4. In a network of N nodes. how many messages are exchanged for each execution of the critical section?

5. (Lamport) Prove the correctness of the following version of the algorithm which explicitly replies to each request rather than deferring a reply which might be interpreted as a lost message. How many messages are exchanged for each execution of the critical section?

   - A node receiving a *Request* message replies with its current ticket number.
   - When a node completes its critical section, it sends a *Release* message with a new (higher) ticket number to all other nodes.
   - A node may enter the critical section if it has received some message with a higher ticket number than its own from every other node.

6. Suppose that there exists an upper bound on message transmission time in the system. Rather than send *Reply* messages, we can send a *Deferred* message and let the absence of a message implicitly indicate a favorable *Reply*. How does this affect the number of messages exchanged?

Chapter 12

Distributed Termination

12.1 Introduction

A sequential program terminates when it has executed its last statement. A concurrent program terminates when all its sequential processes have terminated. Many concurrent programs contain processes executing infinite loops and do not terminate as such. However, a concept of termination does exist even in this case when the set of non-terminating processes are all waiting for communication from each other. If the blocking is not intentional, this is called deadlock.

In a centralized implementation of concurrent programming, it is trivial to detect termination or deadlock – the queue of ready processes is empty. In a distributed system, termination detection is not simple because there is no way to take a 'snapshot' of the global state at an instant of time and then examine it. Instead we will describe algorithms that collect information from a set of processes over a period of time and then decide whether termination has occurred or not.

We are given a set of process connected by directional communications channels. The channels are error-free and deliver the messages in the order sent. The graph is not necessarily acyclic and bi-directional channels can be modeled by two channels so the model is not very restrictive. We do assume the existence of a unique *source process* which has no incoming edges and from which every process is accessible along some path (Figure 12.1).

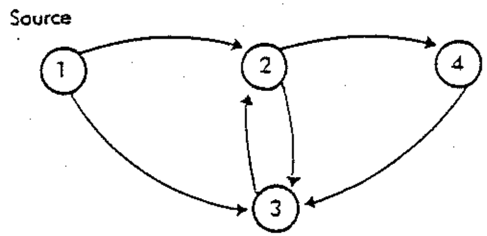


Figure 12.1 Process and channel graph

The computation is started when the source process sends a message along each of its outgoing edges. When a process has received its first message, it may commence computation and may send messages along its outgoing edges and receive additional messages from its incoming edges. A process may terminate when its computation is finished. It will send no more messages, but if it receives more messages, it may be restarted. We want to design a signaling scheme to be superimposed on top of the message communications so that when *all* processes have terminated, the source process will eventually be informed.

The signaling will be done on special channels, one for each of the message channels, but pointed in the opposite direction (the dashed lines in Figure 12.2). We will use the term *signals* to distinguish these data from the messages of the main computation. Whatever the status of a process in terms of its computation – which may be terminated, or its message communications channels – which may be blocked, we assume that it is always able to receive, process and send signals.

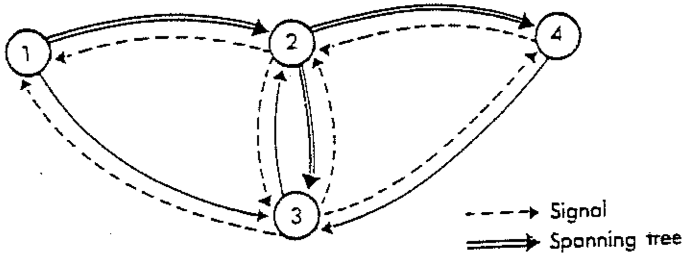


Figure 12.2 Signaling channels and spanning tree

The algorithms will be described within the framework of the Ada program shown in Figure 12.3. There will be a constant number N of nodes each of which runs a task communicating with the outside world. The Main_Process doing the computation will be run as a subtask of the Node task. The main program is used only for initialization – configuration of the channels and initiation of the nodes. The details are not important here and can be found in Appendix D.

Within each node are declared global variables (Figure 12.4) used to maintain the status of the communications channels and to exchange information between the main process and the underlying communications processing. The variable I is initialized to the node ID and remains constant thereafter. The topology of the the channels is described by the variables Incoming and Outgoing within each node. Outgoing(J).Exists is true if there is a channel from I to J and conversely for Incoming. These variables will contain other status fields that differ from one algorithm to the next.

The variables Received_ID and Received_Data are used to record the information coming from the channels in variables local to the node so that the rendezvous can be completed as soon as possible to free the calling node. Once the rendezvous is completed, the messages and signals are processed. The semaphore

```
with Semaphore_Package; use Semaphore_Package;
procedure Distributed_Algorithm is
  N: constant Integer := 4;

  task type Nodes is
    entry Message(Data: Integer; ID: Integer);
    entry Signal(ID: Integer);
  end Nodes;
  Node: array(1..N) of Nodes;

  task body Nodes is
    -- Global variables

    task Main_Process;
    task body Main_Process is ... ;

  begin
    loop
      select
        accept Message(Data: Integer; ID: Integer) do
          Received_ID := ID;
          Received_Data := Data;
        end Message;
        -- Message processing
      or
        accept Signal(ID: Integer) do
          Received_ID := ID;
        end Signal;
        -- Signal processing
      end select;
    end loop;
  end Nodes;
begin
  -- Configure nodes and initiate processes
end Distributed_Algorithm;
```

Figure 12.3 Framework for distributed algorithms

S is used to protect global variables that may be used by both the Main_Process and the Node process.

12.2 The Dijkstra–Scholten Algorithm

If the process graph were structured as a tree, a simple algorithm would suffice to determine termination. A leaf process would decide that it has terminated and send a signal to its parent process. When a process has received notification from all its child processes, it sends a signal to its parent, and so on until the source process is informed. The Ada language has a scheme for termination defined within the language that is based the fact that task dependencies form a tree.

```
type Edge is
  record
    Exists:  Boolean := False;
    -- Other fields
  end record;

Incoming: array(1..N) of Edge;
Outgoing: array(1..N) of Edge;

I:              Integer;
S:              Binary_Semaphore := Init(1);
Received_ID:    Integer;
Received_Data:  Integer;
```

Figure 12.4 Global variables for distributed algorithms

The algorithm can be extended to acyclic directed process graphs as follows. An additional field `Deficit` is added to each edge. On incoming edges this will denote the difference between the number of messages received and the number of signals sent in reply. When a node wishes to terminate, it sends enough signals to ensure that the deficit is zero on each incoming edge. Then it waits until it has received signals from outgoing edges reducing their deficits to zero. As a simplification, a single counter `N_Signals` is sufficient since the requirement is only that all outgoing deficits are zero and no part of the algorithm depends on the identity of the channel from which the signals are received.

If cycles are allowed, the above algorithm fails because we have no leaf processes that can decide to terminate without consulting other processes. The Dijkstra–Scholten (DS) algorithm solves this problem by generating a *spanning tree* during the exchange of messages and then using the tree to order the signals. A spanning tree (Figure 12.2) is a subset of the edges of the graph that forms a tree such that every node is incident with an edge in the tree. The tree will be directed with the source process at its root.

A variable `First_Edge` is added to each node which implicitly creates a spanning tree: the first time that a message is received in a process, the incoming edge upon which the message is received will be added to the tree. Termination requires three steps:

1. Send signals on all incoming edges *except* `First_Edge`.
2. Wait for signals from all outgoing edges.
3. Send signals on `First_Edge`.

The algorithm is shown in Figures 12.5–12.8. Figure 12.6 shows additional processing that must be done by the `Node` process when it receives a message or a signal. The first time that a message is received along any edge `First_Edge` is set. In any case, the deficit is incremented. If a signal is received, the number of outstanding signals is decremented.

If `Main_Process` wishes to send a message, it executes the procedure shown in Figure 12.7 incrementing the signal count.

```
type Edge is
  record
    Exists:  Boolean := False;
    Deficit: Integer := 0;
  end record;

N_Signals:  Integer := 0;
First_Edge: Integer := 0;
```

Figure 12.5 Global variables for termination (DS)

```
-- Message processing
  if First_Edge = 0 then
    First_Edge := Received_ID;
  end if;
  Wait(S);
  Incoming(Received_ID).Deficit :=
    Incoming(Received_ID).Deficit + 1;
  Signal(S);

-- Signal processing
  Wait(S);
  N_Signals := N_Signals - 1;
  Signal(S);
```

Figure 12.6 Receiving messages and signals (DS)

The essence of the algorithm is in the function `Decide_to_Terminate` (Figure 12.8) which `Main_Process` executes when it has no more processing to do. Signals are sent on incoming edges (except `First_Edge`). If there are no outstanding signals to be received on outgoing edges, signals are sent on `First_Edge` and we can terminate. The coding of the function is somewhat convoluted by the desire to protect variables that are global to both processes in the node while at the same time not calling another node from within a critical section since that could lead to deadlock.

The creation of the spanning tree may be traced in the example in Figure 12.2, where the double lines define the spanning tree. This tree would be produced by node 1 initiating the computation by sending a message to node 2 which would relay it to nodes 3 and 4. The tree is not unique and a different sequence of

```
procedure Send_Message(Data: Integer; ID: Integer) is
begin
  Wait(S);
  N_Signals := N_Signals + 1;
  Signal(S);
  Node(ID).Message(Data, I);
end Send_Message;
```

Figure 12.7 Sending messages (DS)

```
function Decide_to_Terminate return Boolean is
  procedure Send_Signals(ID: Integer) is
  begin
    while Incoming(ID).Deficit > 0 loop
      Incoming(ID).Deficit := Incoming(ID).Deficit - 1;
      Signal(S);
      Node(ID).Signal(I);
      Wait(S);
    end loop;
  end Send_Signals;

  begin
    for J in 1..N loop
      if  J /= First_Edge then
        Wait(S);
        Send_Signals(J);
        Signal(S);
      end if;
    end loop;

    Wait(S);
    if N_Signals = 0 then
      if I /= 1 and First_Edge /= 0 then
        Send_Signals(First_Edge);
        First_Edge := 0;
      end if;
      Signal(S);
      return True;
    else
      Signal(S);
      return False;
    end if;
  end Decide_to_Terminate;
```

Figure 12.8 Algorithm for termination (DS)

messages would build a different tree. For example, if node 4 sent message to 3 and it arrived at 3 before the message from 2 then the edge 4 — 3 would be in the spanning tree instead of 2 → 3.

Note that we do not wait for outstanding signals within the function. The reason is that additional *messages* may be received on the incoming edges which will cause the computation in Main_Process to be restarted. The final state of this algorithm is that source process reports termination of the system while the other processes are quiescent in the sense that no node is computing within Main_Process. However, the underlying communications process in Node has no way of knowing whether the entire system has terminated or whether only this node is temporarily without work until a new message arrives. Thus it must continually check for incoming messages. In the next section, we will modify the algorithm to enable all processes to terminate.

The correctness properties of this algorithm are:

Safety If the program decides that termination has occurred, then all processes are quiescent.

Liveness If all processes become quiescent then eventually the program decides to terminate.

For each node, let $D = \sum Deficit$ where the sum is taken over all incoming edges. We leave as an exercise the proof of the following theorem:

Theorem 12.2.1 *The following formulas are invariants in each node:*

$$D \geq 0 \quad (12.1)$$

$$N\_Signals \geq 0 \quad (12.2)$$

$$D > 0 \vee N\_Signals = 0 \quad (12.3)$$

Define an *engaged* process as one that has received a message and has not yet become quiescent. Note that processes may become engaged several times and that processes that are disengaged continue to send and receive signals. Define an *engagement* edge as an edge that is a First_Edge for some node.

Theorem 12.2.2 *The following statements are invariant:*

1. *An engagement edge has non-zero deficit.*
2. *The engagement edges form a tree.*
3. *Every engaged node is reachable from the source node by a directed path formed from engagement edges.*

The program announces termination only if the tree is degenerate and consists only of the source node. Safety of the algorithm follows since the theorem implies that all other nodes are not engaged.

Liveness follows by induction on the height of the tree. If a leaf terminates, it will signal First_Edge because it is not waiting for signals on any outgoing edge. For other nodes, the result follows using the termination of the subtrees as an inductive hypothesis.

12.3 Termination using Markers

As noted in the previous section, the main process cannot assume that if it is quiescent it may terminate. In effect, the communications task is buffering the incoming messages. With buffered channels, we have to ensure that the channels are empty before terminating. On a distributed system, this can be difficult. In Figure 12.9(a) process P2 will note that the channel is empty and may decide to terminate. Then process P1 sends the message (Figure 12.9(b)) and also decides to terminate since it has nothing left to do. The outstanding message which should have caused P2 to continue execution is lost.

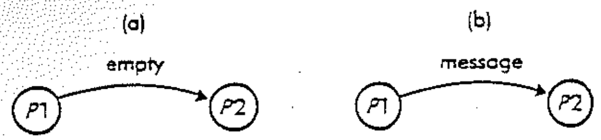


Figure 12.9 Disappearing message

To solve this problem, we must be able to account for all messages so that an empty message channel can be correctly identified. The following termination with marking (TM) algorithm sends special *marker* messages to indicate that a channel has become empty. The source process will initiate checking for termination by sending a marker on its outgoing edges. When a process decides to terminate, it first waits for markers on all incoming edges (if they have not already arrived) and then propagates a copy of the marker message along all outgoing edges. Once a process has received markers from all incoming edges, it can proceed as before: it signals all incoming edges except the first one and waits for signals on the outgoing edges before signaling the first edge.

The TM algorithm is shown in Figures 12.10–12.13. The marker is denoted by a negative integer message. An additional simplification has been made in this algorithm by maintaining a boolean variable for `Active` channels instead of computing non-zero deficits. Incoming edges have an additional field denoting whether a marker has been received or not.

```
type Edge is
  record
    Exists:           Boolean := False;
    Active:           Boolean := False;
    Marker_Received:  Boolean := False;
  end record;
```

Figure 12.10 Global variables for TM algorithm

Message and signal processing (Figures 12.11 and 12.12) are similar to the previous algorithm except for the use of the `Active` field rather than deficits. The modifications to `Decide_to_Terminate` are immediate and left as an exercise.

The main difference is in `Main_Process`, a fragment of which is shown in Figure 12.13 (the complete listing is given in Appendix D). We do not even attempt to terminate before receiving a marker from `First_Edge`. Then markers are sent to all outgoing edges and we wait until markers have been received on all incoming edges. Only now are we assured that the channels are empty and we can commence the sending and receiving signals to collapse the spanning tree.

```
-- Message processing
 if Received_Data < 0 then
   Incoming(Received_ID).Marker_Received := True;
 else
   if First_Edge = 0 then
     First_Edge := Received_ID;
   end if;
 end if;
 if not Incoming(Received_ID).Active then
   Incoming(Received_ID).Active := True;
 end if;

-- Signal processing
 Outgoing(Received_ID).Active := False;
 Wait(S);
 N_Signals := N_Signals - 1;
 Signal(S);
```

Figure 12.11 Receiving messages and signals (TM)

```
procedure Send_Message(Data: Integer; ID: Integer) is
begin
  if not Outgoing(ID).Active then
    Outgoing(ID).Active := True;
    Wait(S);
    N_Signals := N_Signals + 1;
    Signal(S);
  end if;
  Node(ID).Message(Data, I);
end Send_Message;
```

Figure 12.12 Sending messages (TM)

12.4 Snapshots

The TM algorithm is a special case of a more general algorithm that can capture the global state of a system - a *distributed snapshot*. The state of a node will be

```
loop
   exit when Incoming(First_Edge).Marker_Received;
end loop;

-- send markers to all outgoing edges

loop
  exit when Markers_Received;
end loop;

loop
  exit when Decide_to_Terminate;
end loop;
```

Figure 12.13 Main process (TM)

defined as the sequence of messages that have been sent and received along all channels incident with the node. The state of a channel will be the sequence of messages transmitted to the channel but not yet delivered.

The source process initiates the algorithm by recording its state and sending a marker on outgoing channels before any more messages are sent. Upon the first receipt of a marker. a receiving process records its state, records the state of that channel as empty and propagates the marker. At any time after recording its state, if a process receives a marker from another incoming edge, it records the state of that channel as the sequence of messages received between the state it recorded and the receipt of the marker. Figure 12.14 shows the system immediately after

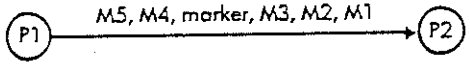


Figure 12.14 Distributed snapshots

P2 has recorded its state. Messages $M1, M2, M3$ received before the marker, are determined by P2 to be part of the state of the channel for this snapshot. Messages $M4, M5$ received after the marker, were sent after P1 recorded its state and thus are not part of the state of the channel.

The algorithm is very flexible and only requires that every node be reachable from some node that spontaneously decides to record the state. In fact. several nodes could concurrently decide to record the state and the algorithm would still succeed.

An implementation of the snapshot algorithm is shown in Figures 12.15–12.17. The data structures are shown in Figure 12.15. Last_Message keeps track of the

```
type Edge is
  record
    Exists:         Boolean := False;
    Last_Message:   Integer := 0;
    Recorded_State: Integer := 0;
    Marker_State:   Integer := -1;
  end record;

State_Recorded:  Boolean := False;
```

Figure 12.15 Global variables for snapshot algorithm

number of the last message received or sent on that edge. The current state of a node is defined as the values of the Last_Message field for all incoming and outgoing edges. To record the state of a node. we record the current value of Last_Message in the field Recorded_State.

Recording the state of a channel is the resposibility of the receiving process. For each incoming edge. an additional field Marker_State records the value of Last_Message when the marker is received. The state of the channel is the difference between Marker_State and Recorded_State. If the state of the node is recorded *before* a marker has been received on some channel, Recorded_State may be less than Marker_State and their difference denotes messages that were sent before the transmitting node recorded its state but received only after the target node recorded its state. Message reception is shown in Figure 12.16. Receipt of a message causes Last_Message to be updated. The first receipt of a marker causes Marker_State to be recorded. A negative value is used as a flag to indicate that a marker has yet to be received. If the state has not yet been recorded, Record_State is called.

```
-- Message processing
  if Received_Data < 0 then
    if Incoming(Received_ID).Marker_State < 0 then
      Incoming(Received_ID).Marker_State :=
        Incoming(Received_ID).Last_Message;
    if not State_Recorded then Record_State; end if;
    end if;
  else
    Wait(S);
    Incoming(Received_ID).Last_Message := Received_Data;
    Signal(S);
  end if;
```

Figure 12.16 Receiving messages (snapshot)

Sending a message is similar (Figure 12.17). A message causes Last_Message to be updated and a marker causes Recorded_State to be updated.

```
procedure Send_Message(Data: Integer; ID: Integer) is
begin
  Wait(S);
  Outgoing(ID).Last_Message := Data;
  Signal(S);
  Node(ID).Message(Data, I);
end Send_Message;

procedure Send_Markers is
begin
  for J in All_Outgoing_Edges  loop
    Wait(S);
    Outgoing(J).Recorded_State := Outgoing(J).Last_Message;
    Signal(S);
    Node(J).Message(-1, I);
  end loop;
end Send_Markers;
```

Figure 12.17 Sending messages (snapshot)

Record_State (Figure 12.18) may be called by either the main process when it spontaneously records its state or by accept Message upon receipt of a message.

