A VIRTUAL MACHINE FOR CONCURRENT PASCAL

ON THE TI 980B

© Tri Manh Pham

A Thesis in the

Department of Computer Science

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

June, 1981

ABSTRACT

A VIRTUAL MACHINE FOR CONCURRENT PASCAL

ON THE TI 980B

Tri Manh Pham

Concurrent Pascal is a programming language designed for the writing of portable operating systems, originally implemented on a DEC PDP-11/45. This thesis describes the transportation of Concurrent Pascal from the PDP-11/45 to the TI 980B. Some comments are made about the portability of this language.

ACKNOWLEDGEMENTS

It is my pleasure to gratefully acknowledge the advice and support of Professor J. W. Atwood during the preparation of this thesis.

I wish to thank Mr. Pierre Desjardins, Université de Montreal, for his availability and assistance in explaining the SOLO operating system.

I wish to thank Professor D. Thalmann of the Department of Computer Science, Université de Montréal, with whom I had fruitful discussions on virtual machines as tools for software portability.

I wish to thank Professor V. Wallentine and Mr. David Neal of the Department of Computer Science, Kansas State University, for having kindly sent me the invaluable technical reports on Concurrent Pascal.

I wish to thank Dr. I. Greenshields for his availability and precious advice in debugging concurrent programs.

I wish to thank all my friends who have helped me one way or another to get this work done.

This work was supported by a research assistantship from the Natural Sciences and Engineering Reasearch Council of Canada, through Dr. Atwood.

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

1.1 Systems implementation languages

Concurrent programming is intellectually intriguing and is essential in the design of operating systems. Operating systems have been written in three types of languages:

1- assembly languages

2- machine-oriented high level languages

3- high level languages

1.1.1 Assembly languages

Assembly languages allow maximum exploitation of the hardware and therefore are advocated for their run-time and memory efficiency. On the other hand, assembly languages are unsafe because of their complete freedom of accessibility to sensitive areas of the hardware. Writing operating systems in assembly code is very fastidious work. Bit-picking cleverness and machine-instruction intricacies tend to camouflage program logic and underlying algorithms. Operating systems written in assembly code are rarely well documented; modifications are hard to implement and often turn out as a source of errors. Improvements may therefore result in actual degradation. Furthermore, systems are not portable from one machine to another.

1.1.2 Machine-oriented high-level languages

While the need for space and time efficiency is a relevant criterion for operating systems, other criteria have increased in importance, largely because of the increasing cost of software development and maintenance:

- High programmer productivity
- Clarity of system structure
- Readability
- Control over programming style
- Ease of testing and debugging
- Portability

Machine-oriented high-level languages have been developed in response to the above-mentioned criteria. They have the following characteristics:

- they allow manipulation of bits and bytes
- they allow access to registers and memory addresses
- they allow driving of peripherals
- they handle interrupts and concurrent processes
- they contain high-level language constructs such as data and control structures

Machine-oriented high-level languages still contain pitfalls inherent to assembly languages. Due to the fact that they allow direct access to the hardware, they cannot be totally safe. Some of the well known machine oriented

high-level languages are: BLISS, PL360, BCPL.

1.1.2.1 BLISS

BLISS [1] was used to implement the HYDRA operating system at Carnegie-Mellon University. This language is designed so as to be especially suitable for use in writing production software systems for a specific machine (the PDP-10): compilers, operating systems, etc. Primary design goals are the ability to produce highly efficient object code and to allow access to all relevant hardware features of the host machine. BLISS may be characterized as an ALGOL-PL/I derivative.

1.1.2.2 PL360

PL360 [2] was created by N. Wirth at Stanford University for the IBM 360 family of computers. Primary design goals were to facilitate structuring, conciseness and clarity in operating system design without sacrificing efficiency. PL360 attempted to reflect the concurrent aspect of operating systems.

1.1.2.3 BCPL

BCPL [3] was originally developed as a compiler writing tool. It has proved to have many qualities which make it highly suitable for systems programming. BCPL has a simple underlying semantic structure which is built around an idealised object machine. This method of design was chosen

in order to make BCPL easy to define accurately and to facilitate machine independence which is one of the fundamental aims of the language.

Machine-oriented high-level languages do represent a significant improvement over assembly languages. The fact that most of them are not portable suggests that better high-level languages are needed.

1.1.3 Machine-independent high-level languages

There are two well-known languages in this category: Concurrent Pascal and Modula.

1.1.3.1 Concurrent Pascal

Concurrent Pascal [9] is the first programming language to present concurrent programming concepts as high-level language constructs. The concept of the abstract data type is used to structure the operating system. Its semantics require a virtual machine which includes a sizable amount of run-time support. BCPL already used the concept of a virtual machine as a tool for writing machine-independent programs. We shall demonstrate that the degree of portability depends on the architecture of the virtual machine.

1.1.3.1 Modula

Modula [4] is a language for multiprogramming invented by N. Wirth. It is largely based on Pascal, but in addition to conventional block structure it introduces a so called module structure. Modula includes general multiprocessing facilities, namely processes, interface modules, and signals. It also allows the specification of facilities that represent a computer's specific peripheral devices.

1.2 Software portability and virtual machines

The cost of developing large programs is so high and the time needed is so long, that it is desirable to use software systems on a number of different machines and/or to use the systems on newer hardware without modification. The fast evolution of mini and micro computers further emphasizes the need for portable software. The primary benefit of a high level language lies in the possibility of defining abstract machines in a manner independent of characteristics of particular hardware. The use of an abstract or virtual machine is a well known technique for writing software whose portability is very difficult to achieve: compilers, operating sytems and real time systems. The design of a virtual machine is therefore crucial in determining the portability of programs written for it.

1.2.1 Virtual machines

Abstraction has always been the most powerful tool enabling the scientist to conquer a complex system. The essence of abstraction consists in hiding irrelevant details so that the human mind can juggle with concepts instead of physical objects. Atomic physics, for example, typifies the use of abstraction to conquer the invisible world of atoms. Progress in computer science as well as in other sciences relies heavily on abstractions. Machine languages, assembly languages, and high-level languages represent different levels of abstraction of the same underlying hardware. A major design concept, closely related to abstraction is the virtual machine. To design a virtual machine amounts to designing an instruction set or primitive operations which describe the possible actions of a system. The machine is called "virtual" because it is simulated by software. Programs written in the same language can be made portable in several ways:

1- by having different compilers for different machines. This is only practical for the most widespread language.

2- by having a single compiler that can be modified to generate code for different machines.

3- by having a single computer that can be simulated efficiently on different machines. This computer is called a virtual computer or virtual machine.

We shall briefly present some well-known virtual machines as background and as illustrations of tools to achieve portability.

The following description of virtual machines is adapted from the article, "Abstract, fictitious, hypothetical, ideal, imaginary and virtual machines" by Thalmann [11].

1.2.1.1 The UNCOL machine

UNCOL [5] (UNiversal Computer Oriented Language) is a very early abstract machine. It was designed to be a universal machine so that any programming language would be translated into machine code of a target computer. UNCOL is the assembly language of the virtual machine UNCOL. Compilers generate code which will be interpreted by each target machine. The UNCOL strategy requires only M+N instead of M*N compilers. The UNCOL approach to portability has been adopted by IBM, DEC and other manufacturers. It was not popular because it did not define a universally accepted virtual machine.

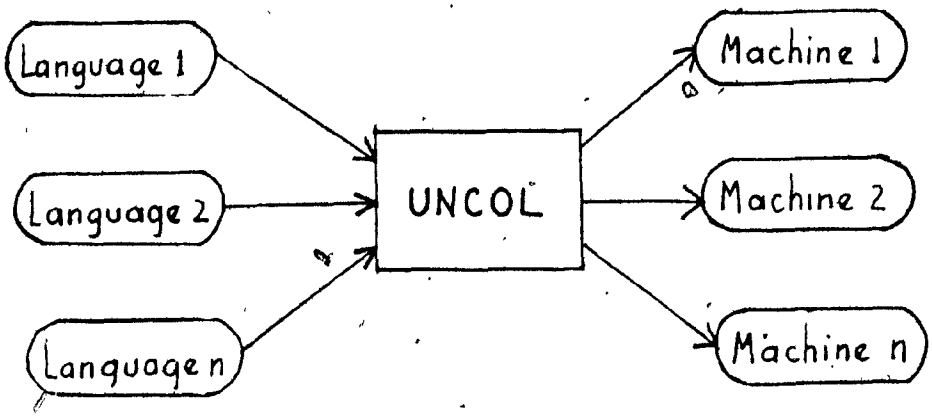

Figure 1.1 The UNCOL approach to portability

1.2.1.2 The JANUS machine

JANUS [6] is a family of abstract machines developed at the University of Colorado by Coleman, Poole and Waite. The goal was to study the problems of producing portable software and in particular portable compilers. JANUS is a symbolic code which is translated to the assembly language of the target computer by a macroprocessor such as Stage2. A method of bootstrapping Pascal using JANUS has been developed at the University of Colorado.

The design of JANUS is based upon the relationship between existing languages and existing hardware. The architecture of the JANUS family of abstract machines includes a processor, a stack, a memory, an accumulator and an index register. This form of model was chosen to simplify the generation of machine code because it favours target computers with a single arithmetic register, or with multiple arithmetic registers, registers/storage arithmetic and stack.

1.2.1.3 OCODE and INTCODE machine

OCODE [13] is the intermediate language used for the BCPL compiler, designed by Richards. The OCODE machine consists of a processor, two memory address registers (S and P) and a memory; it is essentially a stack computer with a rich instruction set. However, even though the OCODE mechanism provided a reasonable mechanism for portability, it was found that time could be saved by using a simpler machine-code INTCODE. The INTCODE machine is not fully specified, but it has 6 control registers and the instruction set includes only 8 machine functions. It allows the BCPL installer to construct a temporary interpretative implementation on the target machine in the minimum time. The method has been used many times and it works very well.

1.2.1.4 The PICA-B machine

The PICA-B machine [10] is an extension of the INTCODE machine which includes two new concepts: a status (interrupt) register I and a memory map of I/O devices. The I register has 3 fields: a bit which indicates whether the interrupt system is enabled or disabled, a second bit which indicates whether the interrupt and device vectors are accessible, and the address of a vector which contains the address of an interrupt routine and a new value for the I register. Three new instructions have been introduced, they

are all concerned with the I register. Device vectors have also been added. With these extensions, the PICA-B machine is an abstract machine tied to the BCPL language but specially designed for developing operating systems.

1.2.1.5 The MUSS machine

The University of Manchester's MUSS system [7] was designed by Morris, Frank and Theaker to provide a compatible environment on all the machines of the MU5 complex (MU5, ICL 1905e and PDP-11/40). It subsequently has been ported to the ICL 2900 and MEMBRAIN MB 7700 computers.

