dim, int pos) { ... }
}

UserInterface UI = new UserInterface(); Arm Robot = new Arm();
```

Given the above classes, in scope, the following will declare a class which can be used to represent the three controllers.

```
public class Control extends Thread {

  private int dim;

  public Control(int Dimension) { // constructor
    super();
    dim = Dimension;
  }

  public void run() {

    int position = 0;
    int setting;

    while(true)
    {
       Robot.move(dim, position);
       setting = UI.newSetting(dim);
       position = position + setting;
    }
  }
}
```

The three controllers can now be created:

```
final int xPlane = 0;   // final indicates a constant
final int yPlane = 1;
final int zPlane = 2;

Control C1 = new Control(xPlane);
Control C2 = new Control(yPlane);
Control C3 = new Control(zPlane);
```

At this point, the threads have been created, any variables declared have been initialized and the constructor methods for the `Control` and `Thread` class have been called (Java calls this the *new* state). However, the thread does not begin its execution until the `start` method is called:

```
C1.start();
C2.start();
C3.start();
```

Note that if the `run` method is called explicitly then the code is executed sequentially.

The second way to create a thread is to declare a class that implements the `Runnable` interface. An instance of the class can then be allocated and passed as an argument during the creation of a thread object. Remember, Java threads are not created automatically when their associated objects are created, but must be explicitly created and started using the `start`method.

```
public class Control implements Runnable {

  private int dim;

  public Control(int Dimension)  { // constructor
    dim = Dimension;
  }

  public void run() {
    int position = 0;
    int setting;

    while(true) {
       Robot.move(dim, position);
       setting = UI.newSetting(dim);
       position = position + setting;
    }
  }
}
```

The three controllers can now be created:

```
final int xPlane = 0;
final int yPlane = 1;
final int zPlane = 2;

Control C1 = new Control(xPlane); // no thread created yet
Control C2 = new Control(yPlane);
Control C3 = new Control(zPlane);
```

and then associated with threads and started:

```
// constructors passed a Runnable interface and threads created
Thread X = new Thread(C1);
Thread Y = new Thread(C2);
Thread Z = new Thread(C2);

X.start(); // thread started
Y.start();
Z.start();
```

Like Ada, Java allows dynamic thread creation. In contrast to Ada, Java (by means of constructor methods) allows arbitrary data to be passed as parameters (of course, all objects are passed by references which are similar to Ada's access variables).

Although Java allows thread hierarchies and thread groups to be created, there is no master or guardian concept. This is because Java relies on garbage collection to clean up objects which can no longer be accessed. The exception to this is the main program. The main program in Java terminates when all its user threads have terminated.

One thread can wait for another thread (the target) to terminate by issuing the `join` method call on the target's thread object. Furthermore, the `isAlive` method allows a thread to determine if the target thread has terminated.

4.5.1 Thread identification

There are two ways that threads can be identified. If the code of the thread is a subclass of the `Thread` class, then the following will define a thread identifier.

```
Thread threadID;
```

Any subclass of `Thread` can be assigned to this object (because of the reference semantics of Java). Note that where the code of a thread is passed through a constructor with the `Runnable` interface, the thread identifier would be the thread object not the object providing the `Runnable` interface. To define an identifier which would identify the object providing the code requires a `Runnable` interface to be defined.