It records the state of the incoming edges and then sends markers on all outgoing edges which will cause the state of these edges to be recorded.

```
procedure Record_State is
begin
    Wait(S);
    for J in 1..N loop
      if Incoming(J).Exists then
        Incoming(J).Recorded_State := Incoming(J).Last_Message;
      end if;
    end loop;
    Signal(S);
    Send_Markers;
    State_Recorded := True;
end Record_State;
```

Figure 12.18 Recording the state (snapshot)

The snapshot algorithm was run on the graph of Figure 12.1 where nine messages were sent along each channel. Node 1 decided to record the state after sending six messages and node 4 independently decided to record the state after sending three messages. Upon termination, the results as collected from each node are shown in Figure 12.19. Node 2 sent four messages on each of the two outgoing channels to nodes 3 and 4. Node 3 received three messages before recording its state leaving only the fourth message to be associated with the channel. Node 4 received messages 1 through 2 and recorded that messages 3 and 4 were in transit. The reader should check the snapshot for consistency – each message sent was either received at the target node or recorded as being in transit in the channel.

12.5 Further Reading

The Dijkstra–Scholten algorithm is from [DS80]. Termination by marking uses ideas from [MC82] and [CL85]. The latter paper is the source of the snapshot algorithm.

12.6 Exercises

1. How many spanning trees are there for the example in Figure 12.1? Describe execution sequences to construct each one.
2. Prove theorems 12.2.1 and 12.2.2.
3. Prove the liveness of the DS algorithm.
4. Suppose that an internal node in the DS algorithm terminates if Decide_to_Terminate is true. Create a scenario demonstrating incorrect behavior of the algorithm caused by a message sent but never received.

```
Node  1
    Outgoing channels
       2 sent 1.. 6
       3 sent 1.. 6
    Incoming  channels
Node  2
    Outgoing channels
       3 sent 1.. 4
       4 sent 1.. 4
    Incoming  channels
       1 received 1.. 4 stored  5.. 6
       3 received 1.. 8
Node  3
    Outgoing channels
       2 sent 1.. 8
    Incoming  channels
       1 received 1.. 3 stored  4..  6
       2 received 1.. 3 stored  4..  4
       4 received 1.. 3
Node  4
    Outgoing channels
       3 sent 1.. 3
    Incoming  channels
       2 received 1.. 2 stored  3..  4
```

Figure 12.19 An example of a snapshot

5. What changes need to be made to Decide_to_Terminate of the DS algorithm to use it in the TM algorithm?
6. In the TM algorithm. what is the purpose of the Active field in outgoing edges?
7. Can a Main_Process of the TM algorithm initiate termination upon receipt of a marker from any incoming edge or does it have to be First_Edge?
8. Consider an algorithm like the DS algorithm but using Active fields like the TM algorithm rather than deficits. What could happen? (*Hint:* consider the scenario in Figure 12.20.)
9. Can Recorded_State equal Last_Message even if the state is spontaneously recorded before a marker has been received?
10. Describe a scenario demonstrating a bug in the implementation of the snapshot algorithm. (*Hint:* If Record_State is called by accept Message. the main process could interleave calls to Send_Message between recording the state and the actual transmission of the marker.)
11. Create extreme scenarios for the snapshot algorithm: one with all channels empty and one with all channels full.

- Node 2 sends message to node 4.
- Node 4 receives message from 2.
- Node 2 sends message to node 4.
- Node 4 sends signal to 2.
- Node 4 receives message from 2.
- Node 2 receives signal from 4.
- Node 4 sends signal to 2.
- Node 2 receives signal from 4.

Figure 12.20 Scenario for DS with Active fields

Chapter 13

The Byzantine Generals Problem

13.1 Introduction

Distributed processing is used not only to create faster systems by exploiting parallelism but also to improve reliability by replicating a computation in several processors. Such a system attempts to be *fault tolerant*, that is to continue to produce correct, or at least reasonable, results even if some components fail. A distributed system is not automatically fault tolerant. Our algorithm for distributed mutual exclusion requires the co-operation of all the processes and will deadlock if one of them fails.

A typical architecture for a fault-tolerant system is shown in Figure 13.1(a). The data from input sensors are replicated along a bus. Several processors each compute the required output. Special circuits compare the values using majority voting and control the outputs. If only one processor malfunctions, it will not affect the output. The computers will be interconnected so that the faulty processor can identified and the system operator notified. In very critical systems like flight controllers, the input sensors and output controllers will also be replicated as shown in Figure 13.1(b).

There are several serious difficulties that must be overcome in designing a fault-tolerant system:

1. When the input sensors are replicated, they may not all give *exactly* the same data. Voting on the outcome is no longer a trivial comparison of two digital values. Even if the difference is physically negligible, the computational algorithms must be designed so that neighboring input values give answers that are near each other.
2. A faulty input sensor or processor may not act 'nicely'. They may produce spurious data or values that are totally out of the range considered by the algorithms.
3. Even if the hardware is not faulty, if all processors are using the same software then system is not fault-tolerant of software bugs. If several different programs are used, they may give slightly different values on the same data. Worse, different programmers are prone to make the same misinterpretations of the program specifications.

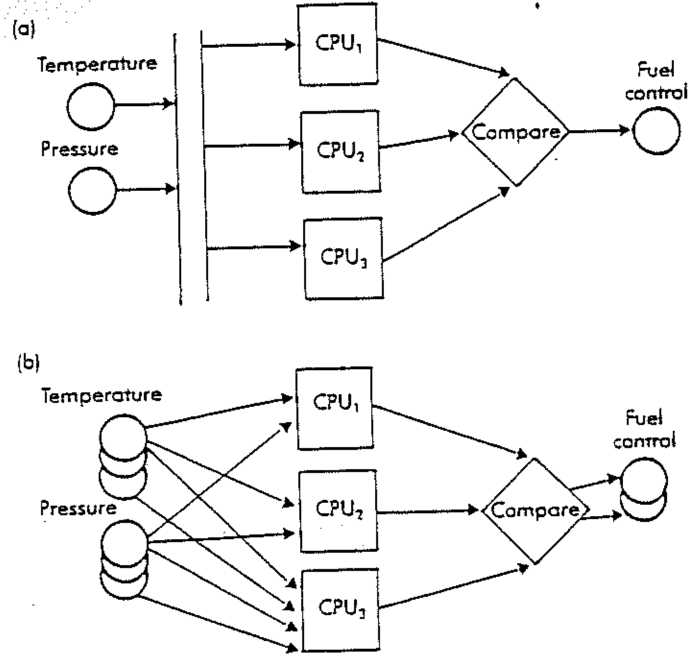


Figure 13.1 Architectures for fault-tolerant systems

The Byzantine Generals problem is an abstraction of one problem encountered in designing reliable systems. As usual we define a specific model and discuss the existence (or non-existence) of algorithms to solve the problem in this model. The applicability of the algorithm depends on the applicability of the model and the willingness of the designer to pay for the overhead of the solution. Solutions under other models can be found in the research literature.

13.2 Description of the Problem

In some of our solutions to the mutual exclusion problem, we allowed a process to die gracefully – it was supposed to terminate in its non-critical section after resetting the protocol variables to a non-interfering value. We indicated that the solutions fail if a process is allowed to fail in its critical section and this is obviously true if a process can assign arbitrary values to the protocol variables. The Byzantine Generals problem considers the situation where faulty processes are actively 'traitorous' and can send any message to the other processes.

A set of units of the Byzantine army is preparing for action against an enemy. Each unit is commanded by a general and these generals communicate with each

other by sending messages over telephone lines. The properties of the telephones are as follows:

1. The lines are error free. More than that, no general can interfere with a telephone line connecting two other generals.
2. The lines are point-to-point like occam channels and thus the sender is unequivocally identified to the receiver.
3. The telephone operators are in constant contact so that the absence of a message means that the sender did not send one and not that the enemy cut the line.

The purpose of these restrictions is to initially limit the problem to coping with the contents of the messages and not with other forms of faults.

The generals must agree on a course of action. Let us assume that this can be described by deciding between simple pre-arranged alternatives like: 'attack' or 'retreat', or 'attack on left' or 'attack on right'. The algorithm must satisfy the following two properties:

1. All loyal generals must take the same decision.
2. Every loyal general must base his decision on correct information from every other loyal general.

The first property means that the enemy cannot divide the Byzantine army simply by having a few traitors issue messages. The second property means that the algorithm used by the loyal generals cannot be random but has to be based on what the other loyal generals actually think.

To see the motivation behind this abstraction. consider an aircraft collision avoidance system with four processors. To ensure that the system is not accidentally engaged, three processors must agree on a course of action. If a collision is impending, the processors should be able to agree to automatically turn the aircraft left or right depending on the position of the other aircraft. It should not be possible for one faulty processor to send messages to the functioning processors that would cause them to split two against one on the proposed course of action and thus fail to engage the system. Furthermore. if all three processors conclude that a right turn is the best maneuver. they should agree to turn right. The faulty processor is not allowed even to have them all agree to turn left. Only if the computation is so close that either left or right would be correct. could the faulty processor influence the decision.

If more than two-thirds of the generals are loyal, there is a solution, that is, an algorithm that will cause them all to take the same action regardless of what messages are received from the traitorous generals. However, if one-third or more of the generals are traitorous, it can be shown that there is no algorithm to solve the problem. In the case of one traitorous general, there is a solution for four generals and none for three. We will describe the algorithm and the impossibility proof for this particular case in the next two sections.

13.3 Algorithm for Four Generals

There are three loyal generals and one traitor. We assume that all the loyal generals follow the same algorithm, in particular that they inform every other general of their proposed plan of action. The communications channel is assumed to be error-free. The absence of a message is detectable, so we assume that every general, even the traitorous ones, sends the messages required by the following algorithm.

Let one of the generals initiate the algorithm. This general will be called the *commander* and the other generals will be called *lieutenants*. The algorithm is as follows:

1. The commander sends his decision.
2. A lieutenant relays the commander's decision to every other lieutenant.
3. Upon receiving both the direct message from the commander and the relayed messages from the other two lieutenants, the lieutenant decides by majority voting on the three messages.

There are two cases to be considered – either the commander is loyal (Figure 13.2) or he is a traitor (Figure 13.3). If the commander is loyal, his decision to attack is presumed to be the correct decision and we must show that both loyal lieutenants make the same decision. In the figure we can see that a loyal lieutenant receives *attack* from the commander and *also* from the other loyal lieutenant. The traitor may send either *attack* or *retreat*, but by majority voting it cannot affect the outcome.

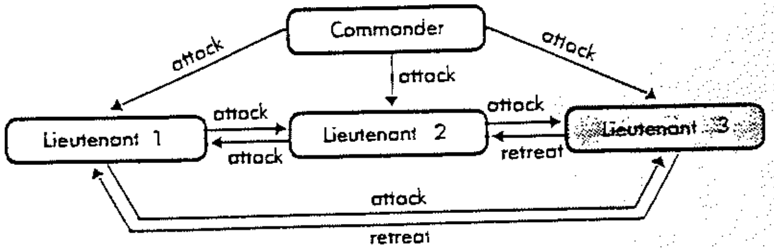


Figure 13.2 Algorithm with loyal commander

If the commander is a traitor, it does not matter what he decides to do as long as the three lieutenants agree on a course of action. Since all lieutenants are loyal, they all correctly relay the commander's message. Thus all three lieutenants receive exactly the same three messages. Thus whatever algorithm they use, they will all take the same decision. In the figure we have shown the commander sending one *attack* message, on *retreat* message and refraining from sending any message to lieutenant 3. If the algorithm agreed upon treats the absence of a message as *attack*, all three lieutenants will attack. Obviously, if the commander sent two or three lieutenants the same message, the majority voting would lead to agreement.

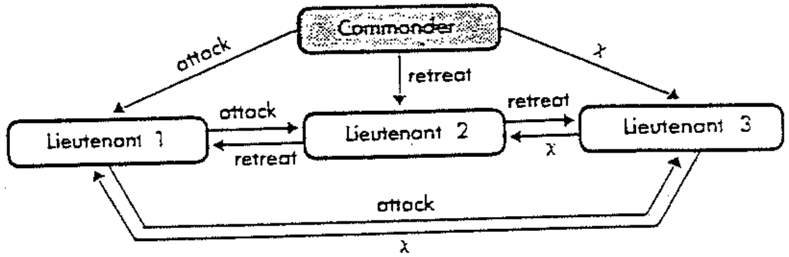


Figure 13.3 Algorithm with traitorous commander

13.4 Impossibility for Three Generals

The above solution does not work for three generals, one of whom is a traitor and in fact, it can be proven that there is no such solution. Suppose that the commander is loyal and that he sends *attack* messages (Figure 13.4). The loyal lieutenant will receive *attack* from the commander and some message, say *retreat*, from the traitorous lieutenant. Since the commander is loyal, the lieutenant must be using some algorithm that causes him to decide to attack.

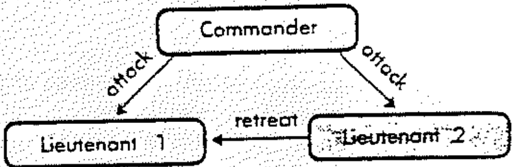


Figure 13.4 Impossibility with loyal commander

On the other hand, if the commander is a traitor, he could send *attack* to lieutenant 1 and *retreat* to lieutenant 2 who will faithfully relay it to lieutenant 1 (Figure 13.5). The situations are indistinguishable from the point of view of lieutenant 1, so whatever decision he takes in the first case he must also take in the second case. So he decides to attack.

A symmetrical argument shows that if a lieutenant receives a *retreat* message from the commander, he must retreat. Thus in the second case, loyal lieutenant 2 will decide to retreat which is not in agreement with lieutenant 1.

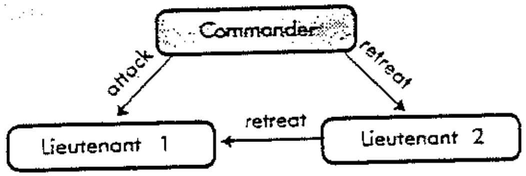


Figure 13.5 Impossibility with traitorous commander

13.5 Further Reading

The Byzantine Generals problem and the results here are from [LSP82]. Books on fault-tolerant computing are [Pra86] and [Shr85]. [SG84] describes how the first launch of the Space Shuttle was delayed because of a fault in the synchronization between the main computer and the backup computer. This example shows that fault-tolerant systems are more complex and will have more faults than an ordinary system but the faults that do occur will not be as catastrophic.

13.6 Exercises

Let us construct a solution to the Byzantine Generals problem in the presence of two traitors.

- The commander sends his decision to each of the $n-1$ lieutenants. Call this a level-2 value.
- Each lieutenant i sends the level-2 value he received to each of the $n-2$ other lieutenants. Call this a level-1 value $v(i)$.
- Each lieutenant k sends the level-1 value he received $v(j)$ to each of the $n-3$ other lieutenants. Call this a level-0 value $v(j)(k)$.
- Eventually, each lieutenant i receives $n-2$ values for lieutenant j: one level-1 value $v(j)$ and $n-2$ level-0 values $v(j)(k)$. By majority vote, lieutenant i can determine a value for lieutenant j.
- Using the $n-2$ values obtained for the other lieutenants in the previous step and the level-2 value from the commander, lieutenant i makes his decision by majority vote.

1. Show that the above algorithm is correct for seven generals and two traitors. (*Hint:* If the commander is loyal, a lieutenant will have four level-0 values and one level-1 value to determine the value for each other lieutenant. But at most two of the five can come from traitors. If the commander is a traitor, five out of six lieutenants are loyal and it is easy to show that they all compute the same value.)
2. Show that the above algorithm is not correct for six generals and two traitors. (*Hint:* A lieutenant obtains only four values from which to compute the value for each other lieutenant. Majority voting cannot help.)

PART III

Implementation Principles

Chapter 14

Single Processor Implementation

14.1 Introduction

Concurrent programming can be implemented by sharing the computational resources of a single processor. We review the resources required to execute a single process and then show how to implement concurrency.

Compilation of a program produces an object program which is stored in memory. The program is executed by a processor which fetches instructions from memory and then executes the computation indicated by the instruction. An *instruction pointer* (IP) keeps track of the current instruction being executed. When an instruction has been executed, the IP is incremented[1] unless the instruction is a jump or procedure call which explicitly modifies the IP. Most instructions read or write memory locations (aside from the object code itself). Additional resources such as general registers or I/O channels can be treated similar to the IP and the main memory. We will discuss the implementation of concurrency in terms of sharing the IP and the main memory among several processes.

14.2 Memory Allocation

Modern programming languages such as Ada are designed to match a specific memory model shown in Figure 14.1.

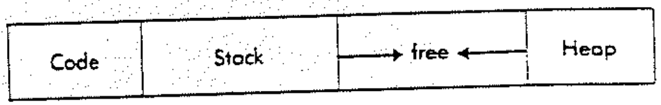


Figure 14.1 Memory model for programming languages

The object code is stored in one segment. The code segment is never modified and can even be stored in *read-only memory (ROM)*. The nested block structure

[1] We are ignoring complications like variable length instructions.

means that memory needed by a procedure call can be allocated on a *stack*. When a procedure is called, a block of stack memory called the *activation record* is allocated which contains memory locations for parameters and local variables as well as room to store the *return address* – the address of the instruction after the procedure call. Upon return from a procedure, the IP is loaded with this value. This technique is what allows us to call the same procedure from several locations in a program. Finally, the activation record contains pointers used by the procedure to access variables in global environments. The third area is called the *heap*. Programming languages contain statements for dynamically creating data structures that are then accessed by pointers rather than variable names. This memory is allocated from the heap.

The size of the code area is known at compilation time. The stack grows and contracts dynamically. Its size is not a function of how many procedures exist or how often they are called since upon exit from a procedure space is automatically reclaimed. Instead it is a function of the maximum depth of procedure calls and of the amount of local memory needed by the procedures. If the program contains recursive procedures, the size of the stack cannot be calculated in advance.[2] If not, the maximum stack size can be calculated, though this may impractical in a large program. Heap storage requirements cannot be calculated in advance since dynamic allocation is usually used for data structures like lists and trees whose size depend on the input to the program.

To summarize:

- Code size is fixed at compilation.
- Stack size can often be estimated before execution. The stack is well behaved: expanding and contracting at one end with no wasted space.
- Heap size is unpredictable. The heap is not well behaved because allocation and de-allocation of variable-sized data areas can occur at any time and at any place within the heap.

What are the implications for concurrency?

- The code areas of the processes are independent and can simply be concatenated together. In fact, code sharing is possible; several processes can use the same object code for a procedure called by any of them.
- The stack contains information particular to each process. The state of the process is defined by its IP (and other registers) and the contents of its memory locations such as procedure variables, parameters, etc. Thus each process will need to have its own stack.
- The heap can be shared since once a data area is allocated it is just like any other memory location and each process will access the locations it was allocated. However, since allocation does modify the heap which is global to all processes, mutual exclusion must be imposed to ensure its consistency.

[2] A similar problem exists if local variables can have their size determined at run-time as allowed in Ada but not in Pascal.

As shown in Figure 14.1, since the stack grows in only one direction, the heap is allocated at the other end of the memory and allowed to grow in the opposite direction. This gives some flexibility in that the same locations can be used by one area and later on by the other. If the two areas meet, then there is actually no more memory to allocate. To implement concurrency, it is not enough to set the bottom of the stack and hope that it does not meet the heap. A stack must be allocated to each process so we must provide the run-time system with an estimate of the stack size of each process (Figure 14.2). This data structure is called a *cactus stack* since a stack is linked to the stack of the process which allocated it.