An abstract machine is used as the target machine for the operating system. This abstract machine has to be emulated on the real machines. All actions in the abstract machine are controlled by reading and writing special control registers. Protection is also governed by a control register which defines the current execution mode; two modes exist: user (slave) and executive (master). The interrupt system of the abstract machine interfaces directly with the operating system CPU manager. Idealised peripherals have also been introduced and they are controlled using registers.

1.2.1.6 The LAMDA machine

The L-machine has been designed by Thalmann [11] to produce machine-independent software for mini and micro computers. It is a very powerful abstract machine with n word-registers and n byte-registers. A memory can be addressed with index registers, by page or indirectly. The L-machine has an i/o system and an interrupt system. Two levels have been defined in the L-machine, they correspond to the classical master and slave modes. A symbolic language SPIP has been developed for the L-machine. A single user disk operating system has been written in SPIP. The L-machine has been related to 4 computers by using a code generator developed by a top-down methodology. Thus, the software development for the L-machine can run on the DGC NOVA, DEC PDP-11, INTERDATA4 and TI TMS 9900.

1.2.1.7 The Concurrent Pascal machine

Concurrent Pascal is an abstract programming language designed by Brinch Hansen [9] to write operating systems. The language extends PASCAL with three concepts: processes, monitors and classes. Three model programs (operating systems) were written in it by Brinch Hansen. The most known is a single-user operating system called SOLO [33]. Hartmann [24] wrote the Concurrent and Sequential Pascal compilers which generate code for an abstract machine.

The implementation of this abstract machine on a

PDP-11/45 consists of an interpreter which executes the abstract code, and a kernel (run-time support) which schedules the execution of concurrent processes. The abstract machine code is quite similar to the P-code [31]. There are about 110 instructions. A quarter of these instructions are used by Concurrent Pascal only, these are the specific instructions for the new concepts introduced.

1.3 Thesis structure

This thesis is structured as follows. Chapter two consists of a description of the language Concurrent Pascal. Chapter three contains detailed view of the CPASCAL virtual machine. Chapter four provides a detailed description of the transportation process. Chapter five suggests a direction for future research in operating systems portability.

CONCURRENT PASCAL AS AN IMPLEMENTATION LANGUAGE

Concurrent Pascal is an abstract programming language for computer operating systems. The language extends sequential Pascal [35] with new concepts for structured concurrent programming. In particular, Concurrent Pascal includes abstract data types to deal with the problems of data sharing (monitor), code sharing (class) and concurrent processing (process).

2.1 Abstract data types

The following description of the abstract data type concept is adapted from the article "Overview of the HYDRA operating system" by Wulf [32]. A central concept in structured programming is the concept of "abstraction". Several authors have noted the close relationship between many programming abstractions and the concept of "type" as it appears in programming languages [14]. Specifically the concept of a "class" in Simula '67 [15] seems especially well suited to expressing these abstractions. A class in Simula defines an abstract data type by specifying both an underlying storage structure and a set of operations which operate on it. Thus, for example, the abstract concept of a set of integers might be introduced into a language by a definition of the form

```
TYPE intset =
```

```
BEGIN
  VAR a:ARRAY[1:100] of INTEGER; n: INTEGER;
  OP union(u, v: intset) RETURNS(intset);
    BEGIN  ...  END;
  OP intersect(u, v: intset) RETURNS(intset);
    BEGIN  ...  END;
END;
```

Such a definition is intended to describe how any particular variable of type intset is to be represented and how operations on this type of variable are to be performed. Thus the declaration "VAR a: ARRAY[1:100] of INTEGER; n: INTEGER;" defines how storage is to be allocated for each variable of type intset. The operator definitions, e.g. that for "union", define how such variables are manipulated. An important property of such definitions is that all the representational information is localized and "hidden" [16] in the type definition; the only way to manipulate variables of a defined type is by invoking the operations defined in the type definition. After having made such definition, the programmer may write such things as declarations of variables of type intset and statements which operate on these sets, e.g.

```
VAR a, b, c: intset;
a := union(b, c);
```

This style of programming captures the notion of abstraction. It effectively separates the application of the abstract "primitives" from the details of their implementation. The programmer, working at a level where intsets are an appropriate medium of expression, need never concern himself with the details of how they are represented or manipulated. Conversely, the implementor of the realization of the type intset may freely alter that realization (to improve efficiency, for example) without concerning himself with the details of how it is used, as long as he preserves the functional properties of the operations.

Concurrent Pascal's concept of abstract data type is derived from Simula's class concept: the combination of a data structure and the code operating on this data structure defines an abstract data type. The abstract data types introduced by Concurrent Pascal are called system types. System types are types in the Pascal sense, that is they are templates from which variables can be defined and initialised. A variable of system type is called a system component. A system component implies that its code be reentrant because components of the same type share a single copy of the procedures associated with the data.

There are three system types: process, monitor and class. They form the building blocks from which concurrent

programs are constructed. Concurrency is expressed by processes. Mutual exclusion and synchronization of processes are provided by monitors. Code sharing is provided by classes. System components may interact with one another by means of access rights. This is the Concurrent Pascal structuring mechanism: to connect program components by access rights into hierarchical systems in

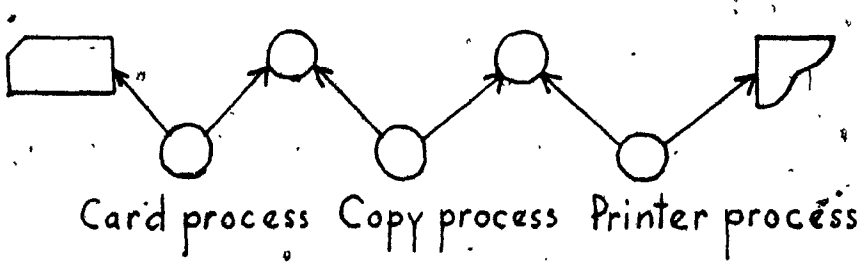

Figure 2.1 A pipeline system

which concurrent processes communicate by calling monitors. Figure 2.1 shows a system that reads cards and prints them on a line printer. The circles are called system components and the arrows, the access rights. The copy process can be decomposed into smaller system types.

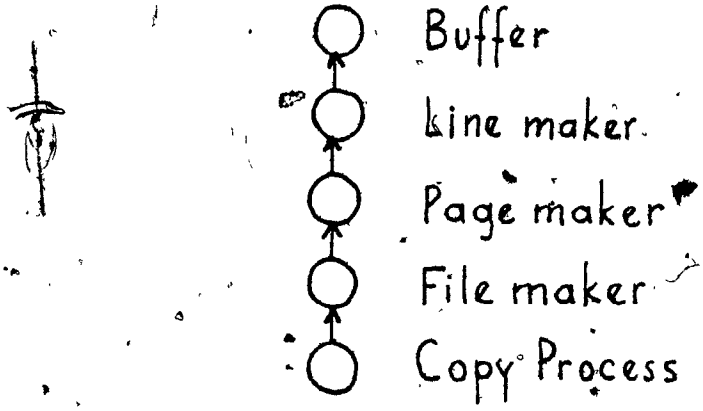

Figure 2.2 Decomposition of the copy process

In addition to access rights, the definition of system types may include definition of constants and non-system types, including global variables and procedures or functions.

2.1.1 Processes

A process is a system component that is executed as an asynchronous sequential program. It consists of three parts:

a- a sequential Pascal program

b- a private data structure (accessed only by the sequential program)

c- the access rights of this process to other system components.

A process cannot operate on the private data of another process. However, processes must be able to share data structures (such as buffers) and to cooperate on common tasks. The shared data on which a process can operate are determined by its access rights. See appendix 3 for more details.

2.1.2 Monitors

A monitor is a system component that controls communication and resource sharing among processes. It defines a shared data structure and a set of synchronizing procedures operating on it. A monitor also defines an initial operation that is executed when its data structure is created with an "init" statement.

The concept of a monitor was introduced by Dijkstra [34] and was further refined by both Brinch Hansen and Hoare [17]. In Concurrent Pascal, monitors are passive entities which are activated by a process which has access to that monitor. Like a process, a monitor can have parameters that define its access rights. It can also include definition of constants and types as well as procedures and functions that are accessible only within the monitor. Processes cannot operate directly on a monitor shared data. They can only call monitor procedures entries that have access to these shared data. During its execution, a monitor procedure entry has exclusive access to shared data. If concurrent processes simultaneously call monitor procedure entries operating on the same data, these procedure calls must be executed strictly one at a time. This implies that the run-time system must handle short-term scheduling of simultaneous calls (mutual exclusion). See appendix 3 for more details.

There are also provisions for monitor procedures to delay a calling process for any length of time by executing a "delay" operation in a queue variable if its request for a resource cannot be satisfied immediately. Concurrent Pascal introduces a simple data type called a "queue" that can be used by monitor procedures to control medium-term scheduling of processes. To prevent deadlocks of monitor calls and ensure that access rights are hierarchical, the following rules are enforced: a routine must be declared before it can be called; routine definitions cannot be nested and cannot call themselves; and a system type cannot call its own routine entries.

2.1.3 Classes

A class defines a data structure and the possible operations on it just like a monitor. But unlike a monitor,

1- simultaneous calls of class procedures at run-time do not exist. The exclusive access of a process to class variables is totally guaranteed at compile time; this means that checking need not be done by the run-time system to guarantee that only one process is in a class when entering it. This makes class calls considerably faster than monitor calls.

2- a variable within a class can be declared as entry variable, which means that it can be accessed by other class procedures.

See appendix 3 for more details.

2.2 Conclusion

Concurrent Pascal provides for high level data and control structures, the definition of a hierarchy of system components, and the elimination of machine dependent features (registers, addresses, interrupts etc.). Program modularity is supported by the concept of processes (which combine modularity and concurrent execution) and monitors (which combine modularity and synchronized execution). A large class of time-dependent programming errors are eliminated at compile time by checking that processes do not refer directly to the same variable. Within monitors, another class of synchronization errors is automatically eliminated by a mutual exclusion of monitor calls enforced during program execution. In addition, the compiler detects recursive monitor calls to prevent a large class of deadlocks.

THE CONCURRENT PASCAL VIRTUAL MACHINE

3.0 The Structure of Concurrent Pascal Programs

Concurrent Pascal was designed and implemented (on a PDP-11/45) by Per Brinch Hansen [33]. It sets out to prove that it is possible to write a well-structured operating system as one single program. Almost all machine-dependent aspects of the sytem are isolated in the virtual machine so that the parts of the system written in high-level language should be highly machine-independent.

A Concurreht Pascal program consists of two distinct logical entities:

1- The common segment which groups together the code shared by all processes: the interpreter, the concurrent code (compiler output of a Concurrent Pascal program), and the data segment of the initial process.

2- The n private (data) segments of the n child processes. Each data segment includes a run-time stack and a heap. Sequential code (compiler output of a Sequential Pascal program) is dynamically loaded into an area allocated in the stack. It is important to note that the concurrent code, which is shared by all processes, needs no protection from process interaction since this protection is guaranteed

by the scope rules of Concurrent Pascal. To permit the sharing of the code itself, the code is re-entrant.

