ON THE SYNCHRONIZATION MECHANISM OF THE ADA LANGUAGE

Abraham Silberschatz

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-222 March 1983

ON THE SYNCHRONIZATION MECHANISM OF THE ADA LANGUAGE[1]

Abraham Silberschatz

Department of Computer Sciences
The University of Texas
Austin, Texas 78712

ABSTRACT

The synchronization mechanism of the Ada language is intended to provide a facility for tasks to synchronize their actions. The accept and select statements are the two main features of the language that deal with the issue of synchronization. This paper points out one major problem that arises in connection with these features and proposes a possible solution to it.

[1]This research was supported in part by the National Science Foundations under Grant MCS 8104017, and in part by the Office of Naval Research under Contract N00014-80-K-0987

1. Introduction

The multitasking mechanisms of the Ada language [1] 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 may communicate and synchronize their actions through:

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

b. select statements -- the non-deterministic control structure based on Dijkstra's guarded command construct [2].

The accept statement provides a task with a mechanism to wait for a predetermined event in another task. On the other hand, the select statement provides a task with a mechanism to wait for a set of events whose order cannot be predicted in advance. Although these mechanisms have a rich range of features they nevertheless ae 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. Resource Scheduling Schemes

A major area of application where the task construct 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 the concept of family-of-entries and the when clause. These two concepts allow one to 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 the task's own variables (i.e., those variables declared locally to the scheduler task), or, formal parameters passed to an accept statement. Such a scheme cannot be coded in Ada in a straightforward manner. One way to accomplish this is to break up a single "logical" operation into a number of operations, that need to be invoked successively. Yet another method is to distribute the scheduling function over a number of different tasks. Both of these methods introduce unnecessary overhead and make programs harder to read and verify.

To illustrate this point, consider a simple resource scheduling scheme. Suppose that one wishes to define a task whose function is to allocate a resource among N competing tasks in the Shortest Job Next order [3]. To accomplish this, we can define the following:

```
type Task-id is new integer range 1..N;

task SJN is;
   entry Record (Who,time: integer);
   entry Acquire (Task-id);
   entry Release;
end
```

It is assumed that each customer task supplies a unique integer number when it calls the entries Record and Acquire. Acquire is defined as a family of entries which is used to simulate a static priority ordered queue.

A customer task must request a resource from the scheduler SJN by executing the following sequence:

```
SJN.Record(Id,Time);
SJN.Acquire(Id);
```

The scheduler task may now be defined. For simplicity we assume that the largest time that can be specified by a customer process is L-1.

```
task body SJN is;
   Free: boolean:= true;
   Min: integer
   Next: Task-id;
   Count: integer:= 0;
   Queue: array(1..N) of Task-id:= (1..N ⇒ L);

   begin
      loop
         select
            accept Record(Who,Time: integer) do;
               Queue[who]:= Time;
               Count:= Count + 1;
            end
         or
            when Free ⇒
               accept Acquire(Next);
               Free:= false;
         or when not Free ⇒
               accept Release;
               Free:= true;
                  if Count > 0 then
                     Min:= L;
                     for I in 1..N loop
                        if Queue[I] < Min then
                           Min:= Queue[I];
                           Next:= I;
                        end if;
                     end loop;
                  end if;
         end select;
      end loop;
   end.
```

Note that the above solution artificially divides the function of acquiring a resource into two parts. This results in unnecessary overhead and, more importantly, makes the program harder to read and verify.

There are many other scheduling schemes which fall in the same category as the shortest job next scheme; namely, requiring processing before queueing. One such additional example is the interprocess communication (IPC) scheme 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 appropriate. 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 allocator must search the Buffers array to see if the requestor's rendezvouz 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 task 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 allocator must do a significant amount of preliminary processing before it even knows whether the requesting task will have to wait.

As in the Shortest Job Next case, the above schemes could not be handled in a straightforward and concise manner in Ada. In the next section we propose a mechanism for effectively dealing with this problem.