```
Runnable threadCodeID;
```

However, once a reference to the `Runnable` object has been obtained there is little explicitly that can be done with it. All the thread-related operations are provided by the `Thread` class.

The identity of the currently running thread can be found using the `currentThread` method. This method has a `static` modifier which means that there is only one method for all instances of `Thread` objects. Hence the method can always be called using the `Thread` class.

4.5.2 Thread termination

A Java thread terminates when it completes execution of its `run` method either normally or as the result of an unhandled exception.

Threads can be of two types: **user** threads or **daemon** threads. Daemon threads are those threads which provide general services and typically never terminate. Hence when all user threads have terminated, daemon threads can also be terminated and the main program terminates. (Daemon threads provide the same functionality as the Ada 'or terminate' option on the select statement – see Section 6.5.) The `setDaemon` method is used to identify daemon threads, but must be called before the thread is started.

Early versions of the language did provide the equivalent of the Ada abort facility. However, these mechanisms have been deprecated as Java views them as being inherently unsafe.

4.5.3 Thread-related exceptions

In Chapter 3, Java `RuntimeExceptions` were introduced. The following `RuntimeExceptions` are relevant to threads.

The `IllegalThreadStateException` is thrown when:

- the `start` method is called and the thread has already been started;
- the `setDaemon` method has been called and the thread has already been started.

The `InterruptedException` is thrown if a thread which has issued a `join` method is woken up by the thread being interrupted rather than the target thread terminating (see Section 7.7.2).

As of Java 5, all threads can set uncaught exception handlers. Essentially, this is a handler that is run if the thread terminates due to an uncaught exception. (It is equivalent to the Ada `Task_Termination` package given in Program 4.2.) The definition of the handler is via the static `UncaughtExceptionHandler` interface included locally in the `Thread` class. It contains a single method `uncaughtException`. An object implementing this interface can be passed to the static method `setDefaultUncaughtExceptionHandler`. The Java Virtual Machine will call the method in this object before the associated thread terminates.

4.5.4 Real-time threads

The Java programming language lacks many facilities for programming real-time systems; hence the development of Real-Time Java. Real-Time Java produces several new thread classes. These will be considered in Section 10.1.

4.6 Concurrent execution in C/Real-Time POSIX

C/Real-Time POSIX provides three mechanisms for creating concurrent activities. The first is the traditional process-level Unix `fork` mechanism (and its associated `wait` system call). This causes a copy of the entire process to be created and executed. The details of fork can be found in most textbooks on operating systems and will not be discussed here. (An example of process creation using fork is given in Section 6.7.) The second is the `spawn` system call which is equivalent in function to a combined `fork` and `exec`.

C/Real-Time POSIX also allows for each process to contain several 'threads' of execution. These threads all have access to the same memory locations and run in a single address space. Thus, they can be compared with Ada's tasks and Java's threads. Programs 4.4 and 4.5 illustrate the primary C interface for thread creation in Real-Time POSIX.

It is not defined whether a POSIX-compliant system must support a single kernel-level thread (see Section 4.1) for each application-level thread created by a call to the `pthread_create` API function. Usually, a kernel will ensure that a sufficient number of threads can execute in order to ensure an application can make progress. The `pthread_setconcurrency` function allows an application to inform the kernel of its desired level of concurrency (i.e. how many kernel threads it would like – a zero means that the kernel should manage the level itself). However, there is no requirement for the kernel to take notice of this request.

All threads have attributes (for example, their stack size and any overflow buffer – called a **guard** in C/Real-Time POSIX). To manipulate these attributes, it is necessary to define an attribute object (of type `pthread_attr_t`) and then call functions to set and get the attributes. Once the correct attribute object has been established, a thread can be created and the appropriate attribute object passed. Program 4.4 shows the typical interfaces.

Program 4.4 A C/Real-Time POSIX interface to thread attributes.

```
typedef ... pthread_t;  /* details not defined */
typedef ... pthread_attr_t;
typedef ... size_t;

int pthread_attr_init(pthread_attr_t *attr);
  /* initializes a thread attribute pointed at by attr to
     their default values */

int pthread_attr_destroy(pthread_attr_t *attr);
  /* destroys a thread attribute pointed at by attr*/

int pthread_attr_setstacksize(pthread_attr_t *attr,
                              size_t stacksize);
  /* set the stack size of a thread attribute */

int pthread_attr_getstacksize(const pthread_attr_t *attr,
                              size_t *stacksize);
 /* get the stack size of a thread attribute */

int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);
  /* set the detach state of the attribute */

int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);
  /* get the detach state of the attribute */

int pthread_attr_setguardsize(pthread_attr_t *attr,
                              size_t guardsize);
  /* set the guard size of a thread attribute */

int pthread_attr_getguardsize(const pthread_attr_t *attr,
                              size_t *guardsize);
 /* get the guard size of a thread attribute */

 ...
/* other attributes associated with scheduling */

/* Unless otherwise stated, all the above integer functions
   returns 0 if successful, otherwise an error number is returned
*/
```

Every created thread has an associated identifier (of type pthread_t) that is unique with respect to other threads in the same process. A thread can obtain its own identifier (via pthread_self).

A thread becomes eligible for execution as soon as it is created by pthread_create; there is no equivalent to the Ada activation state or the Java new state. This function takes four pointer arguments: a thread identifier (returned by the call), a set of attributes, a function that represents the code of the thread when it executes, and the set of parameters to be passed to this function when it is called. The thread can terminate

Program 4.5 A C Real-Time POSIX interface to threads.