The form of a Concurrent Pascal program is shown in

```
Anonymous process
-----------------------------------------------------------
| Constant declarations (real, integer, string ..)        |
|      .                                                  |
|      .                                                  |
|      .                                                  |
| Type declarations (monitor, class, process ..)          |
|      .                                                  |
|      .                                                  |
|      .                                                  |
| Variable declarations (* constructs system       *)     |
|                       (* components by defining  *)     |
|      .              (* instances of system types *)     |
|      .                                                  |
| Routines      (* code of all internal procedures *)     |
|      .                                                  |
|      .                                                  |
|  Code of the anonymous process  (* initializes *)       |
|      .                          (* all system components *)
|      .                          (* defines access rights *)
|      .                        (* among system components *)
-----------------------------------------------------------
```

Figure 3.1 Skeleton of a Concurrent Pascal program

figure 3.1. It consists of 5 sections: constants, types, variables, routines and the code of the initial process. The initial process declares all instances of monitors, classes and child processes, and then starts all child processes.

Concurrent Pascal runs on a virtual machine. This machine is simulated by two distinct programs: the interpreter and the kernel.

The interpreter is an assembly language program (although it could be microprogrammed) which executes virtual code generated by both the Concurrent and Sequential Pascal compilers. It invokes the kernel when a process needs to enter a monitor (to access shared data), to perform I/O, to create another process etc. All input parameters to the kernel are taken from each process stack (private segment) and copied into the kernel/interpreter interface (communication) area called the Process Head.

The kernel is another assembly language program. It multiplexes the processor among the concurrent processes and provides mutual exclusion of monitor calls. The kernel also allocates space for process records, monitors and peripherals.

A picture of central memory will show, in ascending order: the kernel, the interpreter, the concurrent code of the Concurrent Pascal program (e.g. SOLO), the data areas (D0, D1, .., Dn) for the initial process (P0) and the "user" (child) processes (P1, P2, ... , Pn).

If a process within a Concurrent Pascal program wishes to execute a Sequential program, it must load that program (normally from disk) into an area which is allocated in the process data segment and then transfer control to the code just loaded. The sequential program interfaces with the

operating system (concurrent code) via a list of procedure names, called the PREFIX. See figure 3.2. The bodies of these procedures are defined within the process. By PREFIXing sequential programs with different interface lists, a single concurrent process may provide different user programs with varying degrees of access to system resources. Section 3.4 will provide more insight into the execution of a Sequential Pascal program.

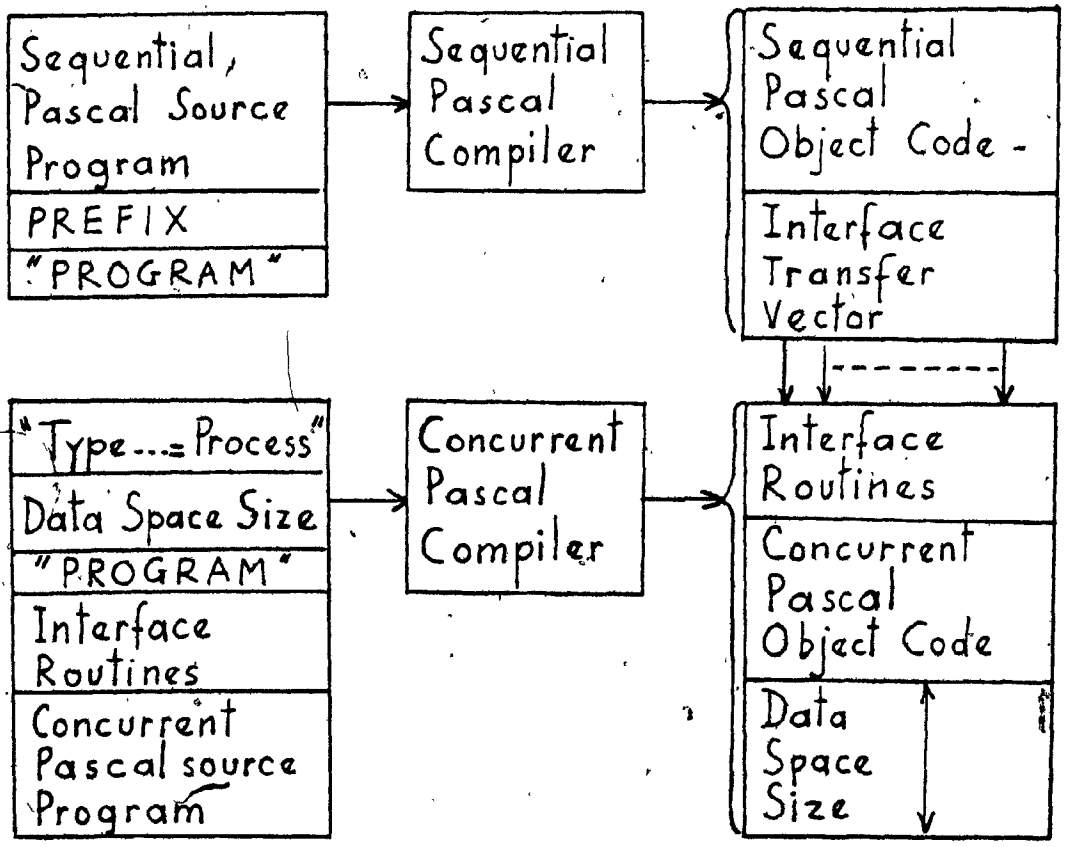

Figure 3.2 Concurrent and Sequential Pascal interface

3.1 Architecture of the virtual machine

The concept of a virtual machine has been used as a standard technique to build portable compilers for sequential languages. The virtual machines that support the intermediate code generated by these types of compilers consist only of an interpreter, mainly because the languages they compile do not incorporate operating system concepts.

Concurrent Pascal's primary objective is to permit the writing of operating systems. To make the language portable, Brinch Hansen had to design an intermediate language which is the "assembly language" of the virtual machine. This virtual machine must support not only normal Sequential Pascal concepts but also Concurrent Pascal concepts. This necessitates an interpreter that executes the virtual code and a kernel that schedules the execution of processes and supports I/O,and concurrency.

The Concurrent Pascal machine is essentially a stack machine. This stack machine is simulated on a PDP-11/45. Unfortunately, certain peculiarities of the PDP-11 architecture pervade the code assembly phase of the compiler. The resultant virtual machine takes on many PDP-11/45 features, and may be termed a "hybrid" between a true stack machine and a PDP-11.

3.2 The PDP-11/45 architecture

Before giving further details of the Concurrent Pascal virtual machine design, it is necessary to present certain aspects of the PDP-11/45 architecture [19]. We will discuss only aspects of the hardware architecture relevant to the Concurrent Pascal virtual machine.

3.2.1 The Central Processing Unit

The CPU executes in one of three modes: kernel, supervisor or user. Each mode includes:

a- a stack register (R6)

b- a set of memory mapping registers

c- a level of priority (0 to 7).

Only kernel mode can alter the level of priority through the SPL instruction. The instructions HALT and RESET are also reserved for kernel mode. The program status word (PSW) contains information on the current status of the CPU, including among other things:

a- the register set (0 or 1)

b- the current mode

c- the previous mode

While kernel mode is the most powerful, and is equivalent to the typical third generation privileged mode, the other two modes are not strictly hierarchically nested.

3.2.2 The registers

The PDP-11/45 is a true general register processor in that any of the registers R0 to R7 can be used for a variety of purposes. The general registers can be used as accumulators, index registers, autoincrement or autodecrement registers, or stack pointers. There are several addressing modes, including register relative, indexed, indirect, autoincrement and autodecrement. When an interrupt or trap occurs, the PSW and the PC are automatically saved on the kernel stack. The kernel mode uses general register set 0, the other two modes use general register set 1.

3.2.3 Memory organization

The PDP-11 memory and instructions are designed to handle both 16-bit words and 8-bit bytes. A word is divided into a high-byte and a low-byte as shown in figure 3.3.

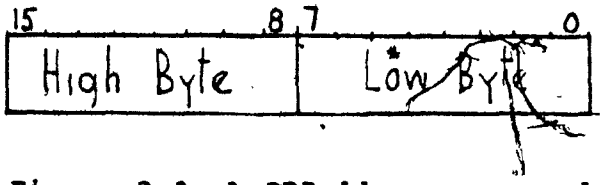

Figure 3.3 A PDP-11 memory word

Bits and bytes are numbered from right to left. Since memory is byte-addressable, consecutive words are found in even addresses. The lowest addresses are reserved for trap and interrupt handling. The highest addresses are reserved for peripheral device registers. A 16-bit word used for

byte addressing can address a maximum of 32K words. However, since the top 4096 word locations are reserved for special registers, the user must use the memory management unit if he wants to expand above 28K words.

3.2.4 Memory management

The PDP-11/45 memory management unit provides the hardware facilities necessary for memory mapping and protection. A simplified view of the address translation process (sufficient to understand its use in the Concurrent Pascal system) is as follows: the 64K byte address space is divided into 8 pages of 8K bytes each. Associated with each page is a page address register (PAR) which defines the starting address in real memory of that page (this starting address must be a multiple of 64 bytes or 32 words), and the length of the page. The top three bits of a virtual (16-bit) address are used as an index into the array of PARs, to determine the bias to be added to the in-page displacement to obtain the real (18-bit) address.

3.2.5 I/O architecture

The PDP-11/45 can perform I/O without the use of explicit I/O instructions, because of the Unibus structure. The PDP-11 has no channels in the usual sense but it does have slow (one word at a time) and fast (one block at a time, via DMAC) devices, each of which interface directly to the Unibus. Instead, each I/O device has one or more status

registers that control its behaviour, plus one buffer register through which all information passes on its way into or out of the device. These special registers are directly addressable by the CPU. To perform an I/O operation, device driver software typically moves start addresses, byte counts and status information to the registers as if they were normal memory. Then, that software initiates the I/O operation by setting the start bit in a status register.

Interrupt handling on the PDP-11/45 is quite simple. No device polling is required to determine which device and therefore which service routine to execute. Every hardware device capable of interrupting the processor has a unique set of two locations reserved for its interrupt vector, which contains the location of the device service routine and the new PSW. This new PSW contains the new processor priority and other information. The interrupt service routines run at processor priority level 7, which provides an effective interrupt mask.

3.3 Concurrent Pascal and the PDP-11/45

3.3.1 Use of the CPU

Concurrent Pascal uses only two of the CPU modes: kernel and user. The kernel (run-time support) executes in kernel mode. The interpreter executes in user mode. The PSW used in kernel mode is defined as follows:

```
KNLPSW:= (current mode is kernel)
         (register set is 0)
         (processor priority is 7)
         (virtual 0 corresponds to real 0)
```

In kernel mode, the CPU operates with the highest priority (7) and accordingly inhibits all interrupts. The PSW used in user mode is defined as follows:

```
USRPSW:= (current mode is user)
         (register set is 1)
         (processor priority is 0)
         (virtual 0 is beginning of interpreter)
```

In user mode, the CPU operates with the lowest priority and all interrupts are enabled.

