ISSN 0105-8517

COMMON CLASS
- a tool for programming the access to shared data

Peter Møller-Nielsen
Jørgen Staunstrup

DAIMI PB-170
December 1983

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

COMMON CLASS

- a tool for programming the access to shared data

Peter Møller-Nielsen and Jørgen Staunstrup

Abstract

The Monitor concept in Concurrent Pascal is too restrictive to express e.g. the "Readers-Writers" kind of access control to shared data. In order to make it possible to measure how the performance of a multiprogram changes when the access control to shared data is refined, we have added a construct, called a Common Class, to the Concurrent Pascal implementation on our multiprocessor system.

1. MOTIVATION

Our experience with a number af actual multiprograms written in Concurrent Pascal shows that the Monitor concept is often much too restrictive. In connection with access to shared data saturation occurs when more than 1+Tl/Tm processors are used, where Tm is the time spent inside - and Tl the time spent outside a critical region, i.e. the region for which mutual exclusion is guaranteed [1]. In order to increase the point of saturation to a satisfactory number of processors, it seems necessary to add a construct to Concurrent Pascal, which allows simultaneous access of processes to shared data to be expressed in a more refined manner. An example is the "Readers-Writers" kind of access control to a shared table. "Readers-Writers" cannot be programmed in Concurrent Pascal. The shared data must be encapsulated in a Monitor, and this implies that all accesses to the shared data are treated as "Writers".

Section 2 of this report offers a construct called a Shared Class, which seems to be a natural modification of the Monitor construct. Like the Monitor, the syntax of a Shared Class ensures that the data that are represented by the Shared Class and the access control mechanisms to the data (i.e. semaphores and associated operations) are always correctly associated. Section 3 offers another construct called a Common Class. This construct is much simpler to add to an existing implementation of Concurrent Pascal. It is shown how a program, which is using Shared Classes, can be transcribed into a program which uses Common Classes and Monitors. Section 4 shows a sequence of programs. They all solve the same problem, but the access control to a shared table shows different degrees of refinement. Section 5 shows how a Shared Class can be transcribed into a Common Class and a Monitor without using Delay- and Continue-statements.

2. THE CONCEPT OF A SHARED CLASS

It would be possible to express "Readers-Writers" (and other strategies for controlling access to shared data) if the Monitor concept in Concurrent Pascal was substituted by the Shared Class. A Shared Class has the following form:

```
sh = shared class ( < parameters > );

     var state  Vs: ... ;   "access to an entry is granted
                             based on the variables Vs."
          data  Vd: ... ;   "the shared data."

procedure entry Pi( < parameters > );

  entry  if Bi(Vs) then ENTRY_STATEMENTi(Vs);

         "Bi(Vs) is a boolean expression of the variables Vs,
          and ENTRY_STATEMENTi is a statement, which manipu-
          lates the variables Vs. The index i indicates that
          this boolean expression and statement is particular
          for the entry procedure Pi."

    begin "The body of the entry procedure Pi."

       Si(Vd) "The manipulation of the shared data."

    end

  exit  EXIT_STATEMENTi(Vs);

begin

    "Initialization of Vs and Vd."

end;
```

The meaning of the constructs in a Shared Class is the following. When a process calls an entry in a Shared Class, access to the body of the entry is not granted based on a mutual exclusion principle, as it is for the Monitor, but access is granted as soon as the boolean expression Bi(Vs) evaluates to true. Then the statement ENTRY_STATEMENTi is executed, and after that the execution of the statement Si(Vd) is initiated. The evaluation of Bi and

the execution of the statement ENTRY_STATEMENTi is done indivisibly, and excluding execution of other entry- and exit-statements of the same Shared Class. When the execution of the statement Si is completed, the statement EXIT_STATEMENTi is executed in an indivisible manner, and excluding execution of other entry- and exit-statements of the same Shared Class. Entry- and exit-statements may only refer to state-variables (i.e. Vs) and the bodies may only refer to the data-variables (i.e. Vd).