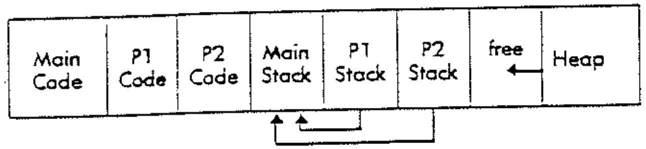


Figure 14.2 Stack allocation for concurrency

A good compiler would make a reasonable first guess for the stack size, but this may not be accurate. In any case, there is a possibility that one process may run out of stack space even when there is free memory. One possibility is to re-arrange the stack allocation at run-time, though this may be difficult on some architectures since the activation records contain pointers to memory locations within the stack. In any case, re-arranging the stacks can time consuming and is not appropriate for real-time systems.

A minor variation on the cactus stack is to allocate the stacks directly from the heap rather than maintaining a separate area for the stacks. This has the advantage that only a single allocation mechanism is needed and the disadvantage that if stack re-arrangement is attempted, it has to be integrated with the heap storage reclamation system.

Heap allocation also poses special problems in concurrent programs. In a sequential system, a request for more memory than is currently available is considered an error. In a concurrent system, it might be reasonable to suspend the requesting process in the hope that another process will eventually free enough memory for the suspended process to proceed. Alternatively, an indication could be returned to the requesting process. In a critical system, the process can be designed to exhibit degraded behavior or to raise an alarm, either of which is preferable to aborting the program.

Heap allocation strategies usually maintain lists of free and allocated memory blocks. Both allocation and release of memory can be time consuming or unpredictable because the time depends on the length of the lists. Concurrent programs with time constraints may need special algorithms which limit the unpredictability of the response time of memory allocation algorithms. In extreme cases, they may have to forego dynamic allocation.

Garbage collection is the term given to reclamation of heap storage that is no longer accessible. It is an alternative to requiring the programmer to explicitly free unused memory. Garbage collection is an essential part of advanced languages like Prolog and LISP. Its main drawback is that classical algorithms require that all computation must be suspended for a long time at unpredictable intervals for the collector to work. Concurrent algorithms exist which allow garbage collection to be done by a separate process, spreading out the overhead throughout the computation. On a multi-processor, a separate processor can be allocated to garbage collection, almost eliminating the overhead on the real-time computation.

14.3 Process Control Blocks

The previous section described the static arrangement of memory required to support concurrency. We still need to dynamically share the (single) processor among the various processes. Since each process will be interrupted or blocked during its execution we need to store data concerning the state of the process. At the very least, the IP must be saved. For each process, we define a data structure called the *process control block* (PCB) used to store the state of the process.

At any instant, at most one process will be executing on the processor. The status of this process is *running*. There may be contention for the processor, that is there may be other processes that are *ready* to execute. The PCBs of these processes are linked together in the *ready queue* (Figure 14.3). Whenever it becomes necessary to change the process that is allocated to the processor, an operating system routine called the *scheduler* takes one process from the ready queue and sets the IP from the PCB. The current IP first saved in the PCB of the executing process which is linked into a queue. This processing is called *context switching* and is the principal overhead of concurrent programming on a single processor.

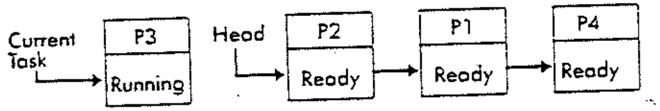


Figure 14.3 Process control blocks

Since the executing process is the one executing on the processor, what can cause it to invoke the scheduler to change the executing process? In the simplest common-memory models (load-and-store or test-and-set), the answer is nothing. Scheduling must be artificially invoked by a hardware timer which interrupts the processor periodically. This time-slicing cuts up the available processing time into fixed-length intervals which can then be allocated to the processes by the scheduler.

In the case of blocking primitives like semaphores, monitors and rendezvous, a process will explicitly indicate its willingness to relinquish the computer. While time-slicing is no longer required, it is usually implemented if for no other reason then to allow the system to interrupt a process in an infinite loop. For each blocking primitive there will be a logically separate queue (Figure 14.4):

- For each semaphore.
- For each monitor and for each condition variable.
- For each entry and for delayed processes in Ada.

Signaling primitives will check the appropriate queue and move a PCB to the ready queue. To separate different aspects of the system into different software modules, the signaling primitive will not attempt to schedule the awakened process but will call the regular scheduler on the updated ready queue. The immediate resumption requirement of the monitor primitive can be implemented by inserting the PCB into the ready queue at the place where the scheduler is known to choose the next process.

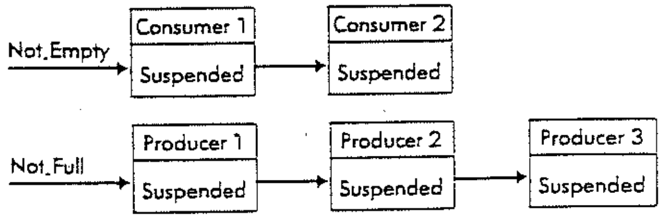


Figure 14.4 Queues for blocking primitives

14.4 Priorities

Scheduling of processes is done by assigning a number to each process called its *priority*. The priorities may be statically assigned at compilation time as in Ada, or they be computed dynamically during run time. Chapter 16 discusses priority allocation strategies.

If the ready queue is ordered in decreasing priority, the next process to run is just the process at the head of the queue. Insertion of a ready process requires the scheduler to search the queue. The insertion will be facilitated if a doubly-linked list is used (Figure 14.5).

If there are a large number of processes and a small number of priority values an alternative data structure is one that maintains an array of queues of PCBs, one queue for each priority value (Figure 14.6). Insertion is simplified since no search is required – the process is simply appended to the tail of the queue for its priority (implicitly breaking ties of equal priority processes on the basis of time of arrival). Selecting the next process to run requires a search for a non-empty queue. If the search is done in order of decreasing priority, high priority processes will be found immediately and only low priority processes will incur significant overhead which is a reasonable design objective.

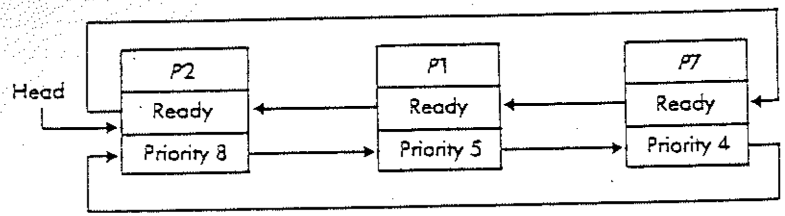


Figure 14.5 Doubly-linked list ordered by priority

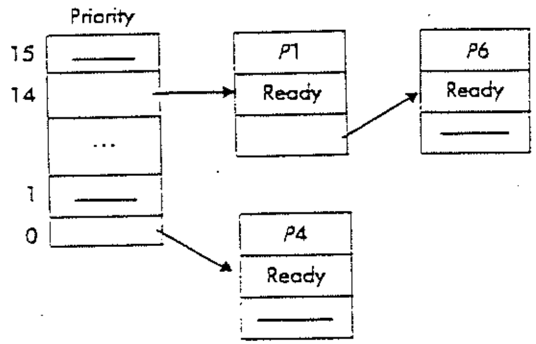


Figure 14.6 Array of queues

Similar techniques are used on the blocking queues. To implement a delay queue, the PCBs are constructed in order of increasing time until wake-up. At every timer update, processes whose wake-up time have passed are moved from the delay queue to the ready queue. We can see how a delay statement in Ada does not ensure that a process will actually be run within the time specified in the delay. *After* the period of time has elapsed, the process will be moved to the ready queue and then it must compete according to priority with other processes.

The FIFO queues required by blocked queue semaphores, monitor condition variables and Ada accept statements are implemented by ordering the PCBs in increasing time-of-arrival. In Figure 14.5, circular lists are used where the tail of the list is linked to the head. Insertion at the tail can now be done in constant time.

14.5 The Ada Rendezvous

Implementation of the rendezvous is more complex because a task may have several entries and it may conditionally wait on those entries (Figure 14.7). Let us consider first a simple accept statement and then the select statement. When

```
loop
  accept Init do ... ;
  select
    when Guard_1 => accept E1(...) do ... ;
  or
    when Guard_2 => accept E2(...) do ... ;
  or
    when Guard_3 => accept E3(...) do ... ;
  end select;
end loop;
```

Figure 14.7 A typical Ada task

an entry is called and the accepting task is not yet ready to accept, the calling task must be suspended. Two obvious choices for data structures are (Figure 14.8):

- Maintain a single list of PCBs for all calling tasks.
- Maintain a separate list for each entry.

The second alternative is to be preferred unless there are very many entries and very few calls expected. If there are separate lists for each entry, when an accept statement is executed one can index directly to the proper list and commence the rendezvous with the first task on the list.

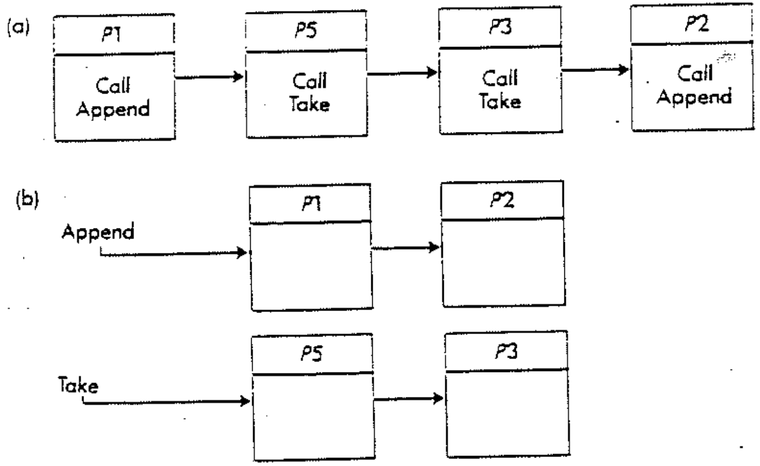


Figure 14.8 Data structures for entry queues

The PCB of the task suspended in the entry queue should have a pointer to a block of memory where the parameters of the call are stored. The accepting task can then copy the parameters, perform the statements of the rendezvous and update the parameters to terminate the rendezvous. Then *both* tasks are inserted into the ready queue and the scheduler is called to find the task with the highest priority.

The other possibility in a rendezvous is that the entry queues are empty and the accepting task must be suspended. The PCB of the accepting task will contain an indication that the task is suspended. The call will cause the accepting task to be awakened and placed on the ready queue.

The select statement is more difficult to implement because entry calls must be checked against a set of open alternatives: those accept statements with guards that evaluate to true. What data structure should be used to describe this set? Again, the choice is between a list and an array (Figure 14.9). If there are many branches in the select statement, few of which are expected to be open, a list will more efficient. If an array is used, every element must be updated with the result of the guard evaluation but it will be easy to locate the element for any particular entry.

This data structure is accessed both by the entry call and by the select statement evaluation:

- On entry call, the data structure must be checked to see if the accepting process is waiting on an open alternative for this entry. If an array is used, the check can be done in constant time, otherwise a list must be searched.
- When the select statement is evaluated, the data structure is created by the evaluation of the guards. Then the entry queues must be checked to see if there are calls queued for the open alternatives. If an array is used, we must sequence through the elements, checking the entry queue(s) for each open alternatives. If a list is used for the open alternatives, the entry queue(s) are checked for each element of the list. This will be more efficient than an array if there are many closed alternatives.

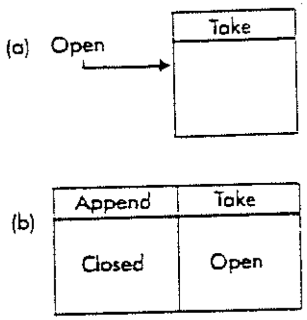


Figure 14.9 Data structures for open alternatives

The other features of the rendezvous are not difficult to implement as extensions of the selective wait:

Else alternative After checking the open alternatives against the entry queue(s), the task does not suspend if a rendezvous is impossible.

Delay alternative The task not only suspends itself, as described above, but also places itself on the delay queue. The processing of the delay queue must be modified so that if this task is awakened, the rendezvous must be canceled. Similarly, if a rendezvous is accomplished during the delay interval, the task must be removed from the delay queue.

Conditional entry call The calling task does not suspend itself on an entry queue if a check of the accepting task shows that it is not suspended awaiting rendezvous or if the alternative for this entry is closed.

Timed entry call As in a conditional entry call except that the calling task suspends itself on the delay queue. As in a selective wait with delay, the delay queue processing and the rendezvous processing must be coordinated.

14.6 Further Reading

Techniques for implementing concurrency are standard and can be found in texts on data structures and operating systems ([Sta80], [Dei85], [PS85]). The special requirements of Ada are described in [BR85]. Algorithms for concurrent garbage collection (called 'on-the-fly' garbage collection in the literature) are presented in [DLM78] and [Ben84].

15.1 Introduction

The implementation of distributed systems is the subject of much
tion. The essential tradeoff is in the design of the connectivity o
Can all processors directly communicate with each other or is the t
limited? It is convenient to distinguish three families of architecture

Common memory (Figure 15.1) Even a small amount of commor
ables processors to communicate and synchronize. A larger amou
memory will hold data needed by all processors. Common-mem
tures are efficient but cannot be expanded to a very large num
sors, both because of the difficulty of constructing the hardware
of increasing contention for the memory bus. This architectu
on very-high performance systems that cooperate to solve cor
intensive problems.

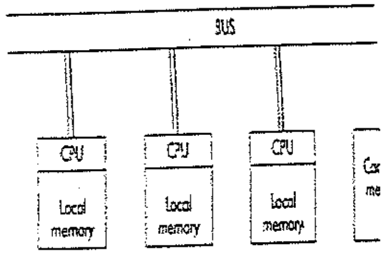
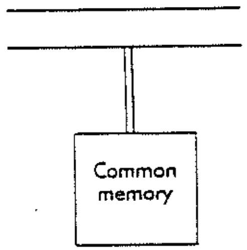


Figure 15.1 Common memory architecture

Networks (Figure 15.2) All processors can still communicate dire
broadcast messages to the whole group. The traffic is increased
structures must be replicated. The hardware scales easily to a r
number of processes but increasing contention eventually limits

t of much experimenta-
nectivity of the system.
or is the topology more
ırchitectures:

of common memory en-
ırger amount of common
nmon-memory architec-
large number of proces-
e hardware and because
architecture is common
o solve computationally



cture

nicate directly as well as
s increased because data
asily to a relatively large
ıally limits the size of the

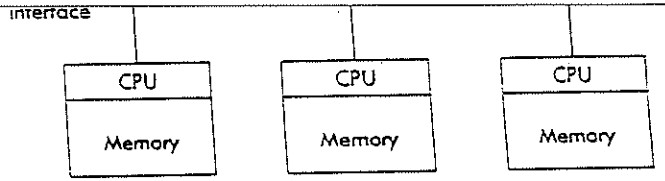


Figure 15.2 Computer network

Point-to-point connectivity (Figure 15.3) Dedicated high-speed communications lines pass data from one processor to another with no contention and no need for addressing. If each processor is connected to only a small number of others, like an array, there is no limit to the size of the system. This architecture is used on high-performance systems that need a very large number of processors. The difficulty is then to construct a program that matches the restrictions of the architecture.

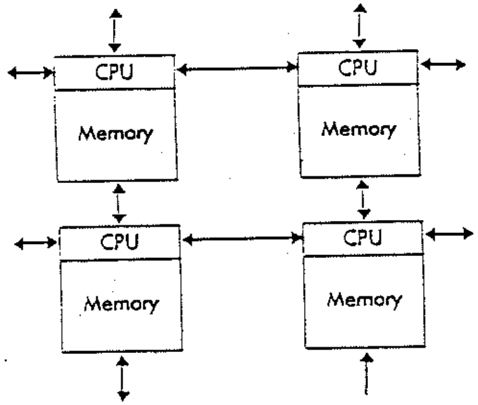


Figure 15.3 Point-to-point architecture

We will describe the transputer system that can deliver very high performance on the restricted occam model and then implementations of Linda on all these architectures. The chapter closes with a discussion of distributed implementation of Ada.

15.2 occam on the Transputer

The transputer is a microprocessor developed by Inmos Corporation to directly support the occam programming model. The IMS 800 transputer (Figure 15.4) contains a fast 32-bit processor, a floating point processor, a small amount (4K bytes) of on-chip memory and four bi-directional high-speed serial communication links. Each directly implements a pair of occam channels, one in each direction.

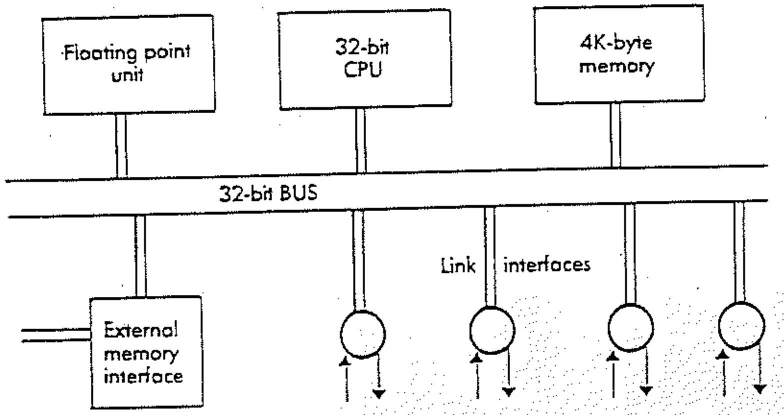


Figure 15.4 IMS 800 transputer architecture

An entire message can be transferred using a single machine instruction. The instruction has three operands: the channel number, the address of the message and the count of the number of bytes in the message. The same instruction is used both for internal channels (between processes that have been allocated on the same transputer) and for external channels (between processes on different transputers). Thus a program that runs on one transputer will run on a multiple transputer system.

The rendezvous semantics of communications require synchronization between input and output. For concreteness, let us assume that one process executes an Output instruction on a channel and then later the other process executes an Input instruction. The execution of the Output instruction depends on the type of channel:

Internal channel The request for communication is noted in a data structure associated with each channel.

External channel The message operands are copied into registers within the link interface.

In both cases, the process is suspended as required by the rendezvous semantics.

Eventually, the process on the other end of the channel reaches the corresponding Input instruction:

Internal channel Checking the channel data structure, it notes that an output request exists and copies the message into its own buffer.

External channel It initializes the registers of its link interface. Now a hardware implemented protocol transfers the message over the channel. The message is transferred byte by byte and each byte is acknowledged through the opposite direction of the link. Note that each link is simultaneously used for data transfer in one direction and acknowledge messages in the other (Figure 15.5).

When message transfer is complete, both processes are released from suspension.