```
typedef ... pthread_t;  /* details not defined */
typedef ... pthread_attr_t;

int pthread_getconcurrency();
  /* returns the last set value of pthread_getconcurrency */

int pthread_setconcurrency(int level);
  /* sets the application's preferred thread concurrency level;
     returns the old level */

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
  /* create a new thread with the given attributes and call the
     given start_routine with the given argument */

int pthread_join(pthread_t thread, void **value_ptr);
  /* suspends the calling thread until the named thread has
     terminated, any returned values are pointed at by value_ptr */

void pthread_exit(void *value_ptr);
  /* terminate the calling thread and make the pointer value_ptr
     available to any joining thread */

int pthread_detach(pthread_t thread);
  /* the storage space associated with the given thread may be
     reclaimed when the thread terminates */

pthread_t pthread_self(void);
 /* return the thread id of the calling thread */

int pthread_equal(pthread_t t1, pthread_t t2);
  /* compare two thread ids
     return non 0 if equal, 0 otherwise */

int pthread_atfork(void (*prepare)(void), void(*parent)(void),
                   void (*child)(void));
  /* used for managing the resources shared by a multi-threaded
     program when a fork is performed */

/* Unless otherwise stated, all the above integer functions
   returns 0 if successful, otherwise an error number is returned
*/
```

normally by returning from its `start_routine` or by calling `pthread_exit` or by receiving a signal sent to it (see Section 7.5.1). It can also be aborted by the use of `pthread_cancel`. One thread can wait for another thread to terminate via the `pthread_join` function.

Finally, the activity of cleaning up after a thread's execution and reclaiming its storage is termed **detaching**. There are two ways to achieve this: by calling the `pthread_join` function and waiting until the thread terminates, or by setting the detached attribute of the thread (either at creation time or dynamically by calling the `pthread_detach` function). If the detached attribute is set, the thread is not joinable and its storage space may be reclaimed automatically when the thread terminates.

In many ways, the interface provided by C/Real-Time POSIX threads is similar to that which would be used by a compiler to interface to its run-time support systems. Indeed, the run-time support system for Ada may well be implemented using POSIX threads. The advantage of providing higher-level language abstractions (such as those of Ada and Java) is that it removes the possibility of errors in using the interface.

To illustrate the simple use of the C/Real-Time POSIX thread creation facility, the robot arm program is shown below.

```
#include <pthread.h>

pthread_attr_t attributes;
pthread_t xp, yp, zp;

typedef enum {xplane, yplane, zplane} dimension;

int new_setting(dimension D);
void move_arm(dimension D, int P);

void controller(dimension *dim) {
  int position, setting;

  position = 0;
  while (1) {
    move_arm(*dim, position);
    setting = new_setting(*dim);
    position = position + setting;
  }
  /* note, no call to pthread_exit, process does not terminate */
}

#include <stdlib.h>
int main() {
  dimension X, Y, Z;
  void *result;

  X = xplane,
  Y = yplane;
  Z = zplane;
  if(pthread_attr_init(&attributes) != 0)
    /* set default attributes */
    exit(EXIT_FAILURE);
  if(pthread_create(&xp, &attributes,
                    (void *)controller, &X) != 0)
    exit(EXIT_FAILURE);
  if(pthread_create(&yp, &attributes,
                    (void *)controller, &Y) != 0)
    exit(EXIT_FAILURE);
  if(pthread_create(&zp, &attributes,
                    (void *)controller, &Z) != 0)
    exit(EXIT_FAILURE);
  pthread_join(xp, (void **)&result);
  /* need to block main program */

  exit(EXIT_FAILURE);
  /* error exit, the program should not terminate */
}
```

A thread attribute object is created with the default attributes. Calls to `pthread_create` create each instance of the `controller` thread and pass a parameter indicating its domain of operation. If any errors are returned from the operating system, the program terminates by calling the `exit` routine. Note that a program which terminates with threads still executing results in those threads being terminated. Hence it is necessary for the main program to issue a `pthread_join` system call even though the threads do not terminate.

In Section 3.1.1, a style of handling the error returns from POSIX was given. It assumes that each call has a macro defined which tests the return value and calls, if appropriate, an error-handling routine. It is therefore possible to write the above code in the following more readable way:

