JAN 166b -0

A Tutorial Introduction to Mosaic Pascal

Johan J. Lukkien, Jan L.A. van de Snepscheut
Computer Science
California Institute of Technology
Pasadena, CA 91125

23 July 1992

In this report we describe a Pascal system that has been developed for programming Mosaic multicomputers. The system that we discuss runs on our Sun workstations, and we assume some familiarity with the use thereof. We assume the reader to be also familiar with programming in Pascal, and with message-passing programs. We describe how the Pascal language has been extended to perform message passing. We discuss a few implementation aspects that are relevant only to those users who have a need (or desire) to control some machine-specific aspects. The latter requires some detailed knowledge of the Mosaic system.

1 Machine overview

The Mosaic is an experimental fine-grain multicomputer. It is being developed by Caltech's Computer Science Department as part of the Submicron Systems Architecture project.

A Mosaic multicomputer consists of a number of nodes. The nodes are connected in a network, that is either a 2-dimensional mesh or, in the case of a small system, a 1-dimensional line. Each node is a single CMOS chip containing a 16-bit processor, 64 Kbytes of memory, a router to handle communication between nodes, and a packet interface between router and processor.

The Mosaic systems that are presently available are hooked up to Sun 3 and Sun 4 workstations. The following systems can be used.

- `hyperion` has an 8-processor system running at 25 MHz
- `psyche` has a 4-processor system running at 25 MHz
- `mosaic` has a 32-processor system running at 25 MHz
- `pallas` has a $4+64$-processor system running at variable clock rate

2 Pascal system overview

The Pascal system consists of a few components: a compiler that compiles programs from Pascal to Mosaic machine language, a monitor that loads a machine language program into a Mosaic system, and a set of include files that can be included in your programs to simplify program development. The monitor, in conjunction with the include files, implements communication between the Mosaic system and your Sun. It offers input from the keyboard, output to the screen, and reading and writing of files.

A major part of the implementation is actually done in Pascal and, therefore, some low-level hooks to the system are provided. We do not discuss them until much later and start by giving some examples of programs that use our include files. First we discuss an ordinary sequential program. Next come some parallel programs that run on a single Mosaic node. Finally we discuss how programs are distributed over a set of nodes. In order to get access to the various programs that you will use, you have to include

```
~jan/mosaic/bin
```

in your path. This you can do by adding the above to the `path` as given in your `.login` or `.cshrc` file.

3 Getting started

Create at least two windows on your Sun workstation. In one of them you login to a Sun that has a Mosaic system connected to it. In another window you run an editor on your Pascal program text. Both windows operate in the same working directory. Use the editor to create file `ex.p` that contains the following text. Do not forget to enter an end-of-line after the final period.

```
program example;

#include <declarations>
#include <monitor>

begin monitor;
      writeln(output, 'my first string')
end.
```

Next, execute

```
cp ~jan/mosaic/config/1 1
```

If you now switch over to the window on the machine connected to the Mosaic system then you can issue the command

```
nmon 1 ex
```

which will cause the program to be compiled and executed. The compiler creates a file `ex.m` containing the machine code for your program. The output of the program's execution is

```
FILE MONITOR
my first string
```

and you can terminate the monitor by a Control-C which returns control to the shell.

> Note: program `nmon` calls the compiler which runs on a Sun 4. When activated from a Sun 3, it delegates the compilation to `mosaic`, so you have to make sure that the Sun 3 that you are using is included in the list of remote hosts that you have access to (see your file `.rhosts`). The easiest way of ensuring this is by letting your file `.rhosts` consists of a single line only, viz.
>
> ```
> +
> ```
>
> which is interpreted as: every machine is included. If you do not have access to `mosaic` you will have to compile on a different machine. In this case, you probably want to do both the editing and compiling on the same Sun 4. You can compile the example program with the command
>
> ```
> mcom ex
> ```
>
> which will produce file `ex.m` containing the machine code for the example program. Sometimes you will find that you have to wait a second or two before the Sun 3 has access to the file created by the compiler on a Sun 4. I have no idea why; don't ask, just wait. (See also section 6.)

This is how you run an ordinary sequential Pascal program on a single Mosaic node. The only "unusual" parts of the program are the `#include` lines and the initial statement that calls procedure `monitor`; other than that it is ordinary Pascal. It may be relevant to observe that the identifier `output` should not be omitted from `write` statements. Similarly, `input` should not be omitted from `read` statements. Here are a few more extensions and restrictions:

- One can use the underscore (_) in identifiers; it is significant.
- One can use subranges in labels of `case` statements and variant records. The labels should not be negative integers.
- The `case` statement has an optional `otherwise` part.
- Function results are not restricted to scalar or pointer types: arrays, records, or sets can be used as function results also.
- The set operator / produces the symmetric set difference, i.e. exclusive or: $x \in (s0/s1)$ is equivalent to $(x \in s0) \neq (x \in s1)$.

- In addition to the boolean operators `and` and `or` we have operators `cand` and `cor`, pronounced "conditional and" and "conditional or" respectively. They differ from the original operators only in that expression c in b `cand` c is only evaluated if b is `true`, whereas it is only evaluated in b `cor` c if b is `false`. In the case of `and` and `or` it is not defined whether just b, or just c, or both b and c are evaluated.
- Declarations can be given in any order with the restriction that a declaration of an identifier must precede its use.
- No runtime checks are performed at all: if you write incorrect programs you are on your own. (If you write correct programs, you are on your own as well.) If an error occurs you may or may not get the error message

  ```
  ***** MONITOR ERROR *****
  ```

  followed by some obscure text.
- You may write constant expressions like `4*5` or `N-1` where you would otherwise have to write a constant like `7` or `N`.
- The `goto` statement is not (yet) implemented.
- No packed data types exist.
- An expression for values of record types exists, which consists of the record type identifier followed by a list of expressions, one for each field. For example:

  ```
  type complex = record re, im: real end;
  var z: complex;

  ... z:= complex(0, 1) ...

  type coordinate = record case polar: boolean of
                      true : (r, phi: real);
                      false: (x, y: real)
                     end;
  var p: coordinate;

  p:= coordinate(true : (29.3, arctan(1)));
  p:= coordinate(false: (p.r * cos(p.phi), p.r * sin(p.phi)));
  ```

- An expression for values of array types exists, which consists of the array type identifier followed by a list of expressions, one for each array element. For example:

  ```
  type vector = array [1..3] of real;
  var v: vector;

  ... v:= vector(0, 1, 4*arctan(1)) ...
  ```

- Integers are represented by 16 bit words and are therefore confined to the range $-32768..+32767$.
- Reals are represented in the IEEE single-precision format. Denormalized numbers, NaNs, and Infs are not (yet) dealt with.
- Sets of integers are restricted to subsets of `set of` 0..15. Other set types are not restricted to 16 elements; for example, a `set of char` is perfectly valid.

Three *compiler directives* exist. They are used to control the compiler's operation somewhat. A compiler directive is part of a comment. It consists of a `$` sign, followed by a letter, followed by a `+` or `-` sign. The following directives exist. The default for all directives is "off". It does not matter where the directive is given: it always applies to the whole program.

- `M+` or `M-` produces or suppresses a listing of the allocation of memory to global variables.
- `O+` or `O-` activates or avoids improvement of the object code (it is not really optimization but it does increase the compile time significantly).
- `S+` or `S-` produces or suppresses a listing of the object code in assembly format.

4 Concurrent programs – one processor

Thus far we have seen how to run a sequential program on a single Mosaic node. Next we discuss how to write concurrent programs. In this section we focus on programs that are executed by a single Mosaic node, and in a later section we turn to distributed programs.

