ON THE MULTITASKING MECHANISMS OF THE ADA LANGUAGE*

Abraham Silberschatz

TR-152 July 1980

* This research was supported in part by the National Science Foundation under Grant MCS 7702463, and in part by the Office of Naval Research under contract N00014-80-C-0592.

Abstract

The multitasking mechanisms of the Ada language are intended to provide a facility for writing concurrent programs. Accept statements, select statements and task procedures are the three main features of the language that deal with the issues of communication and synchronization. This paper points out several problems that arise in connection with these features and proposes various solutions to them.

Key Words and Phrases: Ada, Multitasking, communication, synchronization

CR Categories: 4.20, 4.22, 4.32

1. Introduction

The multitasking mechanisms of the Ada language [1,2] are intended to provide a facility for writing concurrent programs. Central to this facility is the concept of the task, which is a program module that is executed asynchronously. Tasks are disjoint in address spaces; that is, they do not share any variables in common. Tasks communicate and synchronize their actions through:

(a) accept statements and entry declarations -- a combination of procedure calls and message transfer.

(b) select statements -- the sequential control structure based on Dijkstra's guarded commands [3].

(c) task procedures--a mechanism to allow several activities to be executed in parallel within a single task.

We now briefly elaborate on issues (a) and (b) above. Discussion concerning issue (c) will be deferred to section 4.

Central to the communication facility is the accept statement which has the form*:

```
accept <entry name> [<formal parameter list>];
    [do <statements> end]
```

the statements of an accept statement can be executed only if another task invokes the entry-name; at this point in time parameters are also passed. After the end statement has been reached parameter may be passed back, and both tasks are free to continue. Note that either the calling task or the called task may be suspended until the other task is ready with its corresponding communication. Thus the facility serves both

*square brackets ([ ]) denote an optional part, while curly brackets ({ }) denote repetition of zero or more times.

as a communication mechanism and a synchronization tool.

Entry declarations can also specify a family of identical entries, each denoted by an index from a discrete range. In this case every call must be subscripted by an index. This facility is useful in providing additional synchronization power since a caller can channel the call to an appropriate entry.

Choices among several entry calls is accomplished by the select statement which is based on Dijkstra's guarded command concept and which has the form:

```
select
     [when <boolean-expression> =>]
          <select-alternative>
           [<statements>]
{or [when <boolean expression>  =>]
          <select-alternative>
           [<statements>]}
[else <statements>]
end select;
```

A select-alternative may be an accept statement or a delay statement. A delay statement allows the programmer to suspend the execution of a task for at least the given time interval specified.

Execution of a select statement proceeds as follows:

(1) All the boolean expressions appearing in the statement are evaluated. Each select-alternative whose corresponding expression is evaluated to be true is tagged as open alternative.

An alternative with no preceding when clause is always tagged as open.

(2) An open alternative starting with an accept statement may be selected for execution if another task has called upon an entry corresponding to that accept statement. If several open alternatives may be selected an arbitrary one will be chosen for execution. If none can be selected and there is an else part, the else part is executed. If there is no else part, then a delay statement may be selected for execution (provided that the appropriate time interval has elapsed). Otherwise, if there is no else part, then the task waits until an open alternative can be selected.

(3) If no alternative is opened and there is an else part, the else part is executed. Otherwise an exception condition is raised.

A select statement cannot contain both an else part and alternatives starting with delay statements.

The accept statement provides the task with a mechanism to wait for a predetermined event. On the other hand the select statement provides the task with a mechanism to wait for events whose order cannot be predicted in advance. These two constructs (together with task procedures) are the only synchronization tools available in Ada. Although these mechanisms have a rich range of features they nevertheless are not sufficiently powerful to program many common synchronization problems in a straightforward and concise manner. It is our aim here to elaborate on this issue.

