AN ARCHITECTURAL BASE FOR CONCURRENT PASCAL

by

DAVID NICHOLAS NEAL

B.A., University of Minnesota, 1972

A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1977

Approved by:

Major Professor

THIS BOOK CONTAINS NUMEROUS PAGES WITH THE ORIGINAL PRINTING BEING SKEWED DIFFERENTLY FROM THE TOP OF THE PAGE TO THE BOTTOM.

THIS IS AS RECEIVED FROM THE CUSTOMER.

THIS BOOK IS OF POOR LEGIBILITY DUE TO LIGHT PRINTING THROUGH OUT IT'S ENTIRETY.

THIS IS AS RECEIVED FROM THE CUSTOMER.

Document
LD
2668
T4
1977
N42
C.2

113

Table of Contents

Chapter 1 Introduction.....1
Language Concepts.....4
System Types.....5
Processes Monitors and Classes.....7
Architectural Structure of Implementations.....8

Chapter 2 Implementations of Concurrent PASCAL.....13
Brinch Hansen (PDP-11) System.....13
Data Model (Interpreter).....14
Control Model (Kernel).....18
Storage Allocation.....22
KSU (I8/32) System.....24
NUC (I7/16) System.....29
Summary.....30

Chapter 3 Experiences with the Portability of Concurrent PASCAL.....32
Approach to Portability.....33
Language and Run-Time Support.....34
Virtual Machine Implementation.....37
Details and Problems of Implementation.....40
Address Space Structure.....40
Virtual Code and Internal Representations.....44
I/O Structure - Devices.....49
Operating System Idiosyncracies.....51
Conclusion.....53

Chapter 4 Architectural Base Considerations.....56
Levels of Abstraction (Hierarchy).....56
VMM Design Considerations.....60
State Transitions (Control Model).....63
Implementation.....65
Data Model.....72
Motivation.....73
Stack Model and Discussion.....74
Expression Evaluation.....75
Stack Depth.....78
The Hardware Implementation.....81
Logical Stack Structure and Operation.....82
Parameter Passing and Procedure Call.....84
Implementation of the High Speed Stack in Conventional Computer Architecture.....85
Register Set Segmentation.....85
Conclusion.....90

Chapter 5 Conclusion.....91

References.....93

Appendix A Functional Specification of A Concurrent
PASCAL Kernel for the Interdata under OS/32-MT....97
Introduction.....97
Traplevel 1.....99
IO Entry.....100
IO Interrupt.....101
New.....102
Queuetype (Processqueue).....103
Signal.....104
Time.....105
Timer.....106
Clock.....107
Timertrap.....108
Core.....109
Running.....110
Ready.....113
Gate.....114
Exit.....115
Reinitialization.....116
Interpreter.....117
Faults.....118
Transient Initializer.....119

Appendix B Stack Depth of PASCAL Code.....120

Appendix C Scheduler Algorithm.....122

List of Figures

2.1 Data Structures.....15
2.2 Concurrent PASCAL Stack.....15
2.3 Stack in Sequential PASCAL.....17
2.4 Storage Allocation (Brinch Hansen).....23
2.5 Concurrent PASCAL Transportation.....25

3.1 Brinch Hansen Concurrent PASCAL Implementation....36
3.2 KSU Concurrent PASCAL Implementation.....39
3.3 PDP-11 Segmentation Structure.....42
3.4 I8/32 Software Segmentation.....43
3.5 PDP-11 to I8/32 Address Mapping.....45

4.1 Levels of Architectural Support.....59
4.2 Concurrent PASCAL Program.....59
4.3 Multiprogramming State.....62
4.4 Representation of Multiprogramming State.....64
4.5 Blocked Process Structure.....67
4.6 Ready Structure.....68
4.7 Parse Tree for B + C * (D + E).....77
4.8 16 Register example High Speed Stack.....80
4.9 Stack Transitions.....83
4.10 Stack at Point of Procedure Call.....86
4.11 Memory Stack for Function Call.....86
4.12 Register Set Segmentation.....87
4.13 Processor Architecture.....87

Acknowledgements

It is my pleasure to acknowledge the advice and support of my major professor Virgil Wallentine and the helpful comments of my committee members Linda Shapiro and Fred Maryanski. Jim Ratliff and Gary Anderson were both deeply involved in the transportation of Concurrent PASCAL and deserve part of the credit for the final product. The concepts in the architectural base benefited from the influences of John Goettelmann and Mike Ball. I would also like to thank my wife Catherine for her (moral) support and her exhaustive editing of the text.

Thanks are in order to the Kansas State University Department of Computer Science for the extensive use that I have made of their facilities. This document was prepared using a text editor and document processor which I wrote for the KSU Concurrent PASCAL/SOLO system.

The transportation of Concurrent PASCAL was supported in part by the U.S. Army Research Office under grant DAAD-29-76-G-0108.

Introduction

The programming language Concurrent PASCAL [BH1] is the first programming language to present abstractions for concurrent programming to the high level language user. This represents a significant development for the programming of computer operating systems and real time systems. This thesis consists of an analysis of three implementations of Concurrent PASCAL, a description of the process of transporting an implementation from one computer to another, and finally, the synthesis of the various implementations into an informal model of a multiprogramming architecture suitable for use as a semantic base for the language Concurrent PASCAL.

This thesis is structured in the following manner: The remainder of chapter one contains an introduction to the abstractions presented by Concurrent PASCAL, the run-time support (Kernel) for these abstractions, and the three implementations to be discussed. Chapter two consists of a detailed discussion of the three implementations. Chapter three contains a description of the transportation of the initial implementation of Concurrent PASCAL [BH2][BH6] which was carried out by the author [DN]. Chapter four presents the architectural model and chapter five provides the summary. The three appendices consist of the functional specifications of the KSU implementation, a study of the

stack depth of PASCAL virtual code, and a scheduling algorithm.

The contributions of this research work are in the fields of operating systems and computer architecture. Although implementations are discussed, this is not an implementation document. The implementations are supporting background to the work, and the discussions of these implementations provide a survey of the relevant architectural features. The portability of Concurrent PASCAL is of interest in the development of truly portable operating systems. As such, the discussion of portability is included to demonstrate this aspect of the usefulness of Concurrent PASCAL. Although the portability strategies are well known [PW], the discussions of the problems encountered are of interest, especially to those who wish to move the Concurrent PASCAL system.