```
int main() {
  dimension X, Y, Z;
  void *result;

  X = xplane,
  Y = yplane;
  Z = zplane;

  PTHREAD_ATTR_INIT(&attributes);

  PTHREAD_CREATE(&xp, &attributes, (void *)controller, &X);
  PTHREAD_CREATE(&yp, &attributes, (void *)controller, &Y);
  PTHREAD_CREATE(&zp, &attributes, (void *)controller, &Z);

  PTHREAD_JOIN(xp, (void **)&result);
}
```

Finally, it should be noted that conceptually forking a process (program) which contains multiple threads is not straightforward, as some threads in the process may hold resources or be executing system calls. The C/Real-Time POSIX standard specifies that the child process will only have one thread (see POSIX 1003 (Open Group/IEEE, 2004)). The `pthread_atfork` function allows a process to specify three functions that can be called to help the programmer manage such situations.

- `prepare` – this routine is called immediately *before* the fork is called.
- `parent` – this routine is called in the parent immediately after the fork has returned.
- `child` – this routine is called in the child immediately after the fork has returned.

Typically, the `prepare` will try to obtain all the shared locks (therefore, blocking the calling threads until the other threads have released the shared locks). The `parent` and `child` routine will typically release the locks.

4.7 Multiprocessor and distributed systems

Most concurrent programming languages usually attempt to define their semantics so that they can be implemented on single processor or multiprocessor system where there is access to global shared memory. In the multiple processor case, it is usual to assume that all processors in the system can execute all tasks. Hence, when a task is dispatched it can be dispatched to any of the available processors. This is sometimes called **global dispatching**. Hence during its lifetime, a task may migrate from processor to processor.

From a real-time perspective, predictability is a major concern. Fixing tasks to processors and not allowing them to migrate usually results in more predictable response times (see Section 11.14). Hence, some operating systems (for example Linux) provide an API that allows threads to be constrained to execute on a limited set of processors. This is usually called **processor affinity**.

Whilst tasks can be mapped to different processors in a distributed system, it is more usual to provide some other form of encapsulation method to represent the 'unit' of distribution. For example, processes, objects, partitions, agents and guardians have all been proposed as units of distribution. All these constructs provide well-defined interfaces which allow them to encapsulate local resources and provide remote access.

Hence, the production of an application to execute on a distributed system involves several steps which are not required when programs are produced for a single or multiprocessor platform.

- **Partitioning** is the activity of dividing the system into parts (units of distribution) suitable for placement onto the processing nodes of the target system.
- **Configuration** takes place when the partitioned parts of the program are associated with particular processing elements in the target system.
- **Allocation** covers the actual activity of turning the configured system into a collection of executable modules and downloading these to the processing elements of the target system.
- **Transparent execution** is the execution of the distributed software so that remote resources can be accessed in a manner which is independent of their location – usually using some form of message passing.
- **Reconfiguration** is the dynamic change to the location of a software component or resource.

Languages which have been designed explicitly to address distributed programming will provide linguistic support for at least the partitioning stage of system development. Some approaches will allow configuration information to be included in the program source, whereas others will provide a separate **configuration** language. Allocation and reconfiguration, typically, require support from the programming support environment and operating system.

The support that Ada, Java and C/Real-Time POSIX provide for multiprocessor and distributed systems is considered in the following subsections.

4.7.1 Ada

The Ada Reference Manual allows a program's implementation to be on a multiprocessor system. However, it provides no direct support that allows programmers to partition their tasks onto the processors in the given system. The following ARM quote illustrates this.

> NOTES 1 Concurrent task execution may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single physical processor. On the other hand, whenever an implementation can determine that the required semantic effects can be achieved when parts of the execution of a given task are performed by different physical processors acting in parallel, it may choose to perform them in this way. *ARM Section 9 par 11.*

This simply allows multiprocessor execution and also allows parallel execution of a single task if it can be achieved, in effect, 'as if executed sequentially'.

> In a multiprocessor system, a task can be on the ready queues of more than one processor. At the extreme, if several processors share the same set of ready tasks, the contents of their ready queues is identical, and so they can be viewed as sharing one ready queue, and can be implemented that way. Thus, the dispatching model covers multiprocessors where dispatching is implemented using a single ready queue, as well as those with separate dispatching domains. *ARM Section 10.1/2 par 15.*

This allows the full range of dispatching identified above. However, currently the only way that an implementation can provide the mechanisms to allow the programmer to set the processor affinity of their tasks is via implementation-defined pragmas, or non-standard library packages.

Ada defines a distributed system as an

> interconnection of one or more processing nodes (a system resource that has both computational and storage capabilities), and zero or more storage nodes (a system resource that has only storage capabilities, with the storage addressable by more than one processing nodes).

