ISSN 0105-8517

A COMPARISON OF MONITORS AND MESSAGE PASSING

by

Jørgen Staunstrup

DAIMI PB-92
December 1978

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55

A COMPARISON OF MONITORS AND MESSAGE PASSING

Jørgen Staunstrup
Department of Computer Science
University of Aarhus
Ny Munkegade
DK-8000 Aarhus C, Denmark

Abstract

One of the issues which receives much attention in the field of concurrent programming is the communication and synchronization primitives. Many primitives have been proposed but two main philosophies have evolved: monitors and message passing. The two programming languages Concurrent Pascal and Platon represent these two different approaches. By looking at a few algorithms written in the two languages, the two approaches are analyzed and compared.

Keywords
Concurrent programming, message passing, monitors, synchronization primitives.

December 1978

A COMPARISON OF MONITORS AND MESSAGE PASSING

Abstract

One of the issues which receives much attention in the field of concurrent programming is the communication and synchronization primitives. Many primitives have been proposed but two main philosophies have evolved: monitors and message passing. The two programming languages Concurrent Pascal and Platon represent these two different approaches. By looking at a few algorithms written in the two languages, the two approaches are analyzed and compared.

1. INTRODUCTION

A concurrent program describes several computations which take place simultaneously. Each independent computation is called a process. One of the issues that has received much attention is the communication and synchronization of processes. Although the processes are independent they are forced to interact every now and then. If, for example, two processes cooperate on a common task, they need to exchange partial results every now and then. An almost endless list of proposals on how to make processes interact has been made, but two main philosophies have evolved: monitors and message passing. Below these two approaches are analyzed and compared.

In this paper, I will be concerned with concurrent programs written in high level languages only. A particular philosophy for process interaction is realized in a high level language as a set of language primitives. These primitives are then used for expressing all process interaction. A programming language should consist of a small set of primitives which all fit together like building blocks. It is, however, difficult to make such a building block out of a proposed language construct. The construct must be adapted to fit with other constructs of the language. I will therefore not compare monitors and message passing on their own detached from any programming language. Instead, I will compare the two concepts as they are realized in the two languages Concurrent Pascal [Brinch Hansen 1975] and Platon [Staunstrup and Sørensen 1976]. Both of these languages are derived from Pascal [Jensen and Wirth 1974], and both are intended for concurrent programming. Furthermore both languages have now been in active use for several years.

A complete comparison of Concurrent Pascal and Platon would, however, be meaningless. Concurrent Pascal has a very general scope of application. Whereas Platon was designed for a very special application (see section 3). Therefore, only the synchronization and communication aspects of the two languages are discussed. This paper contains several examples of programs written in the two languages. In these examples small deviations from the syntax of the two languages have been made to make all unessential parts of the examples look alike. The reader is assumed to be familiar with Pascal.

2. CONCURRENT PASCAL

Concurrent Pascal is designed for constructing a wide range of concurrent programs. So far, two operating systems and a real time process controller have been published [Brinch Hansen 1977]. The main difference from Pascal is the introduction of the system types: process, class, and monitor. A process is a logically independent component which performs its computations concurrently with other processes. A monitor is a class where the execution of all routines exclude each other. Monitors are the only means for sharing data structures between concurrent processes. The mutual exclusion of monitor routines does only provide short term scheduling of the access to shared data. Medium term scheduling is provided by variables of type (process) queue. Two operations delay and continue are defined on queues. Concurrent Pascal is illustrated by the following example.

Example 1: Resource Allocation

A number of concurrent processes share a common resource, e.g. an external device. Each process needs exclusive access to the resource. The access to the resource is controlled by a resource scheduler. The resource scheduler is a monitor with the two routines request and release. Before a process uses the resource it must be reserved by calling the monitor routine request. When the process has finished using the resource, it calls release. A user process therefore looks as follows:

```
user =
process(r: resource);
    ...
begin
    ...
    r.request;
    use_resource;
    r.release;
    ...
end;
```

An abstract data type _schedqueue_ is used to implement the queue of waiting processes. It is important to realize that it is the programmer who chooses the scheduling algorithm. This is always necessary when the built-in short term scheduling algorithm is inappropriate.

The resource monitor looks as follows:

```
resource =
monitor
      var
          free: boolean;
          q: schedqueue;
      procedure entry request;
      begin
          if free
             then free:= false
             else delay(q.enter);
      end;
      procedure entry release;
      begin
          if q.empty
             then free:= true
             else continue(q.remove);
      end;
begin
      free:= true;
      init q;
end;
```

The details of the type _schedqueue_ are irrelevant for our purposes. A complete program for the resource scheduler can be found in Brinch Hansen 1977.