2. Priority-ordered queues

A major area of application where the multitasking facility of Ada can be utilized is resource scheduling. In such a scheme a scheduler task must be provided whose function is to coordinate orderly access to a resource. Such a scheduler accepts calls from customer tasks and processes them in some fashion. Since the customer might have to be delayed before an access permission can be granted a mechanism must be available to achieve this end. Ada supports such a mechanism by providing: (a) the concept of family-of-entries; and (b) the when clause. These two concepts allow one to partially simulate static priority-ordered queues. A customer can be delayed in one of these queues only at its initial entry to the scheduler.

The vast majority of standard examples in the literature can be cleanly implemented by delaying a customer task just once in a static priority-ordered queue. The problem is that a significant amount of code may have to be executed before the scheduler knows which queue and what priority is appropriate. The code may access either shared variables (i.e. those variables declared locally to the scheduler task), or, private variables (i.e. formal parameters passed to an accept statement). Such a scheme cannot be coded in Ada in a straightforward manner (if at all!). Schedulers written in Ada (see [2], pages 11_21-11_28) tend to use awkward layers of nested calls in which the separate procedures or entries have been introduced solely to allow waiting at their entry points. This introduces unneccessary overhead and makes program harder to read and verify.

One example requiring processing before queuing (in which both shared and private variables are used) is the interprocess communications facility (IPC) from [4]. Suppose many tasks must occasionally communicate via message buffers, where the messages are addressed by a rendezvous code. Since the producer-consumer dialogs are infrequent, dynamic allocation of message buffers from a pool is therefore more appropriate than static allocation for all possible rendezvous codes. A task needing to communicate calls the IPC allocator task with a rendezvous code to obtain a message buffer (Open), writes one or more messages to the buffer (Send), and releases the buffer (Close). Another task capable of handling messages with that rendezvous code would call the allocator with the same rendezvous code (Open), read messages (Receive), and then release the buffer (Close).

The producers and consumers may or may not have the IPC open at the same time. When an Open request is received, the IPC allocater must search the Buffers array to see if the requestor's rendezvous code is already associated with some message buffer. If the code is in use, the subsequent requests for Send or Receive will be directed to the existing message buffer. If the rendezvous code is not currently associated with a message buffer, then one must be allocated (or the request delayed until a message buffer is available.) The association between a rendezvous code and a message buffer is not actually broken until the buffer is empty and no process has that rendezvous code open.

A similar situation may occur in a hierarchical file system where different users may know the same file under different names. When a user attempts to open a file, it may take the file system a number of disk accesses to discover that the file is the same one which has already been opened for exclusive access (e.g., write or update) by another task. As was the case with the IPC, the file allocater must do a significant amount of preliminary processing before it even knows whether the requesting task will have to wait. This cannot be cleanly done (if at all) in Ada.

What we have pointed out above is that Ada lacks an effective mechanism to handle priority scheduling. In this section we develop such a mechanism which is a variant of Kessels conditional wait construct [5]. Central this mechanism is the command

await (<boolean-expression>)

The await statement can appear only in a select-alternative.

With each await statement a list (called an await-list) is associated consisting of entries (activation records) each of which contains:

(a) the formal parameter (if there are any) passed to the accept statement associated with the select-alternative where the await is defined

(b) Information concerning the calling task (if the select alternative is an accept statement)

(c) address of the statement following the await statement.

An activation record is used to save a partial state of a computation so that this state can be restored at a later point in time.

When a task encounters an await statement it evaluates the boolean-expression associated with this statement. If true, the task continues its execution. If false, the task obeys the following:

(i) it creates a new activation record, initializes it and adds it to the appropriate await-list.

(ii) it proceeds with its execution at the first statement following the select statement in which the await is defined.

Thus an await statement provides the programmer with a mechanism to switch between various distinct computations.

With this new mechanism we have to redefine the manner in which the execution of a select statement proceeds. In the introduction we have defined three steps which must be followed in the execution of a select statement. With the addition of the await statement we add another step to the computation which precedes all these three steps:

(0) Consider only the await statements defined in the select statement being executed.