3. The Await Statement

In order to provide Ada with an effective priority scheduling scheme, we propose a mechanism which is a variant of Kessel's conditional wait construct [5]. Central to this mechanism is the command:

await (<boolean-expression>)

The await statement can appear only in an entry operation associated with a select statement.

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

- the local variables of the accept statement, where the await is defined.
- information concerning the calling task.
- 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 with 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. For the sake of brevity, we will be concerned here only with select statements whose select alternative is neither a delay nor a terminate statement.

1. All the boolean-expressions appearing in the select statement are evaluated. These include the boolean expressions in the various when statements as well as those appearing in the await statements. Each accept statement and activation record whose associated boolean expression is evaluated to be true is tagged as open. An accept statement that is not preceded by a when clause is always tagged as open.

2. If several statements or records are tagged as open, an arbitrary single one may be selected for execution:

   - If it is an accept statement, then it can be executed only if another task has invoked an entry corresponding to that accept statement.

   - If it is a record, then it is removed from the associated await-list and the state of the task is restored using the information from the removed record. Note that if a state restoration occurs, the value of the program counter is also being affected.

   If no record or statement can be executed, which implies that only accept statements were tagged as open, then if there is an else part, the else part is executed. If there is no else part, then the task waits until an open statement can be selected.

3. If no accept statement is open, and there is an else part, the else part is executed.

Otherwise an exception condition is raised.

It should be clear that if a boolean expression of an await statement includes formal parameters, 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 contans only the task's own variables, then only one evaluation is needed. Thus, from an efficiency standpoint, one should avoid the inclusion of formal parameters 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 [7]:

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, 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 the task's own variables.

Let us illustrate these concepts by coding the shortest job next scheduling algorithm using our newly proposed scheme:

```
task SJN is
   entry Acquire(Time: integer)
   entry Release;
end
```

A custom task may request a resource by simply executing

SJN.Acquire(Time)

The scheduler task is defined as follows:

```
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;
      end loop;
   end.
```

It would be interesting to compare our solution with the one we presented in Section 2, which uses original Ada. Our solution is much more concise, easier to understand and more efficient. This is because Ada does not provide an effective synchronization scheme.

Let us consider now the more complicated example of the IPC scheme sketched in Section 2. 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 structures needed are:

```
Resource:  array [1..10] of record
                               Rendezvous_code: integer;
                               Hold_count: integer;
                            end
Count, Last_index, Last_channel: integer;
```

Central to the IPC allocator is the select statement described below whose function is to accept two types of calls with 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(I).Hold_count:= Resource(I).Hold_count + 1;
            Buffer_id:= Index;
         end if;
   end;
or
   accept Close(Buffer_id: integer) do
      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;
end select;
```

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

As a final example, consider a controller for the allocation of a group of items from a set of resources. Such a controller can be implemented using our proposed scheme as follows:

```
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) do
               Used:= Used and not Group;
            end;
         end select;
      end loop;
   end Multi_resource_control;
```

4. Conclusion

The accept and select statements are the two main features of the Ada language that provide the programmer with the needed tools for handling communication and synchronization. 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 vaious synchronization problems in a straightforward manner. We have thus proposed the concept of the await statement to resolve this difficulty.

We have demonstrated the usefulness of our proposed concepts by presenting solutions to a number of familiar programming exercises. We have also tried our proposed scheme on other 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).

Finally, we wish to point out that the await statement was specifically designed so that it can be implemented efficiently. If the boolean expressions range only over the task's own 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 own variables only. Additional saving can be obtained by resorting to various optimization techniques (e.g.

Schmid [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.

References

1. "Reference Manual for the Ada Programming Language," United States Department of Defense, July 1980.
2. Dijkstra, E.W., Guarded Commands, Non-Determinacy and Formal Derivation of Programs, CACM 18 8, (August 1975) 453-457.
3. Wegner, P. and Smolka, S.A. Processes, Tasks and Monitors: A Comparative Study of Concurrent Programming Primitives, Proc. of Principles of Programming Languages, (January 1983).
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, CACM 20 7, (July 1977) 500-503.
6. Hoare, C.A.R. Monitors: an operating system structuring concept, CACM 17 10, (October 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.