Implementation of Concurrent Pascal on LSI-11

Mattsson, Sven Erik

1979

Document Version:
Publisher's PDF, also known as Version of record

CODEN: LUTFD2/(TFRT-7168)/1-020/(1979)

IMPLEMENTATION OF CONCURRENT PASCAL ON LSI-11

SVEN ERIK MATTSSON

DEPARTMENT OF AUTOMATIC CONTROL
LUND INSTITUTE OF TECHNOLOGY
AUGUST 1979

Dokumentutgivare
04T0 Lund Institute of Technology
Handläggare Dept of Automatic Control
36T0 Sven Erik Mattsson
Författare
08T0 Sven Erik Mattsson

Dokumentnamn
04T6 REPORT
Utgivningsdatum
06T4 August 1979

Dokumentbeteckning
04T6 LUTFD2/(TFRT-7168)/1-020/(1979)
Ärendebeteckning
06T6

10T4

Dokumenttitel och undertitel
18T0 Implementation of Concurrent Pascal on LSI-11.

Referat (sammandrag)
20T0 This paper considers the moving of the implementation of Concurrent Pascal for the PDP-11/45 computer to the LSI-11 computer. The resulting implementation for the LSI-11 computer is also discussed and described.

Referat skrivet av
42T0 Author

Förslag till ytterligare nyckelord
44T0 Portability Pascal

Klassifikationssystem och -klass(er)
50T0

Indextermer (ange källa)
52T0

Omfång
30T0 20 pages

Språk
38T0 English

Övriga bibliografiska uppgifter
56T2

Sekretessuppgifter
60T0

ISSN
60T4

ISBN
60T6

Dokumentet kan erhållas från
62T0 Department of Automatic Control
Lund Institute of Technology
Box 725, S-220 07 Lund 7, Sweden

Mottagarens uppgifter
62T4

Pris
66T0

SIS-DB 1 DOKUMENTDATABLAD enligt SIS 62 10 12

Blankett LU 11:25 1976–07

INTRODUCTION

Today it is commonly understood that for economic as well as for security reasons it is necessary to use abstract languages when writing concurrent programs. Concurrent Pascal designed by Brinch Hansen [1] is one attempt to solve this problem. It extends the sequential programming language Pascal [2] with facilities for concurrent programming. The language was originally implemented on the PDP-11/45 computer.

The compiler for Concurrent Pascal is written in Sequential Pascal [3]. It is programmed by Hartmann [4]. The compiler generates code for a virtual machine and the virtual machine is emulated by the code interpreter and the kernel, which are written in assembly language.

This paper describes experiences from transferring the compiler, code interpreter and the kernel to the LSI-11 computer.

THE SCENE

This section describes the starting-point and the conditions under which the work had to be done.

A distribution tape with a Concurrent Pascal compiler, a code interpreter, a kernel, the Solo Operating System [1] and two reports titled "Concurrent Pascal Implementation Notes" [5] and "Sequential Pascal Report" [3] were received from the University of Colorado. The compiler, the code interpreter and the kernel received were implemented for the PDP-11/45 computer and the intention was to implement Concurrent Pascal for an LSI-11 computer.

The LSI-11 computer system available has 28K words of RAM memory, a simple alphanumeric display, a printer and a dual diskette unit. Among the available software facilities under the RT-11 operating system [6] the OMSI Pascal compiler [7] can be mentioned. All the computer work was done on the LSI-11 computer itself.

This implementation of Concurrent Pascal for the LSI-11 computer is mainly intended for process control applications and not for implementing general purpose operating systems. The Solo Operating System is too large. It requires a core store of 39K words for programs and data.

MOVING OF THE COMPILER

This section describes the experiences gained and difficulties encountered when moving the Concurrent Pascal compiler from PDP-11/45 to LSI-11.