Using the above concept of a Shared Class, "Readers-Writers" can be programmed as follows:

```
r_w = shared class;

      var state n_readers,      "The number of processes
                                 executing Sr in the Read
                                 entry."
                n_writers       "The number of processes
                                 executing Sw in the Write
                                 entry."
                  : integer;

          data  t : table;      "The shared table on which
                                 the Readers and the Writers
                                 are operating."

   procedure entry  Read(...);

      entry  if n_writers = 0
                then
                   n_readers := n_readers + 1;

         begin

            Sr(t)

         end

      exit   n_readers := n_readers - 1;

   procedure entry  Write(...);

      entry  if (n_readers + n_writers) = 0
                then
                   n_writers := n_writers + 1;
```

```
        begin
           Sw(t)
        end
     exit  n_writers := n_writers - 1;

  begin
     n_readers := 0;      n_writers := 0;
     "Initialization of 't'."
  end;
```

A Monitor can be programmed in a similar way using a single boolean (a "gate") as Vs. The other extreme, which allows unrestricted access to a set of shared data, can be obtained by leaving the statements ENTRY-STATEMENT and EXIT-STATEMENT empty and by using the constant true for the expression B.

3. THE CONCEPT OF A COMMON CLASS

In order to make it possible to measure how the performance of a multiprogram changes when the access control to shared data is refined, we decided to add a suitable construct to the current Concurrent Pascal implementation on our multiprocessor system [2]. The actual implementation of such a construct was based on the following considerations. The added construct should be seen as a tool by means of which we could experimentally study the effect of different refinements of access control to shared data. It should not be seen as a proposal for the syntax and semantics of a new construct to be included in the language Concurrent Pascal. It is our firm belief that such proposals should be postponed until the subject has matured through the study of the effect of different forms of access control refinement on programming complexity and performance. Such a study is best conducted by programming and executing a wide variety of algorithms with different refinement of access control. For this reason, we wanted to add a construct which implied as little change as possible to the Concurrent Pascal compiler and the C-code interpreter on our multi-processor system [2]. To implement the shared class proposed above a rather extensive change is necessary. Instead we chose to keep the Monitor construct unchanged and add a construct called a Common Class. A Common Class is a Shared Class with unrestricted access. From an implementation point of view, a Common Class is a Monitor to which the gate is always left open. For this reason a Common Class is translated as a Monitor except at one point; the C-codes ENTER CLASS, EXIT CLASS, BEGIN CLASS, END CLASS and INIT CLASS are generated instead of the C-codes ENTER MON, EXIT MON, BEGIN MON, END MON and INIT MON (the meaning of the C-codes is explained in [3]). This means that no new C-code has to be added to the C-code machine, and the C-code interpreter (and kernel) can be left unchanged. The changes to the compiler are few and straightforward.

Programs written by means of the Shared Class construct can be systematically transcribed into a program which uses only Monitors and Common Classes. A Shared Class is represented by a pair consisting of a Common Class and a Monitor, to which the Common Class has access. The Common Class contains the shared data (i.e. Vd) and the operations (i.e. Si(Vd)). The Monitor encapsulates a transcription of the entry and the exit statements of the Shared Class. The Shared Class named "sh" is transcribed into the following pair.

```
sh_mon = monitor;

        var  Vs: ... ;

             w:      waiting_room;      "a pool of queue
                                         - variables."
             w_adm:  waiting_room_adm;  "administration
                                         of the pool."

        procedure entry  entry_to_Pi;

          begin

            while not(Bi(Vs)) do
              delay(w( . w_adm.vacant .));

            ENTRY_STATEMENTi(Vs);

            while not(w_adm.empty) do
              continue(w(. w_adm.occupied .))

          end;

        procedure entry  exit_from_Pi;

          begin

            EXIT_STATEMENTi(Vs);

            while not(w_adm.empty) do
              continue(w( . w_adm.occupied .))

          end;

        begin

          init(Vs)

        end;
```


The pair "w" and "w_adm" implements a pool of queue-variables. The operations on the pool are:

- empty : = true, if the pool is empty, i.e. if all queue-variables in the pool are empty.
- vacant : the identity (e.g. an index in an array of queue-variables) of a queue-variable in the pool which is empty.
- occupied : the identity in the pool of a queue-variable which is not empty.