3. PLATON

Platon is designed for programming dedicated systems like terminal concentrators, drivers for external devices, and controllers. In all these applications it is necessary to transmit large amounts of data between the processes. Platon is derived from Pascal by introducing processes, shared variables, and queue semaphores. A process is a logically independent component which performs its computations concurrently with other processes. Variables can be shared by several processes, but only one process at a time can have access to them. Processes exchange access to shared variables by means of queue semaphores [Lauesen 1975]. A queue semaphore combines the synchronizing effect of the binary semaphore [Dijkstra 1968] with the exchange of data. A queue semaphore is a mailbox through which all communication between processes must take place. Following this analogy, shared variables are envelopes in which messages are transmitted. A message is put in an envelope by assigning a value to the shared variable.

A pool of n shared variables of type t is declared as follows:

```
var
   s: shared set n of t;
```

Shared variables cannot be manipulated directly but only through variables of type reference.

```
var
   b: reference;
```

The operation

```
alloc b from s;
```

binds the reference variable b to a shared variable of type t. This shared variable can now be manipulated like any other variable of type t. The two predefined operations wait and signal are used to exchange access to shared variables. At most one process can have access to a shared variable at any time. The following example demonstrates how these constructs are used in a Platon program.

Example 2: Process Communication

Two processes, a producer and a consumer, communicate by sending blocks of data from one process to the other.

```
producer =                          consumer =
process(line: semaphore);           process(line: semaphore);
    var                                 var
        b: reference;                       b: reference;
        s: shared set 1 of block;
begin                               begin
    ...                                 ...
    alloc b from s;                     wait b from line;
    b:= ...;                            use_b;
    signal b to line;                   ...
    ...
end;                                end;
```

The solution is slightly asymmetric because the producer allocates the message buffers.

4. THE EXAMPLES REVERSED

The two examples are repeated below. But this time the resource allocator is written in Platon and the process communication in Concurrent Pascal. The examples reveal some fundamental differences between the two languages. It is, however, not fair to base one's judgement on these two examples only, since each language is tailored to one of the examples.

Example 1: Resource Allocation

First note, that the following well known solution is insufficient:

```
user =
process(res: semaphore);
     var
        r: reference;
begin
     ...
     wait r from res;
     use_resource;
     signal r to res;
     ...
end;
```

This is not a correct solution because the scheduling is not controlled by the programmer but by the short term scheduling algorithms used to implement wait and signal. If this solution is used a fixed scheduling discipline is enforced on all resources. That is clearly not acceptable.

In Platon the resource scheduler has to be a process which can handle two kinds of messages: requests and releases. In each cycle of the scheduler it accepts a message and takes the necessary actions prompted by that message. Again, a variable of type schedqueue is used to handle the pending requests.

An inherent problem in any message passing system is the administration of message buffers, old message buffers must somehow be regained. In Platon the programmer must explicitly return a message buffer when the message has been processed. Any shared variable has a so-called owner semaphore. The predefined operation return is semantically equivalent to a signal to this owner semaphore. In this example, the owner semaphore of a message is used to return the envelope to its sender. Obviously, a different queue semaphore must be used for returning envelopes. By using the owner semaphore the resource scheduler need not explicitly know all its users.

```
resource =
process(mes: semaphore);
      var
          free: boolean;
          q: schedqueue;
          r: reference;
begin
      init q;
      free:= true;
      cycle
          wait r from mes;
          case r of
          request: if free
                       then begin
                              free:= false;
                              return r; "allows the requesting process
                                          to proceed"
                             end
                       else q.enter(r);
          release: begin
                       return r; "allows the releasing process to proceed"
                       if not q.empty
                              then begin
                                     r:= q.remove;
                                     return r; "allows a pending request
                                                  to be completed"
                                    end
                              else free:= true;
                   end;
          end "case";
      end "cycle";
end;
```

```
user =
process(mes: semaphore);
      var
          r: reference;
          s: shared set 1 of (request, release);
          answer: semaphore;
begin
      . . .
      alloc r from s with answer;
      . . .
      r:= request;
      signal r to mes;
      wait r from answer;

      use_resource;
      r:= release;
      signal r to mes;
      wait r from answer;
      . . .
end;
```

The owner semaphore (return address of message) is defined by the alloc operation.

```
      alloc r from s with answer;
```

The shared variable now has answer as the owner semaphore.