An overview of the compiler is given by Brinch Hansen [1]. It is described in detail by Hartmann [4]. The compiler itself is written in Sequential Pascal. The code is about 8000 lines long. It is divided into seven passes of 750-1600 lines each. The source code is readable and in general very interesting and worthwhile reading. This is consistent with one of the objectives of the compiler. In the introduction to his book Hartmann writes

> "Many of the compilation techniques used here are well-known, but, taken as a whole, this compiler is an engineering product that may serve as a prototype for industrial compiler writers".

The Concurrent Pascal compiler is indeed a good program, but even so some problems were encountered. These problems can be divided into three groups. First, the Concurrent Pascal compiler is written in Sequential Pascal. Sequential Pascal is a variant of Pascal and our LSI-11 system has an OMSI Pascal compiler. The problems caused by different Pascal implementations can be viewed as portability problems. Second, the compiler generates code for a virtual machine, but some of the properties (e.g. how numbers are represented) depend on the target machine and have to be considered. Third, genuine errors were found which had to be corrected and which should also be corrected in the PDP-11/45 version.

Portability Problems

The Concurrent Pascal compiler is written in Sequential Pascal. Unfortunately, Pascal is not standardized and there exist several versions. The original Pascal defined by Wirth [2] will in the following be called Pascal and the version of Brinch Hansen [3] will be called Sequential Pascal. Our version is made by Oregon Minicomputer Software Inc. and will be called OMSI Pascal [7]. Some of the differences are of minor importance and can easily be remedied with a good editor. There are, however, differences that can be troublesome and even force the programmer to rewrite large parts of the program.

Differences in the character set are a common portability problem that can be remedied with a good editor. The differences are listed in Table 1. Sequential Pascal allows underscore _ in identifiers, but OMSI Pascal does not. It

was not possible to simply remove the underscore characters. The removal caused ambiguous and illegal constructions. There were for example procedures called TYPE_, REAL_ and CASE_ and a variable called ARG and a procedure called ARG_. These significant underscores were replaced with X.

	Sequential Pascal	OMSI Pascal
Comment brackets	" "	{ }, (* *), /* */
Array brackets	(. .)	[ ]
Pointer symbol	@	↑

Table 1. Differences in the character set of Sequential Pascal and OMSI Pascal.

The size of sets is not defined in Pascal, but it is implementation dependent. OMSI Pascal limits sets to 64 elements and Sequential Pascal to 128 elements. It is easy to understand that this can cause a lot of trouble. In pass 1 (lexical analysis) there were sets of type char, and for this and other reasons discussed below it was decided to rewrite pass 1. In pass 2 (syntax analysis) there were sets of input operators and there are 66 input operators. Fortunately three of these operators (lconstl, messagel and newlinel) are handled directly by the GETSY procedure and are not visible for the parser so they could be removed from the sets. In all other cases the maximum number of elements was less than 64.

In pass 3 (name analysis) there is a function declared in the following way

```
type entryptr=↑entryrec;
function top:entryptr;
begin
  top:=ops[T].defentry
end;
```

Top was used in constructions like

```
with top↑ do ...; and
index:=top↑.noun;
```

According to the Pascal-user and manual report [2] and the Sequential Pascal Report [3] this use of a pointer function is not allowed in either version, although it is a useful extension. The constructs above were rewritten as

```
topl:=top;
with topl↑ do ...; and

topl:=top;
index:=topl↑.noun;
```

If a parameter in Sequential Pascal is prefixed by the word univ, the parameter is said to be of universal type. The prefix univ suppresses compatibility checking of parameters in routine calls. It is only checked that the formal and actual parameter are represented by the same number of store locations. This facility is not defined in Pascal. In pass 4 the following procedures could be found

```
type univset=array[1..8] of integer;
procedure pack(var packedset:integer;
               unpackedset:univ univset);
begin
  packedset:=unpackedset[1]
end;

procedure unpack(packedset:integer;
                 var unpackedset:univ univset);
begin
  unpackedset[1]:=packedset
end;
```