Note that our implementation of the continue-statement deviates from normal Concurrent Pascal [3]. An execution of a continue-statement does not enforce an exit from the entry. Note also that "sh_mon" contains two entry procedures ("entry_to_Pi" and "exit_from_Pi") for each entry procedure (Pi) in the Shared Class. The Common Class associated with "sh_mon" looks as follows:

```
sh = common class(...... , s: sh_mon);

        var  Vd: ... ;

        procedure entry  Pi( ..... );

          begin

            s.entry_to_Pi;

            Si(Vd);

            s.exit_from_Pi

          end;

        begin

          init(Vd)

        end;
```

The detailed transcription of the Shared Class "r_w" above is shown in the next section.

It should be noted that local variables in the entry procedure of a Common Class are allocated anew (and locally to the calling process) for each new invocation. Only the global variables (i.e. Vd) are allocated globally to the processes, and in just one copy.

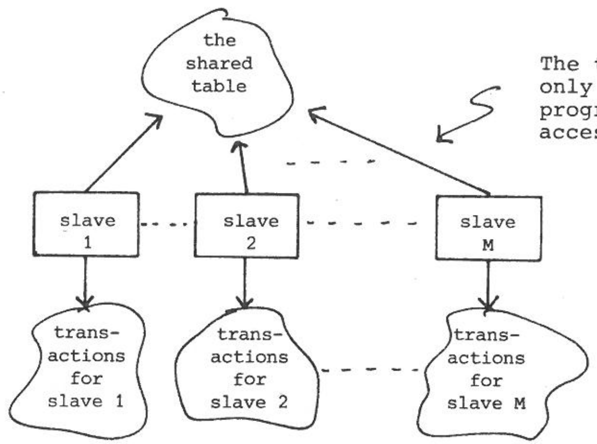
4. AN EXAMPLE

This section contains the most essential parts of three programs. The programs are identical with the exception that the control of access to a shared table is programmed in three different stages of refinement. The first program uses a Monitor to encapsulate the table. The second program uses a single Shared Class, which is transcribed into a pair consisting of a Monitor and a Common Class as described above. The third program uses a further refinement of the access control. This refinement cannot be programmed by means of a single Shared Class.

All programs solve the following problem. The entries of a table is updated according to a set of transactions. The table has a fixed number of entries. Each entry has a key which identifies the entry and a field which contains some information, which is associated with the key. The set of transactions contains only two different kinds of transactions, a WRITE and a READ. A WRITE looks up a key in the table and updates the information field of the entry. Once the entry has been found, the updating must be done indivisibly, i.e. it cannot be mixed with other READs or WRITEs. A READ looks up a key in the table and reads the information field. The reading of the information can be done simultaneously with other READs (but not other WRITEs) of the same information field. Operations on different entries can be done simultaneously. The sequence in which the transactions are carried out is immaterial. The transactions are divided evenly and randomly between a number of identical processes called "slaves". The problem is solved when all slaves have executed all the transactions that was allocated to them. The program for the slave process is as follows:

```
type

   kind_of_transaction = (reading, writing);
   .
   .
   .
   slave = process(t: table; ...);

      var    no_of_transactions: integer;
             .
             .

             kind:                kind_of_transaction;
             key:                 ...;
             info:                ...;

             i:                   integer;
             ...:                 ...;

      .
      .
      .

      procedure new_transaction(var kind: kind_of_transaction;
                                var key:  integer;
                                var info: integer);

         begin
           .
           .
           .
         end;

      begin
        .
        .
        .
          for i := 1 to no_of_transactions do
             begin
                new_transaction(kind, key, info);
                if kind = reading
                   then
                      t.read(key, info)
                   else
                      t.write(key, info)
             end
      end
   end;
```


The structure of the three programs can be depicted as follows:



The three programs differ only with respect to the programming of these accesses.