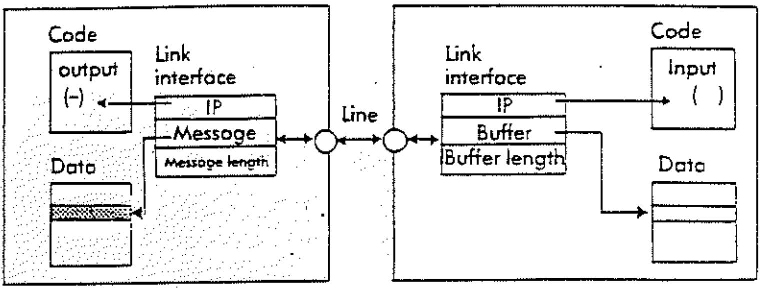


Figure 15.5 Transputer I/O

Remember that occam encourages the creation of a large number of concurrent processes. Scheduling these processes is also implemented by hardware instructions. The transputer maintains two queues of ready processes: high priority and low priority. Normally a process is allowed to run without interruption until it reaches a *descheduling point* which are I/O instructions and termination of processes and loops. Also, high priority processes which become ready because of an external rendezvous occurs will interrupt low priority processes. Otherwise, processes are allowed to execute until they reach a descheduling point. Low priority processes are not run unless the high priority queue is empty. Low priority processes are time-sliced approximately once every 2 milliseconds. This priority model is appropriate if high priority processes are used only to implement short I/O routines.

The advantages of the transputer for high-performance concurrency can be seen by comparing order of magnitude execution times with classical architectures. Channel communication can achieve a rate of about 2 megabytes per second. That means that five 32-bit words can be sent from one transputer to another in 10 microseconds. The hardware scheduler is extremely fast and can do a context switch in one microsecond. Real-time operating systems for classical microcomputers may need 100 microseconds or more to do a context switch in software, and hundreds of microseconds to send a message from one processor to another over a dedicated communications line. Sending a message between large computers over networks can take a millisecond or more when the overhead of the operating system is taken into account.

Clearly. the transputer/occam combination is able to achieve impressive performance in implementing concurrency at the price of restrictions on the allowable hardware and software architecture.

15.3 Implementations of Linda

To implement Linda. we have to implement a global *tuple space* (TS) with insertion and blocking removal of tuples. If there exists common memory on the system. the TS may be stored there. The implementation problem reduces to the classical problem of storing and searching a large data structure. The choice of data structure is dictated by the form of the data. If we have any knowledge that the data are restricted to numbers. are of fixed length. are ordered and so on. we can use this knowledge to choose an optimized data structure. A TS, however. is composed of tuples of arbitrary length, each element of which can be of any type.

With this lack of restriction on the data format. we could store all the tuples in a single linear list. However. searching would be prohibitively expensive. *Hashing* can dramatically improve the search performance by arbitrarily assigning each tuple to one of a large number of very small lists called *bins* (Figure 15.6). A *hashing function* takes the tuple and considering it just as a sequence of bits. computes the number of the bin in which it will be stored. To search for a specific tuple. all we have to do is compute the bin number using the hashing function and search for it in the bin. The hashing function is chosen to be quick to compute and the small bin size ensures that the tuple will be quickly found if it exists.

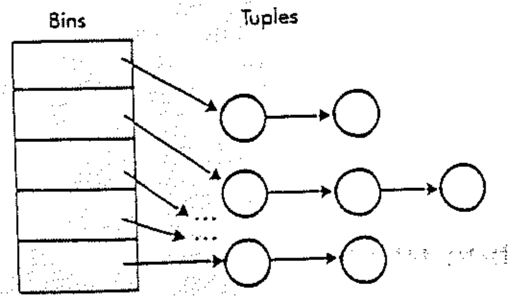


Figure 15.6 Hashing implementation

In Linda. the hashing function is usually computed on the first argument of the tuple only. This leads to a requirement that the first argument of any tuple be an actual value and not a formal parameter.

There exist multi-computer architectures with common memory. However, even on these architectures. it is still more efficient to access local memory than common memory. The recommended programming style in Linda is to use the TS to store common data but to copy tuples to local memory for computation. Thus in the matrix multiplication example. we stored the rows and columns in the TS, but indexing through the vectors to compute the inner product was done after they were copied into local memory.

Hashing can also be used to implement Linda on an architecture without common memory but with high-speed communications lines like a transputer array or higher level generalizations called *hypercubes* (Figure 15.7). The global TS will be distributed. Each tuple will be stored on one of the processors determined by the result of the hashing function. To locate a tuple. compute the identity of the processor where its bin is stored and send a message requesting the tuple to that processor. With high-speed communications lines this can be efficient.

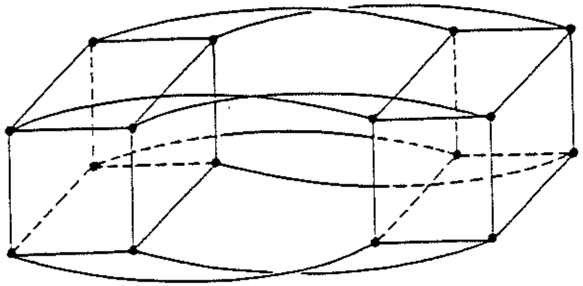


Figure 15.7 Hypercube architecture

Finally, Linda has been implemented on networks where every processor is connected to a single high-speed bus (Figure 15.8). There are two approaches to a network implementation. The TS can be replicated on each processor or it can be distributed among the processors. In the first case. Output broadcasts each tuple. When a processor tries to input or read a tuple. the TS exists locally and the search is efficient. If no matching tuple exists, the process suspends until some incoming tuple matches.

Some mechanism must exist to ensure that the tuple is not removed by two different processors. This can be done by associating each tuple with the processor that created it and requesting permission from this processor before removing the tuple. The 'owning' processor can then ensure that only one processor succeeds in removing the tuple.

The other possibility is to distribute the TS by having each processor locally store the tuples it creates. Now Input and Read must broadcast a request if a matching tuple is not found locally. A receiving processor that matches the request with a local tuple sends it to the requesting processor. It is possible that a processor receives several tuples in answer to its request. In this case, it 'uses' one of them and stores the others. These tuples still exist in the distributed TS even though they may have moved from their original place.

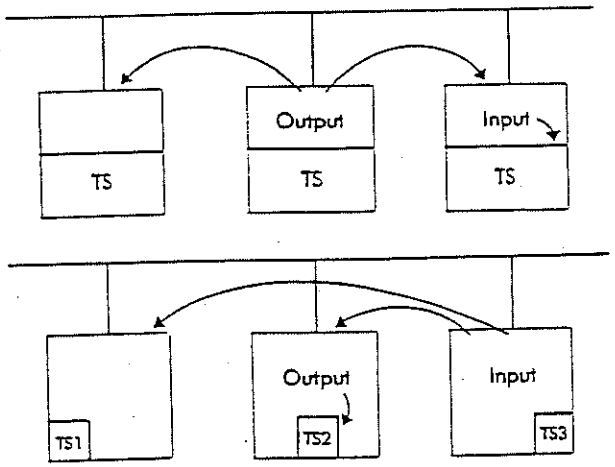


Figure 15.8 Network implementation of tuple space

15.4 Ada on Distributed Systems

It is difficult to implement full Ada on a distributed system. The reason is that the tasking model is integrated into the stack and heap model discussed in the previous chapter on single processor systems.

The first problem is the block structure with its nesting of tasks like ordinary procedures. In Figure 15.9 we see that two tasks can access ordinary global variables as well as global procedures. In a distributed system we have to ask: where are these global data and code located? If they are in one processor, there will be a lot of communication. If they are replicated, we will have problems ensuring consistency. The problem can be complicated if we note that Ada is intended for large systems and the package facility can be used to create a large amount of data and code.

The other problem is the use of dynamic memory allocation. When memory is allocated from a heap, a pointer - an absolute address - is returned. There is no syntactic or semantic limitation in Ada that would prevent two tasks from cooperating on the creation of a list by having each one allocate alternate nodes. If we have one heap on one of the processors, the implementation could be very inefficient. If both processors allocate from their own heaps, how can such a mixed list be implemented?

A related problem is that a pointer can be passed as a parameter in a rendezvous. What does it mean for a task allocated to one processor to receive an absolute address on another?

One solution is to use the concept of *virtual nodes*. During the design of an Ada program. we decide what the maximum number of physically distributed processors will be. These will be called virtual nodes and every component of

```
procedure Main_Program is
  V: array(1..2000) of Integer;

  procedure P(I: in out Integer) is  ... ;

  task T1;
  task body T1 is
  begin
    V(2) := V(3) + 1;
    P(V(2));
  end T1;

  task T2;
  task body T2 is
  begin
    V(2) := V(3) + 2;
    P(V(2));
  end T2;
end Main_Program;
```

Figure 15.9 Block structure in Ada

the program (tasks and packages) will be assigned to some node. Communication between virtual nodes of the program will be limited to parameters that can be easily copied during a rendezvous. Within the virtual node full Ada can be used.

We can always assign more than one virtual node to a single processor. but this design ensures that if nodes are in fact assigned to different processors. they can communicate efficiently over channels. The occam-like restrictions on the program architecture are thus consciously applied by the designer where necessary rather than imposed on all components by the language. Similarly. Linda-like expressibility of common memory is used where needed and not imposed.

This comparison of the three formalisms shows that the designer of a distributed system can choose between expressibility and efficiency and also between a language that imposes (and hence optimizes) one view and a more flexible language that requires more on the part of the designer. Linda and full Ada are very expressive but less efficient than occam in a distributed implementation. Linda and occam impose and optimize a specific primitive while Ada has a more flexible set of primitives.

15.5 Further Reading

The transputer reference manual is [In85]. Implementations of Linda are described in [ACG86]. Virtual nodes in Ada is from [TCN84] which is a study of the use of Ada in distributed environments.

Chapter 16

Real-Time Programming

16.1 Introduction

Real-time programs are programs that must execute within strict constraints on response time. Such programs are used in *embedded computer systems* which have a computer as only one of a set of components that together comprise a large system. Examples are:

- Aircraft and spacecraft flight control systems.
- Industrial plant controllers.
- Medical monitors.
- Communications networks.

The special difficulty in designing real-time systems is in the strict requirement to meet processing deadlines and not in the actual speed of processing. Here are two examples of challenging, complex computer systems which are not real-time:

- Simulation of weather patterns to develop a forecast. This requires the use of powerful supercomputers but it is not a critical failure if the answers are received ten minutes late.
- An airline reservation system must process hundreds or thousands of transactions continuously. A large network of computers, data bases and communications lines is needed to implement the system. The response time requirement for such a system can be expressed statistically. For example: 95 percent of the transactions completed within 5 seconds and 99 percent within 15 seconds.

We do not want to imply that these systems should not be designed to meet deadlines all the time. However, if that would require extremely expensive or unreliable implementation techniques, there is some flexibility in satisfying response time requirements.

A real-time system need not be complex or high-performance. Checking radioactivity levels in a nuclear power plant or pulse rate on a patient may require only the simplest processing that can be done by any micro-computer. However, delay in decreasing plant power or sounding an alarm can be literally fatal. Another problem, common in flight control systems, is that absolute time may be a

parameter in the computation. If an algorithm is designed to sample the aircraft state and control its flight every 50 milliseconds, it must be executed at that rate, not faster and not slower (within some tolerance). In real-time systems, neither deadline slippage nor statistical performance is acceptable.

Of course, even in real-time systems, there will be non-real-time tasks such as logging messages and background testing of the hardware. However, when there are real-time tasks, this affects the design of the system and special techniques are needed. The techniques are not as well developed as concurrent programming techniques that ignore absolute time. Nevertheless, the best approach to real-time systems seems to be the careful application of concurrent programming techniques, modified or restricted as needed to construct a successful program.

16.2 Synchronous and Asynchronous Systems

There are two approaches to the implementation of real-time systems: synchronous clock-driven or asynchronous interrupt-driven. In a synchronous system, a hardware clock is used to divide the available processor time into intervals called *frames* (Figure 16.1). The program must then be divided into segments so that every segment can be completed *in the worst case* during a single frame. A scheduling table is constructed which assigns the segments to frames so that all the segments in a frame are completed by the end of the frame. When the clock signals the beginning of the frame, the scheduler of the underlying system calls the various segments as described in the table. If a computation is too long to fit in one frame, it must be artificially split into smaller segments that can be individually scheduled.

Figure 16.2 shows an Ada task executing as a synchronous scheduler. Each frame commences by sampling an input. The computation is divided into even and odd cycles. Following the computation, control signals are sent to the output

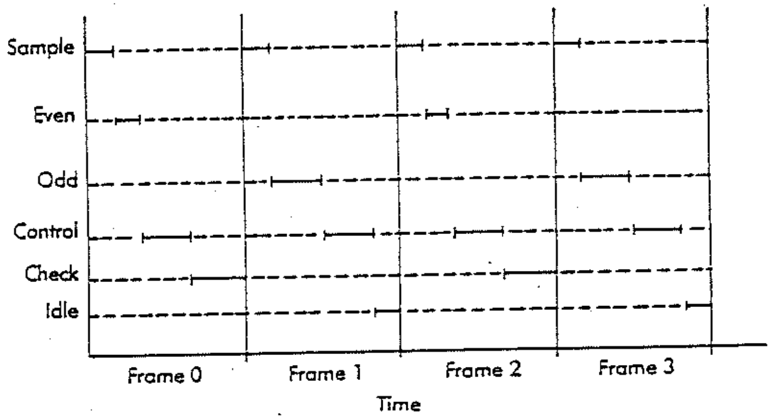


Figure 16.1 Synchronous systems

```
task body Synchronous is
  Frame: Integer range 0..3 := 0;
begin
  loop
    accept Clock_Interrupt;  -- No accept body
    case Frame is
      when 0 => Sample;
                Compute_Even;
                Control_Device;
                Check_Input_Failure;
      when 1 => Sample;
                Compute_Odd;
                Control_Device;
      when 2 => Sample;
                Compute_Even;
                Control_Device;
                Check_Output_Failure;
      when 3 => Sample;
                Compute_Odd;
                Control_Device;
    end case;
    Frame := (Frame+1) mod 4;
  end loop;
end Synchronous;
```

Figure 16.2 Synchronous scheduler

devices. The computation on the even cycles is shorter than on the odd cycles so in those frames we also check for hardware failures. The assignment of segments to frames is shown in Figure 16.1.

An asynchronous system does not attempt to divide the computation into segments. Instead, each segment executes to completion and then the scheduler is called to find the next task to execute. This is the Ada model where the tasks themselves execute asynchronously and the rendezvous exists if it is necessary to synchronize them. Two extensions to this model need to be made in order to write real-time software in an asynchronous system. The various tasks must be given priority so that we can be assured that critical tasks like Sample and Control are always executed even at the expense of background tasks like Check_Input_Failure.

In addition, *pre-emptive scheduling* must be implemented to ensure that a high-priority task can execute as soon as it becomes ready, even if that means suspending the execution of a task of lower priority (Figure 16.3). Ada requires that pre-emptive scheduling be implemented. A higher priority task can become ready upon expiration of a delay or if an interrupt occurs. Closely related to pre-emptive scheduling is *time-slicing* which divides available processor time among several tasks of equal priority. This is implemented by a hardware timer which interrupts processing periodically to allow the scheduler to search for another task to execute. The Ada standard does not require an implementation to implement time-slicing, though most do so.

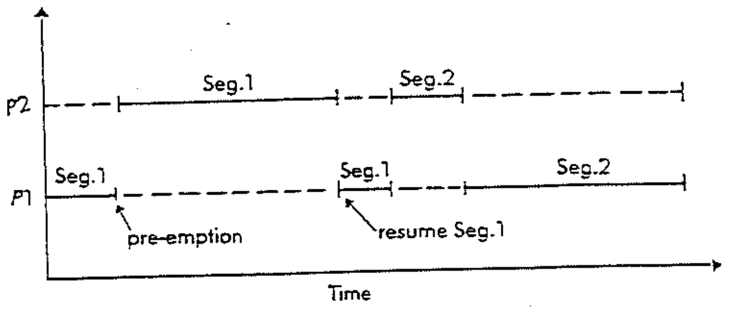


Figure 16.3 Asynchronous systems

The choice between the two types of system hinges on the price one is willing to pay for guaranteed response time. Synchronous systems absolutely guarantee that the tasks will be executed at a rate determined by the timer and the scheduling table provided that there is no overflow of the computation during a frame. However, the scheduling table must be constructed using worst-case timings of each task. If, on the average, the tasks execute faster there is no way to use the wasted processor time. In addition, the fixed size frames will cause fragmentation overhead because segments may not fit in the leftover space in a frame. For example, if every segment is at least 15 milliseconds long, when a 50 millisecond frame has segments totaling more than 35 milliseconds assigned, the rest of the time is wasted. These problems are similar to those experienced in paged virtual memory operating systems. The problems here are more serious because processor time in a real-time system is more significant than memory utilization in a multi-programming computer system.

In addition, since the length of the frames is fixed, segments will be run at integral multiples of the frame length, regardless of the actual requirements. With 50 millisecond frames, a segment that need only run every 75 milliseconds must be run at the higher rate of 50 milliseconds, wasting processor time.

In an asynchronous system, no processor time is ever wasted. As long as there are tasks on the ready queue, some process will always be executing and we can achieve 100 percent utilization. Overall, the computations will be completed faster. However, it is difficult to ensure that any particular computation will be executed within any particular deadline. Obviously, the highest priority task can pre-empt any running task and execute immediately. As we descend in priority, a task will be executed within a time interval that is determined by the actual execution times of the higher-priority tasks. In practice, it may not be possible to compute this time interval.

There is another reason to prefer asynchronous systems. The segmentation of the computation and the construction of the scheduling tables is extremely difficult and error-prone. Moreover, a change in system requirements or in the hardware will invalidate the tables and require their reconstruction. Asynchronous systems by nature dynamically re-adjust themselves to the demand from the various processes.

It seems that the best solution is to used a mixed system. Use a synchronous computation for a few truly time-critical tasks and asynchronous tasks in the background that execute in the available time (Figure 16.4).

```
task body Synchronous is
  Frame: Integer range 0..1 := 0;
begin
  loop
    accept Clock_Interrupt;  -- No accept body
    case Frame is
      when 0 => Sample;
                Compute_Even;
                Control_Device;
      when 1 => Sample;
                Compute_Odd;
                Control_Device;
    end case;
    Frame := (Frame+1) mod 2;
  end loop;
end Synchronous;

task body Check_Input_Failure is ... ;

task body Check_Output_Failure is ... ;
```

Figure 16.4 Mixed system

16.3 Interrupts and Polling

The choice between synchronous and asynchronous designs for a real-time system extends to I/O drivers within the system. In the example of a synchronous system, the procedure Sample will read the channels or registers connected to the input devices and similarly for Control_Device. I/O that is done periodically at the initiative of the program is called *polling*. For polling to work, an input device must be able to maintain its value on the channel until it is sampled by the processor. Other problems can occur if I/O devices are not adapted to polling. The optimal rate of I/O may not be a multiple of frame duration or the I/O device may not respond immediately to a command. However, polling does not introduce any difficulty in the programming itself since I/O is done sequentially within the frame like any other computation.

Just as synchronous systems pay a high price in efficiency to obtain predictability, polling can be extremely inefficient so most I/O devices work on asynchronous *interrupts*. Conceptually, the I/O operation is regarded as an asynchronous process of higher priority than the computational processes. When an I/O device is ready it is scheduled pre-emptively.