In our previous program there was one unusual part: the call of procedure `monitor` as the initial statement. In fact, this procedure call creates two independent processes: one for reading characters from your keyboard, and one for sending characters to your screen. How was this accomplished, because it appeared that reading and writing are done in the standard way via the files `input` and `output`? Indeed, that is what it appeared to be, but in reality this is something that has been changed completely from what you are used to. In fact, `input` and `output` are not files, they are now channels that convey messages of type `char`. A message can be transmitted on a channel with a `write` statement and it can be received by a different process with a `read` statement. Here is an example of a program that consists of two processes. The first process transmits a sequence of integers via channel `c`. The second process receives the integers and stores them in array `a`. When both processes are done, the contents of the array are displayed.

```
program example;

#include <declarations>
#include <monitor>

const n = 100;
```

```
var i, j: integer;
    a: array [1..n] of integer;
    c: channel of integer;

begin monitor;
      cobegin
        for i:= 1 to n do write(c, i*i);
        for j:= 1 to n do read(c, a[j])
      coend;
      for i:= 1 to n do write(output, a[i])
end.
```

Observe the new features that we have introduced. The type `channel of integer` indicates that `c` is a channel whose messages are restricted to be of type `integer`. A message of a different type cannot be transmitted via `c`. The channel types replace the file types in this dialect of Pascal. Files in the Pascal sense do not exist and the only operations on channels are `read` and `write` statements. This implies that we do not have the `put` or `get` operations, and that we do not have a file-pointer or a similar channel-pointer. The second new construct is the `cobegin - coend` statement. It is syntactically similar to the `begin - end` statement but the meaning is different: the statements that are enumerated are executed concurrently, and the whole construct is completed only when all constituent statements are completed. In this context, the constituting statements are called processes. The third relevant aspect is the synchronization that is part of the `read` and `write` statements. The processes that perform these communication statements proceed at some otherwise undetermined rates, but the matching `read` and `write` statements are performed simultaneously. This is implemented by suspending the process that arrives at the communication statement first until the other process reaches its communication statement. Then the value of the expression given in the `write` statement is assigned to the variable given in the corresponding `read` statement, and the two processes proceed from there on. It makes no difference at all which of the two was first, the sending or the receiving process: the communication is synchronous. This is sometimes referred to as zero-slack communication. Consider the next program. It is a modification of the above program and attempts to send two sequences of integers: one sequence in each direction.

```
program example;

#include <declarations>
#include <monitor>

const n = 100;

var i, j: integer;
    a, b: array [1..n] of integer;
    c, d: channel of integer;

begin monitor;
      cobegin
```

```
        for i:= 1 to n do begin write(c, i*i); read(d, b[i]) end;
        for j:= 1 to n do begin write(d, j*7); read(c, a[j]) end
      coend;
      for i:= 1 to n do write(output, a[i]);
      for i:= 1 to n do write(output, b[i])
end.
```

This program will get stuck when executed: both processes are suspended on their `write` statements and, therefore, neither of the two reaches a corresponding `read` statement. This phenomenon is called deadlock. A program that does succeed changes the concurrent part to

```
cobegin
  for i:= 1 to n do begin write(c, i*i); read(d, b[i]) end;
  for j:= 1 to n do begin read(c, a[j]); write(d, j*7) end
coend
```

or to

```
cobegin
  for i:= 1 to n do cobegin write(c, i*i); read(d, b[i]) coend;
  for j:= 1 to n do cobegin write(d, j*7); read(c, a[j]) coend
coend
```

In the first solution the two processes perform the communication on `c` first and then the communication on `d`. In the latter solution these two communications are performed concurrently: there is no interference between the two. In both solutions deadlock does not occur.

Suppose we now want to write a program with three processes: two processes that both send a sequence of integers to the third where they are merged into a single sequence. We cannot write

```
cobegin
  for i:= 1 to n do write(c, i*i);
  for j:= 1 to n do write(c, j*7);
  for k:= 1 to 2*n do read(c, a[k])
coend
```

because this would create two processes writing on the same channel. A channel is a point-to-point connection between two processes: one sender and one receiver. (If you try to run the above program, you do not get an error message; you will get either deadlock, or completely unintelligible results.) What we need to do is to make a receiver with two input channels, one from each of the two sending processes, and to make it select between messages that are coming from either of the two. We do not want to alternate between receiving from the one and receiving from the other: just receive the message that arrives first. To that end we introduce the `select` statement. It is illustrated below.

```
program example;
```

```
#include <declarations>
#include <monitor>

const n = 100;

var i, j, k: integer;
    a: array [1..2*n] of integer;
    c, d: channel of integer;

begin monitor;
      cobegin
        for i:= 1 to n do write(c, i*i);
        for j:= 1 to n do write(d, j*7);
        for k:= 1 to 2*n do
          select
            c : read(c, a[k]);
            d : read(d, a[k])
          end
      coend;
      for i:= 1 to 2*n do write(output, a[i])
end.
```

The `select` statement looks very much like a `case` statement. Each alternative consists of a channel and a statement. An alternative is selected when a communication on the named channel would not lead to suspension. If each choice would lead to a suspension then execution of the `select` statement is suspended until one of the channels becomes available for such a communication. In the example: if the third process executes the `select` statement, it is checked whether the first process is already suspended on `c`; if it is then the first alternative is chosen because the subsequent `read` on `c` will not be suspended. If the first process is not presently suspended on `c` then `d` is checked, and the second alternative may be chosen. If it is not chosen (because the second process was not yet suspended on `d`) then the third process is suspended and the selection is performed as soon as the first process reaches its `write` on `c` or the second process reaches its `write` on `d`. Notice that if the two processes that are connected via some channel both perform a `select` statement on the channel, then neither of the two is performing a `read` or `write` and the selection will not succeed. In the example, the two channels on which the selection is based are input channels to the executing process. There is no restriction, however, on the "direction" of the channels in a `select` statement.

Programs like the one above are also used to implement buffer processes. A buffer is a process that relaxes the slack between a sender and a receiver by storing a number of messages that have already been sent and have not yet been received. This may smoothen the operation of a system if the speed at which messages are sent or received varies. Below we give such a buffer process. The buffer process outputs a message to the receiver only if it currently stores at least one message. Similarly, the buffer inputs a message from the sender only if its buffer capacity admits storing another message. Hence, the selection between the communication actions is not only based on whether the communication would be suspended or not, but also on some local condition. This is expressed by including the condition in the `select` statement. In the example we give a buffer process as a procedure whose input and output channels are parameters. Just like files in Pascal, channels are always passed as reference or `var` parameters.

```
type data    = ... { some arbitrary type } ;
     dataseq = channel of data;
procedure buffer(var ci, co: dataseq);
  const n = 10;
  var b: array [1..n] of data;
      i, j, k: integer;
begin i:= 1; j:= 1; k:= 0;
      while true do
      select
        (k<n) and ci : begin read(ci, b[i]); i:=i mod n + 1; k:=k+1 end;
        (k>0) and co : begin write(co, b[j]); j:=j mod n + 1; k:=k-1 end
      end
end;
```

Observe that this is a nonterminating program. How do we create a number of nonterminating programs that are supposed to run in conjunction with a bunch of terminating programs? One can come up with a `cobegin - coend` construct embedded within a recursive procedure, but often this is not very elegant. A simpler way is to initiate them, but not wait for their completion. This can be done with a `fork` statement. Another reason for having a `fork` statement is that we sometimes do not know in advance how many processes need to be created, and we might want to create them in a loop of one sort or another. Initiating the processes and waiting for their completion may then have to be separate activities, and we need two different statements for them. To that end, the `fork` statement is parameterized in a way that we discuss below, and the same parameter is used in the waiting counter part, the `join` statement. The two statements use a parameter, `v` say, of type `class` that can be used for this purpose only. A process is created through execution of

`fork p(`*arguments*`) : v`

where `p` is a procedure, followed by its list of arguments (if any), and followed by the `class` parameter. In the latter it is recorded which processes have been forked off this way. Completion of processes that have been forked off can be awaited by executing

`join v`

which terminates only after all processes that have been forked off via a `fork` on `class` variable `v` have terminated. By using a number of these `class` variables we can distinguish between different groups of processes. For any `class` variable, at any time, at most one process should be waiting on the `join` statement. It is not the case that, if two processes are trying to execute a `join` statement on the same `class` variable, both are continued when all processes forked on that `class` have terminated. Instead, havoc results because of the multiple `join`.