It turned out that these procedures were used to make the storing more efficient. They pack and unpack information to and from sets declared as

```
type legacytype=(classlegacy,monitorlegacy,
                 processlegacy,queuelegacy);
     legacys=set of legacytype;
```

This effect can be achieved in Pascal using records with variants in the following way

```
type intorset=(integertype,settype);
     univset=record
               case intorset of
                 integertype:(integerarray:
                                array[1..4] of integer);
                 settype:(legacyset:legacys)
             end;
```

```
procedure pack(var packedset:integer;
               unpackedset:legacys);
var setl:univset;
begin
  with setl do begin
    legacyset:=unpackedset;
    packedset:=integerarray[1]
  end
end;

procedure unpack(packedset:integer;
                 var unpackedset:legacys);
var setl:univset;
begin
  with setl do begin
    integerarray[1]:=packedset;
    unpackedset:=legacyset
  end
end;
```

In Sequential Pascal the case clause of a record with variants must have a tag field and a variant field can only be selected if the value of the tag field is equal to one of the labels of that variant. In Pascal records with variants can be used for type conversions. In Sequential Pascal the concept of universal parameters can be used for the same purpose.

Brinch Hansen [1] writes "Our goal was to make a compiler that can compile operating systems on a minicomputer with at least 16K words of core store and a slow disk (50 msec/transfer)". Their solution relies on the Solo Operating System, written in Concurrent Pascal. Under Solo it is possible to let a Sequential Pascal program run other Sequential Pascal programs. The seven passes are written as seven ordinary Sequential Pascal programs and a Sequential Pascal program runs these in turn. The bootstrap problems could be solved in different ways, but it was possible to avoid them. The OMSI Pascal compiler allows separately compiled procedures. The linker under RT-11 can then be used to build a program that is executable under RT-11. The seven passes are compiled separately with a common prefix, which contains declarations of global variables and procedures. The compiler is then linked as an overlaid program, where the seven passes overlay each other.

The standard I/O routines in Sequential Pascal are very rudimentary, for example the user has to define a procedure to write integers. Many of the user defined I/O routines were common to several passes and these were moved from the passes and collected into a separate compile module.

For many reasons pass 1 was rewritten to a large extent. The problem with different sizes of sets is discussed above. Pass 1 decodes the command line from the operator, checks

its validity and opens the files. It was decided that the command structure should follow the RT-11 standard. The OMSI Pascal compiler makes it possible to check if the files exist. To make the programs more readable lower case letters are allowed in the LSI-11 implementation, but they are interpreted as their upper case counterparts.

The identifiers are stored in a table and a hash key is used as an index to the table. The hash key starts at zero and for each character in the identifier the hash key in the PDP-11/45 version is updated as

```
const
  span=26;{number of distinct id chars}
  hashmax=750;{hash table upperbound}
  hashmax1=751;{primelength}
var
  hashkey:0..hashmax;
  hashtable:array[0..hashmax] of
              record
                ...
              end;

....
  hashkey:=hashkey*(ord(ch) mod span+1) mod hashmax1;
```

This updating rule uses one multiplication and two divisions for each character in the identifier. Multiplication and division are fairly slow operations in an LSI-11 even with the Extended Arithmetic Chip. The updating rule was therefore changed into

```
hashkey:=(hashkey+hashkey*16+code(ch)) mod 1024;
if hashkey>=hashmax1 then hashkey:=hashkey-hashmax1;
```

where code(ch) is a simple function which maps letters and digits onto the interval 1..38. The OMSI Pascal compiler implements the multiplication with 16 as shifts and the "mod 1024" is implemented as a mask operation. Consequently, this rule, which contains neither division nor multiplication is much faster.