The motivation for this work stems from two issues. The first issue is that the language Concurrent PASCAL and its implementations have significantly isolated the basic facets of multiprogramming to such a degree that architectural (hardware) support seems feasible. Second, although multiprogramming is the de facto standard in operating systems, hardware support for this standard has not progressed significantly beyond the interrupt.

In operating systems, the support of abstract operating system functions in the underlying architecture have been

demonstrated in the VENUS operating system [BL] where microprogrammed support was included for semaphore operations in the base machine. Certainly operating system considerations, especially for multiprocessing, have influenced such architectures as the CDC6000 series, with its central and peripheral processor structure [CDC]. Architectural support has also been proposed for virtualizable machines [RG]. These concepts attempt to provide the architecture with the knowledge of the structure of the various active virtual machines, thus eliminating software support overhead. Add on support has been provided for the IBM 370 series to assist the operation of the 370 virtual machine facility (VM370 Virtual Machine Assist(VMA) [IBM]). Additional proposed architectural support has addressed such issues as protection [SS] and high level language support (as implemented, for example, in the Burroughs Algol machines [EO]). The above work is intended to support one of the following three concepts: generalized block-structured languages, virtual machines which are faithful replicas of the machine-level (hardware) language and multiprocessing. The goal of this work is to isolate the underlying concepts of multiprogramming from the context of Concurrent PASCAL and demonstrate how these concepts may be represented in the architecture of a Concurrent PASCAL machine.

The reader is assumed to have a basic knowledge of the

Concurrent PASCAL programming language as a background for this work, and enough experience with operating systems to be able to infer the nature of the run-time support necessary for that language. Also assumed is a knowledge of the terminology basic to operating systems work. Some of the commonly used terms in this thesis include:

PCB (Process Control Block) - the basic data structure describing a process to the system;

K - an abbreviation for 1024 units, generally describing computer storage (e.g. 10K = 10240);

byte - a basic unit of storage (generally eight bits);

multiprogramming - the use of a single processor to concurrently execute several processes (as opposed to multiprocessing where two or more processors are used to execute one or more processes).

Some knowledge of the architectures of the Digital Equipment Corporation's PDP-11 computers and/or the Interdata 8/32 computer would be beneficial, though not essential, to understanding the implementation and portability discussions.

Language Concepts

The programming language Concurrent PASCAL has been

designed to provide a convenient, structured language for the expression of solutions to concurrent programming problems, such as those occurring in operating and real time systems. The language facilities directed to these problems combine the control and data structures of PASCAL with abstractions modelled for concurrent processing. In particular, Concurrent PASCAL includes abstract types to deal with the problems of data sharing (monitors), code sharing (classes), and sequential processing (processes). This section provides an overview of these language concepts as a prelude to the discussion of the implementations and architectural support features for Concurrent PASCAL.

System Types

The programming abstractions introduced by Concurrent PASCAL are called system types. The three system types are process, monitor, and class. These three abstract types form the building blocks from which Concurrent PASCAL programs are formulated. There are several important remarks which should be made about these system types and their impact upon a Concurrent PASCAL implementation. The first point is that the system types are types in the PASCAL sense, that is they are templates from which variables can be defined and initialized. This should suggest to the implementor that code be reentrant and storage allocation dynamic. We shall see later how the various implementations

address these problems. Beyond this, system types may interact with one another by means of access rights which are defined by the formal parameters of the type and passed to it in the initialization phase of the system type. For example, processes and classes may have access to monitors and monitors may have access to other monitors. Access rights may be passed among the various system types during initialization. The declaration of process and monitor variables and their initialization is handled by the initial process; which is the primary, and initially the only, active entity in the Concurrent PASCAL program. Declaration and initialization of class type variables occur throughout the Concurrent PASCAL program, wherever common functional processing can be isolated.

In addition to access rights, the definitions of system types may include definitions of constants and non-system types and the declaration of variables, including class variables, and routines (procedures and functions). As abstractions, system types include both data and the operations to be performed on that data as a combined entity. This is considered by Brinch Hansen [BH1] to be one of the fundamental features of the language. As a result of this abstraction, the implementation of a system type variable must include both data space and reference to the functional routines of the type.

Processes Monitors and Classes

Each of the three system types serves a distinct function in a Concurrent PASCAL program. Processes define the active entities of the system. When the initial process, during execution, initializes a process variable, that process variable is given existence as a task and begins active execution concurrently with the initial process and with any previously initialized processes. A process controls its own private data space and may additionally have access to monitors for communication (sharing) with other processes.

A monitor controls the sharing of data between processes. The concept of a monitor was introduced by Dijkstra [ED], and was further refined by both Brinch Hansen [BH4] and Hoare [CH]. In Concurrent PASCAL, monitors are passive entities which are activated by a process which has access to that monitor. The mechanism for monitor access is by procedure or function call to a monitor entry point. The consistency of data in a monitor is guaranteed by two principles: First, the manipulation of monitor data structures can only be affected by routines of that monitor. This guarantees that no unexpected manipulation of the monitor data can be done outside the monitor. Also, the provision must be made in the run-time system such that only a single process may be executing code within a monitor procedure at any one time (mutually exclusive access).

A class type also provides for sharing, but the sharing of code rather than data. The inclusion of this facility in the language can account for considerable savings both in programmer coding and run-time code space, as commonly used functions may be shared by several processes and monitors. The structure and routines (code) are shared, but not the data. As an example, a first-in-first-out (FIFO) list manipulator might be defined by a class. Then several distinct lists may be declared and all will use the same routines.

This completes the brief overview of the facilities unique to the Concurrent PASCAL language. The actual syntax of the language is unimportant to the ensuing discussion. For those desiring a more detailed description, the references [BH1] and [WM] provide a more comprehensive introduction to the language Concurrent PASCAL.

Architectural Structure of Implementations

The language Concurrent PASCAL relies on a specific model of parallel computation and the implementations of the language require certain run-time support. This run-time support may be described as the virtual machine upon which Concurrent PASCAL programs are executed. It is the architectural aspects of this virtual machine that will be of interest in the following discussions. In this section, the model of computation is presented and three

implementations are introduced.

