A Simple Extension of Pascal for Quasi-Parallel Processing

Terje Noodt
Dag Belsnes
Computing Center
University of Oslo

1 Introduction

The University of Oslo has for a number of years been engaged in the development of systems for data communications. The main work investments have been the design of suitable protocols, and the implementation of these in network node machines. Most of the node machines have been of the Nord family, produced by the Norwegian manufacturer Norsk Data A.S.

There exists no suitable language on the Nord for programming real-time stand-alone systems. Therefore, all programming has been done in assembly code. Even though we have felt the need for a high-level language tool, the cost of developing and/or implementing a suitable language was thought to be high.

Some time ago, we looked into the possibility of using the existing Pascal compiler for our purposes. It proved that a simple but usable language tool could be made from Pascal very cheaply. We have called this extension of Pascal for QPP (Quasi-Parallel Pascal). This article describes QPP and its implementation.

2 Basic primitives

The present section first discusses how to establish a suitable process concept. Then the sequencing of processes is treated.

2.1 Processes

The most important task in the design of QPP was to establish a process concept without deviating from Standard Pascal. In this context, a process is a sequential program together with a set of data on which the program operates. We call this set of data the attributes of the process.

In several respects, the Pascal procedure has the characteristics of a process. We have managed to use the procedure as a process, by overcoming the following two obstacles:

1. It is necessary that several processes can be executed simultaneously - that is, the processes must be able to have active phases in quasi-parallel.

2. It must be possible for processes to exchange information - that is, one process must be able to access the attributes of another process.

To transform the procedure concept into a process, point 1. requires that the attributes of a "process-procedure" must be retained while it has a passive phase. That is, a "process-procedure" cannot execute on the stack top as usual, but must have some permanent space in memory.

Point 2. requires some form of looking "into" a procedure. In Pascal, a similar mechanism is given by the record concept. Consider the following program fragment:

```
type
  PROCESS = record
              x, y: T
            end;
  PTRPROCESS = ↑PROCESS;
var
  p: PTRPROCESS;

procedure processprogram;
var
  LOCALS: PROCESS;
begin
  with LOCALS do
  begin
    . . .
  end
end
```

Within the with statement in processprogram the attributes x and y may be accessed directly.

A process is created by calling the function

```
function NEWPROCESS(procedure PROG);
```

This function allocates data space for the procedure PROG on the heap. The function value is a pointer to the record containing the process attributes. In reality, the pointer is a reference to the inside of the procedure object. The Pascal system, however, treats the pointer as if it were generated by the NEW function.

The main program (or another process) may access the attributes through the pointer generated by NEWPROCESS.

The following program fragment shows how a process is generated, and its attributes accessed from the outside:

```
p := NEWPROCESS(processprogram);
 . . .
p↑.y := . . .
 . . .
with p↑ do
  if x = . . .
```

Several processes of the same type may be generated as follows:

```
var
  p1, p2: PTRPROCESS;
 . . .
  p1 := NEWPROCESS(processprogram);
  p2 := NEWPROCESS(processprogram);
```

Processes of different types may be defined by declaring different PROCESS types, or by defining a variant part for each type of process within PROCESS.

Thus, a usable process concept has been established by

1. Implementation of the function NEWPROCESS. In Nord-10 Pascal this is an assembly routine of 15 instructions.

2. Requiring that the programmer stick to the following rules:
   a. Define a record type PROCESS which contains those variables of a process which are to be visible from outside the process.
   b. Declare a variable LOCALS of type PROCESS as the first variable within the process procedure.
   c. Surround the statements of the procedure by
      with LOCALS do begin . . . end

2.2 Sequencing

It must be possible to start and stop the execution of any process, in order that operations occur in the sequence required by the actual application. For this purpose, two operations are implemented (these are modelled after the corresponding primitives in Simula 67):

```
procedure RESUME(p: PTRPROCESS);
```

This procedure transfers control from the caller to the process given by the actual parameter p. The execution of p is resumed at the place where the process last became passive. The caller becomes passive.

```
procedure DETACH;
```

When a process p calls DETACH, it becomes passive. Control goes to the last process x which called RESUME(p).