(i) if no await statements are defined or if each await-list is empty proceed to step 1.

(ii) Tag all activation records whose associated boolean-expression is true as <u>ready</u> records (note that if the boolean expression of an await statement contains only shared variables then the activation records of this statement are either all ready or none is ready; that is, only one evaluation is required.) If no activation record is ready proceed to step 1.

(iii) Pick a ready activation record and remove it from the associated await-list. Restore the state of the task using the information obtained from the removed record.

Note that if a state restoration occurs, the value of the program counter is also being affected.

It should be clear that if the boolean expression of an await statement ranges over private variables (as well as shared variables) then the number of expressions that need to be evaluated (in the process of tagging ready records) will be proportional to the number of entries in the associated await-list. On the other hand, if the boolean expression contains only shared variables only one evaluation is needed. Thus, from an efficiency standpoint one should avoid the inclusion of private variables in an await statement whenever possible.

In order to aid the programmer in achieving this end, we add one more feature to the await statement to give the programmer a closer control over scheduling. We extend the await statement to allow the inclusion of priority information [6].

await (<boolean expression>) [by (<priority-expression>)]

The by clause is optional. When a task encounters an await statement with a false boolean-expression, then the priority-expression is evaluated to produce an integer priority value. This priority value is also stored in the activation record added to the await-list. When an activation record is picked during the execution of a select statement (step (0) (iii)), the activation record with the smallest priority value is removed. If no priority value is present an arbitrary record is removed. This new feature allows the coding of many common synchronization problems (e.g., shortest-job-next, alarm clock, disk schedulers) with the boolean expression restricted to range only over shared variables.

Let us illustrate these concepts by considering a simple resource scheduling scheme. Suppose that one wishes to define a task whose function is to allocate a resource among a number of tasks in the shortest job next order.

```
          task body  SJN is:
               Free:  boolean: = true;
          begin
           loop
            select
             accept  Acquire (Time: in integer) do
              await (Free) by (Time);
               Free: = false;
             end;
           or
            accept release;
              Free := true;
           end select;
          end loop;
         end SJN;
```

It should be pointed out that the shortest job next scheme could not be coded in a straightforward manner in the original Ada (as a matter of fact it is not clear whether it is possible to do it at all!)

Let us consider now the more complicated example of the IPC scheme sketched above. We assume that there are 10 message buffers to be allocated among customer tasks. For brevity we only present those program segments that illustrate our concepts. The main data structure needed are:

```
     Resource:  array (1..10) of record
                                    Rendezvous-number: integer
                                    Hold-count: integer
                                 end
     Count, Index, Last-channel: integer: = 0;
```

Central to the IPC allocator is the select statement described below whose function is to accept two types of calls: Open and Close. Open has two formal parameters: Channel-id -- the rendezvous code supplied by the caller, and Buffer-id -- the index corresponding to the allocated message-buffer.

```
  select
     accept  Open (Channel-id: integer; Buffer-id: out integer) do
      for I in  1..10  loop
       if  Resource(I). Rendezvous-number = Channel-id then
           Resource(I). Hold-count: = Resource(I). Hold-count + 1;
           Buffer-id: = I;
           exit;
       end if;
       end loop,
       await (Count < 10 or Channel-id = Last-channel);
        if Count < 10 then
         for I in 1..10 loop
          if Resource(I). Hold-count = 0 then
           Resource(I). Rendezvous-number: = Channel-id;
           Resource(I). Hold-count = 1;
           Buffer-id: = I;
           Index: = I;
           Last-Channel: = Channel-id;
           Count: = Count + 1;
            exit;
          end if;
         end loop;
       else
         Resource (Index). Hold-count: = Resource(Index). Hold-count +1;
         Buffer-id: =  Index
       end if;

    or
     accept Close (Buffer-id: integer);
       Resource (Buffer-id). Hold-count = Resource (Buffer-id). Hold-count-1;
       if Resource (Buffer-id).Hold-count = 0 then
          Resource (Buffer-id). Rendezvous-number: = 0;
          Count: = Count - 1;
       end if;
    end select;
```

Note that a considerable amount of code needs to be executed before the await statement is encountered.