The shared table and the control of accesses to it looks as follows for the three programs:

<u>Program 1</u>

The table is encapsulated in a single Monitor.

```
   table = monitor;

      var   t: array (. 1 .. ... .) of
                  record
                     key: ...;
                     info: ...
                  end;

      function  search(key_0: ...): integer;
         var i: integer;
         begin  "We assume that 'key_0' is in the table."
            i := 1;
            while t(. i .).key <> key_0 do  i := i + 1;
            search := i
         end;
```

```
procedure entry  read(key: ...; var info_o: ...);
   begin
      info_o := t(. search(key) .).info
   end;

procedure entry  write(key: ...; info_i: ...);
   begin
      t(. search(key) .).info := info_i
   end;

   .
   .
   .

begin
end;
```

Program 2

The table is encapsulated in a single Shared Class, which is transcribed into a Monitor and a Common Class.

```
table_monitor = monitor;

   .
   .

   var   n_readers, n_writers: integer;

         w:      waiting_room;
         w_adm:  waiting_room_adm;

   procedure entry  entry_to_read;
      begin
         while not (n_writers = 0) do
            delay(w(. w_adm.vacant .));
         n_readers := n_readers + 1;
         while not w_adm.empty do
            continue(w(. w_adm.occupied .))
      end;

   procedure entry  exit_from_read;
      begin
         n_readers := n_readers - 1;
         while not w_adm.empty do
            continue(w(. w_adm.occupied .))
      end;
```



```
   procedure entry  entry_to_write;
      begin
         while not ((n_readers + n_writers) = 0) do
            delay(w(. w_adm.vacant .));
         n_writers := n_writers + 1;
         while not w_adm.empty do
            continue(w(. w_adm.occupied .))
      end;

   procedure entry  exit_from_write;
      begin
         n_writers := n_writers - 1;
         while not w_adm.empty do
            continue(w(. w_adm . occupied .))
      end;

   begin
      init w_adm;
      n_readers := 0;   n_writers := 0
   end;

table = common class(t_m: table_monitor);

   var   t: array (. 1 .. ... .) of
              record
                 key: ...;
                 info: ...
              end;

   function  search(key_0: ...): integer;
      var i: integer;
      begin  "We assume that 'key_0' is in the table."
         i := 1;
         while t(. i .).key <> key_0 do  i := i + 1;
         search := i
      end;

   procedure entry  read(key: ...; var info_o: ...);
      begin
         t_m.entry_to_read;
         info_o := t(. search(key) .).info;
         t_m.exit_from_read
      end;

   procedure entry  write(key: ...; info_i: ...);
      begin
         t_m.entry_to_write;
         t(. search(key) .).info := info_i;
         t_m.exit_from_write
      end;

   .
   .

   begin
   end;
```

Program 3

This program resembles program 2, except that the entries of the Common Class is programmed in such a way that the search for the proper entry is allowed to take place simultaneously with other searches and readings and writings of information fields. Note that the "table-supervisor" in program 3 is identical to the "table_monitor" in program 2, except for a few changes of names.

```
table_supervisor = monitor;

  .
  .

  var   n_readers, n_writers: integer;

        w:       waiting_room;
        w_adm:   waiting_room_adm;


  procedure entry  read_request;
     begin
        while not (n_writers = 0) do
           delay(w(. w-adm.vacant .));
        n_readers := n_readers + 1;
        while not w_adm.empty do
           continue(w(. w_adm.occupied .))
     end;


  procedure entry  read_release;
     begin
        n_readers := n_readers - 1;
        while not w_adm.empty do
           continue(w(. w_adm.occupied .))
     end;


  procedure entry  write_request;
     begin
        while not ((n_readers + n_writers) = 0) do
           delay(w(. w_adm.vacant .));
        n_writers := n_writers + 1;
        while not w_adm.empty do
           continue(w(. w_adm.occupied .))
     end;
```