In the model of parallel processing introduced in the language Concurrent PASCAL, all active system entities are processes (or tasks). Explicit management of processes in the Concurrent PASCAL program is provided by the ability to delay and continue the execution of processes within monitors. These facilities allow for the programming of synchronized operations. In fact, Hoare [CH] shows the equivalence of monitors and semaphores. All external events (e.g. IO interrupts and timing signals) in the system must have one or more processes waiting for their occurrence. The processes waiting for an event, as well as those delayed by the Concurrent PASCAL program, are termed blocked. The occurrence of an event causes the transition from the blocked state to the ready state of all the processes waiting for that event. Only processes in the ready state may be given access to a central processor. The continue operation, like an event, causes the associated process to become ready. The mechanism used for this transition is called a Queue variable in Concurrent PASCAL. The implicit process management, including event control, which is inherent to the multiprogramming environment, is outside of the Concurrent PASCAL program's control. Among the facilities which must be provided are: the actual multiplexing of processes on the physical processor(s), the management of processes executing I/O instructions

(including interrupt handling), and the management of timing, clocks, and timing signals (interrupts). In the implementations of Concurrent PASCAL which will be discussed in chapter two, this implicit process management is handled by a Kernel which supports these functions based upon the underlying architecture in which the implementation takes place, providing for a consistent architecture at the Concurrent PASCAL level.

There are three implementations of Concurrent PASCAL which will be discussed as background for the architectural base. The first of these is Brinch Hansen's implementation, done at California Institute of Technology, on the DEC PDP-11/45 computer. This was the first implementation of the language and was conducted by the language designer, Per Brinch Hansen. The Brinch Hansen system encompassed more than just a compiler and run-time support (Kernel). It included an operating system (SOLO) written in Concurrent PASCAL, as well as utility programs (including two compilers) which form a coherent, albeit somewhat spartan, program development system. Excluding the Kernel, which is programmed in assembly language in this as well as the other two implementations, and SOLO, which is written in Concurrent PASCAL, all of the programs of the system are written in a dialect of PASCAL [JW] called Sequential PASCAL [AH2]. This implementation is significant in that it was designed as a portable system [BH6], with at

least two applicable porting strategies. This aspect of the system, as well as a detailed account of the problems encountered in the process of porting the Brinch Hansen system are discussed in chapter three.

The second implementation to be discussed is that undertaken as background for this work by the author at Kansas State University. This implementation was based upon the design of the Brinch Hansen system, but was done on a radically different architecture from the PDP-11 i.e. the Interdata 8/32 (I8/32). In addition to this, the implementation was achieved in a different run-time environment than Brinch Hansen's implementation (the PDP implementation is stand alone, the I8/32 implementation runs in an operating system environment).

The third implementation was undertaken at the Naval Undersea Center (NUC) by Mike Ball and his staff, and is targeted to the Interdata 7/16 (I7/16). This implementation is a departure from the other two in several respects. The first two implementations use a code interpreter for semantic evaluation of programs, while in this implementation, the compilers were extended to output object code for the I7/16 computer. In addition to this, the input output facilities were extended to provide device handling at the Concurrent PASCAL level, using a concept called the IO Machine.

The existence of and interest in various implementations

of Concurrent PASCAL signify the growing acceptance of abstractions in programming languages, especially with regard to languages for systems implementation and concurrent programming. The abstractions presented for concurrent programming by the language Concurrent PASCAL have been previously introduced. In the following chapters, these abstractions will be pursued and extended into the conceptual framework of an architecture based on these abstractions, an architectural base for Concurrent PASCAL.

Implementations of Concurrent PASCAL

In this chapter we provide a discussion of the three implementations of Concurrent PASCAL previously introduced. The discussions will focus on the Kernel level, as it is the goal of the implementations to provide a common architecture at the Concurrent PASCAL level (with the exception of implementation dependent extensions). In addition, for the Brinch Hansen system we will discuss the data model (semantic model) based on the virtual stack machine with its corresponding Interpreter. This model extends to the KSU system as well.

Brinch Hansen (PDP-11) System

As introduced, the Brinch Hansen system [BH2] [BH6] was the initial implementation of Concurrent PASCAL. It is the basis for the other two implementations to be discussed in this chapter. The run-time system of this implementation consists of a Kernel and an Interpreter which execute on a "bare machine" PDP-11. These two program modules form the only PDP machine language code executed in the system. All Sequential and Concurrent PASCAL programs consist of virtual code instructions which are executed by the Interpreter. The Kernel handles the multiplexing of Concurrent PASCAL processes and provides IO drivers, device interrupt handlers, and timing and clock facilities.

Data Model (Interpreter)

The data model of the Concurrent Pascal machine of this system includes two important structures, an execution stack and a program heap. Each instance of a process contains these two data structures. The stack is used by both Concurrent and Sequential PASCAL program execution, while the heap is included for the generation of dynamic variables, allowed only in Sequential PASCAL. Brinch Hansen's structure for a heap and stack consists of a single contiguous memory space, in which the two structures grow toward one another from opposite ends (as in figure 2.1). 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) linkage for procedure return (mark stack);
3) parameter passing; and
4) expression evaluation temporaries.

Figure 2.2 provides a snapshot of the stack during execution of a Concurrent PASCAL program. The structure of the stack allows several interesting uses; for example, actual parameters which are expressions are evaluated using the temporary expression stack, in the same way as any other expression. After evaluation, the parameters are left on the stack and the called procedure may then access the parameters relative to its local base register.