The following method has been used to implement RESUME and DETACH efficiently and with ease.

A Pascal procedure object will normally contain one location for the return address (RA), and one location for the dynamic link (DL). Let CP be a pointer to the currently active process, and consider the main program to be a process with the name MAIN.

The operation RESUME(p) leaves the current program address in CP.RA, and the address of the currently active object (which may be CP itself or an ordinary procedure called by CP) in CP.DL. p.DL becomes the new active object, and execution is resumed at p.RA.

The DETACH operation is restricted to be used to give control back to the main program. It leaves the current program address in CP.RA, and the address of the currently active object in CP.DL. MAIN.DL becomes the new active object, and execution is resumed at MAIN.RA.

The DL location of a process is zero while the process is executing. Thus, CP is found by following the DL chain until DL equals zero. The following function is provided to enable the Pascal program to find CP:

```
function THISPROCESS: PTRPROCESS;
```

2.3 Summary

With a very small effort a primitive but usable process concept has been implemented within Pascal. On the Nord-10, the routines NEWPROCESS, RESUME, DETACH and THISPROCESS consist of ca 60 assembly instructions. No changes have been made to the Pascal compiler or the Pascal run-time library. Although Pascal may operate differently on other computers, the authors believe that our method of implementation may be adapted to most Pascal systems.

On the Nord-10, an ordinary procedure called from a process will execute in the memory space allocated to that process. This requires that the process object be large enough to accommodate such procedure calls. We have solved this problem by letting NEWPROCESS have one extra parameter, giving the largest necessary space for the process.

3 Process Scheduling

Section 2 defines and indicates how to implement a process concept and the basic primitives for process sequencing. To program a real-time system or a simulation model, some

additional concepts are needed. Also in this case SIMULA 67 is used as a source of inspiration. The new programming platform contains:

* a system time concept.

* a "sequencing set" containing the processes scheduled for future execution.

* primitives for process scheduling.

In this section we show how these concepts may be implemented in Standard Pascal, using the basic primitives of section 2.

3.1 Simulated time, Real time

In the case of simulations, the system time is introduced as in SIMULA, but in a real-time environment the system time corresponds closely to the time defined by the computer's real-time clock. The system time is represented by a variable in the main program:

```
SYSTIME:real;
```

The execution of an active phase of a process, called an event, is regarded as not consuming system time. That is, SYSTIME is only updated between the events. How SYSTIME is updated is described below.

3.2 The sequencing set

A process may be scheduled for the execution of a future event. An event is associated with a system time, indicating when the event will occur. This time is represented by a variable local to each process:

```
EVTIME:real;
```

All scheduled processes are collected in a set, the sequencing set, sorted on the EVTIME variable. The sequencing set is represented by a main program variable:

```
SQS:PTRPROCESS;
```

which points to the first member of the set, and a variable

```
NEXTPR:PTRPROCESS;
```

in each process pointing to the next element of the sequencing set.

When an active phase of a process ends, the first process P in the SQS will be the next process to execute an event. The value of SYSTIME is changed to EVTIME of P. If simulated time is used, the simulation is carried on by resuming the process P.

In a real-time system the new value of SYSTIME is compared with the computer's clock. If the difference is positive, the Pascal program makes a monitor call to release the use of the

CPU for the given amount of time. On return from the monitor call the procedure RESUME(P) is called.