Typically one uses one class for all processes that do not terminate at all, and one never performs a `join` statement on this variable. For example, the include procedure `monitor` that we have used before consists of only two statements. It reads

```
procedure monitor;
begin fork reader: eternal; fork writer: eternal end;
```

where `reader` and `writer` are two nonterminating procedures that implement the conversion from keyboard input to messages on channel `input`, and from messages on channel `output` to the screen respectively. Variable `eternal` of type `class` is declared in the include file `declarations`. It is used for forking the nonterminating processes. (In fact the procedure `monitor` is more complicated; we return to that later.)

A `read` or `write` statement has one channel parameter, followed by a list of one or more variables or expressions. Each of them leads to one communication. For example,

```
write(output, 'a', c)
```

is equivalent to

```
write(output, 'a'); write(output, c)
```

The type of `input` and `output` is `channel of char` so that it can be used to communicate characters. For channels of this particular type there is a conversion mechanism, both for reading and writing. If the channel in a `write` statement is of type `channel of char` while the expression is of type `integer`, `real`, or `array [...] of char` then a sequence of characters will be communicated that form the representation of the given value. For example,

```
write(output, 123)
```

is equivalent to

```
write(output, ' ', ' ', ' ', ' ', ' ', '1', '2', '3')
```

In the case of an integer expression the number of characters transmitted can be given by a so-called field width. It is 8 by default, and can be given as illustrated.

```
write(output, 123:4)
```

is equivalent to

```
write(output, ' ', '1', '2', '3')
```

In the case of a real expression the field-width can be given as one number which produces the floating-point representation, or as two numbers (the field-width and the number of fractional digits) which produces the fixed-point representation. The default is a floating-point representation with field-width

16. In the `read` statement an integer or real variable can be given, and a sequence of characters will be read and converted to a numeric value. The first character following the number is read also (in order to determine the end of the number's representation). Just like in Pascal, statements `writeln` and `readln` exist.

We discuss a few more features that are of less importance. The `select` statement is used to select between a number of channels. If the channels are array elements then we need as many alternatives as array elements. Instead of

```
const n = 10;
var c: array [0..n-1] of channel of char;
    d: channel of char;
    k: char;
while true do
select
  c[0]   : begin read(c[0], k); write(d, k) end;
  c[1]   : begin read(c[1], k); write(d, k) end;
   ...
  c[n-1] : begin read(c[n-1], k); write(d, k) end
end
```

which is unacceptable because of the dots, we write

```
const n = 10;
var c: array [0..n-1] of channel of char;
    d: channel of char;
    k: char;
    i: integer;
while true do
select
  for i:= 0 to n-1 do
    c[i] : begin read(c[i], k); write(d, k) end
end
```

This will do the job just fine.

There might be a problem with the fact that the channels are checked in some specific order: first `c[0]`, then `c[1]`, and so on. If the processes sending via `c[0]` and `c[1]` are producing new messages at high rates then a communication via `c[2]` might never occur: this merger is said to be unfair. A fair solution can be obtained by keeping track of the number of the channel that succeeded last time, and trying from the next on.

```
const n = 10;
var c: array [0..n-1] of channel of char;
    d: channel of char;
    k: char;
```

```
    i, j: integer;
j:= 0;
while true do
select
  for i:= 1 to n do
    c[(i+j) mod n] : begin j:= (i+j) mod n; read(c[j], k);
                           write(d, k)
                     end
end
```

Another way of making the merge fair is by using more complicated boolean expressions in the `select` statement. For example, if we have only 2 channels to merge we could write

```
var c: array [0..1] of channel of char;
    d: channel of char;
    x, y: char;
while true do
select
  c[0] and not c[1] :
     begin read(c[0], x); write(d, x) end;
  c[1] and not c[0] :
     begin read(c[1], y); write(d, y) end;
  c[0] and c[1] :
     begin read(c[0], x); read(c[1], y); write(d, x, y) end
end
```

Observe that this more general `select` statement is not so easy to define. In the previous version we could assure that the boolean expression is `true` when execution of the selected statement is begun. As soon as we have a negation before a channel name this can no longer be asserted, since the boolean expression `c[0] and not c[1]` might have been found `true`, but soon thereafter another process may have attempted a `write` on channel `c[1]` so that `not c[1]` is no longer `true`. One should be careful in the use of boolean expressions that are not monotonic functions. It appears that they are useful only to turn unfair programs into fair ones. One property that makes nonmonotonic guards somewhat bearable is the fact that the guards are evaluated as an atomic action: the value of the guards does not change while they are being evaluated. For example, in

```
select
  c     : ...
  not c : ...
end
```

exactly one of the two guards is `true`. Had evaluation not been atomic, the first guard might have been found `false`, then the other process suspends on `c`, and thereafter evaluation of the second guard turns out to be `false` also. The atomicity prevents the other process from changing the boolean value of `c` by not executing parts of the other process while evaluating guards.

Notice that we talk about `c[0] and not c[1]` as an ordinary boolean expression. Of course, it isn't, since `c[0]` and `c[1]` are channels, and not boolean variables. In the guard of a `select` statement, however, this boolean interpretation is valid. The boolean value of the channel is `true` if the "other" process is suspended on the corresponding `read` or `write` statement, and `false` otherwise. Sometimes we are interested in this boolean value outside the context of a `select` statement. The standard function `probe`, whose argument is a channel, returns exactly this value.

One may wonder why we have two different ways of writing down this boolean value of a channel, either as `c` or as `probe(c)`, where the former of the two is allowed in the guard of a `select` statement only. The reason is the following implementation issue. If all guards of a `select` statement are `false` then the executing process is suspended until at least one of the guards is `true`. In order to avoid "busy waiting" where the process keeps trying over and over again, the responsibility of waking up the process that is suspended in a selection is delegated to the processes that change the value of the guards, i.e., to the processes that become suspended in a `read` or `write` statement on a channel that occurs in the guard of a suspended `select` statement. Converting a channel name to a boolean value therefore requires some special actions that affect the state of the channel. In the case of the `probe` only the boolean value is asked for without affecting the state of the channel. This is where the two differ, and because the semantics differ we use a different notation. The change in status of a channel is the one and only reason for resuming execution of a `select` statement. If a guard changes value because it contains shared variables that are modified by one of the processes then this will not cause the suspended `select` statement to resume execution.

The other aspect of the `select` statement is that the process that performs the selection is suspended if no channel has a suspended partner process. Sometimes we want to write a program that selects a channel if it does not lead to a suspension, and does something else otherwise. To that end, the `select` statement can be given an extra alternative. For example, a buffer that repeatedly prints a question mark if no communication on its input or output channels seems possible can be written as follows.

```
while true do
select
  (k<n) and ci : begin read(ci, b[i]); i:= i mod n + 1; k:= k+1 end;
  (k>0) and co : begin write(co, b[j]); j:= j mod n + 1; k:= k-1 end
otherwise write(output, '?')
end
```

By the way, the same effect can be achieved by

```
while true do
 if (k<n) and probe(ci)
     then begin read(ci, b[i]); i:= i mod n + 1; k:= k+1 end else
 if (k>0) and probe(co)
     then begin write(co, b[j]); j:= j mod n + 1; k:= k-1 end
     else write(output, '?')
```

The only reason for having the `otherwise` option is that it is part of the `select` statement and, therefore, guarantees that the guards are evaluated as an atomic action. Atomicity is not guaranteed when one uses calls of `probe` in an `if` statement.