3.3.2 Use of memory management unit

The virtual address space used in kernel mode is 32K words and includes the following: (see figure 3.4)

1- the interrupt vectors and trap locations

2- the kernel and the interpreter.

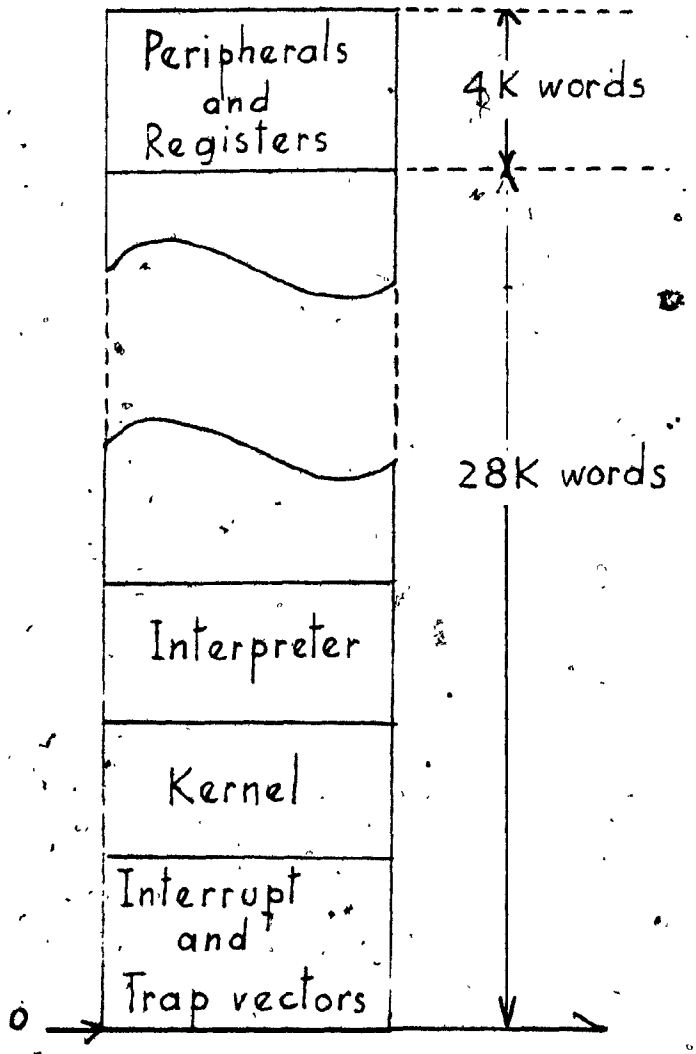

Figure 3.4 Virtual address space in kernel mode

The virtual address space in user mode is more complicated. The address space of each process is divided into two logical segments: a common segment and a private segment (data segment). The common segment is composed of:

1- the interpreter

2- the concurrent code for all processes

3- the data segment of the parent (initial) process

The private segment includes:

1- the process run-time stack

2- the process sequential program heap.

As mentioned before, if a Concurrent Pascal program instantiates n processes, there will be n private segments and one common segment. At any given time, only one process is executing. The addressable memory includes the common segment and the private segment associated with the running process. Associated with each process (in the Process Control Block) is an array MAP

MAP: ARRAY [0..7] of INTEGER

which contains information pertaining to the virtual space allocated when a process was created by its parent process. Each element of the array corresponds to a PAR. process n

3.3.3 The Interpreter on the PDP-11/45

The interpreter is an assembly language program. It consists of :

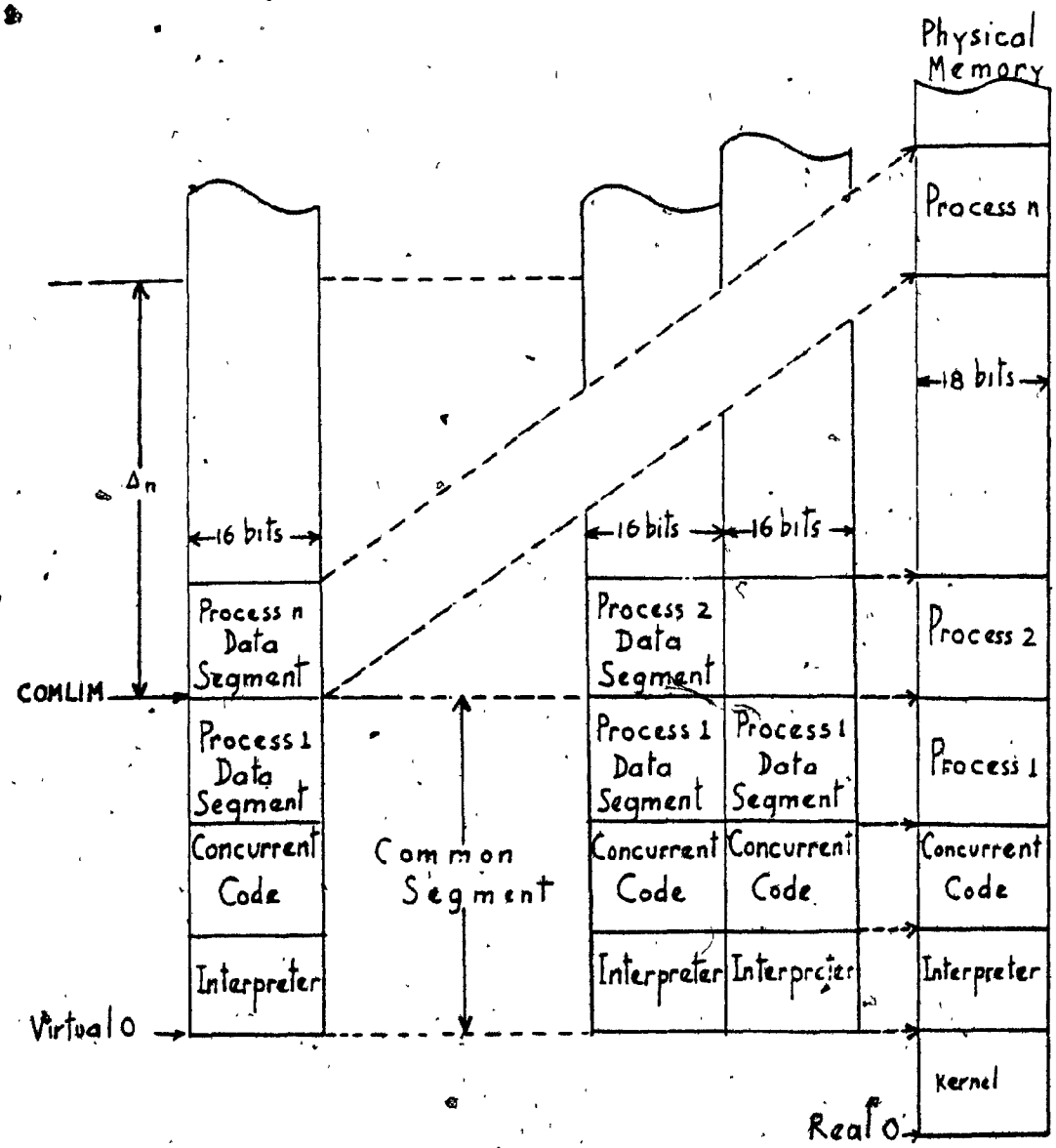

Figure 3.5 Virtual address space for process n

1- a jump table

2- a process head

3- interpreter routines

3.3.3.1 The jump table

The jump table is located at virtual address 0 of user space. As its name implies, it contains entry point addresses of interpreter routines.

3.3.3.2 The Process Head

Process management and physical I/O in the Concurrent Pascal system are implemented by a Kernel, which is invoked by the interpreter when these operations are required. This necessitates that the interpreter and the kernel share a common data structure called the process head. The process head is a record defining the attributes of the running process: run-time, priority, I/O parameters, etc. The process head is saved in the Process Control Block inside the kernel when a process is preempted and restored when the process is resumed. The process head is used to communicate operation codes and arguments to the kernel.

3.3.3.3 The Interpreter routines

A virtual instruction consists of an op-code, possibly followed by some arguments. The op-code and its arguments are integers which occupy one word each. The arguments are always constants which are known at compile time, so the virtual code and the arguments are always read-only. Operands which are only available at run-time are taken from the stack and all results are returned to the stack. There are about 50 different virtual instructions. To make the software interpreter fast (as opposed to microcode) the addressing modes (local or global) and the data types (bytes, word, reals or sets) are encoded into the operation codes. This expands the set of opcodes to 110.

The operation codes include arithmetic and logical operations on word, real and set data types; relational operations on word, real, set and structured data types; operations perfoming address manipulation, index verification and initialization of variables; stack manipulation operations; control transfer operations and operations related to monitors and classes. The specific operations provided by the interpreter are listed in appendix 2. The virtual code of a compiled Concurrent Pascal program constitutes the operating system of a Pascal machine. Virtual code which represents a compiled sequential program may be fetched and run as a procedure by a concurrent process.

The interpreter maintains two important data structures to execute the virtual code of a process: an execution stack and a program heap. Each instance of a process contains these two data structures. The stack is used for both Concurrent and Sequential Pascal program execution, while the heap is used only by Sequential Pascal for the allocation of dynamic variables. The stack and heap grow toward one another from opposite ends of the private segment. The stack provides storage for four basic purposes

1- local and global variable space for the activation of a procedure, sequential program or Concurrent Pascal process.
2- dynamic link for procedure return
3- parameter passing
4- expression evaluation temporary values.

To administer the stack, the heap and the virtual code, the interpreter needs nine registers: W, X, Y, P, Q, S, B, G, and H. Registers W, X, Y are scratch registers. The interpreter uses the real program counter (P) to execute its own code. A virtual program counter Q is used to fetch instructions. The heaptop H defines the current extent of the heap. The contents of the stack are addressed relative to three register bases: a global base register G, a local base register B, and a stack top register S. Temporary variables and parameters of a procedure are addressed relative to B. Permanent variables are addressed relative

to G. Expression evaluation temporaries are addressed relative to S.

The dynamic link includes the stack addresses G, B and S used by a process before a procedure call and a return address Q (old virtual program counter). The link also contains the current line number within the procedure to facilitate the location of run-time errors.

When a process is created, its global and local base registers both point to the permanent variables of that process. It is initialised with no temporaries and an empty heap. When a process calls one of its own procedures the local base register will point to the temporary variables of that procedure while its global base register remains unchanged.

When a process calls a monitor or class procedure, the G pointer will point to the variables of that monitor or class and the local base B will point to the temporary variables of the monitor (or class) procedure. Upon return from a procedure, the temporary variables are popped from the stack and the previous values of the base register are restored by means of the dynamic link.

The base address of a data segment divides the variables from the parameters. Component data segments have

their addresses shifted by the variable length, or component offset, in order to point to the base location. The base location contains either the line number at the point of call for routines or it contains the component index for system components. The parameter portion may contain more than just the explicit parameters. This allows the routine entry to address global component variables. A call of a sequential program places a list of interface routine addresses on the stack before the explicit parameters. After the explicit parameters, the address of the sequential code store is placed on the stack [24].