Let us consider another example which will allow us to compare Ada's synchronization scheme with ours. The example is of a controller for the allocation of group of items from a set of resources (see [2] page 11_23).

```
task  Multi-resource-control is
  type  Resource is (A,B,C,D,E,F,G,H,I,J,K);
  type  Resource-set is array (A..K) of Boolean,
  entry Reserve (Group: Resource-set);
  entry Release (Group: Resource-set);
end;

task body  Multi-resource-control is
  Empty: constant   Resource-set: = (A..K => false);
  Used:  Resource-set: = Empty;
begin
  loop
    select
      accept Reserve  (Group: Resource-set) do
        await ((Used and Group) = Empty)
          Used: = Used or Group;
      end;
    or
      accept Release (Group: Resource-set)
          Used: = Used and not Group;
    end select;
  end loop;
 end Multi-resource-control;
```

it would be interesting to compare our solution with Ada's. Our solution is much more concise, easier to understand and more efficient. This is because in Ada one had to use awkward layers of nested calls in which the separate procedures or entries have been introduced solely to allow waiting at their entry points.

We conclude this section by commenting on two issues concerning our synchronization scheme.

(a) The await statement was specifically designed so that it can be implemented efficiently. If the boolean expressions range only over shared variables then the number of evaluations needed to perform a specific task is minimal. We have introduced the by clause as an extension to the await statement so that a larger number of synchronization schemes could be coded with the boolean expression restricted to shared variables only.

Additional saving can be obtained by resorting to various optimization techniques (e.g. Schmidt [7]). For example, in the Multi-resource-control task described above the await statement needs to be evaluated only after a release call has been accepted.

(b) We have tried our proposed scheme on various scheduling problems. We have found that our algorithms were usually more concise and easier to understand than those coded in the original Ada (this should not surprise the reader; after all our synchronization scheme has a richer range of features than the one in Ada).

3. Select Statement

The select statement as defined in Ada is too restrictive because every select-alternative must either be an accept statement or a delay statement. There are many circumstances where one may want the select-alternative to be more general. To illustrate this we consider an example.

Consider a background task which almost always has useful work to do but whose operation is occasionally modified by an external request (entry call). This situation was illustrated in [8] with the bounded buffer problem. A system is to be designed consisting of two tasks, Producer and Consumer. The Producer task produces some information that the Consumer task consumes. Since the two tasks may run at different speeds, one may construct a buffering scheme which consists of N buffers which can be filled and emptied in some fashion by the Producer and Consumer. This problem has been traditionally programmed by constructing a buffering task which includes the N buffers and the operations Send and Receive that are invoked by the Producer and Consumer respectively. In order to take advantage of the distributed-processing architecture, one should associate each of these tasks with a separate processor (thus increasing parallelism). This, however, will result in an unnecessary copying from the address space of Producer, to the address of the buffer task and then to the address space of the Consumer. Since in a distributed system no shared memory is assumed this extra copying is very expensive; it should be avoided if possible [8]. An alternative is to incorporate the N buffers in the Producer task. This will minimize the amount of copying necessary.

In such an environment the Producer task alternates between two activities: producing new buffers and transmitting the content of the buffers to the Consumer. Thus the central command in the Producer task must be a select statement in which one of the alternatives is an accept statement (whose function is to transmit the contents of the buffers to the Consumer), and the other alternative is the code corresponding to the production of new buffers. New buffers can be produced only if there is at least one empty buffer.