The Ada model for programming distributed systems specifies a **partition** as the unit of distribution. Partitions comprise aggregations of library units (separately compiled library packages or subprograms) that collectively may execute in a distributed target execution environment. The configuration of library units into partitions is not defined by the language; it is assumed that an implementation will provide this, along with facilities for allocation and, if necessary, reconfiguration.

Each partition resides at a single execution site where all its library units occupy the same logical address space. More than one partition may, however, reside on the same execution site.

Partitions may be either **active** or **passive**. The library units comprising an active partition reside and execute upon the same processing element. In contrast, library units comprising a passive partition reside at a storage element that is directly accessible to the nodes of different active partitions that reference them. This model ensures that active partitions cannot directly access variables in other active partitions. Variables can only be shared directly between active partitions by encapsulating them in a passive partition. Communication between active partitions is defined in the language to be via remote subprogram calls (however, an implementation may provide other communication mechanisms).

Categorization pragmas

To aid the construction of distributed programs, Ada distinguishes between different categories of library units, and imposes restrictions on these categories to maintain type consistency across the distributed program. The categories (some of these are useful in their own right, irrespective of whether the program is to be distributed) are designated by the following pragmas.

- **`Preelaborate`** – a preelaboratable library unit is one that can be elaborated without execution of code at run-time.
- **`Pure`** – pure packages are preelaboratable packages with further restrictions which enable them to be freely replicated in different active or passive partitions without introducing any type inconsistencies. These restrictions concern the declaration of objects and types; in particular, variables and named access types are not allowed unless they are within a subprogram, task unit or protected unit.
- **`Remote_Types`** – a `Remote_Types` package is a preelaboratable package that must not contain any variable declarations within the visible part.
- **`Shared_Passive`** – `Shared_Passive` library units are used for managing global data shared between active partitions. They are, therefore, configured on storage nodes in the distributed system.

4.7.2 Java

As with Ada, Java's semantics allow for the parallel execution of threads on a shared memory multiprocessor system and does not support processor affinity. However, the language at least allows the determination of the number of processors on which the application is executing (via the `availableProcessors` method in the `Runtime` class). The full meaning of the execution of Java threads on a shared memory multiprocessor system is defined via the Java Memory Model. This will be considered in Section 5.10.1.

There are essentially two ways in which to construct distributed Java applications.

(1) Execute Java programs on separate machines and use the Java networking facilities.

(2) Use remote objects.

Java networking

The two prominent network communication protocols in use today are UDP and TCP. The Java environment provides classes (in the `java.net` package) which allow easy access to them. The API to these protocols is via the `Socket` class (for the reliable TCP protocol) and the `DatagramSocket` class (for the UDP protocol). It is beyond the scope of this book to consider this approach in detail – see the Further Reading section at the end of this chapter for alternative sources of information.

Remote objects

Although Java provides a convenient way of accessing network protocols, these protocols are still complex and are a deterrent to writing distributed applications. Consequently, Java supports a distributed object communication model through the notion of **remote objects**. This will be discussed further in Section 6.8.4.

4.7.3 C/Real-Time POSIX

C/Real-Time POSIX defines the 'Scheduling Allocation Domain' as the set of processors on which an individual thread can be scheduled at any given time. C/Real-Time POSIX states that (Open Group/IEEE, 2004):

- 'For application threads with scheduling allocation domains of size equal to one, the scheduling rules defined for SCHED_FIFO and SCHED_RR shall be used.'
- 'For application threads with scheduling allocation domains of size greater than one, the rules defined for SCHED_FIFO, SCHED_RR, and SCHED_SPORADIC shall be used in an implementation-defined manner.'
- 'The choice of scheduling allocation domain size and the level of application control over scheduling allocation domains is implementation-defined. Conforming implementations may change the size of scheduling allocation domains and the binding of threads to scheduling allocation domains at any time.'

The details of the actual scheduling algorithms are of no concern here (they will be covered in Chapter 12). With this approach, there is no portable way to specify an affinity between threads and processors.

4.7.4 Processor affinity

From the above, it is clear that neither Ada, Java or C/Real-Time POSIX allows programmer control over the mapping of tasks to processors in a shared memory multiprocessor system. To give a flavour of what could be provided in future by these languages, consider the support that Linux provides.

Since Kernel version 2.5.8, Linux has provided support for multiprocessor systems (Linux Manual Page, 2006) via the notion of CPU affinity. Each process in the system can have its CPU affinity set according to a CPU affinity mask. A process's CPU affinity mask determines the set of CPUs on which it is eligible to run.