We need to discuss two more extensions relating to communication actions. Sometimes we find ourselves writing constructs like

```
const n = 100;
var i, len: integer;
    a: array [1..n] of integer;
    c: channel of integer;

write(c, len); for i:= 1 to len do write(c, a[i])
```

to communicate the first `len` elements of array `a`. We could, of course, write

```
write(c, len, a)
```

to reduce the number of communications from `len+1` to 2. However, it copies `n+1` integers instead of `len+1`. This is not very attractive in the case where `n` is much larger than `len` is. To solve this dilemma, and this one only, we have the construct

```
write(c, len, a for 1 to len)    .
```

It consists of two communications, one which sends 1 integer, viz. `len`, and one for copying elements 1 through `len` of array `a`. Both the upper and the lower bound can be arbitrary expressions of the proper index type. It is not equivalent to either of the other two programs and therefore should be used in conjunction with a similar `read` statement only. Notice that the case `len=0` leads to 2 communications just like the case `len>0` does. Although the second communication does not copy any array elements it does synchronize sender and receiver.

In the last example we had two communications: one for transmitting an integer `len`, followed by one for transmitting `len` array elements. If the array contains integers then there is no problem with using a channel of integers. However, if the array contains elements of a different type, say `char`, then we do have a problem: we need to communicate an integer and a number of `char`s. We get around this problem by using an untyped channel. The only thing it does is to suppress the type checking. An untyped channel is declared as being of type `channel`, without subsequent `of` *element type*. When communicating via such a channel the programmer has to make sure that the types at the sender and receiver side of the communication are the same.

5 Some machine-specific features

In order to construct the processes that implement the communication with the input and output devices, some low-level features have been added. They are of limited use in any other context.

- A variable can be declared together with its address within the memory. This is done by postfixing the declaration with @ *address*. As an example consider the following declaration.

  ```
  var memory:  array [0..32*1024-1] of integer @ 0;
  ```

Through array `memory` the program has direct access to the memory. For example, integer value 17 can be written at memory location 5 by executing

```
memory[5]:= 17
```

Following the declaration

```
reg:  set of 0..15 @ 32767;
```

one can write statements like

```
reg:= reg+[2,4]
```

and

```
if 4 in reg then statement
```

- Integers can be given in hexadecimal representation if the representation is preceded by a $.

- One can include procedures and functions in machine code. The body of a procedure may be a sequence of integer expressions, separated by commas. The value of these expressions is copied directly into the object code: the integers are interpreted as instructions. Such a procedure cannot have local declarations and its body starts with the keyword `code`. When the keyword `code` is preceded by `inline` then the body is expanded inline at every call of the procedure or function. In that case it should not contain an instruction for return from the call (because there will be no instruction for performing the call). Procedures and functions are called with their return address in register `r7`, with a static link in `r6`, with `r12` as a stack pointer, and with `r13` as a stack limit pointer. The static link is omitted if the routine is declared in the outermost block. All parameters are passed via the stack, and the routine should remove them before returning. A function returns its value in register `r0` if it fits in one word (and does something more complicated otherwise). The routines may use registers `r0` through `r7` in any way, and the other registers should not be tempered with. When the subroutine body is executed, stack pointer `r12` points to a word in memory that contains the last parameter. The order of the parameters is: from the first parameter at the highest address to the last parameter at the lowest address. As an example, the next function delivers the setting of the interrupt mask register, and the procedure writes a new value into the same.

```
type word = set of 0..15;
function imr: word;
  inline code $3830;
procedure set_imr(x: word);
  inline code $17c3;
```

- Two functions support the handling of data structures. Function `sizeof` is called with a type identifier as parameter and returns the number of words occupied by a variable of the type. Function `address` is called with a variable and returns the address of the variable. The resulting address is of type `integer`. Function `address` can also be applied to procedures and functions, and returns, as an integer, the address where the code of the procedure or function is found. For arbitrary pointer `p` of type `pt`, `ord(p)` is the integer value of the address pointed to by pointer `p` and `pt(i)` is the inverse operation that makes a pointer out of an integer. In the same spirit, the construct `integer(e)` interprets an arbitrary expression `e` as being of type integer, provided its representation fits in one machine word. And `t(e)` does the same job for a set type `t` and an integer expression `e`.

- functions `maxfree` and `totalfree` return an integer value indicating the size (in words) of the largest chunk of memory available and the total size of all chunks of memory available.
- Procedure `copy` has been added and the effect of `copy(a,b,n)` is to copy `n` words from address `a` onwards to address `b` onwards. All three parameters are of type `integer`.
- Procedure `switch` has been added, whose sole effect is to switch the processor to another process. It has been added to make up for the absence of a timer interrupt. A process switch normally takes place at every interrupt and with every `read` or `write` statement.
- Processes are assigned a priority: an integer number that determines preference of selecting one process over another. Procedure `setpriority(e)` sets the priority of the executing process to `e`. Integer function `priority` yields the priority of the present process. The priority of the initial process is 0 (priorities can be positive or negative) and if a process is created with a `fork` or `cobegin - coend` statement it receives the priority of the process that creates it. The processor maintains a queue of processes that are ready to run. At no time will the ready queue contain a process with a higher priority than the currently executing process. Do not use `maxint` as the priority of any process.

6 An Xwindows interface

The include file "Xgraphics" contains a set of routines that can be used to maintain an X window. The window is actually under control of `nmon`; the filemonitor is used to pass information between `nmon` and the program running in the root Mosaic. Two files are required to do this which implies that when this include file is used, at least four files have to be declared in the configuration file.

In our description here we refer to the manual of X windows where appropriate. The following routines are available.

- `GInit (x, y, w, h, ix, iy, d)`

  This routine initializes the window and displays it on the screen. `x` and `y` are the coordinates of the upper-left corner of the window; `ix` and `iy` the same for the icon. Width and height of the window are given in `w` and `h`. The parameter `d` can have two values: ICONIFIED or DISPLAYED which determines the way the window is displayed initially: as an icon or as a window. The origin of the window lies in the upper-left corner; the positive `y`-direction is down; the positive `x`-direction is right.

- `GStop`

  Removes the window and closes the graphics connection.

- `GIconify`

  Makes an icon of the window (Harmless if the window is already displayed as an icon.)

- `GDisplay`

  Makes a window of the icon. (Harmless if the window is already displyed.)

- `GSetPixel (x, y)`

  Draws a pixel in the current foreground color at position `(x, y)`.

- `GDrawLine (x0, y0, x1, y1)`

  A line is drawn from `(x0, y0)` to `(x1, y1)`.

- `GDrawRect (x, y, w, h, f)`

  A rectangle is drawn with upper-left corner `(x,y)`, width `w` and height `h`. The boolean `f` determines whether the rectangle will be filled. (For insiders: this specification differs from the routine `XFillRectangle`.)

- `GDrawArc (x, y, w, h, arc0, arc1, f: integer)`

  This function draws an arc just like the function `XDrawArc`. The parameter `f` may be one out of ARC_EDGE, ARC_CHORD, ARC_PIESLICE. In the first case the arc is not filled. In the last two cases the arc is filled as described in the manual for X windows.

- `GDrawCircle (x, y, r, f)`

  This function uses `GDrawArc` to draw a circle with radius `r` and middle point `(x,y)`. The boolean `f` determines whether the circle will be filled.

- `GSetMode (m)`

  Sets the drawing mode. The following constants may be given as argument: GCLEAR, GAND, GANDREV, GCOPY, GANDINV, GNOOP, GXOR, GOR and GNOR. They correspond directly to the raster operations defined in X windows. Most important are the modes GCOPY and GXOR. In the first mode all pixels are written directly into the window. In the second mode the logical exclusive or with the previous value is taken to be the new value.