Differences between the Virtual Machines

The Concurrent Pascal compiler generates code for a virtual machine. The virtual machine is emulated on the target machine. In this way the problem with different target machines is moved from the construction of the compiler to a construction of an emulation program. However, for efficiency reasons the representation of integers and reals must depend on the target machine. The compiler must know the ranges in order to test the legality of a numeric constant. It must also know how many bytes an integer or

real occupies in the memory in order to make a correct allocation. This information is given as a set of constants and is easy to change with an editor.

Genuine Errors

Four errors were found: one range error, one endless loop, one uninitialized variable and one more complex error.

In pass 1 the following code could be found

```
....
    const
      idpieclength=9; {ten chars per piece}
      maxpieces=13; {14 pieces => 140 chars}
      blank='          '; {blank padding}
    ....
    type
      piece=array[0..idpiecelength] of char;
    ....
    var
      idtext:array[0..maxpieces] of piece;
    ....
    procedure identifier;
    begin
      pieces:=-1; charindex:=idpiecelength; hashkey:=0;
      repeat
        if charindex=idpiecelength then begin
          charindex:=0; pieces:=pieces+1; idtext:=blank end
        else charindex:=charindex+1;
        idtext[pieces,charindex]:=ch;
        hashkey:=...;
        {read next symbol}
      until {not identifier symbol};
      ....
    end;
```

It is easy to see that an identifier with more than 140 characters will cause a range error in idtext. It may be argued that in practice this error will never occur, but it was corrected all the same.

Identifiers were allowed to contain up to 140 characters, but a string was maximized to 80 characters. A string with more than 80 characters will put the PDP-11/45 Concurrent Pascal Compiler into an endless loop. Consider the following lines

```
const
  maxstringlength=80; {chars}
....
procedure stringchar;
begin
  if stringlength=maxstringlength then {errormessage}
  else begin
    stringlenth:=stringlength+1;
    {store the symbol and read next symbol}
  end
end;
....
procedure string;
begin
  ....
  while {not an special string char} do stringchar;
  ....
end;
```

The endless loop can be removed if stringchar is rewritten as

```
procedure stringchar;
begin
  if (stringlength<maxstringlength) and
     not {end of line} then begin
    stringlength:=stringlength+1;
    {store the symbol and get a new symbol}
  else {fatal error, abort pass 1}
end;
```

In the new version a string constant must be contained in one source line and a string error is considered to be so serious that the lexical analysis is aborted where the error was found.

A serious and difficult error was the following one. In the definition of Concurrent Pascal a variable of type char and an element of a string are equivalent. In the PDP-11/45 version, however, they are not implemented in the same way. A variable of type char occupies two bytes, the redundant byte (here called the high byte) of a character is supposed to have a zero value, but an element of a string occupies only one byte. There are two reasons for storing a simple variable of type char in two bytes. First, an LSI-11 or PDP-11 word is divided into a high byte and a low byte. Word addresses are always even-numbered. Byte addresses can be either even or odd-numbered. This rule complicates the allocation of variables, if variables with an odd number of bytes are allowed. Second, if the high byte is supposed to be zero, a variable of type char can be handled in the same way as an integer. This shortens the instruction list of the virtual machine.

The virtual machine is a stack machine and all manipulations

are done on the stack. A push of an element of a string gives the high byte of the stack the value zero and can then be manipulated in the same way as a simple variable of type char.

Now consider the following Concurrent Pascal Program

```
type stringtype=array[1..10] of char;
var string:stringtype;
    character:char;
procedure setchar(var character:char);
begin
  character:='a';
end;
begin
  setchar(character);
  setchar(string[5])
end.
```

In the first call {setchar(character)} two bytes should be assigned values, but in the second one {setchar(string[5])} only one byte should be assigned. This could not be handled properly in the original version. It produced erroneous code because the call setchar(string[5]) caused an assignment of two bytes. The problem was solved by letting a simple variable of type char still occupy two bytes but with the high byte undefined and handle it as an element of a string in the original version. After extension of the virtual instruction list with two new instructions (pushlocalbyte and pushglobalbyte) it was possible to correct the compiler.