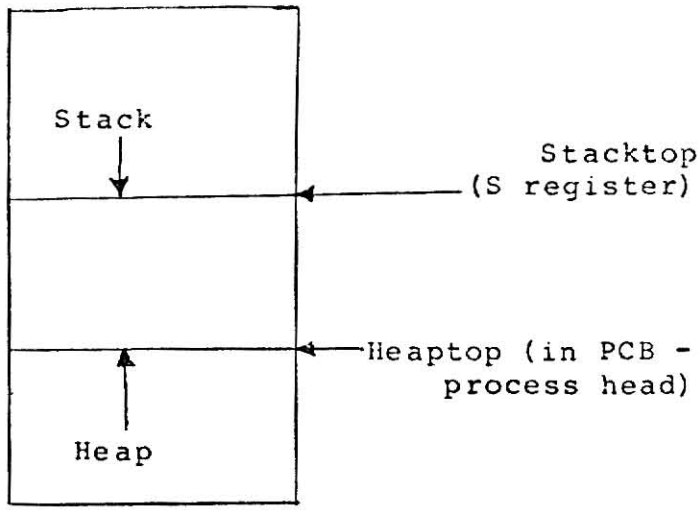


Figure 2.1 Data Structures

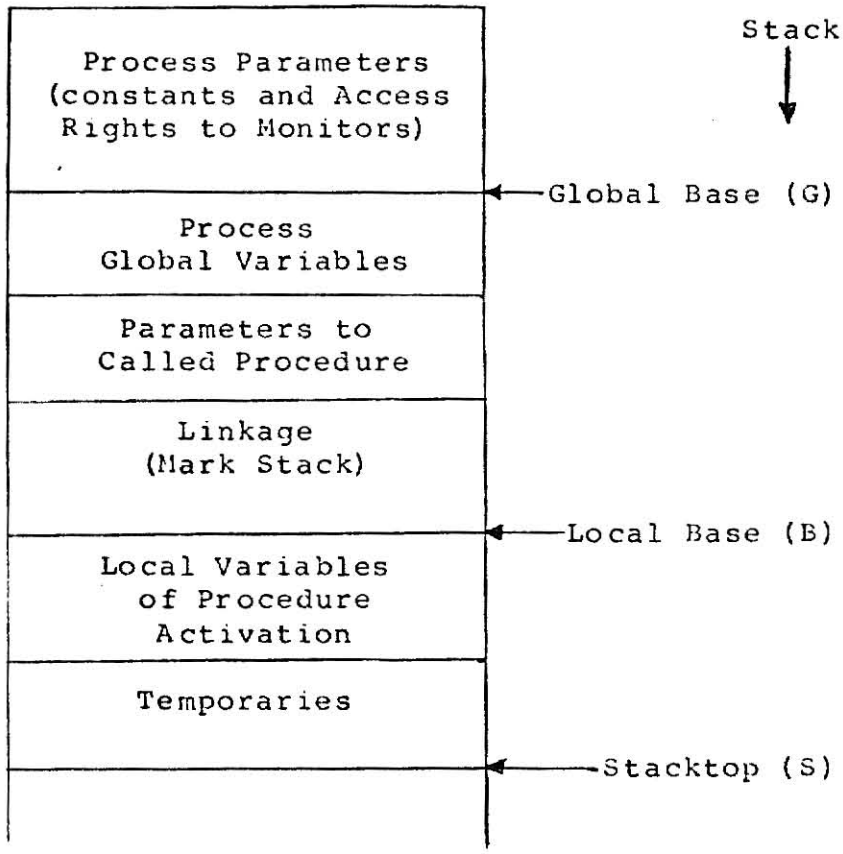


Figure 2.2 Concurrent PASCAL Stack

This method provides efficient and convenient evaluation and passing of parameters. Using the stack for local variables of procedure activation has the advantage that recursion, which is available in Sequential but not Concurrent PASCAL, is implicitly available. In fact, except for data space, it is no more expensive than a normal, non-recursive procedure call. Recursion is not allowed in Concurrent PASCAL for reliability reasons. In particular, allowing monitors to be recursively called would allow deadlock. The parameter mechanism is also general enough to extend to the interface between Concurrent and Sequential PASCAL where the parameters include the entry point addresses for interface routines, which are analagous to supervisor calls (SVC) in more conventional systems. See figure 2.3.

The operations executed on the stack consist of pushes, copies (pops) and the necessary operations for PASCAL semantics (add, subtract, multiply, divide, mod, set operations, etc.). In addition there are instructions for entry and exit to procedures, (Sequential PASCAL) programs, and system types (processes, monitors, and classes). The various entry and initialization instructions set up the correct environment for the execution of the system type, as well as saving the previous environment in the mark stack, for return linkage.