3.3.3.4 Threaded-code implementation

The notion of an interpreter can best be described as a means of recognizing alternative actions and then performing the processing required for the selected actions. The "steering" logic in Brinch Hansen's interpreter uses a technique called "threaded code" [20]. The address of the next virtual instruction is contained in the virtual program counter (register Q). The interpreter uses the instruction code (16-bit integer) as an index into a jump table located at virtual address zero, which is the starting address of the interpreter. The jump table contains starting addresses of routines that perform the desired operations.

Every routine in the interpreter ends with the instruction NEXT, whose effect is:

goto ST(ST(Q)); Q:=Q+2;

where ST(Q) means the contents of the stack locations pointed to by Q. On the PDP-11/45 this instruction is translated by:

MOV @(Q)+,PC

i.e., use the contents of the location pointed at by Q, indirectly to obtain an address; copy this address into the program counter; increment register Q by 2. The auto-increment addressing mode automatically increases the designated register by 2 bytes. This instruction takes only 3 memory cycles. The correct operation of the MOV instruction is dependent on the jump table being located at virtual address zero so that the operation code can be used without bias as an indirect address.

3.3.3.5 Interpreter tracing

Before fetching the next virtual operation code, a conditional assembler toggle permits switching on or off a line printer and console trace of each interpreter routine. This is implemented by executing a kernel call which transfers control to a debugging package resident in the kernel. The information displayed includes: all interpreter registers, the contents of the stack, and the value of the stack pointer before and after execution.

3.3.4 The kernel on the PDP-11/45

The following description of the kernel is adapted from the book "The Architecture of Concurrent Programs" by P. Brinch Hansen. The kernel was first written in a programming language that resembles Concurrent Pascal. It consists of a collection of data structures representing processes, monitors, and peripherals. Each data structure consists of two parts: one defines how the data are represented in store, the other what operations one can perform on these data. The abstract version of the kernel was translated by hand into assembly language (retaining the abstract version as comments). The kernel provides three main functions to the virtual machine:

1- Processor multiplexing

2- Peripheral activation

3- Monitor implementation

3.3.4.1 Processor multiplexing

The computer executes one process at a time. While one process is running, other processes must await their turn in one of three ready queues. Every 17 msec the computer switches from one process to another, to give the illusion that they are executed simultaneously. A process is represented by a data structure within the kernel called a Process Control Block (PCB). When a process is preempted, all registers used to interpret its code are stored in its PCB. The register values are restored when the execution of

the process is resumed. A process queue is represented by a sequence of references to the PCBs. The only operations on a process queue are:

put : enters a process in the queue

get : removes a process from the queue

any : tells whether the queue contains anything

empty : tells whether the queue is empty

The running process is represented by a class. It contains a permanent variable: USER99 is the reference to the running process; it is nil when the processor is idle. Only two operations are defined on the running process: serve and preempt. They start and stop the execution of a process. The ready queues are

top : processes executing monitor code

middle : processes to be resumed after I/O

bottom : compute-bound processes

Two operations can be performed on the ready queues: enter a process in the queue, and select one to be served. An attempt to select a process from an empty queue causes the processor to idle until a peripheral operation has been completed and has entered a process in the ready queue. A clock interrupt has no effect if the processor is idle; otherwise, it preempts the running process, enters it in the ready queue, and selects another process for execution. The clock will only preempt a process when it has used a

reasonable amount of time, and it will never interrupt a process inside a monitor procedure (since this could cause the resource controlled by the monitor to remain idle until the execution of the procedure is completed). The class "running" also contains procedures for process creation.

3.3.4.2 Monitor implementation

Within the kernel, a monitor variable is represented by a data structure called a gate, which only gives one process at a time access to the monitor. A gate is represented by a boolean variable defining whether it is open, and a queue of processes waiting to enter it. At the beginning and at the end of a monitor procedure, a process executes an "enter" and a "leave" operation respectively. (More precisely, kernel routines are called by the interpreter when it executes the virtual instructions "entermonitor" and "exitmonitor"). Enter: if the gate is open, the process enters and closes it; otherwise, the process is preempted to wait outside the gate. Leave: if nobody is waiting outside the gate, it is left open; otherwise, a single waiting process is resumed (by transferring it to the ready queue). These are the short term operations which force processes to enter a monitor one at a time. A monitor can also delay its invoking processes for longer periods of time and resume them again by means of "delay" and "continue" operations on single-process queues. Delay: preempts the running process and enters it in a given single-process queue. The monitor

can now be entered by another process. Continue: forces the running process to leave the monitor and resumes any process that may be waiting in a given single-process queue. It is important to distinguish between a multiprocess queue which the virtual machine automatically associates with a monitor, and a single-process queue which the programmer declares within a monitor. The former is stored within the kernel while the latter is stored in the common segment. When a monitor variable is initialized the kernel executes a procedure that allocates its gate in the kernel heap and initializes it.

The "gate reference" is stored in the stack of the calling process and passed as a parameter to the kernel each time one of the gate operations is executed.

3.3.4.3 Peripheral activation

All I/O operations follow the same pattern. The I/O operation code is copied into the kernel together with the virtual address of the status word to be returned later. The virtual address of the data buffer is converted to a real physical address and transmitted to the device along with additional device-dependent parameters. This necessitates a separate routine to convert the virtual address of the data buffer so that the device registers can use it. The MTPD (Move To Previous Data Space) instruction can only store one word at a time and is used to fetch the

command and to return the status at the end of the I/O operation. A peripheral is represented by a class within the kernel. It defines the "device number" of the peripheral and its current "user" process. An "io" statement in Concurrent Pascal is translated into a call of a kernel procedure that starts a data transfer and preempts the calling process. An interrupt resumes the user process. Only one process at a time can use a peripheral. This must be guaranteed by the operating system written in Concurrent Pascal (and not by the kernel). The main function of the kernel is to make peripherals look uniform with respect to simple input/output operations and their results.

3.4 Execution of a Sequential Pascal program

The following description of a sequential program is adapted from Wallentine's technical report on Concurrent Pascal [29]. A compiled sequential program must have its virtual code loaded into a variable which is declared within the process which is going to execute it. A sequential program must be declared using the keyword PROGRAM

```
PROGRAM EXEC (VAR PARAM: ARGLIST ; STORE: PROGSTORE);
```

The virtual code representing the program to be executed is passed as the second argument STORE. The first argument is actually a list of other parameters. The first of these indicates whether the program terminates successfully. The remaining parameters can be used to pass explicit arguments to the program. Interface procedures are used to allow a

sequential program to access system resources and to perform I/O. For example in figure 3.6, the program EXEC is declared to have access to the procedures ACCEPT and DISPLAY. These two procedures are defined within the process which has the loaded the sequential program and are constructed from calls to system routines which perform character I/O to a terminal. By declaring several programs with different interface procedures, a Concurrent Pascal process may provide different programs with varying degrees of access to system resources. A sequential program must be PREFIXed with the list of all the interface routines which are available for its use. This "prefix" must list the routines in an order identical to the interface procedure list following the program's declaration within the Concurrent Pascal process. For example, a program which is to run under the job process in figure 3.6 should have the following job prefix :

```
Procedure ACCEPT (var C: char)

Procedure DISPLAY (C:char)

Program EXAMPLE

...

user program
```

The declaration JOB=PROCESS; +10000 reserves data space for the job process's heap. This heap can be used by Sequential program executed by the Concurrent Pascal process to allocate data structures through use of procedure NEW.

```
TYPE page = ARRAY[1..15] OF CHAR;
     code = ARRAY[1..N] OF page;
     job = PROCESS; +1000;
VAR  i : INTEGER; parm : idparam;
     store : code; plist : arglist;
     .
     .
     .
PROGRAM EXEC (VAR paramlist:arglist;
                seqcode : code);
  ENTRY accept,display;
  (* beginning of interface procedures *)
  PROCEDURE ENTRY accept (VAR c : CHAR);
     BEGIN
       ...
     END;
  PROCEDURE ENTRY display (c : CHAR):
     BEGIN
       ...
     END;
  (* end of interface procedures *)

BEGIN
  ...
    FOR i := 1 TO n DO
    IO (store[i], parm, device);
    EXEC (plist, store);
         ...
         ...
END
```

Figure 3.6 A simple Job process skeleton

3.5 Initiating the system

The PDP-11/45 version of the Concurrent Pascal system is designed to run on a bare PDP-11 and to use the entire system disk formatted to its own conventions. The system disk contains 4800 sectors of 256 words/sector. The sectors are numered 0 to 4799. The beginning of the disk contains 4 contiguous segments, followed by the disk catalog and files.

Sectors	K words	Name
0..23	6	kernel segment
24..87	16	SOLO segment
88..151	16	other OS segment
152..153	0.5	free page list
154		catalog page map
155..4799		catalog pages and files

To load the system, a human operator must perform a standard initial program load from disk drive 0 which loads the code of the kernel and the interpreter into memory starting at physical address 0. Control is passed to procedure LOADSYSTEMPROGRAM in the kernel which loads the entire system program from disk sector 24 using the length information given in the first three words of the virtual code. Once the virtual code has been loaded, control is given to a procedure which will initialize all kernel classes, and then to the the dispatcher which starts

executing the interpreter.

THE CONCURRENT PASCAL VIRTUAL MACHINE

ON THE TI 980B

4.0 The transportation process

To move the Concurrent Pascal system to the TI 980B, one proceeds as follows:

1- The kernel and the interpreter logic, i.e., the virtual machine, have to be recoded in TI assembly language with the help of the PASCAL-like comments of their PDP-11 version.

2- The correctness of the virtual machine has to be established using the test programs provided

3- The TI disk has to be loaded with an image of the system disk used by SOLO (a single-user operating system written in Concurrent Pascal). SOLO provides an environment in which the Sequential and Concurrent Pascal compilers will run, which can then be used to develop other Concurrent Pascal programs.

To assist in the transportation, a utility program called PREFACE was written. This program assists in the testing process, and in the initial loading of the Concurrent Pascal system.

4.1 The TI 980B architecture

4.1.1 The Central Processing Unit

The CPU executes in one of two modes: privileged and user. Some operations can only be used in privileged mode:

- disable or enable interrupts
- idle (halt) the CPU
- perform I/O
- pass from one CPU mode to the other

The processor status is contained in the status register.

4.1.2 The registers

Eight 16-bit internal registers are directly addressed via the instructions involving registers. Unlike their PDP-11 counterparts, TI registers are more specialized in their functions and there is only one set of registers inside the CPU.