The uninitialized variable was discovered during the work with the problem discussed above. In the very first compilation of the program above pass 5 of the Concurrent Pascal compiler gave an error message that the statement setchar(string[5]) contained a bad operand type. After inspection, it was found that pass 4 told pass 5 that the parameter of setchar was of universal type and since the length of a character was two bytes and an element of a string was only one byte, it is evident that pass 5 could not accept the call setchar(string[5]). It turned out that a boolean variable in pass 4 named universal was not initialized to false.

MOVING OF THE CODE INTERPRETER AND THE KERNEL

The Concurrent Pascal compiler generates code for a virtual machine. The code interpreter is the microprogram of this virtual machine. The kernel multiplexes the processor among concurrent processes and gives them exclusive access to monitors. It contains also the device handlers. The virtual machine, code interpreter and the kernel are described in Reference 1.

In the PDP-11/45 version the code interpreter and the kernel are written as one 4K word long assembly program. It is inconvenient to handle such a large program. It was therefore divided into three parts with a common prefix. The first part is the code interpreter, the second part is the kernel without operations associated with I/O and the third part handles I/O and contains the device handlers.

Fortunately, the PDP-11 family including LSI-11 has a common assembly language [6]. There are of course differences between the computers in the family and an instruction can be special to a version or it can have different effects on different versions.

The Code Interpreter

In the code interpreter it was only the virtual instructions associated with real numbers that had to be modified. In the PDP-11/45 computer a real number is eight bytes long, but in the LSI-11 computer it is only four bytes long. PDP-11/45 and LSI-11 with Extended Arithmetic Chip have different floating point instructions and comparisons between real numbers and conversions between an integer number and a real number are done in software in an LSI-11 computer. It was easy to modify the code interpreter so it could handle real numbers in an LSI-11 computer.

The two new virtual instructions (pushlocalbyte and pushglobalbyte) discussed above were implemented.

An operation table defines the entry points of the code piece that executes virtual instructions. The code generated by the compiler contains indices to the operation table. In the PDP-11/45 version the operation table begins at the virtual address zero. The LSI-11 computer has no virtual addressing mechanism and it is not possible to let the table start at address zero because these locations are reserved for other purposes. In the LSI-11 version the operation table starts at the octal address 1000. Pass 7 of the Concurrent Pascal Compiler was therefore modified to take this into account.

The Kernel

The kernel is a rather complex program and although the assembly code was commented in a language that resembles Concurrent Pascal it was hard to understand it in detail.

Code associated with the management of disks, magnetic tapes and line printers was removed. The debugging facility appeared to be so complicated that it was removed and new debugging facilities were implemented when necessary. Because the LSI-11 computer has no virtual addressing mechanism, the corresponding code was removed or rewritten.

The PDP-11/45 computer has two register sets and the kernel used one and the users the other. The LSI-11 computer has only one register set. The kernel was rewritten so that when the kernel is entered, the registers RØ, R1 and R2 are stacked and the stackpointer SP is stored in the bottom of the kernel stack. The kernel does not use the registers R3, R4 and R5. This makes a kernel call faster.

THE LSI-11 SYSTEM

In this section some of the restrictions and extensions of Concurrent Pascal for the LSI-11 computer are discussed. For further details see the Concurrent Pascal User's Guide [8].

The set of letters is extended with lower case letters, but they will be interpreted as their upper case counterparts. This is easy to implement, it makes it possible to write more readable programs and it is easy to write a Pascal program that converts lower case letters to their upper case counterparts if a program is going to be moved.

Four forms of comment brackets are allowed:

"...", {...}, (*...*) and /*...*/.

The opening and closing comment bracket must have the same form. In the PDP-11/45 version only "..." is allowed. Hartmann [4] states the following rule

> "Symbols must be used unambiguously to make the error recovery of the compiler efficient."