Although the solution is rather complex the communication follows a very regular pattern. A user process "calls" the scheduler by sending a message and immediately wait for an answer. This can be interpreted as a procedure call:

```
      r:= request; "set up parameters"
      signal r to mes; "procedure entry, parameters passed in r"
      wait r from answer; "procedure return, results passed in r"
```

Finally, the return jump from the procedure is the call of return.

There are several problems with the Platon version. The first is the branching (the case statement) caused by the different kinds of messages received on the same queue semaphore. This gives an unnecessarily complicated control structure, but it is also the source of more serious problems which are discussed below. The second problem with the Platon version is its complexity. This is caused by an inherent problem in a message passing system. The scope of a message is not indicated by the block structure of the program, as it is the case with other variables. The points in the program where the message buffer appears and disappears are therefore identified explicitly (by alloc, send, wait, and return).

The following inefficiency in the Platon version was pointed out to me by Nigel Derrett:in a uniprocessor system there is some overhead in switching from one process to another. In the Platon version there are four such switches for each process requesting and releasing the resource.

Example 2: Process Communication

The process communication example written in Concurrent Pascal looks as follows:

```
producer =                          consumer =
process(line: buffer);              process(line: buffer);
    var                                 var
        b: block;                           b: block;
begin                               begin
    ...                                 ...
    b:= ...;                            line.get(b);
    line.put(b);                        use_b;
    ...                                 ...
end;                                end;
```

A monitor is needed to connect the two processes.

```
buffer =
monitor
      var
          buf: block;
          empty: boolean;
          sender, receiver: queue;
      procedure entry put(b: block);
      begin
          if not empty then delay(sender);
          buf:= b;
          empty:= false;
          continue(receiver);
      end;
      procedure entry get(var b: block);
      begin
          if empty then delay(receiver);
          b:= buf;
          empty:= true;
          continue(sender);
      end;
begin
      empty:= true;
end;
```

It is not very surprising that the Platon program is more elegant, since Platon was designed for message passing, whereas Concurrent Pascal has more general goals. It is, however, noteworthy that the greater generality was obtained at an extra cost of:

- extra space: three variables of type block in the Concurrent Pascal version compared to one in the Platon version,
- slower execution: the contents of the buffer is copied twice in the Concurrent Pascal program. No copying takes place in the Platon program. One could argue that the copying could be avoided by introducing references in Concurrent Pascal. But this would be contrary to the philosophy behind Concurrent Pascal [Brinch Hansen 1977].

5. EXTERNAL SELECTION

Consider the situation where a number of processes share an instance of an abstract data type, and let this instance be an output device. The abstract data type has two operations writetext and resupply. Every time a text is written one piece of paper is used. There is only a limited supply of paper, so when the paper is finished, the device cannot be used until it is resupplied. In Platon this is programmed in a way very similar to the resource scheduler:

```
device =
process(mes: semaphore);
   :
   cycle
      wait r from mes;
      case r.type of
      writetext: ...

      resupply: ...

      end; "case"
   end;
end
```

There are, however, several problems with this solution.

It is desirable to design a concurrent programming language in such a way that the compiler can check the process interaction. In Platon this can be achieved by associating a type with a queue semaphore and then only permitting messages of that type to be transmitted via the semaphore. On the other hand a process can only wait on one semaphore at a time. These two things together make it hard to program an algorithm where a process must wait for the first of two messages of different types and take different actions in the two cases. This concept is called external selection because the flow of control is determined by the order in which external events take place. In the above example, the problem was evaded by using a union type.

All messages transmitted via the queue semaphore mes are of this union type. Furthermore the message carries information(r. type) about its type. This of course defeats the goal of having the compiler check the process interaction. Note, that the problem is very elegantly solved by a monitor waiting for the first of many possible routine calls. In Platon what is needed is a construct like:

```
casewait r from
    sem1: s1;
    sem2: s2;
    :
    semn: sn;
end;
```

A construct like this was considered when Platon was designed, but it was omitted because:

-The right syntax was never found. In the above proposal semaphores appear as case labels. This is a new concept that we hesitated to introduce. All other proposals had similar peculiarities.

- It was difficult to find an efficient implementation.

- The importance of the external selection was not realized. In retrospect it is obvious that it was a wrong decision to omit such a construct.

The above example reveals another serious weakness in Platon. The program as it stands does not work. Consider what happens when the printer runs out of paper. From then on the process can accept messages of type resupply only. But all messages arrive on the same queue semaphore, and it is not possible to lock out the writetext-messages. To solve this in Platon, a local queue semaphore must be introduced. All messages that cannot be handled immediately are then signalled to this local semaphore. When more paper is supplied, all the pending messages are handled first.