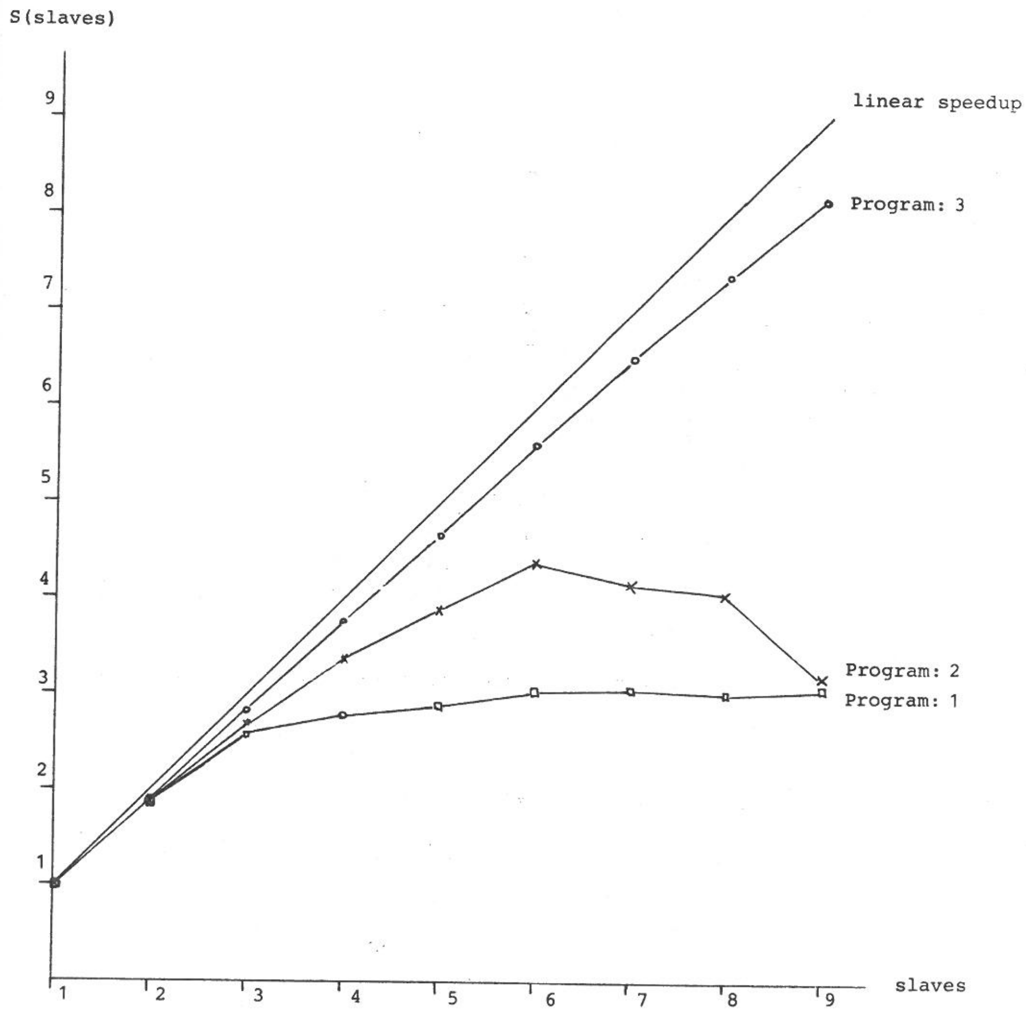


```
  procedure entry  write_release;
     begin
        n_writers := n_writers - 1;
        while not w_adm.empty do
           continue(w(. w_adm.occupied .))
     end;


  begin
     init w_adm;
     n_readers := 0;   n_writers := 0
  end;


table = common class(t_m: table_supervisor);

  var   t: array (. 1 .. ... .) of
              record
                 key: ...;
                 info: ...
              end;

  function  search(key_0: ...): integer;
     var i: integer;
     begin  "We assume that 'key_0' is in the table."
        i := 1;
        while t(. i .).key <> key_0 do  i := i + 1;
        search := i
     end;


  procedure entry  read(key: ...; var info_o: ...);
     var index: integer;
     begin
        index := search(key);
        t_m.read_request;
        info_o := t(. index .).info;
        t_m.read_release
     end;


  procedure entry  write(key: ...; info_i: ...);
     var index: integer;
     begin
        index := search(key);
        t_m.write_request;
        t(. index .).info := info_i;
        t_m.write_release
     end;

     o
     o
  begin
  end;
```