```
#include <sched.h>

int sched_setaffinity(pid_t pid,
  unsigned int cpusetsize, cpu_set_t *mask);

int sched_getaffinity(pid_t pid,
  unsigned int cpusetsize, cpu_set_t *mask);
```

```
void CPU_CLR(int cpu, cpu_set_t *set);

int CPU_ISSET(int cpu, cpu_set_t *set);

void CPU_SET(int cpu, cpu_set_t *set);

void CPU_ZERO(cpu_set_t *set);
```

A CPU affinity mask is represented by the `cpu_set_t` structure, a ‘CPU set’, pointed to by the mask. Four macros are provided to manipulate CPU sets. `CPU_ZERO` clears a set. `CPU_SET` and `CPU_CLR` respectively add and remove a given CPU from a set. `CPU_ISSET` tests to see if a CPU is part of the set. The first available CPU on the system corresponds to a CPU value of 0, the next CPU corresponds to a CPU value of 1, and so on. A constant `CPU_SETSIZE (1024)` specifies a value one greater than the maximum CPU number that can be stored in a CPU set.

The `sched_setaffinity` method sets the CPU affinity mask of the process, whose ID is `pid`, to the value specified by mask. If the process specified by `pid` is not currently running on one of the CPUs specified in mask, then that process is migrated to one of the CPUs specified in mask.

The `sched_getaffinity` method allows the current mask to be obtained.

The affinity mask is actually a per-thread attribute that can be adjusted independently for each of the threads in a thread group.

4.8 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 4.5 outlines this simple system: a task T takes readings from a set of thermocouples (via an analog-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-analog converter, DAC). Both T and P must communicate data to S, which presents measurements to an operator via a console. Note that P and T are active; 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, the 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.

(2) T, P and S are written in a sequential programming language (either as 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.

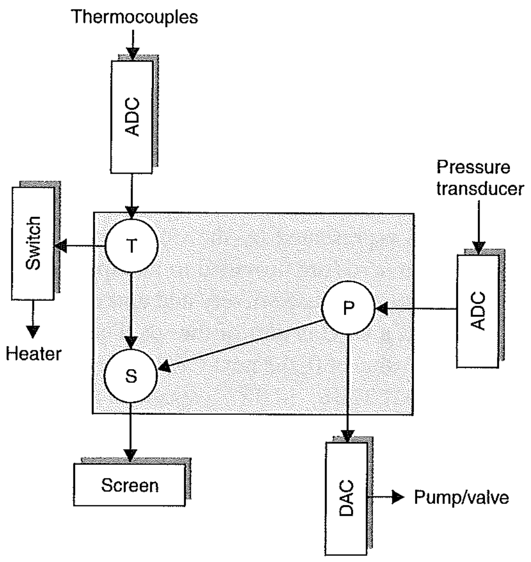

Figure 4.5 A simple embedded system.

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 new Integer range 10..500;
  type Pressure_Reading is new Integer range 0..750;
  type Heater_Setting is (On, Off);
  type Pressure_Setting is new Integer 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 console
  procedure Write(PR : Pressure_Reading); -- to console
end IO;

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

Sequential solution

A simple sequential control program could have the following structure:

```
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 console
    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 conversion 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 Controller is
  TR : Temp_Reading;
  PR : Pressure_Reading;
  HS : Heater_Setting;
  PS : Pressure_Setting;
  Ready_Temp, Ready_Pres : Boolean;
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 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 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. In a concurrent programming environment, this can be rectified by coding each system as a task.