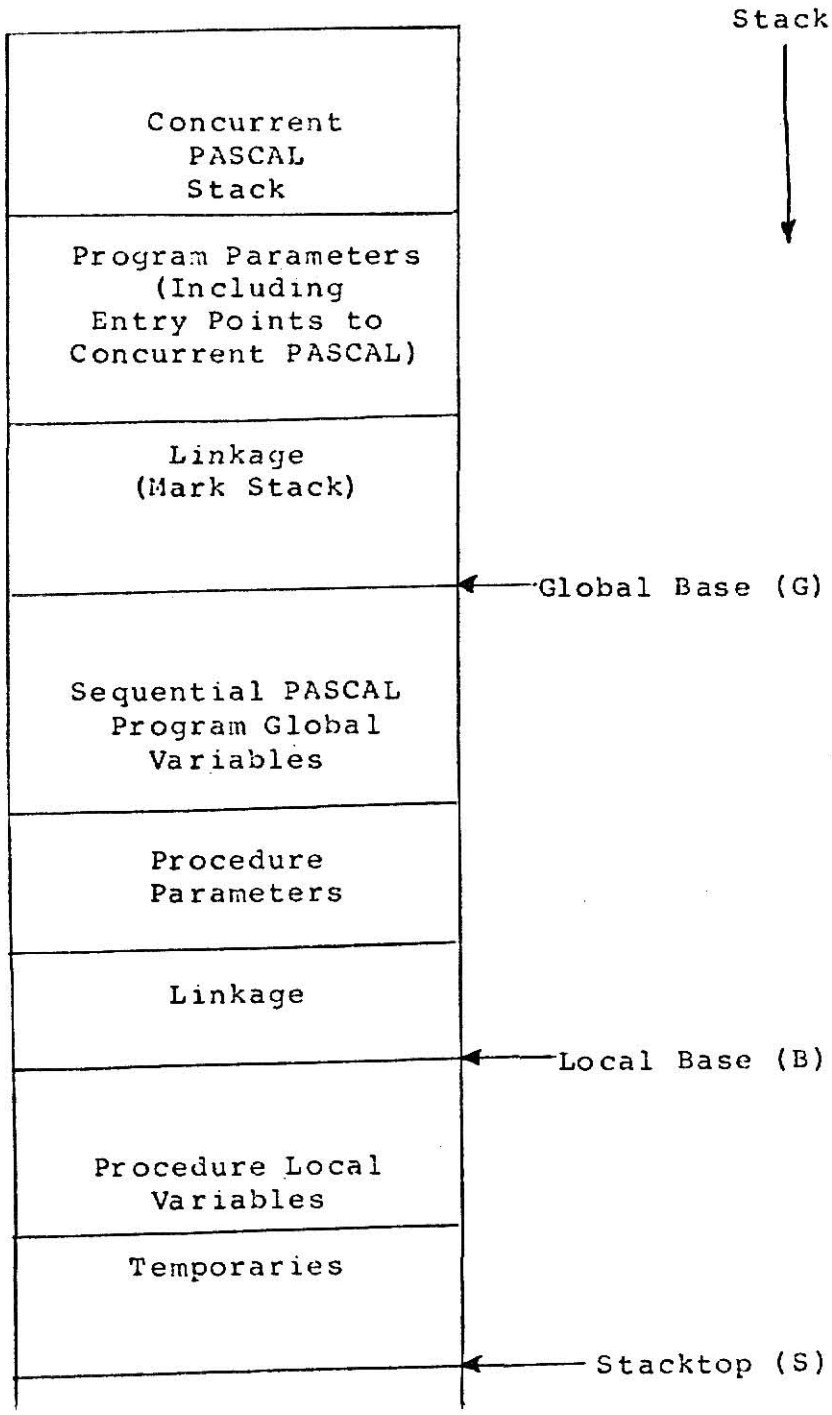


Figure 2.3 Stack in Sequential PASCAL

The instructions which cause initialization of the various system types must interface to the Kernel, since these types must have certain Kernel data structures associated with them. Instructions also exist allowing entry to and exit from (as well as the ability to delay and continue processes within) monitors. These instructions also interface to the Kernel since their evaluation causes the implicit manipulation of Kernel variables, i.e. the state of processes and/or monitors can be changed by these instructions.

Control Model (Kernel)

There are three major data structures owned and manipulated by the Kernel in this implementation. These are process records, monitor gates, and device blocks. The process records are maintained by the use of various process queues, which are doubly linked lists of process records. A process record may be on at most one process queue. Processes not on a process queue are assumed to be in one of the following states: running - in which case the processor registers hold the current state of the process (it is referenced via the current process pointer); delayed - in which state the Kernel has released the process from its purview and relegated it to the Concurrent PASCAL program by means of a Concurrent PASCAL queue variable; IO wait - in which case the device block for which the process is waiting

contains a reference to the process (this is similar to the Concurrent PASCAL queue variable, since only one process may access a device at any time). The assumption that all processes are either in one of the above states or on a Kernel process queue may, in fact, be invalid as there are several scenarios in which a process may become lost to the Kernel. These scenarios are, of course, due to errors in the Concurrent PASCAL program executing at the time. Such situations include delaying two processes on a single (Concurrent PASCAL) queue variable (the first process will be lost to the system), and a process activating an IO device while another process is awaiting completion of a transfer to that device (the waiting process will be lost). These features can be the cause of much confusion ("head scratching") during the development of Concurrent PASCAL programs.

The basic process queues of the Kernel include the three ready lists (prioritized by monitor nesting, IO completion, and compute bound), signal queues (process awaiting timing or operator signals), and monitor gate queues. Monitor gate queues help control the exclusive access of a process to a monitor. This is handled in the following manner: The gate variable consists of a boolean ("open") specifying whether any process is currently executing within the monitor. A monitor entry instruction from the Interpreter causes the Kernel to test the "open" boolean. If the boolean is true,

the Kernel sets it false signifying that a process now has access to the monitor and exits, allowing the process access to the monitor code to continue execution. If the boolean was already false, the process is preempted from execution and added to the gate queue, signifying that it must wait for exclusive access to the monitor. On monitor exit, this process is reversed, leaving the "open" boolean true if no process is waiting for the monitor, and otherwise allowing a single process from the gate queue to move to the ready state. The monitor operations "delay" and "continue" also include an implicit monitor exit. For a detailed monitor specification which is similar to the Brinch Hansen implementation, and which is formally verified see Saxena and Brendt [SB] and for a discussion of signaling strategies (of which "delay" and "continue" are one method) see Howard [JH].

Control flow within the Kernel is relatively straightforward. The design level documentation of the Brinch Hansen system consists of the expression of the logic of the Kernel in terms of Concurrent PASCAL-like abstractions (Kernel classes). The reader is referred to the "Concurrent PASCAL Machine" manual [BH2] for a discussion of these classes. These classes convey an understanding of the several individual parts of the Kernel. The overall flow of the Kernel can be simply stated and easily understood. The Kernel is entered either from the

Interpreter -- in order to

a) initialize or terminate a system type,

b) gain or release access to a monitor,

c) await a timing or operator signal,

d) initiate I/O or

e) process a semantic error in the Concurrent PASCAL program

-- or by means of an interrupt (timer or device). Events which cause transition from a blocked state to ready (IO completion, continue of a Concurrent PASCAL queue variable, completion of timing interval) also cause the ready list to be rescheduled, causing preemption of the running process if a higher priority process has become ready. Blocking transitions (IO request, entry to an active monitor, delay of a Concurrent PASCAL queue variable) cause the running process to be preempted and reference to the process to be made in the correct blocking data structure (Concurrent PASCAL queue variable, Kernel process queue, or device block). Most Kernel entries cause the running process time slice to be updated and if this exceeds a Kernel defined maximum (16.7 milliseconds), the process is preempted, unless it is executing within a monitor. Each of the Kernel classes handles a section of this control flow, isolating a particular function. The final act of a Kernel activation tests to see if there is an active process and if so, returns to it; if not the Kernel attempts to select a ready

task to be the current process. If none are available, the Kernel causes the processor to idle, otherwise the Kernel returns to the selected process.

Storage Allocation

The allocation of storage in the Brinch Hansen system is very straightforward. The compiler for Concurrent PASCAL computes the total global storage necessary for monitors and also the stack space necessary for the initial process. This storage is then allocated at run-time during initialization of the Concurrent PASCAL program. Each process is allocated storage for all its variables (i.e. stack, heap, and globals - including class variables) during initialization of that process. Kernel variables (monitor gates and process records) are pre-allocated during initialization in data space dedicated to the Kernel. It is possible at run time to overflow either the reserved Kernel data space or the process data space. The first of these conditions is caused by declaring too many processes and monitors in the Concurrent PASCAL program. The second is caused by attempting to use more main storage than there is available on the processor. Memory is never de-allocated in this implementation. Even if a process completes execution, its data space remains allocated. The storage allocation is illustrated in figure 2.4.