The implementation of interrupts is not the same as that of software processes. Associated with each interrupt is a process called the *interrupt handler*. The address of an interrupt handler is stored in a fixed memory location called the *interrupt vector* (Figure 16.5). An interrupt is initiated by an I/O device. This causes the interrupt handler to be scheduled, pre-empting whatever software process happens to be running. The scheduling is done by the hardware and not by the software scheduler. A partial context switch is done; at the very minimum, the IP of the running process must be stored so that it can be restored upon completion of the interrupt. Unlike software processes, interrupt handlers should terminate and should be as short as possible since they run at the highest priority.

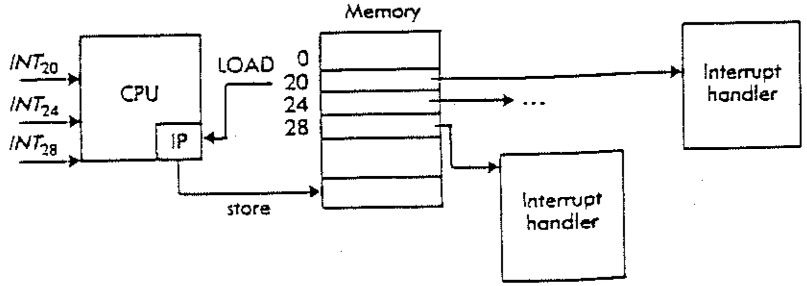


Figure 16.5 Interrupts

The interrupt mechanism is so useful that it is also used by the processor itself to indicate faults like division by zero. In addition, many computers contain instructions available to the programmer that are implemented by interrupts:

Trap Used to catch instructions that have not been implemented in hardware. The interrupt handler is a software emulation of the instruction.

Breakpoint Used by the debugger to replace an ordinary instruction. The interrupt handler is the debugger software which allows the programmer to query the status of the computation.

Supervisor calls Interrupts are used rather than procedure calls to ensure that the operating system runs at higher priority than ordinary processes.

Despite the differing implementation, interrupt handlers should be regarded as ordinary asynchronous processes for which concurrent programming techniques of synchronization and communication are applicable.

16.4 Interrupt Handlers and Software Processes

Suppose we have a set of software processes running at priorities 1-10. Conceptually, an interrupt occurs with a higher priority, say 20, so that it pre-empts any software process. What priority should be given to the interrupt handler process? The simplest decision is to run the interrupt handler at an even higher priority, say 30, so that it is uninterruptible. This simplifies the programming of device drivers, debuggers, operating systems, etc. since they have become critical sections running under mutual exclusion from all other critical sections.

While interrupt handlers are critical sections, the problem remains of how do we synchronize and communicate with the software processes? In Figure 16.6 we see that even though the variable N is incremented under mutual exclusion in the interrupt handler, no such mutual exclusion exists in the software process.

```
  N: Integer := 0;

task body Interrupt_Handler is
begin
  N := N + 1;
end Interrupt_Handler;

task body Software_Process is
begin
  loop
    ...
    N := N + 1;
    ...
    N := N + 1;
    ...
  end loop;
end Software_Process;
```

Figure 16.6 Synchronization with interrupt handlers

There are two solutions to this problem:

1. Transfer all critical sections from the software processes to software-invoked interrupt handlers.
2. Use non-blocking producer–consumer schemes to transfer data to and from the interrupt handlers.

The first solution returns us to the realm of concurrent programming, but the overhead may be unacceptable for real-time programs. In fact, it is possible to implement concurrent programming by decomposing all the software into interrupt handlers! There is no operating system as such and every statement can assume mutual exclusion if the handlers themselves are not interrupted. This design would be appropriate in a system that is demand-driven by I/O device requests, each of which requires relatively little processing.

The second solution is more often used in real-time systems. For concreteness, let us assume that an interrupt handler is reading data from an input device and passing it to a computational process. The producer–consumer solutions that we programmed require the producer to suspend itself when the buffer is full and similarly for the consumer when the buffer is empty. Suspending the consuming computational process is no problem, but we do not want to suspend the high-priority interrupt handler producer. Obviously, the buffer should be designed so that overflow is rare, but we still have to decide what to do if it does occur. There are two choices:

Lost data New data are discarded if the buffer is full.

Fresh data New data overwrite 'stale' data if the buffer is full.

Figure 16.7 shows the difference in programming the two designs.

```
-- Lost data
  if Count = N then
    null;
  else
    B(In_Ptr) := Value;
    Count := Count + 1;
    In_Ptr := (In_Ptr+1) mod N;
  end if;

-- Fresh data
  if Count = N then
    Out_Ptr := (Out_Ptr + 1) mod N;
  else
    Count := Count + 1;
  end if;
  B(In_Ptr) := Value;
  In_Ptr := (In_Ptr+1) mod N;
```

Figure 16.7 Non-blocking producers

The lost data option can be chosen when sequential processing of the data is important and when retry can be attempted. In a communications system, it is important to deliver a complete message and occasional retransmission of lost data will not significantly degrade the performance of the system. On the other hand, the fresh data option is chosen in control systems which try to predict or control future behavior based on past samples. It is less important to know where the aircraft was half a second ago or what the patient's pulse rate was ten seconds ago than it is to know the value of those signals *now*.

16.5 Nested Interrupts

Computer hardware usually provides the ability to assign interrupts to a set of *levels*. Through software commands, it is possible to dynamically change the set of levels from which interrupts will be accepted. The previous section considered the simple case where all levels are enabled except when an interrupt handler is executing in which case all levels are disabled. The other extreme would be to leave all levels enabled. This is equivalent to pre-emptive scheduling with no priority and has little to recommend it.

One design which is commonly used is to consider the levels as priorities and enable higher priority interrupts during the execution of an interrupt handler. This is called *nested interrupts* because higher priority interrupt handlers are executed as is they were nested local procedures (Figure 16.8). This design is attractive because each handler can assume that is it executed as a critical section with respect to lower priority interrupt handlers as well as the software processes. Furthermore, since interrupts of the same priority are disabled, an interrupt handler does not have run under mutual exclusion with 'itself'. For example, if we use one of the buffering schemes described earlier to communicate with a software process, an interrupt handler can safely execute instructions like `In_Ptr := In_Ptr+1` without invoking a synchronization primitive.

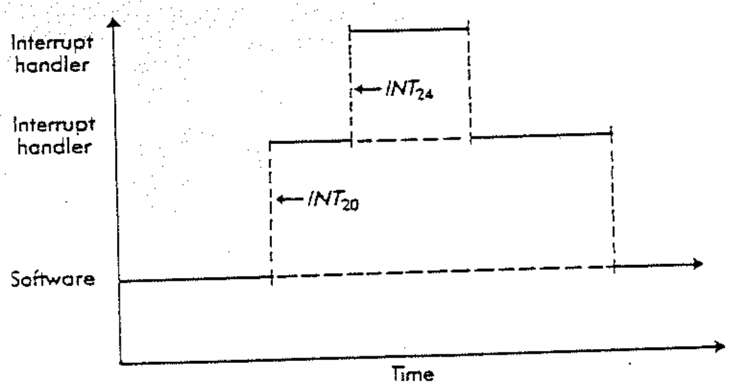


Figure 16.8 Nested interrupts

16.6 Scheduling Algorithms for Real-Time

How should priorities be assigned to processes in a real-time system? In this section we present (without proofs) two results on scheduling algorithms.

Assume that we are given a set of processes Pi each of which is characterized by a repetition interval Int_i and an execution duration Ex_i. That is, every Int_i seconds, process Pi requires Ex_i seconds of processor time. A *priority assignment* is an assignment of numbers Pri_i to the processes. The processes are scheduled by a pre-emptive scheduler which interrupts a running process if a higher priority process becomes ready.

The *response time* of a request to execute a process is the time between the request and the completion of the process.[1] A scheduler overflows if a process must execute another cycle before the previous one has been completed. A priority

[1] Here we are using process to mean one execution of a cycle of the process.

assignment is *feasible* if a pre-emptive scheduler will execute all processes without overflow.

For example, let $Int_1 = 2$, $Int_2 = 5$, $Ex_1 = 1$ and $Ex_2 = 2$. Assigning priorities such that $Pri_1 > Pri_2$ is feasible. In Figure 16.9(a), P1 completes its execution during the first second of the first two-second interval, leaving an an additional second for P2. At the end of two seconds, P1 pre-empts P2 for its next interval, relinquishing the processor again after one second. This shows that the assignment is feasible since P2 has received $Ex_2 = 2$ seconds within $Int_2 = 5$ seconds.

Conversely, if $Pri_2 > Pri_1$, P1 cannot pre-empt P2 which will proceed to execute for two seconds before relinquishing the processor (Figure 16.9(b)). Thus an $Int_1 = 2$ interval has expired without P1 receiving $Ex_1 = 1$ seconds of processing time and the assignment is not feasible.

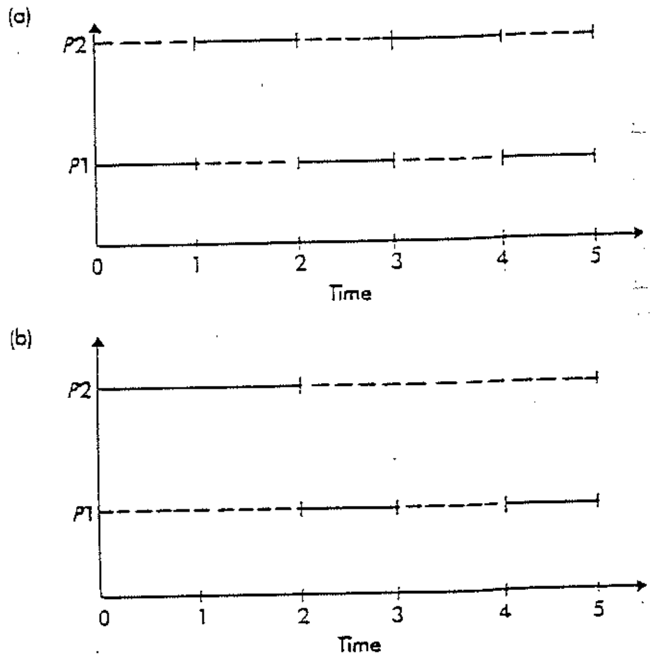


Figure 16.9 Feasible priority assignments

Theorem 16.6.1 *The longest possible response time for a process occurs when it is requested at the same time as all higher priority processes.*

This means that in checking a priority assignment for feasibility, it is sufficient to consider a scenario where all processes are in phase and request service at the same time. This was done in Figure 16.9 to show that only one of the two possible priority assignments is feasible.

The *rate monotonic scheduling algorithm* (RMSA) assigns priorities to processes in decreasing order of the intervals between requests. That is, the faster a process needs to be executed, the higher its priority, regardless of the duration of the process or the 'importance' of the process. It is possible to prove the following surprising result:

Theorem 16.6.2 *RMSA is optimal: if there exists a feasible priority assignment for a set of processes then RMSA is also feasible.*

Despite the simplicity of the RMSA, there are two problems that may limit its applicability:

- Processor time can be wasted as shown in Figure 16.10. Here $Ex_i = 1$, $Int_1 = 3$, $Int_2 = 4$, and $Int_3 = 5$. The process requests are shown beneath the axis and the processor allocations above. It can be seen that there is unused processor time that cannot be used by other processes scheduled by the algorithm.
- The algorithm is based on a model of fixed repetition rates and fixed execution durations. In practice, this means always using the maximum rate and maximum execution time which causes under-utilization of the time frames.

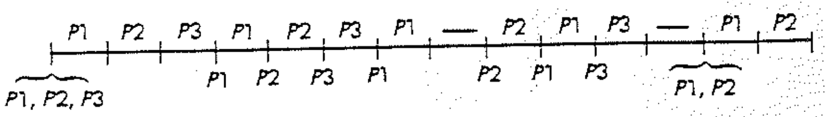


Figure 16.10 Rate monotonic scheduling algorithm

A *deadline* is the time of the next request for a process. The *deadline driven scheduling algorithm* (DDSA) dynamically assigns priorities such that the process with the closest deadline receives the highest priority. This algorithm can fully utilize the processor.

Theorem 16.6.3 *DDSA is feasible if and only if:*

$$\sum_i \frac{Ex_i}{Int_i} \leq 1$$

For the example above, $\frac{1}{3} + \frac{1}{4} + \frac{1}{5} \simeq 0.78$, so it is feasible. In fact, the DDSA would still be feasible if P3 needed 2 seconds to execute because $\frac{1}{3} + \frac{1}{4} + \frac{2}{5} \simeq 0.98$. However, if P1 needed 2 seconds, it would not be feasible: $\frac{2}{3} + \frac{1}{4} + \frac{1}{5} \simeq 1.12$.

DDSA is not always immediately applicable because hardware processes like interrupt routines may have fixed priorities. In addition, it may impose a serious overhead on the real-time scheduler.

16.7 Priority Inversion in Ada

It is not possible to implement rate monotonic scheduling in Ada because of a problem known as *priority inversion*. This occurs because the rendezvous is executed at the higher of the two priorities rather than executing it at the priority of the calling task. In addition, Ada priorities are not considered when choosing among calling tasks queued at an entry (FIFO scheduling is used instead), nor in the choice of alternative in a `select` statement (the choice is non-deterministic regardless of the priorities of the queued calling tasks).

Suppose task `High` is executing at priority 8, task `Medium` at priority 7 and task `Low` at priority 6. They may all require the services of task `Server`. What priority should be given to `Server`? Suppose it is given priority 10, and suppose that `Server` is engaged in a rendezvous with `Low`. If `High` suddenly becomes ready to execute, it ought to be able to pre-empt `Server` which is doing work for `Low`. But this is impossible because the rendezvous is executing at priority 10. A priority inversion has occurred between `Low` and `High`.

Now let us assume that `Server` is given priority 5 and is engaged in a rendezvous with `Low`. Let `High` and `Medium` become ready for execution and let `High` call `Server`. Since rendezvous execute under mutual exclusion, it is reasonable to assume that `High` should wait for the conclusion of the rendezvous with `Low`. Then `High` should be scheduled in preference to `Medium`. But upon the conclusion of the rendezvous, `Server` must issue another `select` or `accept` before it can engage in another rendezvous. In the competition between `Server` at priority 5 and `Medium` at 6, `Medium` will win, even though `Server` should be working for `High`. A priority inversion has occurred between `Medium` and `High`.

16.8 Further Reading

[Mac80] discussed real-time systems in the context of Ada. The real-time scheduling algorithms are from [LL73]. Priority inversion in Ada is from [CS87] and an alternative priority strategy for Ada is suggested in [GS84]. Algorithms for real-time memory allocation are given in [For88].

Appendix A

Ada Overview

A.1 Introduction

This appendix is intended to give an overview of the part of the Ada programming language used in this book. It assumes that the reader is experienced in a high-level programming language. preferably Pascal since Ada is built upon its principles.

Lexically, Ada is written in ASCII characters. It is free-form, meaning that punctuation marks rather than format on the page are used to indicate grammatical units. Identifiers are arbitrary length sequences of letters, numbers and underscores beginning with a letter. Upper and lower case letters are not distinguished. Certain identifiers like procedure, if, loop, end are reserved and may not be used for other purposes. Any Ada book will contain a list of the reserved words. A comment starts with a pair of minus-signs and continues until the end of the line.

An Ada program is a procedure which contains a declarative part followed by a sequence of executable statements (Figure A.1). Every variable and every local procedure must be declared before it is used. An initial value may be given for a variable. Procedure parameters must be declared and the procedure calls must have the correct number and type of parameters. A procedure nested within another can access the declarations in the enclosing scope. Procedure parameters have a *mode* associated with them: in, out, in out. in parameters may only be read within the procedure, out parameters may only be written and in out parameters may be both read and written. In the absence of an explicit mode, in is assumed. All parameters of a function must be in parameters. An explicit return statement is used to terminate the execution of the function and return the computed result:

```
procedure Main_Program is
  G: Integer := 0;   -- Global variable
                     -- with initial value
  procedure Inner(P: in Integer) is
    L: Integer;  -- Local variable
  begin
    L := P + 1;
    G := L + G; -- Global variable accessible
  end Inner;

begin
  G := G * 5;    -- Sequence of statements
  Inner(7);      -- including procedure call
  G := G + 1;
end Main_Program;
```

Figure A.1 Ada program structure

```
function Square(I: in Integer) return Integer is
begin
  return I * I;
end Square;
```

A.2 Types and Statements

Ada contains predefined types: Integer, Float, Character, Boolean. The boolean literals are False, True and the character literals are ASCII characters in single quotes: 'a', 'X'. The type String is also predefined; its values are character sequences within double quotes: "Good morning". The usual arithmetic operations are defined on integer and floating point values.

Statements are terminated by semicolons. They include assignment statements and control statements: if, loop, case, return. elsif can be used to program additional branches within an if then else statement:

```
if A > B then
  C := A;
elsif A < B then
  C := B;
else
  C := 0;
end if;
```

A loop statement can be written with no conditional clause in which case it indicates an infinite loop. It can also be controlled by a for clause or a while clause and in addition, exit statements can terminate a loop at any point within its sequence of statements:

```
for I in 1..10 loop
  A(I) := 0;
end loop;
while X > 0 loop
  X := X / 2;
end loop;
loop
  X := X / 2;
  exit when X <= 0;
end loop;
```

The last example shows how an exit statement can simulate a Pascal repeat statement. We use exit extensively in the examples because we are usually interested in waiting for something to occur and this gives a positive statement of the condition for leaving the loop.

Ada provides an extensive facility for creating user defined types:

Enumerations

```
type States is (Off, Run, Start);
```

Arrays

```
type Arr is array(1..100) of Integer;
```

Records

```
type Rec is
  record
    Field1: Integer;
    Field2: Boolean;
  end record;
```

Pointers

```
type Rec_Pointer is access Rec;
```

Tasks See Chapter 8.

Arr is declared above as a constrained array type, which means that its indices are constrained at compilation type and all arrays of this type range over the same indices. *Aggregates* denote values that are entire arrays or records. We use them to initialize arrays:

```
A: array(1..100) of Integer := (others => 0);
```

It is also possible to declare an unconstrained array type and then supply the constraint when the array variable is declared. In particular, strings are implemented this way. To declare a string variable, the index bounds must be given:

```
A1: Arr;               -- A1 has 100 elements
A2: Arr;               --   so does A2

S1: String(1..80);     -- S1 has 80 elements
S2: String(1..40);     -- S2 has 40 elements
S3: String;            -- Invalid.
```

Attributes are functions that are used to access predefined characteristics. For example. States'First and States'Last evaluate to Off and Start. respectively. If the declaration of States is modified, these attributes will evaluate to the new first and last values of the type. There is a rich set of attributes in Ada. Some of the most useful are those defined on arrays: First, Last, Length and Range. Using attributes makes it possible to write programs that need not be modified if a declaration is changed.

```
S: String(1..80);
if S'Length < 50 then       -- fewer than 50 elements in S
  S(S'Last) := S(S'First); -- store first element last
```

```
end if;
for I in S'Range loop     -- loop for all elements in S
  S(I) := '*';
end loop;
```

One feature that we will occasionally use is the addition of a *range constraint* to integer variables. This clearly documents the values that such a variable can have. If an attempt is made to assign a value outside the range, the Ada system will detect the fault.

```
N: Integer range 0..1 := 0;
```

A.3 Packages

The language described so far is just a variant of Pascal. Along with tasking, the package facility is the main distinguishing mark of Ada. Standard Pascal does not contain any structure larger than the procedure except an entire program. Most Pascal compilers do supply a module structure, but this is not done in any uniform way.