- `GSetColor (ind, r, g, b)`

  The color with index `ind` is defined with the specified intensities of red, green and blue. Intensities range from 0 to 65535. If a color cannot be assigned due to hardware limitations, `nmon` tries to find a matcvhing color that is as close to the requested color as possible.

- `GSetfgColor (c)`

  Selects the color to be used for drawing and printing text. The argument is an index in a color lookup table and has to be defined earlier.

- `GSetPixelBlock (x, y, w, h, start)`

  `(x, y)`, `w` and `h` specify a rectangle of pixels to be assigned directly. The values are found in memory from address `start` onwards. It is expected that `h` arrays of `w` pixels each are found consecutively at that address.

- `GText (x, y, start, len)`

  At position `(x,y)` in the window, `len` characters are printed. They are found in the memory at address `start` onwards. The start position is the upper-left corner of the box containing the character. Two constants are available for computing the size of a text: TEXT_WIDTH and TEXT_HEIGHT give the width and the height of a character respectively.

- `GClear (x, y, w, h)`

  The specified rectangle is drawn in the background color.

- `GSetEventMask (m)`

  With this routine it is determined which events are received by the Mosaic. Six constants are defined to specify events. KPRESS, KRELEASE, BPRESS, BRELEASE, MNOTIFY and RESIZE. The corresponding events are: press/release of a mouse button, press/release of a keystroke, motion of the mouse pointer and resize of the window. A mask is specified as the logical or (or as the sum) of a number of these event types. For instance, after a call "GSetEventMask (BPRESS + MNOTIFY)" events are passed to the Mosaic in case the user moves the mouse or presses a mouse button. An event may be received by the function

- `GNextEvent (e)` An event is a variable of the type:

  ```
  type Event = record t, x, y, val:  integer end.
  ```

  The type is declared in the include file. The field `t` is the type of the event. Its value is one of the contstantsas mentioned above. The field `val` is the value of the button in case of a BPRESS/BRELEASE event. In case of a KPRESS/KRELEASE event it is the ordinal value of the corresponding character. `x` and `y` are the position of the pointer in case the type is MNOTIFY; in case the type is RESIZE they are the new width and height of the window respectively. In all other cases they are meaningless.

`Nmon` takes care of maintaining the screen so it is unnecessary for an application to maintain a copy of the screen for repainting.

7 Intermezzo on the compiler / preprocessor interface

The compiler is a shell script called `mcom`. It combines a preprocessor and the actual compiler. In general it is useful to have the possibility of sharing certain declarations among several programs. The preprocessor is capable of textually including such files in a program. It simply copies the file containing the program except lines that begin with a # (the very first character). The keyword `include` is then expected followed by the name of a file. This name is either given between < and > or between single quotes. Two instances of such include commands are `#include <ff>` and `#include 'ff'`. In both cases, `ff` is a file that is processed in the same way recursively. In the first case the file is searched in the standard include directory

```
~jan/mosaic/include
```

; in the second case `ff` is a relative or absolute pathname. The preprocessor does not check for any recursive references. Trailing spaces in the filename are significant. The preprocessor is the regular *C* preprocessor.

Script `mcom` is called as follows.

`mcom` *file* `[-o` *object*`]` `[-i` *include directory*`]`

The program is expected on the file *file*`.p` . The object file has the name *file*`.m` but this name can be changed by specifying the `-o` argument. The `-i` argument defines a directory that the preprocessor should consider to be the default directory in which to look for include files.

Using the script is the normal way to compile a Pascal source. It can be used for instance in a Unix "make" file. Usually `mcom` is called by the program `nmon`, but you can call it directly if you want to. (For example, if you want to run the compiler on your own Sun 4 instead of on `mosaic`.)

The script `mcom` described above uses two programs: the preprocessor and the compiler. The bare compiler is a program named `mpc`. It is called as follows.

`mpc` *prog*

The compiler expects the source code of the program on the file *prog*`.p`. It compiles the program and it produces either one error message on the standard output (yes, only one!) or, if the program is found to be syntactically correct, a code file. The name of this code file is *prog*`.m`. The name of the preprocessor is `tprep` and it is called from a command line as follows.

`tprep` *filename* `[-i` *include directory*`]`

The preprocessor reads the file *filename* and it produces on standard output a file that is used as input for `mpc`. It eliminates the lines `#include ....` as described above.

In order to make error reporting by the compiler look as if it operated on the original files that were input to the preprocessor rather than on the output thereof, two compiler directives are used, viz. `$F` and `$L`. You are strongly discouraged to use them yourself, but you might see them in use if you peek at the preprocessor's output (which is also discouraged).

Here are the include files that you have been using so far. The first one is `declarations`.

```
type textchannel = channel of char;
var eternal: class;
    px: integer @ $ffff;
    py: integer @ $fffe;
```

The role of the two variables `px` and `py` is clarified in the section on multiple processors. The second include file is `monitor`. We do not give its listing here, but we describe its features. We have discussed its operation on channels `input` and `output`, but more operations exist. Seven channels are used, six of type `textchannel` and one of type `channel of integer`. They are

`input`	– standard input as before
`output`	– standard output as before
`clock`	– time of day clock
`filein`	– file input
`fileout`	– file output
`endoffilesig`	– if the program reads a file, the end of the file is signaled on endoffilesig
`command`	– command channel for controlling file i/o

If a program uses files residing in the Sun's file system, the last four channels are used. In order to read a file, a program specifies the name of the file on the channel `command`. The name is followed by the command `READ` (defined in the include file). The contents of the file appears on the channel `filein`. The program can read this and when the end of the file is reached, the monitor communicates once along `endoffilesig` (notice that this communication must be accepted before continuing). The program must read the whole file until the end is reached before `filein` can be connected to another file.

When a file is written, the filename is specified on the channel `command` followed by `WRITE`. The contents of the file can be written on the channel `fileout`. The end of the file is signaled by communicating `ENDFILE` along `command`. Instead of writing a file, the program can append to it by specifying the command `APPEND` instead of `WRITE`. A program can have two files open at the same time: one for reading and one for writing.

The time of the day can be read from channel `clock`. It is expressed with two integers, say $t0, t1$ and they are assigned a value through execution of `read(clock, t0, t1)`. The time $t0$ is expressed in multiples of 100 seconds, whereas $t1$ is expressed in multiples of 10 milliseconds. We have $0 \leq t1 < 10000$ and $0 \leq t0 \leq maxint$ which provides a range of about a month. The time is an absolute time, not the time since the beginning of program execution. Therefore, you need to read the clock twice and the difference is the time elapsed between the two readings.

Finally, a program can stop the monitor `nmon` on the Sun by communicating `STOPIT` along `command`. We give an example of a program that prints file `test` on standard output.

```
program example;

#include <declarations>
#include <monitor>

var c: char;
```

```
    eof: boolean;

begin monitor;
      write(command, 'test', READ); eof:= false;
      while not eof do
      select
        filein      : begin read(filein, c); write(output, c) end;
        endoffilesig: begin read(endoffilesig, c); eof:= true end
      end;
      write(command, STOPIT)
end.
```

8 Concurrent programs – multiple processors

In this section we finally consider the case of writing programs for execution on a collection of processors. We have taken a rather extreme point of view. The programmer gives a program for each processor and has to make sure that they cooperate in a decent fashion. If the programs have something in common, or are even almost identical, then you can take advantage thereof through the mechanism of the include files —you need not copy your program text for every processor. The communication between processes running on different processors is not the same as for processes running on the same processor. The reason is that we found ourselves using very many channels, the number growing rapidly with the number of processors. Instead, we prefer a slightly different mechanism, known as the "multiplexer". The multiplexer provides each processor with a number of incoming links. There are no "outgoing" links connected to them. (We use the word "link" to distinguish them from the ordinary channels.) The number of links is usually small, typically 5. It is possible to send a message from any processor to any processor's incoming link: there is no fixed correspondence between a processor and incoming links on other processors. The multiplexer uses the following statements.