3.3 Process scheduling

The following procedures define a small but convenient set of operations for discrete event scheduling. All procedures are written in Standard Pascal. The amount of Pascal code is about 40 lines. For a detailed description see the appendix.

procedure PASSIVATE;

The caller process ends its active phase, and the next event is given by the first element of the SQS. SYSTIME is updated, and in the real-time case the program may request a pause before the next process is resumed.

procedure HOLD(del:real);

Equivalent to PASSIVATE, except that the caller is put into the SQS with an event time equal to SYSTIME+del.

procedure ACTIVATE(p:PTRPROCES; del:real);

The process p is scheduled to have an event at the time SYSTIME+del.

procedure CANCEL(p:PTRPROCESS);

If the process p is scheduled to have an event, this event is cancelled. That is, p is removed from the SQS.

3.4 Summary

Based on the basic primitives discussed in section 2, we have defined a set of additional primitives suitable for discrete event scheduling. These primitives are implemented by Standard Pascal procedures and data structures. The system time concept is introduced in two variations: simulated time and real time. In the implementation the difference between the two time concepts is only visible as a small modification of the procedure PASSIVATE. An important consequence is that it is possible to test out a program by simulation and afterwards use the same program as a part of a real time system.

4 Concluding remarks

As an example, the Bounded Buffer problem has been programmed in the appendix.

At the University of Oslo, QPP has been used to program the UNINETT node. UNINETT is a computer network of the central computers of all universities in Norway, plus several other governmental computers. Each institution has a node machine

which hooks one or more computers into the network. At the University of Oslo, this node is a Nord-10. The size of the UNINETT node program is about 2200 lines of QPP code. In the development of this program, keeping to the restrictions of QPP was neither hampering nor the cause for any serious problems. The UNINETT project has shown that a considerable amount of development time may be gained by going from assembly code to a "primitive" high-level language tool. In cases where a full-fledged language tailored to the actual application (such as Concurrent Pascal) is not available, there seems to be good reason to select a solution such as ours.

The UNINETT node program was developed on a Nord-10 running the MOSS operating system. The first step in testing the program was to run it under MOSS as a simulation, using simulated time. Then the program was run in real time under MOSS. Finally, the program was transported to the UNINETT node machine, where it runs in real time. The node machine has a rudimentary operating system only, which supports stand-alone systems of this kind. The small size of the code which implements the QPP process primitives, has allowed us to easily make different versions to adapt to the environment in which the UNINETT program was to be run. It has proved very valuable to run the program as a simulation before it was run in real time. Development time was also saved by testing under an operating system with utilities such as interactive debugging, a file system etc. The errors remaining after transporting the program to the node machine have been few.

The reader who compares QPP with for instance Concurrent Pascal, will remark that QPP contains no primitives for the protection of shared data. Such a mechanism could be useful in QPP, but is not strictly necessary. The reason is that processes run in quasi-parallel rather than true parallel. An active phase of a process is regarded to take zero time, and thus is an indivisible operation. Time increases only when control is transferred from one process to another. It is the programmer who decides at which points in the program this may occur.

Appendix

This appendix contains a simple example of the use of QPP. A producer process generates characters which are read by a consumer process. The rate of production/consumption is up to the processes themselves, and in order to remove some of the time dependency between the processes, they are connected by a bounded buffer. However, since the buffer may get full (or empty) there is still need for some synchronization of the processes. This is achieved by the use of the ACTIVATE and PASSIVATE primitives.