Using operating system primitives

Consider a POSIX-like operating system which allows a new task/thread to be created and started by calling the following Ada subprogram:

```
package Operating_System_Interface is
  type Thread_ID is private;
  type Thread is access procedure; -- a pointer type

  function Create_Thread(Code : Thread) return 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);
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 sequential 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 tasks 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.

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.

Although an improvement, one major problem remains with this two-task solution. Both `Temp_Controller` and `Pressure_Controller` send data to the console, but the console is a resource that can only sensibly be accessed by one task at a time. In Figure 4.5, control over the console 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 complete definition of the required behaviour of `Screen_Controller`). This has transposed the problem from one of concurrent access to a passive resource to one of inter-task communication, or at least communication between a task and some other concurrency primitive. It is necessary for tasks `Temp_Controller` and `Pressure_Controller` to pass data to `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 programming languages, and are considered in the following chapters.

The concurrent version of this simple embedded system has a structure that is common in many control systems. Each task is iterative, it get its data at the start of each iteration, it processes the data and it writes its results at the end of each iteration. This example will be revisited throughout this book as various aspects of real-time systems are discussed.

4.9 Language-supported versus operating-system-supported concurrency

Although this book is focusing on concurrent real-time languages, it is clear that an alternative approach is to use a sequential language, like C, and a real-time operating system (such as one that conforms to the POSIX API).

There has been a long debate among 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) There are many different types of operating system; defining the concurrency in the language makes the program more portable.
(3) An embedded computer may not have any resident operating system available.

These arguments were clearly the ones which held the most weight with the designers of Ada and Java. 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) Operating system API standards, such as POSIX, have emerged and therefore programs are more portable.

The need to support multiple languages was one of the main reasons why the civil aircraft industry when developing its Integrated Modular Avionics programme opted for a standard applications–kernel interface (called APEX) supporting concurrency rather than adopting the Ada model of concurrency (ARINC AEE Committee, 1999). However, it should be noted that certain compiler optimizations may lead to race conditions if the compiler does not take into account potential concurrent execution of the program (Buhr, 1995; Boehm, 2005). This is particularly true for multiprocessor systems. As a consequence of this, even C++ is adding language-defined support for multithreading into the next version of its international standard.

Summary

The application domains of most real-time systems are inherently parallel. It follows that the inclusion of the notion of task/thread within a real-time programming language makes an enormous difference to the expressive power and ease of use of the language. These factors in turn contribute significantly to reducing the software construction costs whilst improving the reliability of the final system.

Without concurrency, the software must be constructed as a single control loop. The structure of this loop cannot retain the logical distinction between system components. It is particularly difficult to give task-oriented timing and reliability requirements without the notion of a task being visible in the code.

The use of a concurrent programming language is not, however, without its costs. In particular, it becomes necessary to use a run-time support system (or operating system) to manage the execution of the system tasks.

The behaviour of a task is best described in terms of states. In this chapter, the following states are discussed:

- non-existing
- created
- initialized
- executable
- waiting dependent termination
- waiting child initialization
- terminated.

Within concurrent programming languages, there are a number of variations in the task model adopted. These variations can be analysed under six headings.

(1) **structure** – static or dynamic task model;
(2) **level** – top-level tasks only (flat) or multilevel (nested);
(3) **initialization** – with or without parameter passing;
(4) **granularity** – fine or coarse grain;
(5) **termination** –
   - natural
   - suicide
   - aborted
   - untrapped error
   - never
   - when no longer needed;

(6) **representation** – fork/join, cobegin, explicit task declarations.

Ada and Java provide a dynamic model with support for nested tasks and a range of termination options. C/Real-Time POSIX allows dynamic threads to be created with a flat structure; threads must explicitly terminate or be killed.

Although Ada, Real-Time Java and C/Real-Time POSIX all support multiprocessor implementations, they do not provide mechanisms to set the processor affinity of a task.

Further reading

Ben-Ari, M. (2005) *Principles of Concurrent and Distributed Programming*. New York: Prentice Hall.

Burns, A. and Wellings, A. J. (2007) *Concurrent and Real-time Programming in Ada*. Cambridge: Cambridge University Press.

Butenhof, D. R. (1997) *Programming With POSIX Threads*. Reading, MA: Addison-Wesley.

Goetz, B. (2006) *Java: Concurrency in Practice*. Reading, MA: Addison-Wesley.

Hyde, P. (1999) *Java Thread Programming*. Indianapolis, IN: Sams Publishing.

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

Oaks, A. and Wong, H. (1997) *Java Threads*. Sebastopol, CA: O'Reilly.

Nichols, B., Buttlar, D. and Farrell, J. (1996) *POSIX Threads Programming*. Sebastopol, CA: O'Reilly.

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

Exercises

4.1 A particular operating system has a system call (*RunConcurrently*) which takes an unconstrained array. Each element of the array is a pointer to a parameterless procedure. The system call executes all the procedures concurrently and returns when all have terminated. Show how this system call can be implemented in Ada using the Ada tasking facilities. Assume that the operating system and the application run in the same address space.

4.2 Write Ada code to create an array of tasks where each task has a parameter which indicates its position in the array.

4.3 Show how a `cobegin` can be implemented in Ada.

4.4 Can the `fork` and `join` method of task creation be implemented in Ada without using intertask communication?

4.5 How many POSIX processes are created with the following procedure?