This rule implies that the opening and closing comment bracket should not be identical.

The boolean operator & can be written AND. This is in fact also allowed by the PDP-11/45 compiler, although it is not declared in the Concurrent Pascal Report.

The array brackets (. and .) can be written [and].

Process Scheduling and Real-Time Control

This implementation of Concurrent Pascal for the LSI-11 computer is mainly intended for process control applications. In these applications the computer is expected to control a plant and to communicate with an operator. Two major tasks can be recognized. The first should control the plant and has to be run periodically. The second task should communicate with the operator and the control task. In many cases the control task must not be delayed by the communication task. So in these cases the communication task has to wait until the control task has completed its execution, but this does not matter if the execution time of the control task is short.

In the PDP-11/45 version the computer in principle switches from one process to another every 17 ms to give the illusion that they are executed simultaneously. This means that it is not possible for the Concurrent Pascal programmer to achieve

the effect discussed above. In the LSI-11 version a priority is associated with every process. It is described by a nonnegative integer. A small priority number corresponds to a higher priority. All scheduling is done according to the priority. If two processes have the same priority, the first-come-first-served rule applies. The initial process will start with priority zero. The other processes will start with priority one. A process can change its priority during its execution by means of the standard procedure setpri(x). The priority of the calling process is set to x. In a process control system the processes should cooperate and not compete, so it is assumed that the programmer chooses proper priorities for the different processes to make the system work well. If negative priorities occur a run time error is generated. Changes of priorities are significant events and a process may be suspended if it lowers its priority.

It is desirable to get the kernel and consequently also the assembly program as short as possible and write as much as possible in Concurrent Pascal. Consequently, the standard routines defined in the LSI-11 version are simple constructions.

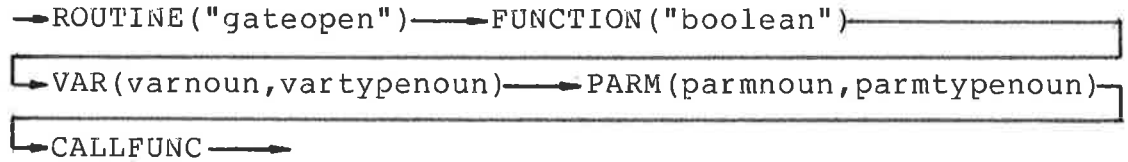
The standard function realtime returns an integer defining the real time in ticks mod 32768 after system initialization. The time between two ticks is 20 ms and 32768 ticks are 10 minutes and 55.36 seconds. A call of the standard procedure sleep(x) delays the calling process until the real time defined by the function realtime is x. Only one process at a time can sleep. If a process calls sleep when another process is sleeping, or if x<0, a runtime error occurs. A process can wake up a sleeping process by calling the standard procedure awake. If no process is sleeping the call is ignored. The standard routines realtime, sleep(x) and awake can be used to implement timetables and schedulers.

The standard function gateopen(x) applies to monitors. The result is a boolean value, true if the monitor is free, false otherwise. Note that the use of this function does not violate any of the strict access rules of Concurrent Pascal. The intention is that a high priority process may check the availability of a monitor before trying to enter it. If the monitor is not free, the process may choose some alternative action.

It was fairly easy to introduce new standard routines into the compiler, because the pattern of the already existing ones could be followed. However, gateopen(x) caused some problems. The parameter should be of any monitor type. A variable of system type is called a system component. It is either a process, a monitor or a class. In pass 5 where the compatibilities of operands are checked it is not possible to find out if a system component is a process, a monitor or

a class. So if the model was followed a process or a class would be accepted as a legal parameter. The compiler produced code that could be used by the code interpreter so that part caused no problems.