The three programs have been executed on our multiprocessor system in order to measure the changes in performance as access control was refined from program 1 to program 3. The execution times depend on many irrelevant quantities, e.g. the total number of transactions handled by the slaves, the speed of the actual C-code machine etc. The measurements are therefore displayed in terms of the "Speed-up". The Speed-up S(n), which is a function of the number of slaves, is defined as:

$$S(n) = T(1)/T(n)$$

where T(i) is the time it takes for a system with i slaves to solve the problem (i.e. to execute all transactions). S(n) still depends on some parameters e.g. the quotient between the time spent retrieving a transaction and the time spent searching the table for a single key, the fraction of WRITEs among the transactions etc. The graphs below show - for a particular choice of these parameters - the Speed-up for the three programs and for different numbers of slaves. The measured values illustrate how the saturation point of the shared object (i.e. the table) is moved to higher values by refining the access control.



5. ANOTHER TRANSCRIPTION OF THE SHARED CLASS

The transcription of a Shared Class, which was shown in section 3, can often be optimized (with respect to execution time) by using some properties of the access constraints implemented by the Shared Class. For example, EXIT_FROM_READ in program 2 in section 4 can be written as:

```
procedure entry  exit_from_read;
   begin
      n_readers := n_readers - 1;
      if (n_readers = 0) and (not w_adm.empty)
         then
            continue(w(. w_adm.occupied .))
   end;
```

and ENTRY_TO_READ as:

```
procedure entry  entry_to_read;
   begin
      while not(n_writers = 0) do
         delay(w(. w_adm.vacant .));
      n_readers := n_readers + 1
   end;
```

Below is shown a quite different transcription of the Shared Class in program 2. The delay- and continue-statements (and the associated objects 'waiting_room' and 'waiting_room_adm') are substituted by busy-waiting loops.