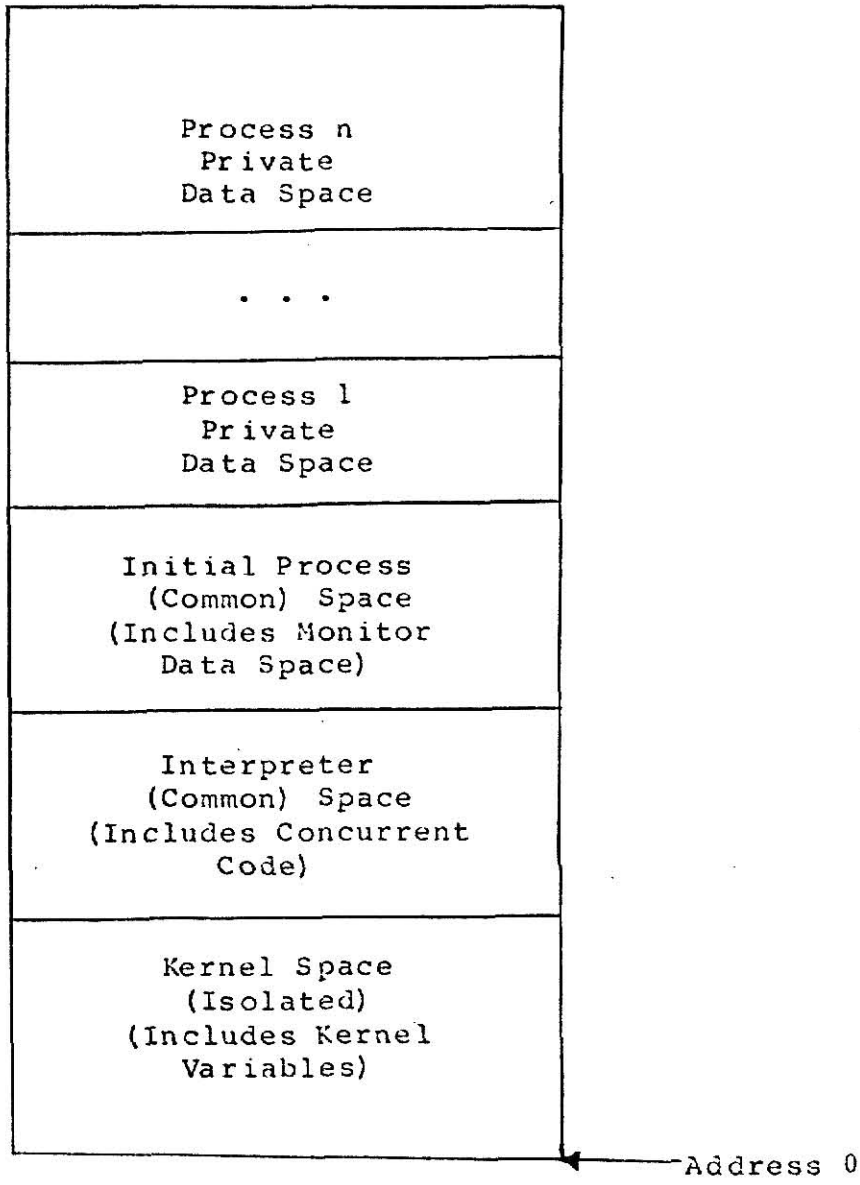


Figure 2.4 Storage Allocation (Brinch Hansen)

The goals of the Kernel for Concurrent PASCAL as described by Brinch Hansen must include the provision of concurrent processing abstractions at a language level which is not as prone to errors as assembly language. In addition, when programming must be done in assembly language, abstractions can be used to aid the thinking process and through discipline and structure restrict the probability of error. In these respects, the Brinch Hansen system earns a well deserved place in the development of programming languages.

KSU (I8/32) System

The KSU system [DN] is based on Brinch Hansen's design of the Kernel and Interpreter. The virtual machine architecture was faithfully reproduced by programming the Kernel and Interpreter in Interdata CAL assembler language [I3]. The logic flow of the Kernel was duplicated and the IO drivers have been written to simulate the IO calls used by the SOLO operating system (using the Concurrent PASCAL IO function). Having completed the above effort (a system of approximately 5000 lines of assembler code) and simulating the virtual disk structure of the SOLO system, we were able to load and execute the SOLO system using the virtual code file included in Brinch Hansen's distribution tape (SOLO files). The details and problems of the system transportation are covered in chapter 3.