From the descriptions of the intermediate languages in Reference 4 it can be found that a legal call of gateopen is input to pass 4 as

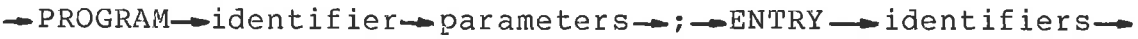


Varnoun and vartypenoun point to a description in the symbol table of the parameter used in the call of gateopen (parmnoun and parmtypenoun describe in the same way how the parameter is declared) and here it is possible to find out if the parameter is of monitor type. When routine("gateopen") is found a boolean variable called monitorcheck is set to true and this causes the compiler to check if the variable described by var(varnoun, vartypenoun) is of monitor type. This is not the most proper way to do it, but it was easy to implement. The system component type should of course be split up into three different types, but this is laborious to implement and it can easily cause new errors.

Sequential Programs

In the PDP-11/45 version a Concurrent Pascal program can execute a program written in Sequential Pascal and it is possible for the Sequential Pascal program to use routines defined in the Concurrent Pascal program. It was decided that in the LSI-11 version this should be done with programs compiled with the OMSI Pascal compiler.

The process that controls the execution of the sequential program must include a declaration of the sequential program



A program declaration consists of program identifier, a parameter list and a list of access rigths.

The access rigths of a program are specified by a list of identifiers of routines defined within the process in which the program is declared. The sequential program may call these routines during its execution.

The problem of communication between the Concurrent Pascal program and the OMSI Pascal programs is solved as follows.

When a sequential program is called, the absolute addresses of the routines specified in the ENTRY list are stacked with the first address on the top of the stack. So when the sequential program is entered the addresses to the entry routines are known. An interface procedure can use these addresses to make a proper jump. The interface procedure must know which of the procedures the user wants to call. Suppose that there are n explicit (defined below) entry routines in the ENTRY list. If the following declarations are done in the sequential program

```
type entryroutines=(entryl, entry2,..., entryn);
procedure entprocedure(entryroutine:entryroutines);
  external;
```

and if the i:th explicit entry routine is a procedure and declared as

```
procedure routinei({the same parameterlist as in the
                    declaration of the procedure in
                    the process});
begin
  entprocedure(entryi)
end;
```

then it is easy to write the assembly procedure entprocedure (six statements). Entprocedure is available in a support library.

It is natural but not necessary to give the OMSI Pascal procedure the same name as in the Concurrent Pascal process. The important issue is that entprocedure(entryi) will call the i:th explicit entryroutine, if entryi is the i:th element of the type entryroutines. Entry routines which are functions turned out to be impossible to implement efficiently and are not allowed.

The Concurrent Pascal compiler and the OMSI Pascal compiler handle constant parameters in diffent ways, therefore all parameters of entry routines must be variable parameters. It is desirable that the standard types are compatible. OMSI Pascal limits sets to 64 and therefore a set is in the LSI-11 version of Concurrent Pascal limited to 64 in contrast to the PDP-11/45 version where a set may contain up to 128 elements. OMSI Pascal stores a variable of type boolean or char in one byte. This problem is discussed above and the rules given there are extended to variables of type boolean. However, with this the problem is not completely solved. OMSI Pascal stores also variables of type boolean or char in high bytes. If a record contains two simple successive elements of type boolean or char this record will be stored in different ways. It is laborious to change the allocation rules of the Concurrent Pascal compiler. Therefore records with two simple successive elements of type boolean or char or records with a simple element of

these types at the end are not allowed as parameters of an entry procedure.

Unfortunately, it is not possible for the compilers to check that the procedures are declared in the same way in the two programs and that the rules given above are followed. The programmer must be careful and check it himself.

A concurrent program must ensure that a device is not used by more than one process at a time Consequently the Concurrent Pascal program must support the calls of the read and write procedures in the OMSI Pascal programs. The first two procedures in the ENTRY list are reserved for this purpose and they must always be defined in the Concurrent Pascal program. Since they are used implicitly by read and write, they must not be declared in the OMSI Pascal program. The rest of the routines in the list are called explicit routines.