Register address	Name	Function
0	A	Primary arithmetic register
1	E	Extended arithmetic register
2	X	Index register
3	M	Maintenance register
4	S	Storage register
5	L	Subroutine link register

6	B	Base register to point to address of operands
7	PC	Program counter

The TI addressing modes are: program counter relative, indexed, immediate, extended, and base register relative. The base register relative mode allows access to an operand address too far for program counter relative. The status register permits, among other things:

- I/O bus and DMAC interrupt control
- setting different address spaces

4.1.3 Memory organization

There are two external memory bounds registers which allow the upper and lower-most portions of memory to be protected. They are called LLR (Lower Limit Register) and ULR (Upper Limit Register). The TI 980B memory and instructions are designed to handle only words. (Some byte operations are available, but their semantics are so different from that of the PDP-11 that it was impossible to use them.) Unlike the PDP-11, bits are numbered from left to right. There is no virtual memory in the usual sense, but there are separate addressing spaces as defined by the LLR and ULR.

4.1.4 Memory management

There is no memory management on the TI 980B. For the purpose of implementing Concurrent Pascal, we only need the LLR. The LLR divides physical memory into two distinct address spaces corresponding to the 2 CPU modes:

1- The privileged address space which covers the entire memory (one-to-one mapping with physical memory)

2- The user address space which runs from the lower limit bound to the top of memory. Its address zero corresponds to the content of the LLR plus one, in real absolute address.

Before the privileged mode can be set, it is necessary to load the LLR with its memory address which physically separates the kernel from the Interpreter and the virtual code.

4.1.5 I/O structure

Slow devices (card reader, line printer, console) are attached to the CPU-controlled data bus. Fast devices (disk, tape) are connected to the Direct Memory Access Channel (DMAC). Special I/O instructions and registers are designed to perform I/O on the data bus and on the DMAC. To start I/O on the data bus, the instructions WDS and RDS are used in conjunction with external and internal registers. Initiating a DMAC transfer requires the instruction ATI, coupled with a list of memory words. Interrupt handling on

the TI 980B is more cumbersome than on the PDP-11/45; the programmer has to examine an interrupt status word to determine the interrupting device. This gives him complete control over device priorities. There are three types of interrupts on the TI 980B. These interrupts, in order of priority, are as follows:

1- internal interrupts (supervisor calls, power failure, ...)

2- DMAC interrupt (disk devices)

3- data bus interrupt (tty, card reader, line printer, ...)

The priority interrupt option on the TI 980B is not available in our installation.

4.2 Concurrent Pascal and the TI 980B

4.2.1 Use of the CPU

The kernel (run-time support) executes in privileged mode. The interpreter executes in user mode. The status register used in privileged mode is defined as follows:

KNLPSW:= (privileged mode is enabled)
(lower limit bias is disabled)
(data bus interrupt is disabled)
(DMAC interrupt is disabled)

The status register used in user mode is defined as follows:

USRPSW:= (privileged mode is disabled)
(lower limit bias is enabled)
(data bus interrupt is enabled)
(DMAC interrupt is enabled)

4.2.2 Use of the memory with LLR

The address space used in privileged mode includes:

1- The interrupt and trap vectors

2- The remainder of memory

The address space used in user mode includes:

1- The interpreter

2- The concurrent code

3- The private segments of all processes

See figure 4.1.

4.2.3 Simulating PDP-11/45 virtual memory

4.2.3.1 Address translation

One of the major problems to be overcome in simulating the PDP-11/45 environment is to provide for the translation of PDP-11 virtual addresses to TI 980 physical addresses. This is necessary because the Concurrent Pascal virtual machine expects addresses to be byte oriented, while the interpreter must use word addresses to address physical memory. The most straightforward transformation is to separate the 16-bit virtual address into a 3-bit index into an array of bias values, and a 13-bit displacement (in bytes) into the 8K byte page. This operation must be done

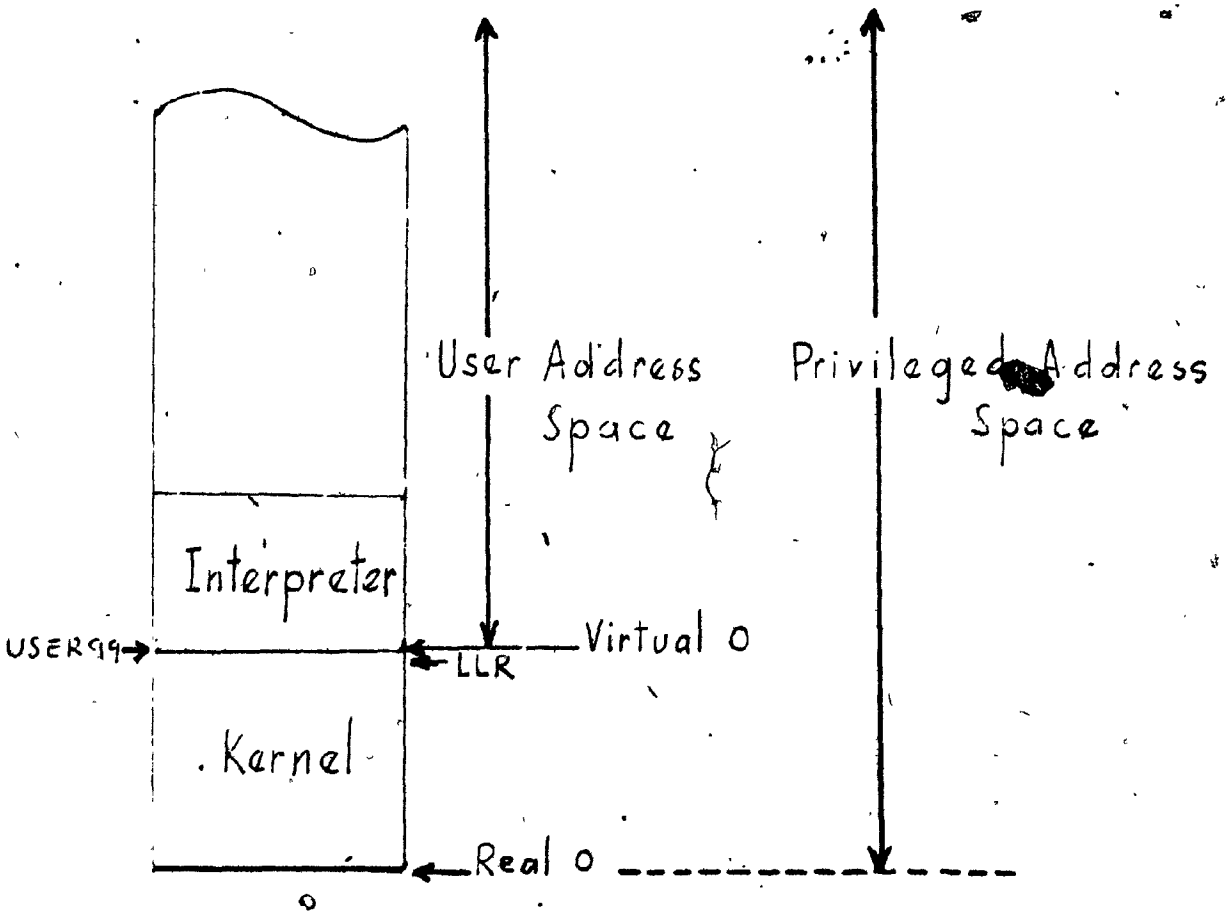

Figure 4.1 The two address spaces on the TI 980B

with care, however, as it leads to a 17-bit address if the operand is in the top 32K words of memory.

In order to keep the arithmetic within the 16-bit range, we may note that all operations in the interpreter (except some which are specifically identified as byte-manipulation operations) operate on an integral number of words. Therefore the displacement can be divided by two before being added to a word-address bias value, thus

keeping the arithmetic within 16 bits. (Byte operations must take note of the byte number before dividing by two.)

A faster algorithm, which is more specific to the Concurrent Pascal context, is the following: we note that the address space of a running process is the common segment and the private segment of that process. These are separated by a process-specific number of words DELTA. See figure 3.5. If we define COMLIM as the upper limit of the common segment (in words), then the virtual-to-real conversion may be made in the following way. Let V be the virtual PDP-11 byte address and R the TI physical address.

```
R := V/2
IF R >= COMLIM THEN R := R + DELTA
```

The TI assembler code for this sequence takes 6.75 microseconds, assuming that the current value of DELTA is stored in the current process head in the interpreter. The real-to-virtual conversion (this is needed because values stored back onto the stack must be in PDP-11 format) is

```
IF R > COMLIM THEN R:= R - DELTA
V := R*2
```

The two conversions are in-line coded in the interpreter.

4.2.3.2 Memory allocation

Although the TI memory is not segmented, allocation of memory must be done in segments of 8K bytes and in blocks of 32 words, because features of the PDP-11/45 memory management unit have influenced the design of the virtual machine. The important condition that USER99, a pointer to the running process located at the beginning of the interpreter, must sit on a block boundary, caused considerable difficulty as the TI does not have macro facilities to make it self-adjustable. This requires a re-computation of the location of USER99 after every update of the virtual machine.

4.2.4 The Interpreter on the TI 980B

In performing the hand translation from the PASCAL comments to TI assembler, we have found it necessary to understand the PDP-11 macro assembler implementation in detail; most of the time the comments reflect intent and implementation but there are a sufficient number of cases where the direct relationship is violated to require a detailed examination of the entire text. This is not to say that the comments should always be changed to reflect the details of the assembler version more accurately (indeed, much of the power of using a high-level language for commentary would thereby be lost), but rather that the comments have to be interpreted only as commentary, and not as a compilable program specification.

4.2.4.1 Register allocation

The TI 980B contains only one set of registers; the register file (A, E, X, M, S, L, B), the PC, and the status register must be saved on entry to the kernel and restored on exit. The interpreter uses the A, E, and X registers as general working registers W, Y, and X (X is used occasionally for indexing). The B register is used to access code and data outside the interpreter. The PC is used as the real program counter. Even though the M, S, and L registers cannot be used to directly address operands in memory, it is useful to keep three of the interpreter addresses Q, S, G, B, H in these registers so that they may be quickly copied into the B register, since the B register cannot be loaded from memory. Thus we have the following register allocation scheme:

R0 (A) ---> W

R1 (E) ---> Y

R2 (X) ---> X

R3 (M) ---> Q

R4 (S) ---> S

R5 (L) ---> G

R6 (B) ---> Z

R7 (P) ---> P

The remaining two interpreter registers (B, H) are kept in the process head.

4.2.4.2 Simulating the bit/byte numbering system

When moving character strings around or testing bits in sets, we must bear in mind their peculiar PDP-11 ordering. With regard to bytes in a string, the PDP-11 devices output the lower-order byte of a word before the high-byte. This must be followed when inputting or outputting to the console, the printer, the card reader,etc., because the virtual machine expects data to be in PDP-11 format.