A *package* in Ada is a set of data and procedure declarations that can be compiled, placed in a library and later accessed by other packages or by the main program. For example, the standard Ada library contains a package called Text_IO which contains procedures to read and write from a terminal. The program can access the entities in a package by placing a *context clause* before the program:

```
with Text_IO;
procedure Main_Program is
begin
  Text_IO.Put_Line("Good morning");
end Main_Program;
```

The Text_IO package contains a procedure called Put_Line which takes a parameter of type String, prints the parameter on the terminal followed by the control sequence for end-of-line. The dotted notation is used to access the procedure within the package that is mentioned in the with clause just as if it were a field within a record.

Since we often want to access many of the entities within a package, the context clause can include a use clause which allows us to omit the name of the package. However, if this results in any ambiguity, the full dotted notation must be used.

```
with Text_IO; use Text_IO;
procedure Main_Program is
begin
  Put_Line("Good Morning");
end Main_Program;
```

Packages are used to decompose large systems into modules. The decomposition itself does not help if we still have to read the entire program to understand what is going on. Packages are written in two parts to separate out the *interface* to the entities in the package from the *implementation*. The users need only see the interface to the package and the implementation can be changed without affecting them. The interface part is called the package *specification* and consists of data declarations and procedure declarations only. The implementation part is called the package *body* and consists of the procedure bodies that implement the procedures that appear in the specification as well as additional data declarations and procedures (Figure A.2).

```
package Geometry is    -- Specification
  function Circle_Area(Radius: in  Float) return Float;
  function Square_Area(Side:   in  Float) return Float;
end Geometry;

package body Geometry is
  PI: Float := 3.14159;   -- Internal data
  function Square(X: in Float) return Float is
  begin                   -- Internal procedure
    return X * X;
  end Square;

  function Circle_Area(Radius: in  Float) return Float is
  begin
    return PI * Square(Radius);
  end Circle_Area;
  function Square_Area(Side:   in  Float) return Float is
  begin
    return Square(Side);
  end Square_Area;

end Geometry;
```

Figure A.2 Package specification and body

In this book, we typically demonstrate concurrent programming primitives on small examples that can be written in a single program. The package facility is used to provide simulations of the various primitives that can be hidden from the reader. For example, the semaphore package specification provides a semaphore type and specifications of the Wait and Signal primitives. Their implementation in Ada is hidden in the package body which would be of interest only to Ada programmers.

Both the package specification and the package body are *compilation units* which can be separately compiled and saved in a library. Tasks are not compilation units: they must be textually included within another unit, usually a package or the main program. However, it is possible to separately compile a procedure, package or task body as shown in Figure A.3. The body is replaced by the word separate. This *subunit* can then be placed in a file and separately compiled where a clause identifies the parent unit. The use of separate is transparent to the programmer. Visibility within the subunit is the same as if it were textually included in the parent unit. However, any change in the parent unit will require recompilation of the subunit.

```
procedure Main is
  task Buffer is
    entry Append(I: in  Integer);
    entry Take  (I: out Integer);
  end Buffer;

  task Producer;
  task Consumer;

  task body Buffer   is separate;
  task body Producer is separate;
  task body Consumer is separate;
begin
  ...
end Main;

separate(Main)
task body Buffer is ... ;

separate(Main)
task body Producer is ...;

separate(Main)
task body Consumer is ...;
```

Figure A.3 Separate compilation of subunits

To summarize, Ada has:

- Pascal-like structured control statements.
- Pascal-like block structure of procedures.
- The package facility for modularization.
- Built-in concurrent programming (tasks).

Additional aspects of Ada not used in this book are:

- User defined types that significantly extend the Pascal types.
- User processing of errors called exceptions.
- Representation clauses that map programs to hardware.

A.4 Further Reading

The reference manual for Ada is [DOD83]. There are many textbooks on Ada: [Bar89] and [Coh88] are written for experienced programmers who wish to learn Ada, while [WWF87] is appropriate for beginners. [Boo83] is an introduction to designing large systems in Ada, specifically by using packages to implement object-oriented programming.

Appendix B

Concurrent Programs in Ada

B.1 Introduction

Most of the examples shown in this book can be run on Ada systems. The purpose of this appendix is to describe the additional work that must be done to turn the program fragments into executable programs.

Since concurrent programs may not terminate, we must find some means of observing their behavior. One way is to insert print statements into the tasks and the other is to use a good debugger to follow the state of the tasks. It is probably best to use both methods: print statements to get an overall view of the execution and a debugger to observe the state at selected points.

Printing in Ada is done by using the services of the `Text_IO` package. The following three procedures should suffice:

Put(S) Prints the string S.

New_Line Sends the control sequence for a new line.

Put_Line(S) Like `Put(S); New_Line`.

`Put(S)` can be used several times to build up a line in pieces. Note that since the execution of tasks is interleaved, `Put` statements from several tasks may print on the same line. `Put_Line` will probably execute as an atomic operation.

To print numbers, an appropriate package must be created. This is done by compiling the following two-line program:

```
with Text_IO;
package Integer_Text_IO is new Integer_IO(Integer);
```

Then a context clause referencing `Integer_Text_IO` provides the procedures:

Put(N) Prints the integer N.

Put(N,W) Print the integer N in a field of width W.

Procedures from `Text_IO` and `Integer_Text_IO` may be combined in the same program as long as context clauses for both packages appear.

Experienced Ada programmers may prefer to use the `Image` function and string concatenation to prevent interleaving between two `Put` statements:

```
Put_Line("Ticket number = " & Integer'Image(Number(I));
```

The examples are written with infinite loops. Either replace them with terminating loops using `for`-clauses or make sure you know how to terminate a program that is looping (usually Control-C or Control-Break).

The examples were developed on the VAX Ada compiler from Digital Equipment Corporation. Implementation specific features of the VAX Ada compiler are:

- The package `Integer_Text_IO` is predefined and does not need to be compiled as stated above.
- Time-slicing is disabled unless explicitly enabled by a `pragma Time_Slice` clause.
- The standard suggests the use of a `pragma Shared` clause on global variables accessed by more than one task. Instead, VAX Ada supplies a `pragma Volatile` clause with slightly different semantics which should be used for these variables.
- In VAX Ada, both the time slice and the priority of a task can be changed during a debugging session.

The examples were also tested using the Alsys Ada compiler on a PC compatible computer. `pragma Shared` is supported by this implementation. To correctly execute the concurrent programming examples, the default parameters must be changed as follows:

```
default.bind(timer=>fast,slice=>10).
```

B.2 Common Memory

Figure B.1 shows a complete program for the first attempted solution to the mutual exclusion problem. It will only run on systems that implement time-slicing, otherwise the busy-wait loops will never terminate. The tasks are declared within a main procedure. The main procedure body is `null` since all the computation goes on in the dependent tasks. The main procedure will terminate when the dependent tasks terminate. Either add `for`-clauses to the loops in the task body or abort the program from the terminal.

This example will alternately print the message from each of the critical sections. If one of the idling sections is replaced by a very long computation, we can demonstrate how the other task is delayed in spite of lack of contention. The second attempt will deadlock after printing only one result. It will be difficult to demonstrate livelock and starvation since the time slice is long relative to the execution of the statements.

```
with Text_IO; use Text_IO;
procedure First is
  pragma Time_Slice(0.01);        -- VAX/Ada
  Turn: Integer := 1;
  pragma Volatile(Turn);          -- VAX/Ada

  task T1;
  task body T1 is
  begin
    loop
      Put_Line("Task 1 idling");
      loop exit when Turn = 1; end loop;
      Put_Line("Task 1 critical section");
      Turn := 2;
    end loop;
  end T1;

  task T2;
  task body T2 is
  begin
    loop
      Put_Line("Task 2 idling");
      loop exit when Turn = 2; end loop;
      Put_Line("Task 2 critical section");
      Turn := 1;
    end loop;
  end T2;
begin
  null;
end First;
```

Figure B.1 First attempt at mutual exclusion

B.3 Semaphores

Semaphores are implemented in a package which supplies both binary and general semaphores (Figure B.2). The package must be mentioned in a context clause in a program which uses it (Figure B.3). Since the semaphores are declared as task types. data structures containing semaphores may be created.

The implementation of semaphores is standard (Figure B.4). For binary semaphores. successive accept statements are sufficient. We include them in a select statement with a terminate alternative so that the user program can terminate. Note that if Wait and Signal are not balanced, the task will not terminate. Since entry queues are FIFO, this is an implementation of a blocked queue semaphore.

The general semaphore includes an initializing entry. A select statement accepts both Wait and Signal with a guard on the Wait to make sure that it does not decrease the semaphore value below 0. Note that this is not a blocked queue semaphore since there is no guarantee that the select will choose a Wait following a Signal.

```
package Semaphore_Package is

  task type Binary_Semaphore is
    entry Wait;
    entry Signal;
  end Binary_Semaphore;

  task type Semaphore is
    entry Init(N: in Integer);
    entry Wait;
    entry Signal;
  end Semaphore;

end Semaphore_Package;
```

Figure B.2 Semaphore package specification

```
with Text_IO; use Text_IO;
with Semaphore_Package; use Semaphore_Package;
procedure Mutex is
  S: Binary_Semaphore;

  task T1;
  task body T1 is
  begin
    loop
      Put_Line("Task 1 is idling");
      S.Wait;
      Put_Line("Task 1 critical section");
      S.Signal;
    end loop;
  end T1;

  task T2;
  task body T2 is
  begin
    loop
      Put_Line("Task 2 is idling");
      S.Wait;
      Put_Line("Task 2 critical section");
      S.Signal;
    end loop;
  end T2;

begin
  null;
end Mutex;
```

Figure B.3 Mutual exclusion using semaphores

```
package body Semaphore_Package is

  task body Binary_Semaphore is
  begin
    loop
      select
        accept Wait;
        accept Signal;
      or
        terminate;
      end select;
    end loop;
  end Binary_Semaphore;

  task body Semaphore is
    Count: Integer;
  begin
    accept Init(N: Integer) do
      Count := N;
    end Init;
    loop
      select
        when Count > 0 =>
          accept Wait do
            Count := Count - 1;
          end Wait;
      or
        accept Signal do
          Count := Count + 1;
        end Signal;
      or
        terminate;
      end select;
    end loop;
  end Semaphore;

end Semaphore_Package;
```

Figure B.4 Implementation of semaphores

B.4 Monitors

Monitors are difficult to implement directly in Ada because there is no direct way of implementing condition variables by having a task leave a rendezvous, call another entry and then return to the original rendezvous. A close approximation is given in Figures B.5 and B.6. Figure B.5 supplies the tasking services to implement a monitor. Enter and Leave are essentially semaphores which control mutual exclusion to the monitor. Condition variables are implemented as task types providing the entries Signal and Wait. The function Non_Empty calls an additional entry in the condition task.

```
package Monitor_Package is
  task Monitor is
    entry Enter;
    entry Leave;
  end Monitor;

  task type Condition is
    entry Signal;
    entry Wait;
    entry Waiting(B: out Boolean);
  end Condition;

  function Non_Empty(C: Condition) return Boolean;
end Monitor_Package;
```

Figure B.5 Tasking services for a monitor

Using Monitor_Package one can write 'monitors' as packages. Figure B.6 shows the monitor for the producer–consumer which exports two procedures: Append and Take. Note that Enter and Leave have to be explicitly programmed – something that would normally be done implicitly by the monitor implementation. Signal is programmed to automatically release the monitor since it is restricted to be the last statement in a procedure. Now it is possible to write the program for the producer–consumer program using the monitor (Figure B.7). We have each task print the value of the elements produced and consumed so we can check that the program correctly preserves the order of the elements.

The implementation of mutual exclusion on a monitor is a simple binary semaphore (Figure B.8). The implementation of a condition variable is more difficult. If no processes are waiting, Signal simply releases mutual exclusion. If a process executes Wait, the condition task executes a nested accept Signal thereby blocking the waiting process until the condition is signaled. Since the signaling process holds mutual exclusion on the monitor, the awakened process automatically inherits the right to continue execution.

The code for the monitor package is complicated by the requirement that queries on Non_Empty always succeed. This is the reason for the loop within the accept Wait. If the queue status is queried, the select is re-issued. Only a rendezvous with Signal will exit the loop.

```
package Producer_Consumer_Monitor is
  procedure Append(V: in  Integer);
  procedure Take  (V: out Integer);
end Producer_Consumer_Monitor;

with Monitor_Package; use Monitor_Package;
package body Producer_Consumer_Monitor is
  Not_Empty, Not_Full: Condition;
  In_Ptr, Out_Ptr:     Integer := 0;
  Count:               Integer := 0;
  Buffer:              array(0..19) of Integer;

  procedure Append(V: in Integer) is
  begin
    Monitor.Enter;
    if Count = Buffer'Length then
      Monitor.Leave;
      Not_Full.Wait;
    end if;
    Buffer(In_Ptr) := V;
    In_Ptr := (In_Ptr + 1) mod Buffer'Length;
    Count := Count + 1;
    Not_Empty.Signal;
  end Append;

  procedure Take(V: out Integer) is
  begin
    Monitor.Enter;
    if Count = 0 then
      Monitor.Leave;
      Not_Empty.Wait;
    end if;
    V := Buffer(Out_Ptr);
    Out_Ptr := (Out_Ptr + 1) mod Buffer'Length;
    Count := Count - 1;
    Not_Full.Signal;
  end Take;
end Producer_Consumer_Monitor;
```

Figure B.6 Monitor for producer consumer

```
with Text_IO; use Text_IO;
with Integer_Text_IO;
with Producer_Consumer_Monitor; use Producer_Consumer_Monitor;
procedure Producer_Consumer is
  pragma Time_Slice(0.01);   -- VAX/Ada

  task Producer;
  task body Producer is
    N: Integer := 0;
  begin
    loop
      N := N + 1;
      Put("Produce  ");
      Integer_Text_IO.Put(N);
      New_Line;
      Append(N);
    end loop;
  end Producer;

  task Consumer;
  task body Consumer is
    N: Integer;
  begin
    loop
      Take(N);
      Put("Consume ");
      Integer_Text_IO.Put(N);
      New_Line;
    end loop;
  end Consumer;

begin
  null;
end Producer_Consumer;
```

Figure B.7 Producer consumer solution with monitors

```
package body Monitor_Package is

  function Non_empty(C: Condition) return Boolean is
    B: Boolean;
  begin
    C.Waiting(B);
    return B;
  end Non_empty;

  task body Monitor is
  begin
    loop
      accept Enter;
      accept Leave;
    end loop;
  end Monitor;

  task body Condition is
  begin
    loop
      select
        when Wait'Count = 0 =>
          accept Signal do Monitor.Leave; end Signal;
      or
        accept Wait do
          loop
            select
              accept Signal;
              exit;
            or
              accept Waiting(B: out Boolean) do
                B := True;
              end Waiting;
            end select;
          end loop;
        end Wait;
      or
        accept Waiting(B: out Boolean) do
          B := Wait'Count /= 0;
        end Waiting;
      end select;
    end loop;
  end Condition;

end Monitor_Package;
```

Figure B.8 Monitor package implementation

Appendix C

Implementation of the Ada Emulations

C.1 Introduction

This chapter presents the Ada emulations of occam and Linda. While the Linda emulation packages can be used with only the minimal knowledge of Ada required for this book, the occam emulation requires a sophisticated knowledge of the language.

C.2 occam

Aside from the prioritized alternate and repetitive alternate features, the concurrent programming primitives of occam are very similar to Ada and occam programs can be directly translated into Ada. However, to preserve the spirit of occam, an emulation should try to preserve the concept of 'channel' which is declared in a scope global to the communicating tasks rather than placing the declaration of the entries in the accepting task. In the case of the matrix multiplication example, the tasks around the boundary of the matrix (8 out of 9 in the example) would require special programming.

We have developed a way to emulate occam channels as Ada tasks so that the computational tasks can all be of the same type. Unfortunately, this requires a non-trivial data structure and initialization scheme which must be created separately for each new problem. The matrix multiplication example is given in full detail and can be used as a model for solving other problems.

Each channel is declared as an additional task with an entry named Output. The process executing an occam Output statement can simply call this entry in the channel task. The problem is with processes executing occam Input statements. We want to allow them to execute select statements. Thus the channel tasks must *call* entries in the accepting tasks. In Ada, this requires that the channel task know the name of the accepting task and the entry name. This is accomplished during initialization of the channel task. The problem is that it is impossible to pass a 'task' as a parameter. We can only pass a pointer to a specific task type.

Rather than have different code for each channel,[1] we use variant records to pass a pointer to a general task type. The variant record and the channel code will need modification for each problem, but this is not difficult to do.

We give the complete listing of the matrix multiplication program which can be used as a guide to writing other emulations. The main program (Figure C.1) declares all the tasks and channels. Task_Types has a value for each type of task in the program and is used as the discriminant in the variant record Tasks. This variant record assumes a transputer-like solution where every process uses at most four channels which are initialized with channel indices. The varying part contains a field for the task.

The channels are numbered sequentially and the Configure procedure assigns channel indices to each of the tasks. The computations are detailed but not difficult if following on a diagram (Figure C.2). Initialization is done by calling the Init entry of each task with a pointer to the data structure of type Tasks. The task stores its channel indices and calls the Destination entry of each outgoing channel to configure the channel.

The channel task body (Figure C.3) stores the task pointer and the channel index. It then suspends on a select statement waiting for some process to execute an output statement. The select has a terminate alternative so that the channels can terminate once the processes themselves have done so. Once a value has been received, the channel stores it and then calls the destination process. A case statement on the discriminant of task record is used to call the correct task and the channel index is used as an index in an entry family. Calls to tasks which are not destinations of any channel cause an exception to be raised.

While the configuration of the system is extremely difficult, the programming of the processes themselves is straightforward (Figure C.4).