`send(` *dxdy* , *link* , *expression* `)`

`receive(` *link* , *variable* `)`

The destination of a message is given by an identification of the target processor and the index of the incoming link on that processor that will receive the message. The target processor is identified by giving its distance in the `x` and `y` direction in the processor network. The particular encoding thereof is provided by the function `dxdy` which is called the distance in the `x` direction and the distance in the `y` direction. One may also use function `destination` which is called with the `x` and `y` coordinate of the target processor. Both functions pack the signed magnitude representation of the distance in the x-direction and in the y-direction into a single word.

```
function sign_magnitude(x): integer;
begin if x<0 then sign_magnitude:= 128-x else sign_magnitude:= x end;
```

```
function dxdy(dx, dy: integer): integer;
begin dxdy:= sign_magnitude(dx) * 256 + sign_magnitude(dy) end;

function destination(x, y: integer): integer;
begin destination:= dxdy(x-px, y-py) end;
```

The last function uses the global variables `px` and `py` that were declared in the include file `declarations` and that are initialized upon loading the program. (Actually, this include file also contains the above three function declarations.) For example, consider a message that is to be sent to link 1 on the processor where `px=2` and `py=0`. The statement

```
send(destination(2, 0), 1, 'abc')
```

causes one message of three characters to be sent. This statement is completed as soon as the packet interface has been set up and the message has left the system. It is not necessarily the case that the message has been received by the other processor: it is not the synchronous communication mechanism as provided by `read` and `write` statements on channels. The message can be received on the other processor by the statement

```
receive(1, a)
```

Upon completion the message is received in variable `a`.

```
a='abc'
```

Note that a processor can send messages to itself by setting $dxdy$ to 0:

```
send(0, 1, 'abc')
```

or send to its neighbor in the $+y$ direction without referring to `px` or `py`:

```
send(1, 1, 'abc')          .
```

In order to use the multiplexer, the system has to be informed of the number of incoming links per processor. You do so by including the compiler directive

{`$X`*number*}

in your program. Links are numbered from 0 on (that is, from 0 up to and excluding the number following the `X`).

No type checking is performed on messages sent across links. This implies that you have to be extra careful in matching the type of the expression and the variable, just like in the case of an untyped channel.

In both the `send` and the `receive` statements you can use the `for - to` construct that is also available in communicating arrays with `write` and `read` statements on a channel.

You have to be aware of two restrictions. No two processes should be engaged in receiving a message via the same link simultaneously. (If it is attempted nevertheless, the havoc that results is similar to attempting simultaneous `read` statements on one channel.) The second restriction is that any message that arrives on a processor should be received through a `receive` statement without performing any other message transmissions (`send`s or `receive`s). The multiplexer does not provide a buffered fully-connected network between all links: the message sits in the network until the corresponding `receive` statement is executed. Failure to receive a message that is being delivered on one link can lead to suspension of subsequent communications on other incoming links on the same processor, or even suspension of outgoing communications. In the case of long messages, even other processors can be affected because a long message can be spread out over the buffer capacity of a number of physical inter-processor links.

In order to meet the restriction that a process be ready to perform the appropriate `receive` operation when a message arrives, one sometimes needs to split the receive operation in two parts. These two parts are called `set_receive` and `wait_receive`. The effect of

`set_receive(` *link*, *variable* `); wait_receive(` *link* `)`

is identical to

`receive(` *link*, *variable* `)`

but allows some other statement to be executed between the two halves of the `receive` statement. The first half provides all information to the multiplexer to enable reception of an incoming message on the named link. The second half suspends the executing process until the message arrives. One often inserts a statement between these two that notifies the sending process that the receiver is ready for accepting an incoming message.

The `send` statement sends a message consisting of the last argument, prefixed with a header. The header consists of two words, and contains the length of the message and the link number. The `receive` statement expects such a header at the beginning of each message and strips it off before returning the message in its last argument. When bootstrapping the system is it sometimes necessary to send a message that is not augmented with such a header. To that end, the statement

`send(` *dxdy* `, *,` *message* `)`

can be used. It sends the message indicted by the last argument to the processor indicated by the first argument but does not add any header to the message. It cannot be handled by a `receive` statement.

How do we execute our multi-processor programs? Assume that we have two different programs and a four-processor system. One program, say `control` is to run on processor 0, whereas the other three processors are to execute the other program, `work` say. First, copy a configuration file

```
cp ~jan/mosaic/config/4 4
```

and then compile and execute the four-processor program through the command

```
nmon 4 control work
```

The file **4** is a configuration file which causes **nmon** to load the first program (**control**) in processor 0, and the other program (**work**) in processors 1, 2 and 3. An example of a simple 4-processor program is given in the appendix.

9 More on nmon

Program **nmon** loads your program into the Mosaics and maintains the communication with them. It also checks whether someone else is already using the Mosaic system that you plan to use. If so, you are entered in a queue and you have to wait for your turn. The other users that are ahead of you are displayed, together with the time at which the first one of them obtained access to the system. When your turn has come, **nmon** generates a bell signal and waits for you to type a return. If this does not happen within a minute, you are kicked out of the queue.

When the queue is not empty it is displayed every minute for the user who has access to the system. This can be suppressed by the specification of the **w** flag (see below).

For the user of the system it is possible to see whether other someone else is waiting. This is done by typing the "quit"-key (normally, Control-\ on Sun workstations). The program prints the waiting queue or, if the queue is empty, the string "Nobody's waiting for you".

The programs that are loaded into the Mosaic system are determined by the so-called configuration file. In the examples it was called **1** or **4**. The configuration file consists of three major parts.

- Mosaic system layout. The description of the system consists of a declaration of all processors. A declaration of a processor is given by its number and the amount of available memory. As an example,

  ```
  memorysize of 0 is 65536
  ```

  declares processor 0 to have 65536 words of memory.

- The identity of the Mosaic processor that **nmon** communicates with. For example, if the configuration file contains the line

  ```
  root is 0
  ```

  then **nmon** will load the programs for all processors that are listed in the configuration file, but it communicates with processor 0 only. Every system has to have exactly one such **root** processor.

- Information needed for compilation and programs to be loaded. For every processor in the system, a (Pascal) program must be specified which should run on that processor. For example, if we specify

  ```
  0 runs work
  ```

  then processor 0 will be loaded with object code of program **work**. This object code is expected in file **work.m**. If the code file does not exist, the compiler is called with the appropriate arguments. The compiler is called also if the source file or the loader is newer than the code file.

The command for running the compiler can be specified as well as the name of the standard include directory.

`compiler is` *compiler*

include directory is *include directory*

Defaults are `~jan/mosaic/bin/mcom` and `~jan/mosaic/include` respectively.

- Note: If you want to run the compiler on your own Sun 4 instead of on `mosaic` you have to write something like

  `compiler is rsh triton cd` *working directory* \; `jan/mosaic/bin/mcom`

  (but, please, list the name of your machine rather than mine). Make sure that you have permission to go from the Sun 3 to the Sun 4 that you are listing here. Invoking the compiler can be suppressed by including

  `no compilation`

  in the configuration file.

On arbitrary places in the file the construct @*number* may be included. Such a construct refers to an extra argument, given in the command line after the configuration file and this argument is inserted there. Counting starts at 0. In the examples that we have given so far, this was always the name of a program to be executed. The program does not check for any recursive references. As an example we give a configuration file for a stand-alone system. The file `one` contains the following lines. It is the one that we have been using all along. It also illustrates how comments may be included in a configuration file: everything following a `*` upto the end of a line is ignored. Here is the contents of `1`.

```
memorysize of 0 is 65536  *
root is 0                 * the Mosaic with which nmon communicates
0 runs @0                 * the program is passed as an argument
```

If you include a line of the form

`echo` *rest of line*

then *rest of line* will be printed when `nmon` encounters the line.