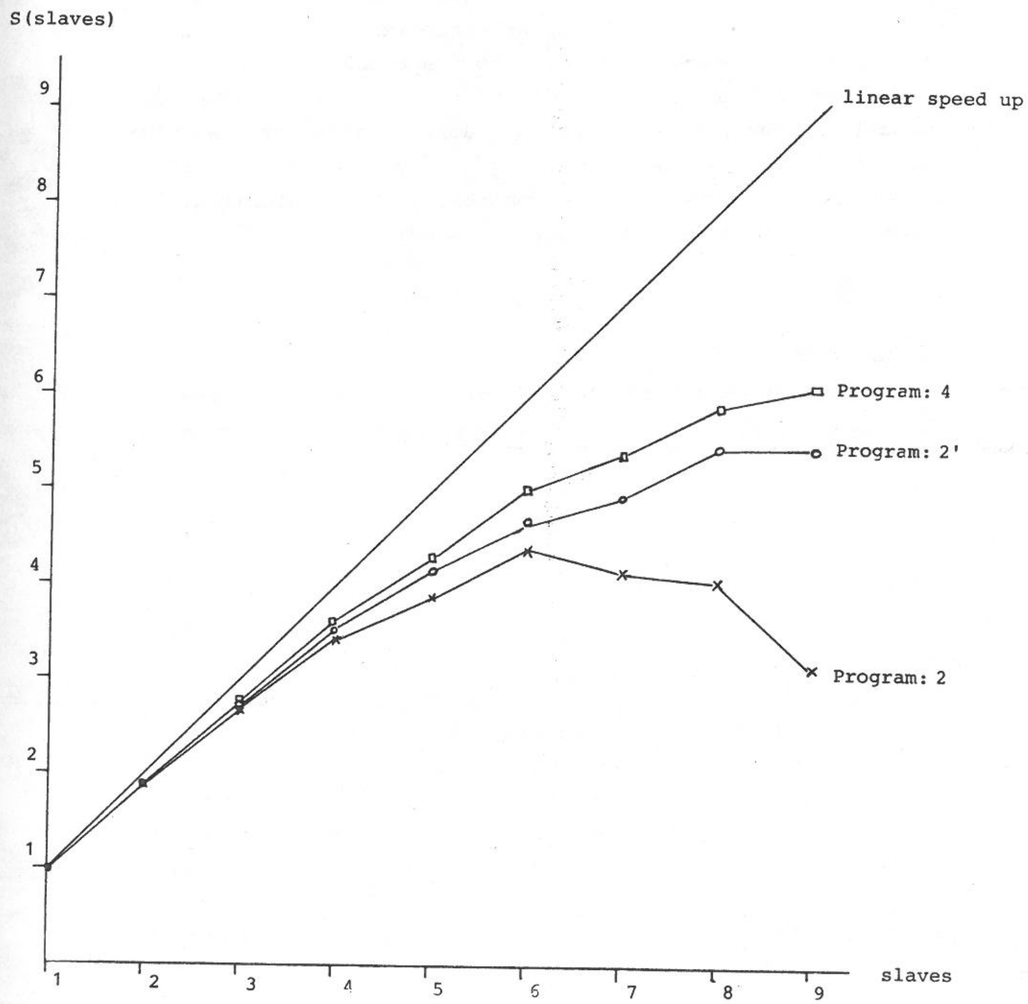
```
table_monitor = monitor;

   .....

   var   n_readers, n_writers: integer;

   function entry  entry_to_read: boolean;
      begin
         if n_writers = 0
            then
               begin
                  n_readers := n_readers + 1;
                  entry_to_read := true
               end
            else
               entry-to-read := false
      end;
```



```
   procedure entry  exit_from_read;
      begin
         n_readers := n_readers - 1;
      end;

   function entry  entry_to_write: boolean;
      begin
         if (n_writers + n_readers) = 0
            then
               begin
                  n-writers := n-writers + 1;
                  entry_to_write := true
               end
            else
               entry_to_write := false
      end;

   procedure entry  exit_from_write;
      begin
         n_writers := n_writers - 1;
      end;

   begin
      n_readers := 0;   n_writers := 0
   end;

table = common class(t_m: table_monitor);

   var   t: array (. 1 .. ... .) of
               record
                  key: ...;
                  info: ...
               end;

   function  search(key_0: ...): integer;
      var i: integer;
      begin  "We assume that 'key_0' is in the table."
         i := 1;
         while t(. i .).key <> key_0 do  i := i + 1;
         search := i
      end;

   procedure entry  read(key: ...; var info_o: ...);
      begin
         while not (t-m.entry-to-read) do;
         info_o := t(. search(key) .).info;
         t_m.exit_from_read
      end;
```

```
procedure entry  write(key: ...; info_i: ...);
   begin
      while not (t_m.entry_to_write) do;
      t(. search(key) .).info := info-i;
      t_m . exit_from_write
   end;

   .
   .
   .

begin
end;
```

The graphs below compare the performance of three transcriptions of a Shared Class. Program 2 was described in section 4. Program 2' uses the same transcription as program 2, but includes the optimization suggested above. Program 4 uses a transcription based on busy-waiting.



The transcription which uses busy-waiting is slightly better than the other two. This may be explained as follows. The transcription which uses busy-waiting is inherently faster than transcriptions using delay- and continue-statements (compare the entry procedures EXIT_FROM_... in the three programs). An adverse effect is caused by the heavy traffic on the entries of the Monitor called 'table_monitor'. This traffic will saturate the 'table_monitor' for a relatively small number of processes. However, if the time spent executing an entry in the 'table_monitor' is small compared to the total time spent in an entry of the Common Class called 'table', the relative loss is small for the processes that are doing useful work. Whether a process loses its power executing a busy-waiting loop or is delayed in a queue-variable is immaterial when a static allocation of processors to processes is used.

Acknowledgement

Particular thanks to Ole Caprani for his contributions at all stages of this work and valuable suggestions for improvements.

References

[1] Møller-Nielsen and Staunstrup: Saturation in a Multiprocessor. IFIP'83, September 1983.

[2] Møller-Nielsen and Staunstrup: Early Experience from a Multiprocessor Project. DAIMI, PB-142, January 1982.

[3] Brinch Hansen: The Architecture of Concurrent Programs. Prentice-Hall, 1977.