[1] More precisely, for each channel calling a different destination task type.

```
procedure Matrix is

  type Task_Types is (Multiplier, Source, Sink, Zero, Result);

  Size: constant Integer := 3;
  type Vector is array(1..Size) of Integer;
  Matrix1: array(1..Size) of Vector :=
    ((1,2,3),(4,5,6),(7,8,9));
  Matrix2: array(1..Size) of Vector :=
    ((1,0,2),(0,1,2),(1,0,0));

  type Channels is range 1..2*Size*(Size+1)+1;

  type Tasks(Task_Type: Task_Types);
  type Task_Ptr is access Tasks;

  task type Multiplier_Task is
    entry Init(Coeff: Integer; T: Task_Ptr);
    entry Input (Channels)(I: in  Integer);
  end Multiplier_Task;

  task type Source_Task is
    entry Init(V: Vector; T: Task_Ptr);
  end Source_Task;

  task type Sink_Task is
    entry Init(T: Task_Ptr);
    entry Input (Channels)(I: in  Integer);
  end Sink_Task;

  task type Zero_Task is
    entry Init(T: Task_Ptr);
  end Zero_Task;

  task type Result_Task is
    entry Init(ID: Integer; T: Task_Ptr);
    entry Input (Channels)(I: in  Integer);
  end Result_Task;

  type Tasks(Task_Type: Task_Types) is
    record
      North, East, South, West: Channels;
      case Task_Type is
        when Multiplier => M: Multiplier_Task;
        when Source     => S: Source_Task;
        when Sink       => T: Sink_Task;
        when Zero       => Z: Zero_Task;
        when Result     => R: Result_Task;
      end case;
    end record;
```

Figure C.1 Main program for matrix multiplication

```
task type Channel_Task is
  entry Destination(T_Ptr: Task_Ptr; Chan: Channels);
  entry Output(I: in Integer);
end Channel_Task;

M: array(1..Size, 1..Size) of Task_Ptr;
S: array(1..Size) of Task_Ptr;
T: array(1..Size) of Task_Ptr;
Z: array(1..Size) of Task_Ptr;
R: array(1..Size) of Task_Ptr;

Channel: array(Channels range 1..Channels'Last-1) of Channel_Task;

task body Multiplier_Task is separate;
task body Source_Task     is separate;
task body Sink_Task       is separate;
task body Zero_Task       is separate;
task body Result_Task     is separate;
task body Channel_Task    is separate;

procedure Activate is
begin
  M := (others => (others => new Tasks(Multiplier)));
  S := (others => new Tasks(Source));
  T := (others => new Tasks(Sink));
  Z := (others => new Tasks(Zero));
  R := (others => new Tasks(Result));
end Activate;

procedure Configure is
  N: Channels;
begin
  N := 1;
  for I in 1..Size loop
    S(I).South := N;  M(1,I).North    := N;
    N := N + 1;
    Z(I).West  := N;  M(I,Size).East  := N;
    N := N + 1;
    T(I).North := N;  M(Size,I).South := N;
    N := N + 1;
    R(I).East  := N;  M(I,1).West     := N;
    N := N + 1;
    for J in 2..Size loop
      M(I,J).West := N;  M(I,J-1).East := N;
      N := N + 1;
    end loop;
    if I /= Size then
      for J in 1..Size loop
        M(I,J).South := N;  M(I+1,J).North := N;
        N := N + 1;
      end loop;
    end if;
  end loop;
end Configure;
```

Figure C.1 Main program for matrix multiplication (*continued*)

```
procedure Init is
begin
  for I in 1..Size loop
    R(I).R.Init(I, R(I));
    S(I).S.Init(Matrix2(I), S(I));
    T(I).T.Init(T(I));
    Z(I).Z.Init(Z(I));
    for J in 1..Size loop
      M(I,J).M.Init(Matrix1(I)(J), M(I,J));
    end loop;
  end loop;
end Init;

begin
  Activate;
  Configure;
  Init;
end Matrix;
```

Figure C.1 Main program for matrix multiplication (*continued*)

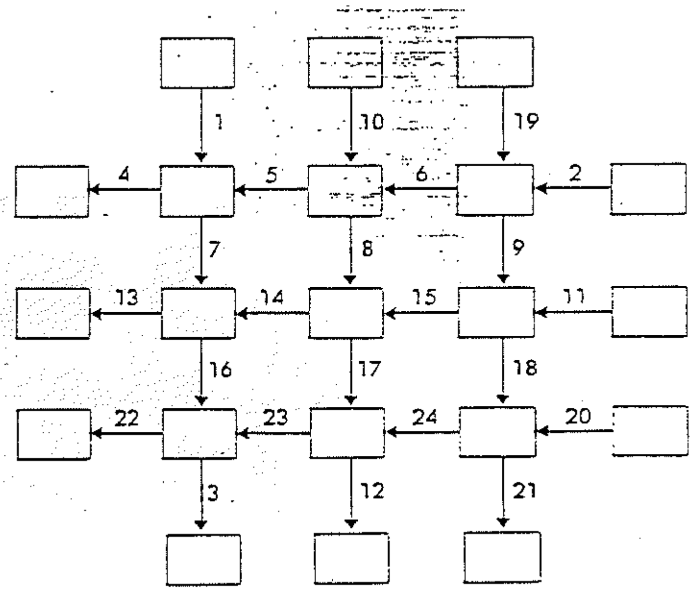


Figure C.2 Assignment of channels to processes

```
separate(Matrix)
task body Channel_Task is
  T:  Task_Ptr;
  Ch: Channels;
  Save: Integer;
  Invalid_Channel: exception;
begin
  accept Destination(T_Ptr: Task_Ptr; Chan: Channels) do
    T  := T_Ptr;
    Ch := Chan;
  end Destination;
  loop
    select
      accept Output(I: in Integer) do
        Save := I;
      end Output;
    or
      terminate;
    end select;
    case T.Task_Type is
      when Multiplier => T.M.Input(Ch)(Save);
      when Source     => raise Invalid_Channel;
      when Sink       => T.T.Input(Ch)(Save);
      when Zero       => raise Invalid_Channel;
      when Result     => T.R.Input(Ch)(Save);
    end case;
  end loop;
end Channel_Task;
```

Figure C.3 Channel task

```
separate(Matrix)
task body Multiplier_Task is
  North, East, South, West: Channels;
  X, Sum, A: Integer;
begin
  accept Init(Coeff: Integer; T: Task_Ptr) do
    A := Coeff;
    North := T.North; East := T.East;
    South := T.South; West := T.West;
    Channel(North).Destination(T, T.North);
    Channel(East) .Destination(T, T.East);
  end Init;
  for N in 1..Size loop
      accept Input(North)(I: in Integer) do
        X := I;
      end Input;
      Channel(South).Output(X);
      accept Input(East) (I: in Integer) do
        Sum := I;
      end Input;
      Sum := Sum + A*X;
      Channel(West).Output(Sum);
  end loop;
end Multiplier_Task;

separate(Matrix)
task body Source_Task is
  South: Channels;
  Vec: Vector;
begin
  accept Init(V: Vector; T: Task_Ptr) do
    Vec := V;   South := T.South;
  end Init;
  for N in 1..Size loop
    Channel(South).Output(Vec(N));
  end loop;
end Source_Task;

separate(Matrix)
task body Sink_Task is
  North: Channels;
begin
  accept Init(T: Task_Ptr) do
    North := T.North;
    Channel(North).Destination(T, T.North);
  end Init;
  for N in 1..Size loop
    accept Input(North)(I: in Integer);
  end loop;
end Sink_Task;
```

Figure C.4 Tasks for occam matrix multiplication

```
separate(Matrix)
task body Zero_Task is
  West: Channels;
begin
  accept Init(T: Task_Ptr) do
    West := T.West;
  end Init;
  for N in 1..Size loop
    Channel(West).Output(0);
  end loop;
end Zero_Task;

with Text_IO; use Text_IO;
with Integer_Text_IO; use Integer_Text_IO;
separate(Matrix)
task body Result_Task is
  East: Channels;
  Ident: Integer;
begin
  accept Init(ID: Integer; T: Task_Ptr) do
    Ident := ID;  East := T.East;
    Channel(East).Destination(T, T.East);
  end Init;
  for N in 1..Size loop
    accept Input(East)(I: in Integer) do
      Put(Ident); Put(N); Put(I); New_Line;
    end Input;
  end loop;
end Result_Task;
```

Figure C.4 Tasks for occam matrix multiplication (*continued*)

C.3 Linda

Two packages have been written to embed Linda primitives in Ada. Tuple_Defs creates and accesses tuples and Tuple_Package manages the tuple space. Figure C.5 contains the specification of package Tuple_Defs. A user of the package need only know that there exist types Tuple_Element and Tuple and functions that convert from these types to ordinary data types.

Vector is an unconstrained array of integers used in the matrix multiplication example. Tuples can consist of a sequence of one to four values of type Tuple_Element. Each of these can be of types: Integer, Character, Boolean, String, Vector. Formal values are defined which can be used in the creation of tuples. Finally, conversion routines convert ordinary values to tuple elements and conversely, extract values from tuples. The extraction functions take a index denoting the element to be extracted.

```
package Tuple_Defs is

  type Vector   is array(Positive range <>) of Integer;

  type Int_Ptr  is access Integer;
  type Char_Ptr is access Character;
  type Bool_Ptr is access Boolean;
  type Str_Ptr  is access String;
  type Vec_Ptr  is access Vector;

  type Tuple_Types is (None, Ints, Chars, Bools, Strs, Vecs);

  type Tuple_Element(Tuple_Type: Tuple_Types := None) is
    record
      case Tuple_Type is
        when None => null;
        when Ints  => I: Int_Ptr;
        when Chars => C: Char_Ptr;
        when Bools => B: Bool_Ptr;
        when Strs  => S: Str_Ptr;
        when Vecs  => V: Vec_Ptr;
      end case;
    end record;

  Null_Element: constant Tuple_Element:=(Tuple_Type=>None);

  Formal_Int:   constant Tuple_Element := (Ints,  null);
  Formal_Char:  constant Tuple_Element := (Chars, null);
  Formal_Bool:  constant Tuple_Element := (Bools, null);
  Formal_Str:   constant Tuple_Element := (Strs,  null);
  Formal_Vec:   constant Tuple_Element := (Vecs,  null);

  type Tuples is array(1..4) of Tuple_Element;

  Null_Tuple: constant Tuples:=(others=>(Tuple_Type=>None));

  function Int(I:  Integer)   return Tuple_Element;
  function Char(C: Character) return Tuple_Element;
  function Bool(B: Boolean)   return Tuple_Element;
  function Str(S:  String)    return Tuple_Element;
  function Vec(V:  Vector)    return Tuple_Element;

  function Int(T:  Tuples; Index: Integer) return Integer;
  function Char(T: Tuples; Index: Integer) return Character;
  function Bool(T: Tuples; Index: Integer) return Boolean;
  function Str(T:  Tuples; Index: Integer) return String;
  function Vec(T:  Tuples; Index: Integer) return Vector;

  function Create_Tuple(T1, T2, T3, T4: Tuple_Element :=
       (Null_Element)) return Tuples;
  function Match(T1, T2: Tuples) return Boolean;

end Tuple_Defs;
```

Figure C.5 Tuple definitions

```
package Tuple_Package is

  function  Input (T: Tuples) return Tuples;
  function  Read  (T: Tuples) return Tuples;
  procedure Output(T: Tuples);

  function  Input (T1, T2, T3, T4: Tuple_Element :=
        Null_Element) return Tuples;
  function  Read  (T1, T2, T3, T4: Tuple_Element :=
        Null_Element) return Tuples;
  procedure Output(T1, T2, T3, T4: Tuple_Element :=
        Null_Element);

end Tuple_Package;
```

Figure C.6 Tuple Space package

The package Tuple_Package defines the Linda primitives (Figure C.6). Each primitive can take one to four elements as parameters or an entire tuple. Input and Read are functions which return a matching tuple.

```
with Text_IO; use Text_IO;
with Integer_Text_IO; use Integer_Text_IO;
with Tuple_Defs; use Tuple_Defs;
with Tuple_Package; use Tuple_Package;
procedure Matrix is
  Result: Tuples;
  task type Workers;
  Worker: array(1..2) of Workers;
  task body Workers is separate;

begin
  Output(Char('A'), Int(1), Vec((1,2,3)));
  Output(Char('A'), Int(2), Vec((4,5,6)));
  Output(Char('A'), Int(3), Vec((7,8,9)));
  Output(Char('B'), Int(1), Vec((1,0,1)));
  Output(Char('B'), Int(2), Vec((0,1,0)));
  Output(Char('B'), Int(3), Vec((2,2,0)));
  Output(Char('N'), Int(1));

  Put_Line("  Row    Col   Result");
  for I in 1..3 loop
    for J in 1..3 loop
      Result := Input(Char('C'), Int(I), Int(J), Formal_Int);
      Put(I,4); Put(J,6); Put(Int(Result,4),8); New_Line;
    end loop;
  end loop;
end Matrix;
```

Figure C.7 Matrix multiplication main program

The main program for matrix multiplication is very simple (Figure C.7). It creates the matrix row and column tuples, initializes the task counter and then reaps the result tuples. Note the use of the conversion functions to create tuple elements from ordinary values and the use of the formal integer tuple element when reading the result. The example shows two worker tasks, but this is trivial to modify by changing the array declaration. The worker tasks are shown in Figure C.8. The extraction functions are used in the computation of the inner product to extract values from the tuple.

```
separate(Matrix)
task body Workers is
  Next, Row_Tuple, Col_Tuple: Tuples;
  Element, I, J, Sum: Integer;
begin
  loop
    Next := Input(Char('N'), Formal_Int);
    Element := Int(Next, 2);
    Output(Char('N'), Int(Element+1));
    exit when Element > 3 * 3;
    I := (Element - 1)  /  3 + 1;
    J := (Element - 1) mod 3 + 1;
    Row_Tuple := Read(Char('A'), Int(I), Formal_Vec);
    Col_Tuple := Read(Char('B'), Int(J), Formal_Vec);

    Sum := 0;
    for N in 1..3 loop
      Sum := Sum + Vec(Row_Tuple,3)(N) * Vec(Col_Tuple,3)(N);
    end loop;

    Output(Char('C'), Int(I), Int(J), Int(Sum));
  end loop;
end Workers;
```

Figure C.8 Worker tasks for matrix multiplication

As shown above, a familiarity with essential points of the package specifications suffices to write programs that emulate Linda. The rest of the section contains information on the implementation of the emulation.

Tuple elements are implemented by variant records each of which contains a pointer to the element value (Figure C.5). A null pointer denotes a formal tuple element. A tuple is a fixed length array of tuple elements where unused elements are filled with the null element. Note that Create_Tuple is not used in the matrix multiplication example since the Linda primitives have been defined to take sequences of tuples and call Create_Tuple themselves (Figure C.6). Function Match is used internally by the tuple space manager. The body of Tuple_Defs is straightforward (Figure C.9). The only non-trivial part is matching tuple elements which must be done on a case by case basis for each type.

```
package body Tuple_Defs is

  function Int(I: Integer) return Tuple_Element is
  begin
    return (Ints, new Integer'(I));
  end Int;

  function Char(C: Character) return Tuple_Element is
  begin
    return (Chars, new Character'(C));
  end Char;

  function Bool(B: Boolean) return Tuple_Element is
  begin
    return (Bools, new Boolean'(B));
  end Bool;

  function Str(S: String) return Tuple_Element is
  begin
    return (Strs, new String'(S));
  end Str;

  function Vec(V: Vector) return Tuple_Element is
  begin
    return (Vecs, new Vector'(V));
  end Vec;

  function Int(T: Tuples; Index: Integer) return Integer is
  begin
    return T(Index).I.all;
  end Int;

  function Char(T: Tuples; Index: Integer) return Character is
  begin
    return T(Index).C.all;
  end Char;

  function Bool(T: Tuples; Index: Integer) return Boolean is
  begin
    return T(Index).B.all;
  end Bool;

  function Str(T: Tuples; Index: Integer) return String is
  begin
    return T(Index).S.all;
  end Str;

  function Vec(T: Tuples; Index: Integer) return Vector is
  begin
    return T(Index).V.all;
  end Vec;
```

Figure C.9 Tuple definitions package body

```
  function Create_Tuple(T1, T2, T3, T4: Tuple_Element :=
      (Null_Element)) return Tuples is
  begin
    return (T1, T2, T3, T4);
  end Create_Tuple;

  function Element_Match(E1, E2: Tuple_Element)
      return Boolean is
  begin
    case E1.Tuple_Type is
      when None => return True;
      when Ints =>
        return E1=Formal_Int  or else E2=Formal_Int  or else
               E1.I.all = E2.I.all;
      when Chars =>
        return E1=Formal_Char or else E2=Formal_Char or else
               E1.C.all = E2.C.all;
      when Bools =>
        return E1=Formal_Bool or else E2=Formal_Bool or else
               E1.B.all = E2.B.all;
      when Strs =>
        return E1=Formal_Str  or else E2=Formal_Str  or else
               E1.S.all = E2.S.all;
      when Vecs =>
        return E1=Formal_Vec  or else E2=Formal_Vec  or else
               E1.V.all = E2.V.all;
    end case;
  end Element_Match;

  function Match(T1, T2: Tuples) return Boolean is
  begin
    for J in Tuples'Range loop
      if T1(J).Tuple_Type /= T2(J).Tuple_Type
                  or else
         not Element_Match(T1(J), T2(J)) then
           return False;
      end if;
    end loop;
    return True;
  end Match;
end Tuple_Defs;
```

Figure C.9 Tuple definitions package body (*continued*)

Package Tuple_Package (Figures C.6 and C.10) contains the TS which is a fixed array of tuples. An exception is defined if TS is full. Tasks are declared which are responsible for mutual exclusion to the TS and for blocking Input and Read requests if the tuple is not available. There are two versions of each primitive: one which takes a tuple parameter and the other which takes a sequence of tuple values. Non-blocking primitives are not implemented.

```
with Tuple_Defs; use Tuple_Defs;
package body Tuple_Package is

  Tuple_Space: array(0..50) of Tuples :=
     (others => Null_Tuple);
  Out_of_Tuple_Space: exception;

  task Space_Lock is
    entry Lock;
    entry Unlock;
  end Space_Lock;

  task Suspend is
    entry Release;
    entry Notify;
    entry Request;
  end Suspend;

  task body Space_Lock is separate;
  task body Suspend    is separate;

  function Find_Tuple(T: in Tuples) return Integer is
  begin
    Tuple_Space(0) := T;
    for I in reverse Tuple_Space'Range loop
      if Match(T, Tuple_Space(I)) then return I; end if;
    end loop;
  end Find_Tuple;

function Find_Tuple_or_Suspend(T: Tuples; Remove: Boolean)
    return Tuples is
  T1: Tuples;
  I: Integer;
begin
  loop
    Space_Lock.Lock;
    I := Find_Tuple(T);
    if I /= 0 then
      T1 := Tuple_Space(I);
      if Remove then Tuple_Space(I):=Null_Tuple; end if;
      Space_Lock.Unlock;
      return T1;
    else
      Suspend.Notify;
      Suspend.Request;
    end if;
  end loop;
end Find_Tuple_or_Suspend;
```

Figure C.10 Body of Tuple Space package

```
procedure Output(T: Tuples) is
  I: Integer;
begin
  Space_Lock.Lock;
  I := Find_Tuple(Null_Tuple);
  if I = 0 then raise Out_of_Tuple_Space; end if;
  Tuple_Space(I) := T;
  Suspend.Release;
end Output;

procedure Output (T1, T2, T3, T4: Tuple_Element :=
   Null_Element) is
begin
  Output(Create_Tuple(T1, T2, T3, T4));
end Output;

function Input(T: Tuples) return Tuples is
begin
  return Find_Tuple_or_Suspend(T, Remove => True);
end Input;

function  Input  (T1, T2, T3, T4: Tuple_Element :=
   Null_Element) return Tuples is
begin
   return Input(Create_Tuple(T1, T2, T3, T4));
 end Input;

function Read(T: Tuples) return Tuples is
begin
  return Find_Tuple_or_Suspend(T, Remove => False);
end Read;

function  Read(T1, T2, T3, T4: Tuple_Element :=
   Null_Element) return Tuples is
begin
   return Read(Create_Tuple(T1, T2, T3, T4));
end Read;

end Tuple_Package;
```

Figure C.10 Body of Tuple Space package (*continued*)

Output searches for a null tuple in TS and replaces it with the parameter. Input and Read search for a matching tuple. If found, it can be returned (and deleted for Input). If not, the process suspends. The reason for the two-instruction suspend will be described when the task body for Suspend is shown.

The tuple space uses two tasks (Figure C.11). Task Space_Lock implements a semaphore to ensure that only one task accesses TS at any time.