This formulation of the Producer can be coded within the framework of Ada in various ways; none however would be considered satisfactory. One way to code the Producer is to rely on busy waiting. Another way is to duplicate code. Yet another way is to "cheat" by using a delay statement with time interval equal to zero. The most natural way would be to code the Producer as follows:

```
task body Producer is

     Buffer: array (0..9) of portion;
     In, Out: integer: = 0;
     P: portion;
     begin
     loop
       select
         when Out < In =>
          accept Receive (X: out portion) do
            X: = Buffer (Out mod 10);
          end
          Out:= Out + 1

       or when Out < In + 10 =>
          PRODUCE (P)
          Buffer (IN mod 10):=P
          In:= In + 1;
       end select;
    end loop;
  end Producer;
```

Unfortunately, the above is not a valid Ada program because a select alternative must be either an accept statement or a delay statement.

We propose that the select statement be modified to as follows:

(a) The syntax is as originally definied

(b) A select-alternative may be:

i) an accept statement

ii) a delay statement

iii) an arbitrary statement; in this case the when clause <u>must</u> precede this alternative.

(c) Execution of a select statement proceeds as defined above, with the additional understanding that a select-alternative which is an arbitrary statement can always be selected for execution provided that the boolean expression in the when clause (which must be present) is true.

It should be clear that the modified select statement is more general than the original one. It corresponds more closely to the Dijkstra's guarded command [3] and Hoare's alternative command [9].

4. Task Procedures

Procedures in a task specification can be called concurrently from several other tasks. One of the main reasons for introducing task procedures in Ada is to provide a safe mechanism for handling resource scheduling schemes. With this mechanism one can encapsulate the resource within the scheduling task thereby ensuring that users of the scheduler cannot directly access the component resources or their internal representation. Thus most properties of the resources (e.g., mutual exclusion, ordering constraints, priorities) are defined and enforced by the scheduler task.

Although it is important for a language such as Ada to provide a mechanism for the safe handling of resource scheduling schemes, we nevertheless disagree with the choice of the task procedure concept as a mechanism to achieve this end. We can think of three reasons in support of our claim.

(1) If several task procedures can simultaneously access the same shared variables, time dependent errors can occur [10]. Much has been said on this subject; suffice it to say that in recent years we have seen a trend in language design to disallow concurrent activities within the same name space (see, CSP[9], Concurrent Pascal [11], Modula [12], DP [13].)

(2) The multitasking mechanisms of Ada were designed to be implemented on two different kinds of architecture:

(i) Shared-memory--a few physical processors share a common memory

(ii) Distributed-processing--a large number of processors with local memory but no shared memory.

Task procedures can be efficiently and effectively implemented in a shared memory architecture; this however is not the case in distributed-processing architecture. This is due to the fact that in such an environment concurrency is achieved by multiplexing rather than by having real concurrent activities. This defeats the purpose of the distributed processing architecture.

(3) The only method for achieving safe resource scheduling schemes in Ada is to force the user of a resource to always reach the resource through the scheduler. The task procedure concept was specifically designed to support such a scheme. However, forcing the encapsulation of a resource with the scheduling task may result in a significant amount of overhead. There are many circumstances where this overhead would not be acceptable.

A good example is the common case of a scheduler which simply allocates from a pool of uniform resources (e.g., the IPC allocator described above, a pool of tape drives on a large computer system). Another example is the case where Ada is implemented on a distributed processing architecture. In such an environment each access would have to go through some interprocessor communications channels in order to reach the central scheduler. Since there is no shared memory, call-by-reference is impossible. Call-by-value must be used between processors. Every extra layer of schedulers between the user task and the resource therefore requires an additional transfer of the parameters associated with every access to the resource. This is clearly very expensive!

In a real system that magnitude of overhead would cause the centralized scheduler to be rejected in favor of some less structured but more efficient technique. We believe however that one could provide more effective and efficient mechanisms in Ada to allow the safe handling of resource scheduling schemes.

One approach is to divorce the resource from the scheduler except during allocation and release. Such an approach was advocated in [4, 14], by proposing a new type of module called manager. When an allocation is made, the scheduler ("manager" of the resource) grants the requesting process a "capability" for the allocated resource. Thereafter the process accesses the resource directly via the capability. Another approach using a capability scheme was described in [15]. The advantage of a capability scheme is that accessing of the resource is via the central scheduler only when coordination is actually required.