4.2.4.3 Threaded-code implementation

The address of the next virtual instruction is contained in the virtual program counter Q. This address is a PDP-11 virtual byte address, which must be converted to a real TI word address. In addition, it must be incremented by two. As the virtual program counter must always point at a word address, the overhead introduced by the required conversion led us to decide to keep the local value of Q in the M register as a real word address. Detailed analysis of the uses of Q can be found in [30]. As an autoincrement feature is not available, this becomes a sequence of five instructions: four to move the contents of the M register to the B register, increment the M register, fetch the op-code, and branch indirectly through the jump table; one at the end of each routine to branch to the start of the fetch sequence, located at address one of user space. (Address zero contains USER99). The TI assembler sequence for

fetching a virtual instruction is:

```
FETCH   RMO  M,B        move M to B
        RIN  M,M        increment M
        LDX  0,BR       load X with op-code
        BRU  *TABLE,X   branch indirectly through table
```

The branch instruction at the end of each routine is:

```
        BRU  =1         branch to location 1 (FETCH)
```

The correct operation of this instruction is dependent on the LLR being set to the address of USER99-1.

4.2.4.4 Stack addressing

Operands on the stack are addressed relative to S, G, or B. An analysis similar to that of Q may be made to show the utility of keeping these values as word addresses, at least inside the interpreter. Most operations address only word boundaries, so the same algorithms may be used for virtual/real and real/virtual conversions when required. The exceptions are the byte-oriented operations, which must identify the byte in question prior to conversion of a virtual (byte) address to a real (word) address. Because of the opposite addressing conventions in the PDP-11/45 and the TI 980B, byte operations fetch an entire word to the A register, manipulate the byte in question, and then restore the entire word.

4.2.5 The kernel on the TI 980B

We only discuss relevant features of the implementation that require siginificant changes.

4.2.5.1 Processor multiplexing

Since the TI possesses only one set of registers, all pointers and registers that characterize the environment of a process must be saved whenever there is a context switch from user mode to privileged mode, even if the process is not preempted. The restoration of a process environment when it resumes execution must be delayed until the last moment when the registers are no longer needed for other computations.

4.2.5.2 Simulating the PDP-11/45 real arithmetic

As SOLO and the two compilers use real numbers, we were forced to write two conversion routines situated in the kernel to convert PDP-11 real numbers into TI 980 real numbers and vice-versa. These conversions were necessary because programs on the distribution tape have PDP-11 format real numbers in their "constant" declaration area. The TI floating point instructions used inside the interpreter are translated into supervisor calls which transfer control to the TI Floating Point Package physically integrated into the kernel code segment. It is worth mentioning that the TI Floating Point Package required minor modification in calling format and register usage to adapt to the

interpreter.

4.2.5.3 Peripheral activation

A rough calculation and comparison of the time used in context switch and the time elapsed between two card-column interrupts led us to let the card reader generate interrupts only on the first column and to hold the CPU for the remainder of the card. The first interrupt is necessary because of a relatively large segment of time involved in mechanical motion and reading the gap separating the edge of the card and its first column. The console interrupts on every input or output character. The disk signals only at the end of its data transfer. The line printer signals only at the end of a line printed. The TI kernel contains a virtual-to-real address conversion routine similar to that of the PDP-11 version.

4.3 Initiating the system

After implementing the virtual machine, a virtual disk of 4800 PDP-11 sectors must be initialized with the new TI kernel and interpreter, the SOLO code, a free-page list, a catalog, and the SOLO files. Our overall porting strategy proceeds as follows:

1- transfer the virtual disk to the TI disk. (the virtual disk resides on the distribution tape)

2- replace the PDP-11 virtual machine by the TI virtual machine

Given that the TI disk (a Diablo 44) can accomodate twice as much data space as SOLO requires, we decided to incorporate the virtual disk as a TI standard file. The TI kernel and interpreter code segment which we have created cannot fit into the virtual disk, because it exceeds the 24 virtual sectors occupied by the PDP-11 virtual machine on the distribution tape. This, in turn, motivated us to store the TI virtual machine as a TI standard file. To move the SOLO virtual disk into a TI standard file requires the following steps:

1- Transfer the SOLO tape (virtual disk) to a CYBER mass storage unit

2- Download the virtual disk into a TI file using a TI utility.

This roundabout route was dictated by the lack of a TI tape drive. The TI virtual machine is assembled from cards and stored into a TI binary file. Since the virtual machine runs on a bare machine, a special utility that we wrote, called PREFACE, assists in loading the virtual machine into memory. The virtual machine, in turn, loads the SOLO code from an absolute TI address, and starts executing it.

4.4 Validating the system

One of the requirements for software portability is to provide test programs to validate the newly transported software. Brinch Hansen provided us with programs to test the new kernel and interpreter. The kernel tests are carefully progressive in that they proceed from the most basic functions of the kernel to the most sophisticated ones. The areas scanned by the kernel tests are:

1- system loading

2- processor multiplexing

3- device drivers.

As thorough as it appears, the kernel tests overlooked the disk and card reader drivers in the context of Concurrent Pascal. The interpreter tests did cover the entire spectrum of interpreter routines. The test programs are extracted from another distribution tape containing both the sources and binaries as separate files. To retrieve the test programs, the following steps are taken

1- Transfer the tape contents to a Cyber mass storage unit

2- Write a Pascal program to convert the binaries into binary card images

3- Read the cards into a TI standard file

Again, PREFACE provided us with mechanisms to read in the binary cards, display the contents of the TI files, and of central memory, on a peripheral device, and other

indispensable functions to help the debugging and testing. For more details of PREFACE, refer to appendix 1.

4.5 Portability of Concurrent Pascal

Poole and Waite [25] define software portability as follows:

> "Portability is a measure of the ease with which a program can be transferred from one environment to another. If the effort required to move the program is much less than that required to implement it initially, then we say it is highly portable."

An operating system must be intimately concerned with the hardware it supports and as such is inherently machine-dependent. Brinch Hansen succeeded in separating the machine-independent logic of an operating system from its machine-dependent part. Concurrent Pascal, which embodies the machine-independent logic, runs on a virtual machine which hides the hardware dependency. The virtual machine technique for software portability is not new. The problem of portability is now confined to the portability of the virtual machine. The Concurrent Pascal virtual machine suffers from an invasion of many PDP-11/45 features, which tarnishes its nature, i.e., to stay above hardware peculiarities. As a consequence, all virtual machines designed to support the Concurrent Pascal virtual code are

forced to simulate not only the orignally intended idéal stack machine but also the idiosyncrasies of the PDP-11/45 architecture.

The major difficulties encountered in emulating the PDP-11/45 fall into three broad areas:

1- the virtual memory

2- the I/O devices and I/O architecture

3- the internal representation of data (real arithmetic representation and the bit-byte reverse order).

One notable flaw of the Concurrent Pascal system is the use of reals in the SOLO operating system and the two compilers. The Concurrent Pascal virtual machine has a dependence on the PDP-11 format for real arithmetic. In implementing the entire Concurrent Pascal system, it is obvious that real arithmetic must be provided for. However, the task of getting SOLO running is made considerably more difficult by the fact that both SOLO and the two compilers make use of real arithmetic. If they had not, then SOLO could have been made to work prior to implementation of support for real arithmetic. In summary, although the Concurrent Pascal kernel and interpreter have proved to be relatively difficult to transport, in assessing the portability of the system, one has to consider the total effort expended in the original development, including the

compilers and SOLO itself. From this perspective, Brinch Hansen experiment in portability appears to have been relatively successful.

4.6 Conclusion

The Concurrent Pascal virtual machine totals only 4 percent of the code of the entire Concurrent Pascal system. However, as it must simulate the PDP-11/45 architecture, and must normally be coded in assembly language, especially one that does not have macro facilities, the effort involved in its transportation far exceeds the 4 percent of the initial implementation effort. This is attributed to the vast difference between the TI 980B and the PDP-11/45 architectures. In short, Concurrent Pascal proves to be only moderately portable. There are some peripheral items which might improve overall portability:

1- better documentation including such things as a flow diagram of the kernel modules and their calling formats

2- a more detailed explanation of the kernel modules and interpreter routines.

3- a more comprehensive set of test programs which guarantee that the virtual machine is truly fit and ready to tackle the real operating system.

CONCLUSION

In this document, we have presented a panoramic view of different virtual machines whose common goal is to facilitate the production of portable operating systems. We have described the Concurrent Pascal language and its virtual machine, and finally, we have analyzed the portability of Concurrent Pascal vis-a-vis the TI 980B.

Concurrent Pascal has raised considerable interest in the programming field as shown by the many implementations around the world. The abstract data types introduced by Concurrent Pascal impart structuring, clarity and safety to operating systems. However, Concurrent Pascal's portability leaves much room for improvement. Its virtual machine architecture resembles too much that of the PDP-11/45 and therefore deviates from its original goal, i.e. staying above any particular hardware.

Since the design of the virtual machine is crucial in determining the portability of programs written for it, one obvious area of its design worth exploring is the parameterization or virtualization of machine-dependent features, which is absent in most virtual machines. This is consistent with the objective of the virtual machine: stand at a level of abstraction high enough to enable transportation across a wide spectrum of machine

architectures. This requires that the virtual instruction set and its run-time support refrain from being hardware specific in its original version.

In more concrete terms, this necessitates that

1- The virtual instruction be abstract enough and provide enough parameters for its run-time support to function efficiently.

2- The run-time support, which is 100 percent machine-dependent, be top-down designed and modularized, so that the highest-level modules interfacing the interpreter are the least hardware-specific and the lowest-level modules will handle all hardware idiosyncrasies. The top-down design of the operating system should extend to the design of its virtual machine as well. The degrees of freedom from particular hardware provided by this methodology corresponding to the levels of abstraction of the virtual machine could be a direction for future research in operating system portability.

REFERENCES

1. Wulf, W. A., Russell, D. B., Habermann, A. N., BLISS: a language for system programming. Comm. ACM, Vol. 14, Dec 1971, 780-790.
2. Wirth, N., PL360, a programming language for the 360 computers, Journal of the ACM, Vol. 15, 1968, 37-74.
3. Richards, M., BCPL: a tool for compiler writing and systems programming, Spring Joint Computer Conference, Vol. 34, 1969, 89-93.
4. Wirth, N., Modula: a language for modular multiprogramming, Software - Practice and Experience, Vol. 7, 1977, 3-35.
5. Steel, T. B. Jr., UNCOL, the Myth and the Fact, Annual review in automatic programming, Vol. 2, Pergammon Press, N.Y., 1961, 325.
6. Coleman, S. S., Poole, P. C., and Waite, W. N., The Mobile Programming System: JANUS, Software - Practice and Experience, Vol. 4, 1974, 5-23.
7. Theaker, C. J., Frank, G. R., MUSS: a portable operating system, Software - Practice and Experience, Vol. 9, 1973, 633-643.
8. Morris, D. et al., Machine-independent operating systems, IFIP Congress Proc., Toronto, North Holland, 1977, 819-825.
9. Brinch Hansen, P., The Programming Language Concurrent Pascal, IEEE Transaction on Software Engineering, Vol. 1, 1975, 199-207.

10. Abramson, H. et al., The PICA-B Computer - an Abstract Target Machine for a Transportable Single-user Operating Environment, Proc. ACM 1978, Washington, ACM, 1978, 301-309.