In the RC 4000 system [Brinch Hansen 1970] which is also based on message passing, this problem is solved by introducing two new standard routines called waitevent and getevent. This routine allows a process to select other messages than the first from a given queue. In this example, waitevent could be used to filter out all messages of type writetext.

In Concurrent Pascal the same problem is solved by distinguishing between short and medium term scheduling. Short term scheduling is concerned with implementing the indivisible access to the shared variables. Medium term scheduling is concerned with the delays caused by the logic of the algorithm used. In this case medium term scheduling takes care of delaying writetext-operations when there is no paper.

In Concurrent Pascal one would therefore write (ignoring the problem that only one process can be delayed in each queue-variable):

```
device =
monitor
    :
   procedure entry writetext(...);
   begin
     if paper = 0
        then delay(morepaper);
     ...
     paper:= paper-1;
   end;
end;
```

Concurrent Pascal does, however, have a similar problem on the medium term level. Processes are always delayed in a single queue even if they wait for the first of several events. Let B1 and B2 be two boolean expressions, and assume that a process has to be delayed until either of the two conditions is satisfied. Different actions S1 and S2 are to be taken depending on which condition becomes true. This is "solved" by collapsing the two conditions into one:

```
     if not(B1 or B2)
        then delay(q);
     "B1 or B2"
     if B1
        then S1
        else S2;
```

So Concurrent Pascal does not have means for expressing external selection on the medium term level. A construct called the guarded region has been proposed to capture this [Brinch Hansen and Staunstrup 1978]:

```
when
    B1: S1;
    B2: S2;
end;
```

The process is delayed until the first of the two guards (B1, B2) becomes true. It then executes the corresponding statement. If both guards become true only one of the statements are executed, which one is unknown.

6. CONCLUSION

This paper has analyzed and compared two concurrent programming concepts: monitors and message passing. The concepts are compared on the basis of their realization in two programming languages which are in active use: Concurrent Pascal and Platon. No claim is made that the paper contains an exhaustive list of assets or weaknesses of the two languages. Only examples where some insight can be gained from the differences have been included. Problems which are common to the two approaches have been omitted.

The examples presented in this paper point out the following fundamental problems with the two languages.

Platon:

1) no direct means for expressing external selection,
2) no distinction between short and medium term scheduling,
3) complex program structure,
4) unnecessary mixing of data and control flow because messages carry type information in addition to the transmitted data,
5) inefficiency caused by the frequent switching of the processor from one process to another.

Concurrent Pascal:

a) no direct means for expressing external selection on the medium term level,
b) somewhat inefficient for transmitting large amounts of data.

I believe that 1), 2), and 4) are problems that could be removed in a future language based on message passing. Whereas 3) is an inherent problem caused by the fact that the scope of shared variables is dynamically controlled, and the scope of other variables is statically determined. 5) is of course mainly a problem in a uniprocessor system.

Problem a) with Concurrent Pascal is solved by introducing guarded regions which are hard to implement efficiently however. Problem b) only exists in a uniprocessor system without explicit pointers. In a multiprocessor implementation the data has to be copied (transmitted) anyway.

Acknowledgements

The ideas presented in this paper emerged through many interesting discussions with Sven Meiborg Sørensen and others at Aarhus University, Ole Caprani and Anders P. Ravn from Copenhagen University, and Per Brinch Hansen at University of Southern California.

References

Brinch Hansen, P., The nucleus of a multiprogramming system. Comm. ACM 13, 4 (April 1970), 238-250.

Brinch Hansen, P., The programming language Concurrent Pascal. IEEE Transactions on Software Engineering 1, 2 (June 1975), 199-207.

Brinch Hansen, P., The architecture of concurrent programs. Prentice-Hall, Inc., Englewood Cliffs, N.J., July 1977.

Brinch Hansen, P., and Staunstrup, J., Specification and implementation of mutual exclusion. IEEE Transactions on Software Engineering 4, (September 1978).

Dijkstra, E.W., Cooperating sequential processes. In Programming Languages, F. Genuys (ed.), Academic Press, New York, N.Y. 1968.

Jensen, K., and Wirth, N., PASCAL user manual and report. Lecture Notes in Computer Science, No. 18, Springer Verlag 1974.

Lauesen, S., A large semaphore based operating system. Comm. ACM 18, 7 (July 1975), 377-389.

Staunstrup, J., and Sørensen, S.M., Platon. A high level language for systems programming. In Minicomputer Software, J.R. Bell (ed.), North Holland 1976.