Suspension on non-existing tuples is difficult. We decide that every Output instruction will awaken all suspended processes which then proceed to try to

```
separate(Tuple_Package)
task body Space_Lock is
begin
  loop
    select
      accept Lock;
      accept Unlock;
    or
      terminate;
    end select;
  end loop;
end Space_Lock;

separate(Tuple_Package)
task body Suspend is
  Suspended: Integer := 0;
begin
  loop
    select
      accept Release;
       for I in 1..Suspended loop
        accept Request;
      end loop;
      Suspended := 0;
    or
      accept Notify;
      Suspended := Suspended + 1;
    or
      terminate;
    end select;
    Space_Lock.Unlock;
  end loop;
end Suspend;
```

Figure C.11 Tuple Space tasks

match the new tuple, suspending themselves again if necessary. The problem is how do we know how many processes must be awakened? The obvious way to do this is to have Output call an entry, here named Release, which will then loop accepting all of the Request'Count processes which are suspended. It is clear that a suspended process must have unlocked TS before calling the Request entry, otherwise no other process could ever execute Output. We have to watch out for the following scenario:

1. Initially no processes suspended and empty TS.
2. P1 executes Input and locks TS.
3. P1 does not find the required tuple.
4. P1 unlocks TS.
5. P2 executes Output with matching tuple and locks TS.
6. P2 notes that no processes are suspended.

7. P2 unlocks TS.
8. P1 suspends waiting for a tuple that is actually there.

The solution is to have suspended processes make an initial call to the Suspend task to Notify that it will suspend. Since this is executed while holding the lock, the count of suspended processes will always be correct. There is no problem holding the lock, because the accept body is empty and will never block the calling task. Notify will release the lock on the TS which will allow processes to execute Output and find out exactly how many processes are suspended. Note that the solution works because Input and Read use simple entry calls without timeouts.

Termination of tasks in a library package such as Tuple_Package is not defined in the standard though the VAX Ada implementation will terminate these two tasks. If problems occur on other compilers, Tuple_Package can be declared as a local package within the main program.

Appendix D

Distributed Algorithms in Ada

D.1 Introduction

In this appendix we present the details of an Ada implementation of distributed algorithms that is suitable for laboratory experimentation. It has been used to implement the Ricart-Agrawala algorithm for distributed mutual exclusion, the DS and TM algorithms for distributed termination and the snapshot algorithm. The complete code of the TM algorithm will be given followed by discussion of the other algorithms. Not included will be the Put_Line statements scattered throughout the code to instrument the program.

D.2 The TM Algorithm

The nodes of the distributed system are declared as an array of tasks, one task per node. Communications among the nodes can be done simply by indexing the node ID and calling the Message or Signal entry. The main process is declared as a subtask within the node task. The permits concurrency between the main process and the communications process yet allows them to access the same global variables declared in the node task. They cannot both be tasks in a package because it is not possible to create a data structure of packages to index the nodes. The main process must be a subtask of the communications task and not conversely because accept statements must appear in the outermost level of a task.

Figure D.1 shows the outer framework of the program. Each node has two entries for initialization purposes. The first is used to give a node its ID and the number of incoming and outgoing edges. Then the node executes an accept Configure statement for each edge. It is passed the ID of the task at the other end of the edge. Entry calls in the main program (Figure D.2) construct the node topology. Once configuration is complete, Main_Process can be initiated and the main loop entered.

```
with Semaphore_Package; use Semaphore_Package;
procedure TM is
  type Node_Count is range 0..4;
  subtype Node_ID is Node_Count range 1..Node_Count'Last;

  task type Nodes is
    entry Init(ID: Node_ID; N_I, N_O: Node_Count);
    entry Configure(C: Node_ID);
    entry Message(M: Integer; ID: Node_ID);
    entry Signal(ID: Node_ID);
  end Nodes;
  Node: array(Node_ID) of Nodes;

  task body Nodes is
    -- Global variables
    -- Main Process task
  begin
    accept Init(ID: Node_ID; N_I, N_O: Node_Count) do
      I := ID;
      N_In  := N_I;
      N_Out := N_O;
    end Init;
    for J in 1..N_In loop
      accept Configure(C: Node_ID) do
        Incoming(C).Exists := True;
      end Configure;
    end loop;
    for J in 1..N_Out loop
      accept Configure(C: Node_ID) do
        Outgoing(C).Exists := True;
      end Configure;
    end loop;
    Main_Process.Init;

    -- Main loop
  end Nodes;
begin
  -- Configure node topology
end TM;
```

Figure D.1 Framework of TM algorithm

The global variables are shown in Figure D.3.[1] The only additions to the declarations shown in the description of the algorithm are the variables N_In and N_Out which are used during configuration only. The main loop of the node process is straightforward (Figure D.4). A terminate alternative has been added to the select statement. When Main_Process terminates, the node task will

[1] In VAX Ada global variables used by several processes should be declared with pragma Volatile.

```
Node(1).Init(1,0,2);
Node(1).Configure(2); Node(1).Configure(3);

Node(2).Init(2,2,2);
Node(2).Configure(1); Node(2).Configure(3);
Node(2).Configure(3); Node(2).Configure(4);

Node(3).Init(3,3,1);
Node(3).Configure(1); Node(3).Configure(2);
Node(3).Configure(4); Node(3).Configure(2);

Node(4).Init(4,1,1);
Node(4).Configure(2); Node(4).Configure(3);
```

Figure D.2 Configure node topology

```
type Edge is
  record
    Exists:  Boolean := False;
    Active:  Boolean := False;
    Marker_Received: Boolean := False;
  end record;

Incoming: array(Node_ID) of Edge;
Outgoing: array(Node_ID) of Edge;
N_In, N_Out: Node_Count := 0;

First_Edge:  Node_Count := 0;
N_Signals:   Natural := 0;

I: Node_ID;
S: Binary_Semaphore := Init(1);
Received_ID: Node_ID;
Received_Data: Integer;
```

Figure D.3 Global variables

also be able to terminate since all its sibling node tasks will eventually be waiting on the terminate alternative. Main_Process (Figure D.5) has an initiate entry to prevent it from commencing execution until the node process completes the configuration of the global variables.

There is a difference between the the processing done in the source node and the processing done in other nodes. The source node sends two messages followed by a marker on each outgoing channel. Then it simply waits for termination. Internal nodes go through the following stages:

1. Wait to be engaged (a message has been received on First_Edge).
2. Send five messages on each outgoing channel.
3. Wait for a marker on First_Edge.

```
loop
  select
    accept Message(M: Integer; ID: Node_ID) do
      Received_ID := ID;
      Received_Data := M;
    end Message;
    if Received_Data < 0 then
      Incoming(Received_ID).Marker_Received := True;
    else
      if First_Edge = 0 then
        First_Edge := Received_ID;
      end if;
    end if;
    if not Incoming(Received_ID).Active then
      Incoming(Received_ID).Active := True;
    end if;
  or
    accept Signal(ID: Node_ID) do
      Received_ID := ID;
    end Signal;
    Outgoing(Received_ID).Active := False;
    Wait(S);
    N_Signals := N_Signals - 1;
    Signal(S);
  or
    terminate;
  end select;
end loop;
```

Figure D.4 Main loop of communications task

4. Send markers on each outgoing channel.
5. Wait for marker on each incoming channel.
6. Execute the Decide_to_Terminate function until successful and then terminate.

To demonstrate the algorithms, it was assumed that each node sends each message on all outgoing channels. so a loop over all existing outgoing channels is programmed in Send_Message (Figure D.6).

Received_Markers (Figure D.7) is straightforward - checking if markers have been received from each incoming channel. Decide_to_Terminate (Figure D.8) has been discussed in Chapter 12.

```
task Main_Process is
  entry Init;
end Main_Process;

task body Main_Process is
  Count: Integer := 0;
  Markers_Sent: Boolean := False;
begin
  accept Init;
  if I = 1 then
    Send_Messages(Count);
    Send_Messages(Count);
    Send_Messages(-1);
    loop
      exit when Decide_to_Terminate;
    end loop;
  else
    loop
      if Count < 5 then
        if First_Edge /= 0 then
           Count := Count + 1;
           Send_Messages(Count);
        end if;
      elsif not Markers_Sent then
        if Incoming(First_Edge).Marker_Received then
          Send_Messages(-1);
          Markers_Sent := True;
        end if;
      elsif Received_Markers then
          loop
            exit when Decide_to_Terminate;
          end loop;
          exit;
      end if;
    end loop;
  end if;
end Main_Process;
```

Figure D.5 Main process task

```
procedure Send_Messages(Data: Integer) is
begin
  for J in Node_ID loop
    if Outgoing(J).Exists then
      if not Outgoing(J).Active then
        Outgoing(J).Active := True;
        Wait(S);
        N_Signals := N_Signals + 1;
        Signal(S);
      end if;
      Node(J).Message(Data, I);
    end if;
  end loop;
end Send_Messages;
```

Figure D.6 Sending messages

```
function Received_Markers return Boolean is
begin
  for J in Node_ID loop
    if Incoming(J).Exists and then
      not Incoming(J).Marker_Received then
         return False;
    end if;
  end loop;
  return True;
end Received_Markers;
```

Figure D.7 Receiving markers

```
function Decide_to_Terminate return Boolean is
  procedure Send_Signals(ID: Node_ID) is
  begin
    Node(ID).Signal(I);
    Incoming(ID).Active := False;
  end Send_Signals;

begin
  for J in Node_ID loop
    if  J /= First_Edge and then
        Incoming(J).Active then
          Send_Signals(J);
    end if;
  end loop;

  if N_Signals = 0 then
    if I = 1 then
      null;
    elsif First_Edge /= 0 then
      Send_Signals(First_Edge);
      First_Edge := 0;
    end if;
    return True;
  else
    return False;
  end if;
end Decide_to_Terminate;
```

Figure D.8 Checking for termination

D.3 The Ricart–Agrawala Algorithm

This algorithm for distributed mutual exclusion can be implemented exactly as described in Chapter 11 or it can fit into the paradigm described here. Instead of three tasks. the request and reply tasks are converted into a node communications task which contains the main process as a subtask (Figure D.9). When compared with the distributed termination algorithms, the mutual exclusion algorithm is simpler because the process graph must be complete, each process connected to each other. Thus we can dispense with the Incoming and Outgoing data structures. Finally. each main process makes a constant number of entries to the critical section and then terminates. The terminate alternative allows the node processes to terminate.

```
procedure RA is
  task type Nodes is
    entry Init(ID: Node_ID);
    entry Request_Message(Num: Integer; ID: Node_ID);
    entry Reply_Message;
  end Nodes;
  Node: array(Node_ID) of Nodes;

  task body Nodes is
    -- Global variables
    -- Main process
  begin
    accept Init(ID: Node_ID) do
      I := ID;
      Main_Process.Init;
    end Init;
    loop
      select
        accept Request_Message(Num: Integer; ID: Node_ID) do
          Received_Number := Num;
          Received_ID     := ID;
        end Request_Message;
        Received_Request;
      or
        accept Reply_Message;
        Received_Reply;
      or
        terminate;
      end select;
    end loop;
  end Nodes;
begin
  for J in 1..N loop
    Node(J).Init(J);
  end loop;
end RA;
```

Figure D.9 Distributed mutual exclusion

D.4 The Dijkstra–Scholten Algorithm

The main process of the DS algorithm implementation is shown in Figure D.10. The source process sends two messages on all outgoing nodes while the internal processes send up to five messages. The source process will terminate upon receiving a final signal. However, the internal processes may be repeatedly disengaged and re-engaged as new messages come in. This is modeled by having the process execute Decide_to_Terminate after sending each message and then checking if First_Edge still indicates that the process is engaged. Thus internal nodes will not terminate and the execution of the program must be manually interrupted.

```
task body Main_Process is
  Count: Integer := 0;
begin
  accept Init;
  if I = 1 then
    Send_Messages;
    Send_Messages;
    loop
      exit when Decide_to_Terminate;
    end loop;
  else
    loop
      loop
        exit when First_Edge /= 0;
      end loop;
      if Count < 5 then
        Count := Count + 1;
        Send_Messages;
      end if;
      loop
        exit when not Decide_to_Terminate or First_Edge /= 0;
      end loop;
    end loop;
  end if;
end Main_Process;
```

Figure D.10 Main process of DS algorithm

D.5 Snapshots

The implementation of the snapshot algorithm is straightforward except for printing the answers. The main process (Figure D.11) sends messages and occasionally initiates spontaneous recording of the state. When all messages have been sent, the main process waits in a loop until State_Recorded is true and markers have been received on all incoming channels. Then it prints the recorded state. As a final detail, an additional semaphore Print (which is global to all nodes) is used so that the lines of printing from the nodes do not interleave.

```
task body Main_Process is
  function Write_State return Boolean is
  begin
    if not State_Recorded then return False; end if;
    for J in Node_ID loop
      if Incoming(J).Exists and not Incoming(J).Marker_Passed then
        return False;
      end if;
    end loop;
    Wait(Print);
      -- Print state of this node
    Signal(Print);
    return True;
  end Write_State;

begin
  accept Init;
  for J in 1..9 loop
    Send_Messages(J);
    case I is
      when 2 | 3 => null;
      when 1 => if J = 6 then Record_State; end if;
      when 4 => if J = 3 then Record_State; end if;
    end case;
  end loop;
  loop exit when Write_State; end loop;
end Main_Process;
```

Figure D.11 Main process of snapshot algorithm

Bibliography

[ACG86] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8): 26–34, 1986.

[Bar89] J.G.P. Barnes. *Programming in Ada*. Addison-Wesley, Reading, MA, 1989.

[Ben84] M. Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Trans. Program. Lang. and Syst.*, 6(3): 333–44, 1984.

[BEW88] D. Bustard, J. Elder, and J. Welsh. *Concurrent Programming Structures*. Prentice Hall International, Hemel Hempstead, 1988.

[BH77] P. Brinch Hansen. *The Architecture of Concurrent Programs*. Prentice Hall, Englewood Cliffs, NJ, 1977.

[Boo83] G. Booch. *Software Engineering with Ada*. Benjamin/Cummings, Menlo Park, CA, 1983.

[BR85] T.P. Baker and G.A. Riccardi. Ada tasking: From semantics to efficient implementation. *IEEE Software*, 2(2): 34–46, 1985.

[Bur85] A. Burns. *Concurrent Programming in Ada*. Cambridge University Press, Cambridge, 1985.

[CDJ84] F.B. Chambers, D.A. Duce, and G.P. Jones (eds). *Distributed Computing*. Academic Press, London, 1984.

[CG89] N. Carriero and D. Gelernter. How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys* (forthcoming).

[CL85] K.M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Computer Syst.*, 3(1): 63–75, 1985.

[CM84] K.M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. and Syst.*, 6(4): 632–46. 1984.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, Reading, MA, 1988.

[Coh88] N.H. Cohen. *Ada as a Second Language*. McGraw-Hill, New York, 1988.

[CS87] D. Cornhill and L. Sha. Priority inversion in Ada. *Ada Letters*, VII(7): 30–2, 1987.

[Dei85] H. M. Deitel. *An Introduction to Operating Systems*. Addison-Wesley, Reading, MA, 1985.

[Dij68] E.W. Dijkstra. Co-operating sequential processes. In F. Genuys (ed.) *Programming Languages*, Academic Press, New York, 1968.

[Dij71] E.W. Dijkstra. Hierarchial ordering of sequential processes. *Acta Informatica*, 1(2): 115–38, 1971.

[DLM78] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11): 966–75, 1978.

[DS80] E.W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1): 1–4, 1980.

[DOD83] US Department of Defense. *The Ada Programming Language*. US Government Printing Office, Washington, DC, 1983.

[For88] R. Ford. Concurrent algorithms for real-time memory management. *IEEE Software*, 5(5): 10–23, 1988.

[Fra86] N. Francez. *Fairness*. Springer Verlag, New York, 1986.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. and Syst.*, 7(1): 80–112, 1985.

[Gri81] D. Gries. *The Science of Programming*. Springer Verlag, New York, 1981.

[GS84] D. Gifford and A. Spector. The TWA reservation system. *Commun. ACM*, 27(7): 650–65, 1984.

[Hoa74] C.A.R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10): 549–57, 1974.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Commun. ACM*. 21(8): 666–77, 1978.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, Hemel Hempstead, 1985.

[How76] J.H. Howard. Proving monitors. *Commun. ACM*, 19(5): 273–9. 1976.

[In85] Inmos Limited. *Transputer Reference Manual*. Prentice Hall International, Hemel Hempstead, 1985.

[In88] Inmos Limited. *occam 2 Reference Manual*. Prentice Hall International, Hemel Hempstead, 1988.

[JG88] G. Jones and M. Goldsmith. *Programming in occam 2*. Prentice Hall International, Hemel Hempstead, 1988.

[Lam74] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8): 453–5, 1974.

[Lam86] L. Lamport. The mutual exclusion problem. I-II. *Journal ACM*, 33(2): 313–48, 1986.

[Lam87] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1): 1–11, 1987.

[LL73] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogamming in a hard-real-time environment. *Journal ACM*, 20(1): 46–61, 1973.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. and Syst.*, 4(3): 382–401, 1982.

[Mac80] L. MacLaren. Evolving toward Ada in real time systems. *SIGPLAN Notices*, 15(11): 146–55, 1980.

[MC82] J. Misra and K.M. Chandy. Terminating detection of diffusing computations in Communicating Sequential Processes. *ACM Trans. Program. Lang. and Syst.*, 4(1): 37–43, 1982.

[MP81] Z. Manna and A. Pnueli. Verification of concurrent programs: the temporal framework. In R.S. Boyer and J.S. Moore (eds) *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[OG76] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5): 279–85, 1976.

[Pet81] G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3): 115–6, 1981.

[Pet83] G.L. Peterson. A new solution to Lamport's concurrent programming problem using small shared variables. *ACM Trans. Program. Lang. and Syst.*, 5(1): 56–65, 1983.

[PM87] D. Pountain and D. May. *A Tutorial Introduction to occam Programming*. McGraw-Hill, New York, 1987.

[Pra86] D.K. Pradhan (ed.). *Fault-Tolerant Computing*. Prentice Hall, Englewood Cliffs, NJ, 1986.

[PS85] J. Peterson and A. Silberschatz. *Operating Systems Concepts*. Addison-Wesley, Reading, MA, 1985.

[Qui87] M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, New York, 1987.

[RA81] G. Ricart and A. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1): 9–17, 1981.

[Ray86] M. Raynal. *Algorithms for Mutual Exclusion*. MIT Press, Cambridge MA, 1986.

[SG84] A. Spector and D. Gifford. The Space Shuttle primary computer system. *Commun. ACM*, 27(9): 874–900, 1984.

[Shr85] S.K. Shrivastava. *Reliable Computer Systems*. Springer Verlag, Berlin, 1985.

[Sta80] T.A. Standish. *Data Structure Technqiues*. Addison-Wesley, Reading, MA, 1980.

[Sta82] E.W. Stark. Semaphore primitives and starvation-free mutual exclusion. *Journal ACM*, 29(4): 1049–72, 1982.

[TCN84] M. Tedd, S. Crespi-Reghizzi, and A. Natali. *Ada for Multimicroprocessors*. Cambridge University Press, Cambridge, 1984.

[WWF87] D.A. Watt, B.A. Wichmann, and W. Findlay. *ADA: Language and Methodology*. Prentice Hall International, Hemel Hempstead, 1987.

Index