11. Thalmann, D., Evolution in the design of Abstract Machines for Software Portability, Proc. 3rd Intern. Conf. on Software Engineering, Atlanta, Georgia, IEEE press, 1978, 333-340.

12. Richards, M., The implementation of BCPL, Software portability, Brown (Eds), Cambridge Press, London, 1977.

13. Richards, M., Bootstrapping the BCPL compiler using Intcode, Machine Oriented Higher Level Languages, van der Poel Maarsen, L. A., (Eds), 1974, 253-264.

14. Dahl, O. J., Dijkstra, E. W., Hoare, C. A. R., Structured programming, Academic Press, N.Y., 1974.

15. Dahl, O. J., and Nygard, K., Simula - An Algol based simulation language, Comm. ACM, Vol. 9, Sept 1966.

16. Parnas, D. L., On the criteria to be used in decomposing systems into modules, Comm. ACM, Vol. 12, Dec 1971.

17. Hoare, C. A. R., Monitors: an operating system structuring concept, Comm. ACM, Vol. 17, 1974, 549-57.

18. Brinch Hansen, P., The Architecture of Concurrent Programs, Prentice-Hall Inc., Englewood Cliffs, N.J., 1977.

19. PDP-11/45 Processor Handbook, Digital Equipment

Corporation, Maynard, MA, 1973.

20. Bell, J. R., Threaded code, Comm. ACM, Vol. 16, June 1973, 370-372.
21. Assembly Language Manual, TI 980 Minicomputer, Texas Instruments Corporation, Digital Systems Division, Dallas, TX, 1976.
22. Desjardins, P., personal communications.
23. Thalmann, D., Abstract, fictitious, hypothetical, ideal, imaginary and virtual machines. Publication 329. Departement d'Informatique et de Recherche Operationnelle, Universite de Montreal, 1979.
24. Hartmann, A. C., A Concurrent Pascal compiler for minicomputers. Lecture notes in computer science. Springer Verlag, Berlin, Heidelberg, New York, 1979.
25. Poole, P. C., and Waite, W. M., Portability and Adaptability, Software Engineering, Springer-Verlag, Berlin, Heidelberg, New York, 1973.
26. Brinch Hansen, P., Concurrent Pascal implementation notes, Information Science, California Institute of Technology, 1976.
27. Thalmann, D., Ecriture de systemes d'exploitation portables pour mini-ordinateurs, These de Doctorat es science, Geneve, 1977.
28. Neal, D., An architectural base for Concurrent Pascal, Technical Report, Department of Computer Science, Kansas State University, Manhattan, Kansas, 1977.
29. Wallentine, V. and McBride, R., Concurrent Pascal - A

tutorial, Technical Report, Department of Computer Science, Kansas State University, Manhattan, Kansas, Nov. 1976.

30. Atwood, J. W. and T. Pham, A Concurrent Pascal interpreter for the Texas Instruments 980B, International Symposium on Mini- and Micro- computers, Montreal, Conference Proceedings, Nov 1977, 41-48.
31. Nori, K. V. et al., The Pascal-P compiler: Implementation notes, Berichte des Instituts fur Informatik, Zurich, ETH, 1975.
32. Wulf, W. A., Overview of the HYDRA operating system, Comm. ACM, Vol. 17, June 1974.
33. Brinch Hansen, P., The SOLO operating system, Software - Practice and Experience, Vol. 6, 1976, 199-207.
34. Dijkstra, E. W, Hierarchical ordering of Sequential Processes, Acta Informatica, Vol. 1, No. 2, 1971, 115-138.
35. Wirth, N., Systematic programming, Prehtice-Hall Inc., Englewood Cliffs, N.J.,1973.

PREFACE USER MANUAL

Immediately after the bootstrap, the standard TI boot loader asks the user to enter a system file name

SFN = ?

User enters PREFAC, which will be loaded from D0 into upper memory and given control. PREFAC (file name of PREFACE) will ask user to enter one of many commands:

ENTER COMMAND

Here are the commands that the user can use.

LOADVM : will load the memory image of the kernel and the interpreter, in other words the virtual machine from the kernel file on D1, into memory starting at address 0, and give it control.

GENCOR : This is the most frequently used command to start execution of a Concurrent Pascal program (test programs or operating systems). It essentially processes the object code which is the assembly output of the virtual machine and which resides on D0's file SMEFIL. GENCOR asks user to enter SMEFIL when it prompts him with:

VFN = ?

Once the object code has been converted into memory image form and loaded into memory starting at location 0, control is given to it.

STORVM : Stores the virtual machine in memory image form on D1 at beginning of "kernel" file. This command is the opposite of LOADVM.

MT1.D1 : Takes test programs from tape and stores it on D1 at sector address specified by the user. MT1.D1 will prompt user for a destination sector address on D1:

SECTOR ADDRESS IN DECIMAL = ?

MT2.D1 : Takes the SOLO binary code from tape and stores it on D1 at a sector address specified by the user (similar to MT1.D1). These two tape related commands are not currently in use because the tape drive is not connected.

CRTOD1 : Reads a deck of binary cards (Concurrent Pascal test program) and puts it on D1 at a sector address specified by the user. PREFACE asks the user to enter a starting sector address:

SECTOR ADDRESS IN DECIMAL = ?

D1TOLP : Dumps on the line printer the contents of sectors as specified by the user. Same prompt as previous command.

FROM SECTOR =? TO SECTOR =?

APPENDIX 2 : INTERPRETER OPERATIONS

Arithmetic and Logical Operations

Add	word/real	In	set
Or	word/set	Build	set
Subtract	word/real/set	Empty	set
Multiply	word/set	Increment	word
And	word/set	Decrement	word
Divide	word/real	Truncate	real
Mod	word	Convert	word
Negate	word/real	Successor	word
Not	word	Predecessor	word
Abs	word/real		

Relational Operations

Equal	word/real/set/structure
Not equal	"
Not greater	"
Not less	"
Greater	word/real/structure
Less	"

Address Manipulation and Verification

Constaddr	Variant
Localaddr	Range
Globaladdr	New
Field	Newinit
Index	Initvar
Pointer	

Stack Operations

Push	const	Copy	byte
	local		word
	global		real
	indirect		set
	byte		tag
	label		structure
Pop			
Funcvalue			
Setheap			

Control Transfer Operations

Jump
Falsejump

Casejump	
Call	procedure/function
Enter	"
Exit	"
Callprog	sequential program
Enterprog	"
Exitprog	"
Initclass	initialize class
Beginclass	initial class statement
Endclass	"
Enterclass	class procedure
Exitclass	"
Initmon	initialize monitor
Beginmon	initial monitor statement
Endmon	"
Entermon	monitor procedure
Exitmon	"
Initproc	initialize process
Beginproc	concurrent process
Endproc	"
Callsys	call prefix routine in concurrent process
Enterproc	"
Exitproc	"
Delay	kernel request to delay process
Continue	kernel request to continue process
Start	request to start job
Stop	kernel request to stop job

Wait	kernel request to wait for alarm clock

Miscellaneous

Newline	note current line in source program
Attribute	return process attributes
Realtime	return real time in seconds
IO	perform input/output

Kernel operations

Initgate

Entergate

Leavegate

Endprocess

Initprocess

Stopjob

Wait

Systemerror

APPENDIX 3

PROCESSES

A process is a system component that is executed as an asynchronous sequential program. It consists of a private data structure and a sequential program that operates on the data. A process cannot operate on the private data of another process, but processes can share certain data structures. The shared data on which a process can operate are determined by its access rights. The notation

```
Type P = process (A1: R1; ....Ak: Rk);
         var V1: T1; ...., Vm:Tm;
         begin
            S1; ......Sn;
         end;
```

declares a process P with the following attributes:

1- global variables V1, ... Vm of type T1, ... Tm which are private data defining the current state of the process.

2- parameters A1, ... Ak of type R1, ... Rk called the access rights of process P and which represent constants and other system components on which the process can operate.

3- statements S1, ... Sn operating on the global data and accessible system components.

A process can also include definitions of constants, procedures and functions which are accessible only within the process. The statement INIT P executed by the embedding process allocates storage for the global variables of process P, starts its execution as a concurrent process and replaces the formal parameters of the process by the actual parameters which remain accessible to the process after initialization. The execution of a process terminates normally at the end of its statement, but in some cases can repeat the execution of a set of statements forever. This is done by means of the CYCLE statement, which defines a sequence of statements to be executed indefinitely.

MONITORS

A monitor is a system component that controls communications and resource sharing among concurrent processes. It defines a shared data structure and a set of synchronizing procedures operating on it. The execution of these procedures exclude one another in time. A monitor also defines an initial operation that will be exceuted when its data structure is created. The notation

```
Type M = monitor
         var V1: T1; ... Vm: Tm;
         procedure entry P1 ( ); begin .... end;
         .....
         procedure entry Pn ( ); begin .... end;
         begin
```

S

end;

declares a monitor M with the following attributes:

1- global variables V1, ... Vm of type T1, ... Tm defining the current state of the monitor

2- procedure entries P1 ... Pn, that processes can call to operate on the global data

3- an initial statement S, defining the initial values of the global data.

As for a process, a monitor can have parameters that define its access rights. It can also include definitions of constants and types as well as procedures and functions that are accessible only within the monitor. Processes cannot operate directly on a monitor's shared data. They can only call procedure entries that have access to these shared data. A call to a procedure entry Pi within a monitor M is denoted by

M.Pi ( )

Such a monitor procedure call is executed as part of a calling process just like any other procedure.

CLASSES

A system component that cannot be called simultaneously by several other components is called a class. A class defines a data structure and the possible operations on it, just like a monitor. But unlike a monitor,

1- the execution of class procedure entries do not exclude one another in time

2- a global variable within a class can be declared as an entry variable.

For example

```
Var C : class;
        var .... entry Vi: Ti; ... ;
        .....
        begin .... end;
```

The value of variable Vi can be accessed outside the class by using the notation

C.Vi

but assignment to an entry variable can only be made by a procedure of the class in which it is declared. One advantage of classes over monitors is that class calls are much faster than monitor calls, because the virtual machine does not have to schedule simultaneous calls at run-time, since such calls cannot occur.

SCOPE RULES

A scope is a region of program test in which an identifier is used with a single meaning. An identifier must be introduced before it is used. (The only exception to this rule is a sequential program declaration within a process type: It may refer to routine entries defined later in the same process type. This allows one to call sequential programs recursively.)

A scope is either a system type, a routine, or a "with" statement. A system type or routine introduces identifiers by declaration; a "with" does it by selection. When a scope is defined within another scope, we have an outer scope and an inner scope that are nested. An identifier can only be introduced with one meaning in a scope. It can, however, introduced with another meaning in an inner scope. In that case, the inner meaning applies in the inner scope and the outer meaning applies in the outer scope.

System types can be nested, but routines cannot. Within a routine, "with" statements can be nested. This leads to the following hierarchy of scopes

(nested system types

(non-nested routines

(nested "with" statements)))