Flags

The operation of `nmon` can be modified by the specification of flags. Each flag controls an option. The setting or resetting of a flag can be done in the command line, in the configuration file, or, if `nmon` is used interactively, by a command to `nmon`. A general call of `nmon` has the format

`nmon [-`*flags*`]` *configuration file program files*

The options and flags are:

- verbose. Flag `v` and the word `verbose` in the configuration file turn on verbose mode. If this option is set, `nmon` displays what is happening.
- compile. Flag `c` and the sentence `no compilation` suppress the compilation phase of `nmon`.
- loading. If the flag `l` is specified or the sentence `no loading` is included in the configuration file, `nmon` does not reset, bootstrap, and load the Mosaic system. It simply connects to the Mosaic system and it assumes that a program is running already.
- monitor. The sentence `no monitor` and the flag `m` cause `nmon` to omit the startup of the monitor after loading the Mosaic system.
- queueing. The flag `q` and the sentence `no queueing` in the configuration file, cause `nmon` to stop if the Mosaic system is not available. `nmon` stops in that case, with an exit status of 1.
- reset only. The sentence `reset only` and the flag `r` restrict the actions of `nmon` to resetting the Mosaic system.
- multiple access. The sentence `multiple access` and the flag `a` have the effect that multiple access to the Mosaic system is allowed. If this flag is set it is possible to have more than one `nmon` using the system at the same time. This can be used, for instance, to access more than one Mosaic processor and communicate directly with each of them. An `nmon` requesting multiple access should not load any programs but just connect to a Mosaic in which a monitor runs. Its configuration file should look like

  ```
  memorysize of 0 is 65536
  no loading
  root is 1              * for example
  multiple access
  ```

  A second `nmon` is allowed only if the current user of the system is the same as the one requesting multiple access. If the first user stops using the system, any `nmon` using the system by multiple access will stop also. `nmon` generates a warning if you acquire multiple access. You can suppress this warning with the `w` flag.
- interactive. The flag `j` turns `nmon` into interactive mode. It starts an interpreter that reads commands from the command line and executes these commands. (See below about the use of the interpreter.)
- warnings. The sentence `no warnings` and the flag `w` have the effect that the user is not warned if another user is waiting for access to the Mosaic system.

Monitor

After loading a Mosaic system, `nmon` acts as a monitor for the user program. In this way standard- and file i/o is implemented. `nmon` cooperates with a Pascal procedure that should be included in the user program that runs in the root of the Mosaic system. This procedure was discussed before, but you can write your own if you want to. `nmon` takes care of buffered terminal input/output. Certain escape codes can be typed to use a file as input or to log the output of the Mosaic system on a file. The codes consist of an escape character (ASCII 27, the escape key) followed by one of `r`, `w`, `l` or a period. The `r` causes `nmon` to use another file than the keyboard as input. The `w` causes `nmon` to write the output to a file instead of to the screen. The `l` causes `nmon` to log the output on a file. The `.` means termination of the `w` and `l` commands. `nmon` prompts the user for filenames.

The monitor is stopped either by typing Control-C or by the program running on the Mosaic system. If so desired, you can use your own monitor instead of the one provided by us. You can do so by including the line

`monitor is` *shell command*

in the configuration file. Instead of starting the monitor, `nmon` executes the shell command.

Interactive mode

If the `j` flag is specified when `nmon` is called, `nmon` is put into interactive mode. It prompts the user for commands. The prompt is `[`*machine*`] nmon-` where *machine* is the name of the host on which `nmon` has been started. We give an overview of the commands that can be given.

- `read` *config* `[`*parameters*`]`. A new configuration file is read with the appropriate parameters. If not enough parameters are given, `nmon` prompts the user.
- `mon`. A connection to the Mosaic system is made and the monitor is started. It is possible that this must be tried a couple of times because of synchronization errors with the program in the Mosaic.
- `load`. The programs are loaded into the Mosaic system.
- `reset`. The Mosaic system is reset.
- `run`. This command restarts `nmon` according to the current settings of the flags.
- `comp [`*name*`]`. If no argument is specified, `nmon` compiles all programs specified in the configuration file, if necessary. If an argument is specified, it compiles this program. In this case it does not check whether it's necessary.
- `names`. Prints the names of the current configuration file, the compiler, and the like.
- `par`. The current values of the command line parameters are printed.
- `@`*number* `:=` *string*. In this way a command line parameter is assigned a new value. The configuration file is read again.
- `cd` *directory*. Changes to another working directory.

- `flags`. Prints the current values of the flags.
- One character out of `vclmqrwad`. Toggles the value of the flag whose name starts with the specified character.
- `e`. Exit: `nmon` stops.

A line that starts with a period is interpreted by a shell (a Bourne shell).

Queueing

As mentioned earlier, `nmon` maintains a queue of waiters for the Mosaic system. The name of the Sun to which the Mosaic system is connected is used as a capability to determine a file that contains the queuing information. For example, file `mercury.table` is used as a queue table and the file with the name `mercury.lock` is used as a lockfile to establish exclusion for the Mosaic system connected to `mercury`.

Remarks:

- If for whatever reason `nmon` must be stopped by the Unix command `kill`, simply use the `kill` without extra arguments which generates a termination signal. When the command `kill -KILL` is given, `nmon` is not able to update the queue and therefore the Mosaic system will not be accessible.
- Processes in the queue communicate using (Unix) signals. Therefore it is required that they all have the same user identification. For the waiters, user `jan` has been chosen. Therefore, `nmon` should have the set-user-id bit set and it should be owned by `jan`.
- Queue tables must be readable and writable for the user `jan`.
- The table and lock files reside by default in the directory `~jan/mosaic/queue`. This default can be changed by specifying in the configuration file

  `queue directory is` *directory*

  You may want to use the queueing mechanism also for other purposes. For example, if you have your own way of using the Mosaic system but you don't want to interfere with others, you may specify in your configuration file

  `main program is` *shell command*

  `nmon` does the queueing but instead of its normal operation it will execute the shell command.

Appendix. An example program

The following program is an implementation of the sieve of Eratosthenes using dynamic creation and termination of sieve processes. The program runs on a single processor. You can find the program in file ~jan/mosaic/examples/prime.p (the directory examples contains some other programs as well). A sieve process receives a stream of natural numbers, the first of which is a prime. The process filters multiples of this prime from the rest of the stream. The parameter max determines whether a new sieve must be started as a successor of the current sieve. In this way a pipeline of sieve processes is started recursively. The pipeline is stopped by passing a termination signal.

```
program prime;

#include <declarations>
#include <monitor>

const terminate = -1;

type integers = channel of integer;

var bound, primes_per_line: integer;
    to_sieve, from_sieve: integers;
    comm: char;

procedure generator;
  var i: integer;
begin for i:= 2 to bound do write(to_sieve, i);
      write(to_sieve, terminate)
end;

procedure sieve(max: integer; var from_left, out: integers);
   var my_prime: integer;
       to_right: integers;
   procedure filter(var from, into: integers);
     var p: integer;
   begin read(from, p);
         while p <> terminate do
         begin if p mod my_prime <> 0 then write(into, p);
               read(from, p)
         end;
         write(into, p)
   end;
begin read(from_left, my_prime); write(out, my_prime);
      if my_prime < max
```

```
      then cobegin sieve(max, to_right, out);
                   filter(from_left, to_right)
           coend
      else filter(from_left, out)
end;

procedure print;
  var p, cnt: integer;
begin writeln(output, 'Primes between 2 and ', bound:0);
      read(from_sieve, p); cnt:= 0;
      while p <> terminate do
      begin write(output, p:6); cnt:= cnt+1;
            if cnt mod primes_per_line = 0 then writeln(output)
            read(from_sieve, p)
      end;
      if cnt mod primes_per_line <> 0 then writeln(output)
      writeln(output, cnt:0, ' prime numbers printed')
end;

begin monitor;
      writeln(output, 'Prime number generator')
      writeln(output, 'All primes are generated upto an upper bound');
      primes_per_line:= 8;
      repeat
         write(output, 'Type q(uit), u(pper bound) or p(rimes per line) ');
         readln(input, comm);
         case comm of
           'u': begin write(output, 'bound = '); read(input, bound);
                      cobegin
                          sieve(round(sqrt(bound)), to_sieve, from_sieve);
                          generator;
                          print
                      coend
                end;
           'p': begin write(output, 'Number of primes per line = ');
                      read(input, primes_per_line)
                end;
           'q': ;
           otherwise writeln(output, 'oops... ')
         end
      until comm = 'q';
      write(command, STOPIT)
end.
```