```
for(i=0; i<=10;i++) {
  fork();
}
```

4.6 Rewrite the simple embedded system illustrated in Section 4.8 in Java and C/Real-Time POSIX.

4.7 If a multithread process executes a POSIX-like fork system call, how many threads should the created process contain?

4.8 Show, using concurrent tasks, the structure of a program to control access to a simple car park. Assume that the car park has a single entrance and a single exit barrier, and a full sign.

4.9 Explain, with the help of the following program, the interactions between Ada's rules for task termination and its exception propagation model. Include a consideration of the program's behaviour (output) for initial values of the variable `C` of 2, 1 and 0.

```
with Ada.Text_Io; use Ada.Text_Io;
procedure Main is
  task A;

  task body A is
    C : Positive := Some_Integer_Value;
    procedure P(D : Integer) is
      task T;
      A : Integer;
      task body T is
      begin
        delay 10.0;
        Put("T Finished"); New_Line;
      end T;
    begin
      Put("P Started"); New_Line;
      A := 42/D;
      Put("P Finished"); New_Line;
    end P;

  begin
    Put("A Started"); New_Line;
    P(C-1);
    Put("A Finished"); New_Line;
  end A;

begin
  Put("Main Procedure Started"); New_Line;
exception
  when others =>
    Put("Main Procedure Failed"); New_Line;
end Main;
```

4.10 For every task in the following Ada program indicate its parent and guardian (master) and if appropriate its children and dependants. Also indicate the dependants of the `Main` and `Hierarchy` procedures.

```
procedure Main is
  procedure Hierarchy is
    task A;
    task type B;

    type Pb is access B;
    Pointerb : Pb;

    task body A is
      task C;
      task D;
      task body C is
      begin
        -- sequence of statements including
        Pointerb := new B;
      end C;
      task body D is
        Another_Pointerb : Pb;
      begin
        -- sequence of statements including
        Another_Pointerb := new B;
      end D;
    begin
      -- sequence of statements
    end A;

    task body B is
    begin
      -- sequence of statements
    end B;

  begin
    -- sequence of statements
  end Hierarchy;

begin
  -- sequence of statements
end Main;
```

4.11 To what extent can Figure 4.2 be used to represent the state transition diagram of (a) C/Real-Time POSIX pthreads and (b) Java threads?

4.12 Given the following:

```
  public class Calculate implements Runnable

  {

    public void run()
    {
      /* long calculation */
    }
  }

Calculate MyCalculation = new Calculate();
```

what is the difference between:

```
  MyCalculation.run();
```

and

```
  new Thread(MyCalculation).start();
```

4.13 Explain how a Java thread can be protected against being destroyed by an arbitrary thread.