The program also contains a complete implementation of the concepts defined in section 3. Names corresponding to concepts and primitives in QPP are written in capital letters, while small letters are use for variables particular for the example.

```
program prodcon;
const
  buflength = 16;
  buflgml = 15;
type

(*  definition of bounded ring buffer  *)

  bufindex = 0..buflgml;
  buf=record
        p,c:bufindex;
        txt:packed array[bufindex] of char;
    end;
  ptrbuf=↑buf;

(*  definition of the data structure of the processes  *)

  PTRPROCESS=↑PROCESS;
  processtype=(producer,consumer);
  PROCESS=record
      NEXTPR:PTRPROCESS; EVTIME:real; INSQS:boolean;
      case processtype of
        producer:(outbuf:ptrbuf; outcha:char);
        consumer:(inbuf :ptrbuf; incha :char);
    end;

var
  SQS:PTRPROCESS; SYSTIME:real;
    ptrpro,ptrcon:PTRPROCESS;

(**       basic primitives       **)

function NEWP(procedure p; siz:integer):PTRPROCESS; extern;
function THISP:PTRPROCESS; extern;
procedure RESUME(p:PTRPROCESS); extern;
procedure DETACH; extern;
```

```
(**       sequencing routines       **)

procedure INTOSQS(p:PTRPROCESS);
var rp,rpo:PTRPROCESS;
  begin
   with p↑ do
   begin
     rp:=SQS; rpo:=nil;
     while (rp<>nil) and (rp↑.EVTIME<EVTIME) do
     begin rpo:=rp; rp:=rp↑.NEXTPR end;
     if rpo=nil then SQS:=p else rpo↑.NEXTPR:=p;
     NEXTPR:=rp; INSQS:=true
   end;
  end;

procedure CANCEL(p:PTRPROCESS);
var rp,rpo:PTRPROCESS;
  begin
   with p↑ do
   if INSQS then
    begin
      INSQS:=false; rp:=SQS; rpo:=nil;
      while rp<>p do begin rpo:=rp; rp:=rp↑.NEXTPR end;
      if rpo=nil then SQS:=rp↑.NEXTPR else rpo↑.NEXTPR:=rp↑.NEXTPR;
    end;
  end;

procedure PASSIVATE;
var p:PTRPROCESS;
  begin
     p:=SQS; if p=nil then DETACH else SYSTIME:=p↑.EVTIME;
     (* if realtime then monitor call PAUSE(SYSTIME-CLOCK)      *)
     SQS:=p↑.NEXTPR; p↑.INSQS:=false; RESUME(p)
  end;

procedure HOLD(del:real);
var p:PTRPROCESS;
  begin p:=THISP; p↑.EVTIME:=SYSTIME+del; INTOSQS(p); PASSIVATE end

procedure ACTIVATE(p:PTRPROCESS; del:real);
  begin CANCEL(p); p↑.EVTIME:=SYSTIME+del; INTOSQS(p) end;
```

```
(**       buffer routines      **)

function bufempty(bp:ptrbuf):boolean;
  begin bufempty:=(bp↑.p=bp↑.c) end;
function buffull(bp:ptrbuf):boolean;
  begin buffull:=(((bp↑.p+1) mod buflength)=bp↑.c) end;
function putchar(bp:ptrbuf; ch:char):boolean;
  begin with bp↑ do
     if ((p+1) mod buflength)=c then putchar:=false else
     begin txt[p]:=ch; p:=(p+1) mod buflength; putchar:=true end;
  end;
function getchar(bp:ptrbuf; var ch:char):boolean;
  begin with bp↑ do
     if p=c then getchar:=false else
     begin ch:=txt[c]; c:=(c+1) mod buflength; getchar:=true end;
  end;

(**        processes      **)

procedure pproducer;
var LOCALS:PROCESS;
begin DETACH;
   with LOCALS do
   while true do
    begin
      (*  produce next character  *)
      if bufempty(outbuf) then ACTIVATE(ptrcon,Ø);
      while not putchar(outbuf,outcha) do PASSIVATE
    end
end;

procedure pconsumer;
var LOCALS:PROCESS;
begin DETACH;
   with LOCALS do
   while true do
    begin
      if buffull(inbuf) then ACTIVATE(ptrpro,Ø);
      while not getchar(inbuf,incha) do PASSIVATE;
      (*  consume character  *)
    end
end;

(**       main program     **)

begin
   ptrpro:=NEWP(pproducer,1ØØ); ptrcon:=NEWP(pconsumer,1ØØ);
   new(ptrpro↑.outbuf); ptrcon↑.inbuf:=ptrpro↑.outbuf;
   RESUME(ptrpro)
end.
```