10 Appendix. Another example program

Next comes a program that runs on four processors. It computes a Mandelbrot picture and consists of 4 files. The first file is called `complex` and contains the definition of a type for complex numbers and three operations on them.

```
type complex = record re, im: real end;

function mpy(z0, z1: complex): complex;
begin mpy:=complex(z0.re*z1.re-z0.im*z1.im, z0.re*z1.im+z0.im*z1.re) end;

function add(z0, z1: complex): complex;
begin add:= complex(z0.re+z1.re, z0.im+z1.im) end;

function modulus(z0: complex): real;
begin modulus:= sqrt(sqr(z0.re) + sqr(z0.im)) end;
```

The second file is `mandeldecs`. It only contains some definitions that are shared between the last two programs.

```
const size = 500;
      datalink = 0;
      synclink = 1;

type picture  = record org: complex; step: real; max, line: integer end;
     scanline = record proc, lno: integer;
                       l: array [0..size-1] of integer
                end;
#include <declarations>
```

The third file is `control.p` and contains the program that runs on the Mosaic that is connected to the Sun. It distributes the work over the other three processors and collects the results. These are written to a file called `pixels`. The grain size of the work is a whole scan line. Observe that this program forks three processes for receiving scanlines from the workprocesses. this is done to meet the requirement that the receiver be ready to accept a message whenever the sender is going to send it.

```
program control;

#include 'complex'
#include 'mandeldecs'
#include <monitor>
{$X3}

var p: picture;
```

```
    s: array [0..2] of scanline;
    sz: real;
    hor, ver, i, j: integer;
    avail: array [0..2] of channel of integer;

procedure guard(i: integer; p: picture);
  var v: integer;
begin p.line:= i; send(dxdy(i+1, 0), datalink, p);
      repeat set_receive(i, s[i]);
             send(dxdy(i+1, 0), synclink, 0);
             wait_receive(i);
             read(avail[i], v);
             if v<size then begin
               p.line:= v; send(dxdy(i+1, 0), datalink, p)
             end;
             read(avail[i], v)
      until v>=size
end;

begin monitor;
      writeln(output, 'computing Mandelbrot sets');
      write(output, 'left bottom corner coordinates : ');
      read(input, p.org.re, p.org.im);
      write(output, 'size : '); read(input, sz); p.step:= sz/size;
      write(output, 'number of iterations : '); read(input, p.max);
      for ver:= 0 to 2 do fork guard(ver, p): eternal;
      write(command, 'pixels', WRITE);
      i:= 0;
      for ver:= work to size+work-1 do
        select for j:= 1 to work do
          avail[(i+j) mod work] :
            begin i:= (i+j) mod work;
                  write(avail[i], ver);
                  write(fileout, s[i].lno:3);
                  for hor:= 0 to size-1 do
                    write(fileout, s[i].l[hor]:3);
                  writeln(fileout);
                  write(avail[i], ver)
        end end;
      write(command, ENDFILE);
      writeln(output);
      writeln(output, 'done');
      write(command, STOPIT)
end.
```

The last file is `work.p` and contains the program that does the actual computation. It repeatedly receives a message containing information about the size and location of the picture, and about the scan line to be computed.

```
program work;

#include 'complex'
#include 'mandeldecs'

{$X2}

var i, k: integer;
    p: picture;
    s: scanline;

function mandel(c: complex): integer;
  var k: integer;
      a: real;
      z: complex;
begin k:= 0; z:= c; a:= 0;
      while (k<p.max) and (a<4.0) do begin
        z:= add(mpy(z, z), c); a:= sqr_modulus(z); k:= k+1
      end;
      mandel:= k
end;

begin while true do begin
        receive(datalink, p);
        for i:= 0 to size-1 do
          s.l[i]:= mandel(add(p.org,
                              complex(i*p.step,
                                      (size-1-p.line)*p.step)));
        s.lno:= p.line;
        receive(synclink, k);
        send(destination(0, 0), px-1, s)
end   end.
```

11 Appendix. Another useful file

This appendix lists an include file that may be of use in communicating messages on mesh network. The file is included as

```
#include <send.receive>
```

It defines operations `SEND` and `RECEIVE` that take one of the arguments `north`, `south`, `east`, or `west` as its first parameter, and a message as its second parameter. It implements a communication protocol that allows `SEND` and `RECEIVE` to be used without having to worry about the requirement that a message be sent only when the receiver is ready for its reception. For example, execution of

```
SEND(south, 29)
```

in processor (x, y) will be matched with

```
RECEIVE(north, v)
```

in processor $(x, y-1)$ and will result in setting v to 29. On a network that is a line in the x-direction, one would use only the `east` and `west` connections. (Notice that the definitions of `SEND` and `RECEIVE` are on single lines starting with `#`; they are long lines that did not fit in the format of this report but they are really single lines.)

```
type connection = record dst, golink, gilink, dolink, dilink: integer;
                         signal: channel of integer
                  end;

var dummy: integer;

#define SEND(c,msg)  begin read(c.signal,dummy); send(c.dst,c.dolink,msg) end
#define RECEIVE(c,v) begin set_receive(c.dilink,v); send(c.dst,c.golink,dummy);
                          wait_receive(c.dilink) end

procedure guard_connection(var c: connection);
begin setpriority(priority+2);
      set_receive(c.gilink, dummy);
      while true do begin
        wait_receive(c.gilink);
        set_receive(c.gilink, dummy);
        write(c.signal, dummy)
end   end;

procedure init_connection(var c: connection;
                          dst, golink, gilink, dolink, dilink: integer);
begin c.dst:= dst;
```

```
      c.golink:= golink;
      c.gilink:= gilink;
      c.dolink:= dolink;
      c.dilink:= dilink;
      fork guard_connection(c) : eternal
end;

{$X8}

var north, south, east, west: connection;

procedure init_connections;
begin init_connection(north, dxdy(0, +1), 0, 1, 2, 3);
      init_connection(south, dxdy(0, -1), 1, 0, 3, 2);
      init_connection(east,  dxdy(+1, 0), 4, 5, 6, 7);
      init_connection(west,  dxdy(-1, 0), 5, 4, 7, 6)
end;
```

If a 4 by 4 torus network is to be simulated on a line of 16 processors numbered from (0,0) to (15,0) then the last procedure is changed as follows.

```
procedure init_connections;
begin init_connection(north, destination((px+4) mod 16, 0), 0, 1, 2, 3);
      init_connection(south, destination((px-4) mod 16, 0), 1, 0, 3, 2);
      init_connection(east,  destination(px div 4 * 4 + (px+1) mod 4, 0), 4, 5, 6, 7);
      init_connection(west,  destination(px div 4 * 4 + (px-1) mod 4, 0), 5, 4, 7, 6)
end;
```