The first implicit procedure should be a procedure which inputs a single character from the terminal. It will be used implicitly by read and readln in the sequential program. The procedure should have one variable parameter of type char.

The second one should be a procedure which outputs a single character to the terminal. It will be used implicitly by write and writeln, and also to print error messages from the seqeuntial program. The procedure should have one variable parameter of type char.

A sequential program must of course be loaded into the store before it can be executed. At present there is no disk handler available in Concurrent Pascal for the LSI-11 computer, so the RT-11 disk handler is used. The loading of sequential program is handled by the standard procedure

```
loadseq(seqprogspec,varspace)
```

In order to load sequential programs safely, all sequential programs should be loaded before any process is initialized. Seqprogspec should be a variable of type seqprogspectype defined as

```
type seqprogspectype=record
                       filename: array[1..14] of char;
                       stacktop,
                       heaptop,
                       startaddress,
                       loadaddress: integer
                     end;
```

The user must declare this type and the compiler can only check that the parameter has the correct length. The filename component of seqprogspec should contain the filename of a relocatable version of the sequential program.

Loadseq will give the rest of the components of seqprogspectype appropriate values. Varspace should be an integer and should specify the data space (in words) needed to execute the sequential program.

The parameter list of the declaration of the sequential program must contain exactly one variable parameter and no constant parameter, and the variable parameter must be of the type seqprogspectype (defined above). The parameter used at a call of a sequential program should contain the values received at loading. A sequential program is not reentrant, but it can be restarted without loading. If two processes want to execute the same sequential program, two copies must be loaded.

CONCLUSIONS

The implementation process described in this paper was pleasant and instructive. The implementation is a tool of significant value for teaching, research and engineering. It has been used with success in an undergraduate course.

The Concurrent Pascal Compiler is a large program consisting of 8100 lines of source code. It is divided into seven passes each 750-1600 lines long. It takes a full hour to create a runnable version of the Concurrent Pascal compiler on the LSI-11 with diskettes as mass memory. The compiler requires a code space of 11K words and a data space of 7K words. The mass memory requirements are 182 256-word blocks.

It takes 23 seconds to compile a program containing only begin end. Excluding these 23 seconds the compilation time varies between 5-8 lines/s again with diskettes as system units.

The code interpreter is 1.2K words long and the kernel is 2.0K words long including an I/O part of 0.4K words.

The total manpower to implement Concurrent Pascal for LSI-11 is estimated at four man-months.

ACKNOWLEDGEMENTS

I want to thank Professor Karl Johan Åström who proposed this project and Leif Andersson for his never ending interest, support and valuable suggestions during this project.

REFERENCES

1. P. Brinch Hansen,'The Architecture of Concurrent Programs', Prentice Hall, Inc., Englewood Cliffs, New Jersey 07632, 1978.
2. K. Jensen, N. Wirth,'Pascal-User and Manual Report', Springer Verlag, Berlin, 1975.
3. P. Brinch Hansen, A.C. Hartmann,'Sequential Pascal Report', Information Science, California Institute of Technology, 1975.
4. A.C. Hartmann,'A Concurrent Pascal Compiler for Minicomputers', Springer Verlag, Berlin, 1977.
5. P. Brinch Hansen,'Concurrent Pascal Implementation Notes', Information Science, California Institute of Technology, 1976.
6. RT-11 System Reference Manual, Order No. DEC-11-ORUGA-C-D,DN1,DN2, Digital Equipment Corporation Massachusetts, 1976.
7. OMSI PASCAL-1 Documentation Version 1.1, Oregon Minicomputer Software Inc., 2340 SW Canyon Road, Portland, Oregon 97201.
8. S.E. Mattsson,'Concurrent Pascal User's Guide', Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1979. CODEN: LUTFD2/(TFRT-7167)/1-009/(1979)