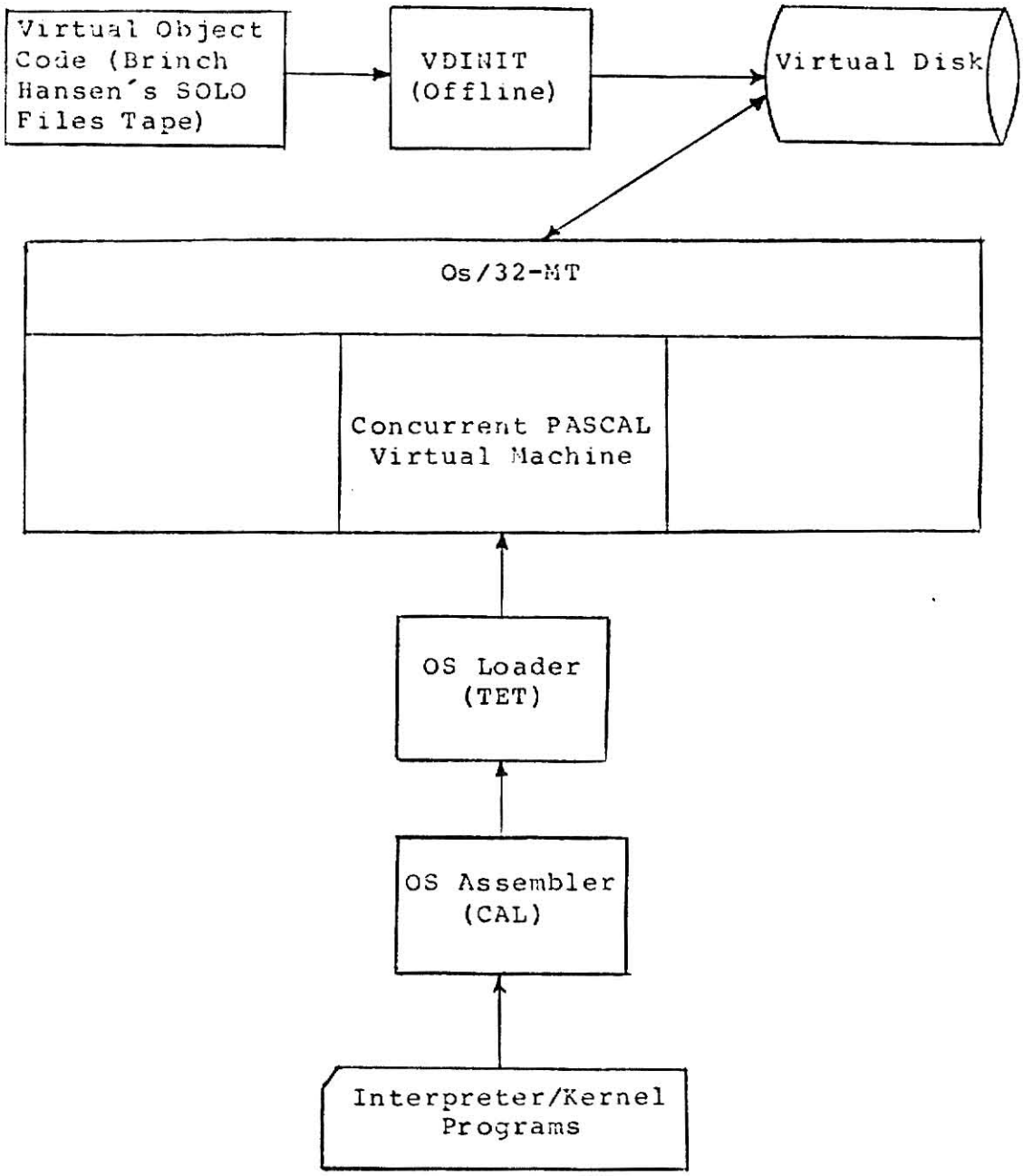


Figure 2.5 Concurrent PASCAL Transportation

This transportation is illustrated in figure 2.5. After using the system for approximately six months, during which time hopefully all bugs were expurged from the Kernel, the virtual code was modified slightly to be more efficiently interpreted on the Interdata hardware, and a microcode routine (the Interdata 8/32 [I2] supports user microprogramming) was added to speed up interpretation. These aspects are also covered in chapter 3. In this section I shall cover the software architecture of this implementation and its impact upon the ensuing discussion.

The Interdata 8/32 implementation was designed to execute in the environment of Interdata's OS/32-MT operating system [I1]. The decision to use the operating system environment as well as to execute as a user (i.e. non privileged) task imposed the following constraints and benefits to the implementation:

a) Segmentation registers (i.e. the Memory Access Controler or MAC [I2]) were not accessable to the implementation,

b) Operating system services (SVC) were available for IO and timing, and

c) system supported segmentation (i.e. reentrant library segment and task common) was available.

The unavailability of the segmentation registers was not a major problem. Segmentation was necessary to the implementation, dictated by the addressing structure of the

virtual code, which was predicated on the address spaces of the PDP-11, but the segmentation was done in software. The differences in the structure of the PDP-11 and Interdata segmentation schemes would have made the use of the MAC cumbersome. This is further explored in chapter 3. The software segmentation is one of the major inefficiencies of the system but a rather difficult one to remedy, as it takes a significant change to the virtual code.

A point of interest in the structure of the implementation is the use of the re-entrant library segment. The Kernel and Interpreter code were written re-entrantly. These programs were then combined to form a re-entrant library segment (RL) as supported by the OS/32-MT operating system. This has several benefits in the KSU system. First, only one copy of the Kernel/Interpreter code need exist for several users of our system. As this code uses approximately 7.5K bytes of main memory, and as many as five users are often active, this amounts to about 30K savings. In addition, the address structure of the virtual machine limits the address space of each Concurrent PASCAL process to 64K bytes, including common (i.e. monitor data, concurrent code space, and Kernel variables). If included in the task partition rather than the library, the Kernel and Interpreter could further limit process space. Finally the re-entrant code more nearly simulates a 'real' machine architecture where the code is implemented in

hardware/firmware, giving insight into the structure of a Concurrent PASCAL machine.

The operating system environment supported the implementation in the following ways. Most important, the OS/32-MT system supported multiprogramming within a task partition. This facility was provided by the ability of the task to accept a trap (interrupt, signal) generated by the operating system, signifying the completion of an event. Proceed IO and timing facilities were available through system calls and generated traps upon completion. The IO facilities were particularly supportive to the implementation. Device drivers did not have to be written, only translation of the parameter blocks as provided by SOLO (using the Concurrent PASCAL IO function) to forms expected by OS/32-MT and translation of status upon completion. Notable exceptions to this simple strategy were the console device and command functions - see chapter 3.

The major interest of the KSU system lies in the aspect of portability. It is an example of the utility of high level languages and abstractions for concurrent programming. Using these techniques an operating system, probably the least portable software product, was not only moved from one machine architecture to another, but was also implemented as a subsystem of another operating system (as in, for example, VM370 [IBM]), on minicomputers and with less than one man-year of effort!

NUC (I7/16) Implementation

The Naval Undersea Center implementation [MB] [CZ] represents a significant departure in implementation strategy from the preceding two implementations. The areas of difference include code generation, Kernel structure and IO facilities.

In the area of code generation, the NUC system includes Concurrent and Sequential PASCAL compilers which output object code for the Interdata 7/16 computer. These compilers were produced by the addition of four passes to the initial five passes of the Hartman compilers for the PDP-11 system [AH1]. The current implementation uses cross compilation (on a PASCAL compiler for the Univac 1110 [MBU]) and produces optimized code for the 7/16 (using 'peephole' optimization [WW]). These compilers have also been transported to the KSU system, allowing cross compilation for the I7/16 on the I8/32.