The capability scheme protects the internal representation of the resource and provides control over the updating of shared variables. It does not ensure that constraints on the order of operation be enforced if accesses can be made without going through the central scheduler (e.g. open a file before reading it). The problem was considered in [16, 17]. The approach taken was to explicitly describe the allowable sequence of operation for each module by means of augmented regular expressions.

For the reasons listed above we strongly suggest that the designer of Ada consider replacing the task procedure construct by more structured and efficient constructs. A good starting point might be the previous work described above.

5. Conclusion

The accept statement, select statement and the task procedures are the three features of the language that provide the programmer with the needed tools for handling communication and synchronization. We have pointed out several problems that arise in connection with those features and have proposed various solutions to them.

We have shown that the restriction that a calling task can be delayed only at its initial entry to another task does not allow one to code various synchronization schemes in a straightforward manner. We have proposed the concept of the await statement to resolve this difficulty.

We have shown that the select statement as defined in Ada is too restrictive because every select alternative must either be an accept statement or delay statement. A modification to remedy this situation was proposed.

Finally, we have argued that the task procedure mechanism has several deficiencies. Discussion concerning possible alternatives was presented.

References

[1] Preliminary Ada Reference Manual, ACM SIGPLAN Notices 14, 6 (June 1979), part A.

[2] Ichbiah, J.D., et al, Rationale for the Design of the ADA Programming Language, ACM SIGPLAN Notices 14, 6 (June 1979), part B.

[3] Dijkstra, E.W., Guarded commands, nondeterminancy, and formal derivation of programs, Comm ACM 18, 8 (Aug. 1975), 453-457.

[4] Silberschatz, A., Kieburtz, R.B., and Bernstein, A.J., Extending Concurrent Pascal to Allow Dynamic Resource Management, IEEE Transactions on Software Engineering 3, 3 (May 1977), 210-217.

[5] Kessels, J.L.W., An alternative to event queues for synchronization in monitors, Comm ACM 20, 7 (July 1977), 500-503.

[6] Hoare, C.A.R., Monitors: an operating system structuring concept, Comm ACM 17, 10 (Oct 1974), 549-557.

[7] Schmid, H.A., On the Efficient Implementation of Conditional Critical Regions and the Construction of Monitors, Acta Informatica 6 (1976), 227-279.

[8] Silberschatz, A., On the Decomposition of Distributed Systems into Modules, Technical Report #73, University of Texas, 1980.

[9] Hoare, C.A.R., Communicating Sequential Processes, Comm ACM 21 8 (August 1978), 666-677.

[10] Brinch Hansen, P., Operating System Principles, Prentice-Hall, Englewood Cliffs, N.J., 1977.

[11] Brinch Hansen, P., The Programming language Concurrent Pascal, IEEE Trans. on Software Engr 1, 2 (June 1975), 199-207.

[12] Wirth, N., Modula: a programming language for modular multiprogramming, Software Practice and Experience, 7, 1 (Jan 1977), 3-35.

[13] Brinch Hansen, P., Distributed processes: a concurrent programming concept, Comm ACM 21, 11 (Nov. 1978), 934-940.

[14] Kieburtz, R.B., and Silberschatz, A., Capability Managers, IEEE Transactions on Software Engineering, 6 (Nov. 1978), 467-477.

[15] Andrews, G.R., and McGraw, J.R., Language Features for Process Interaction, ACM SIGPLAN Notices 12, 3 (March 1977), 114-127.

[16] Lauer, P.E., Torrigiani, P.R., and Shields, M.W., COSY--A System Specification Language Based on Paths and Processes, Acta Informatica 12, 2 (1979), 109-158.

[17] Kieburtz, R.B. and Silberschatz, A., Access Right Expressions, Technical Report #44, University of Texas, 1978.