The Kernel of the NUC system does not follow the design by Brinch Hansen, but does produce the correct architecture at the Concurrent PASCAL level. Significant improvements were made on the Kernel in the areas of timing and IO facilities, as the NUC Kernel supports general timing waits, both periodic (from the point of last call) and interval (from now), and IO device control from the Concurrent PASCAL level. The flow of control through the Kernel seems more straightforward than Brinch Hansen's, as the class concepts

were not used and the logic was programmed in a more 'conventional' manner (that is, more in-line coding, without the use of abstractions for understandability). Whether or not this design is more understandable (and thus more portable) remains for an implementor with initial access to both designs to determine.

The IO facilities of the NUC system merit special attention. Included in the Kernel instead of device drivers is an interpreter called the IO Machine. Programmable from the Concurrent PASCAL level (as arrays of integers containing instructions to the interpreter) the IO Machine supports the transmission of commands and data to and from devices, and allows the process to await termination of requests (i.e. interrupt). The IO machine shelters the Concurrent PASCAL program from the details of transfer (IO bus vs. channel; byte, halfword, or block transfer, etc.). The success of the IO machine in this implementation was somewhat less than hoped for by the implementors, due to the vagaries of the Interdata hardware. The concept, however, remains valid as an implementation technique, and is crucial to the design of an architecture which can support a general purpose operating system (including support for devices not initially implemented) written in Concurrent PASCAL.

Summary

The three implementations of Concurrent PASCAL which

have been discussed provide a background from which certain architectural aspects of a Concurrent PASCAL machine can be derived. The implementations have been presented in enough detail to allow the abstraction of the underlying architectural concepts, without immersing the reader in a myriad of implementation details. In chapter 3 there is an extended discussion of the implementations, centering on the portability aspect of the system designed by Brinch Hansen, and the transportation of that system to the Interdata 8/32.

Experiences with the Portability of Concurrent PASCAL

The programming language Concurrent PASCAL [BH1] was designed for the structured programming of computer operating systems, using the abstract structuring concepts of processes, monitors, and classes. In order for this language and its concepts to be of use in the design and programming of real operating systems, implementations must exist for real computers. Per Brinch Hansen, the designer of Concurrent PASCAL, and his staff at California Institute of Technology have implemented Concurrent PASCAL on the DEC PDP-11 [BH2]. This implementation has been widely distributed both in a version suitable for the PDP-11 and in a version intended to be moved to other computers. The language implementation includes a compiler for Concurrent PASCAL and a compiler for Sequential PASCAL, a subset of Standard PASCAL as defined by Wirth [JW], which are both written in Sequential PASCAL. The Brinch Hansen distribution system also includes an operating system (SOLO) which is written in Concurrent PASCAL and provides a single user environment for the development of both Sequential and Concurrent PASCAL programs. This chapter describes the porting strategy and experiences and the insight gained during the process of moving the Brinch Hansen system to an Interdata 8/32 computer.

Approach to Portability

Several techniques have been proposed or used in moving programs and/or programming systems from one computer to another [PW]. Among these techniques are "standard" languages, macro implementations, and abstract machine architectures. With regard to the Brinch Hansen system, two of these approaches are immediately applicable. Similar to the PASCAL-P implementation system [NA], Brinch Hansen designed his Concurrent and Sequential PASCAL compilers to translate language source into code for an abstract stack machine. The architecture of the abstract stack machine is sufficiently close to that of the PDP-11 to be interpreted efficiently on that architecture yet sufficiently abstract to allow mapping onto an architecture radically different from the PDP-11. In addition to the design of the compiler output, the compiler itself, programmed by Hartman [AH1], was written in a dialect close enough to "Standard PASCAL" to allow another transportation strategy. The multipass (seven passes for the Hartman Sequential and Concurrent PASCAL compilers) structure allows a group with access to a "Standard PASCAL" compiler to use several of the analysis passes of the Hartman compilers and add their own synthesis passes, adapted to the target architecture of their project. This approach has been used in the adaptation of Concurrent PASCAL for the Interdata 7/16 and is currently in progress for the Interdata 8/32 (Ball et al. [MB], [CZ]).

The approach taken in our project was to interpret the virtual code produced by the Hartman compilers directly in Interdata CAL assembly language. The reasons for this approach were twofold. First, we did not at the time have access to a "Standard PASCAL" compiler (although one probably could have been found, since we did have access to an IBM 370). Second, it was felt that the implementation of an interpreter was a much easier task, and manpower (as opposed to computer power) was a premium quantity in our project. This proved to be a valid assumption as we were able to produce a system which was capable of running Brinch Hansen's SOLO system and also several pedagogical systems written in Concurrent PASCAL (see Wallentine [WM]), with a manpower outlay of approximately four man-months.

Language and Run-Time Support

In addition to the decision of language portability strategy, the design of Concurrent PASCAL requires the provision of run-time support. This support is in the form of a Kernel which implements the virtual machine upon which Concurrent PASCAL programs are executed. Among the functions provided by the Kernel are:

a) processor multiplexing--the Concurrent PASCAL program defines process entities, but it is the Kernel's duty to give these concurrent processes the illusion of executing

simultaneously;

b) allocation of main memory to processes and monitors (shared data structures);

c) isolation of the Concurrent PASCAL program from such details as processor registers, interrupts and exception condition faults (e.g., arithmetic overflow traps, etc.); and

d) Input/Output facilities.

The software structure of the Brinch Hansen Implementation is illustrated in figure 3.1.

The design strategy for the Kernel was to provide a more abstract machine at the Concurrent PASCAL language level. As such, the level of abstraction may be debatable from a theoretical viewpoint, particularly with respect to how much policy belongs in the Kernel. For example, it can be argued that processor scheduling (as opposed to the mechanism by which the processor is allocated/de-allocated) belongs at a higher level than the base machine (Kernel) level. A similar point may be made for memory allocation policies. These questions, however, are more concerned with adaptability than portability and will be discussed in the following chapter.

The implementation of the Kernel in our project consisted of reproducing the logic of Brinch Hansen's Kernel in the assembly language of the Interdata 8/32 and providing the I/O facilities as required by the SOLO device drivers.

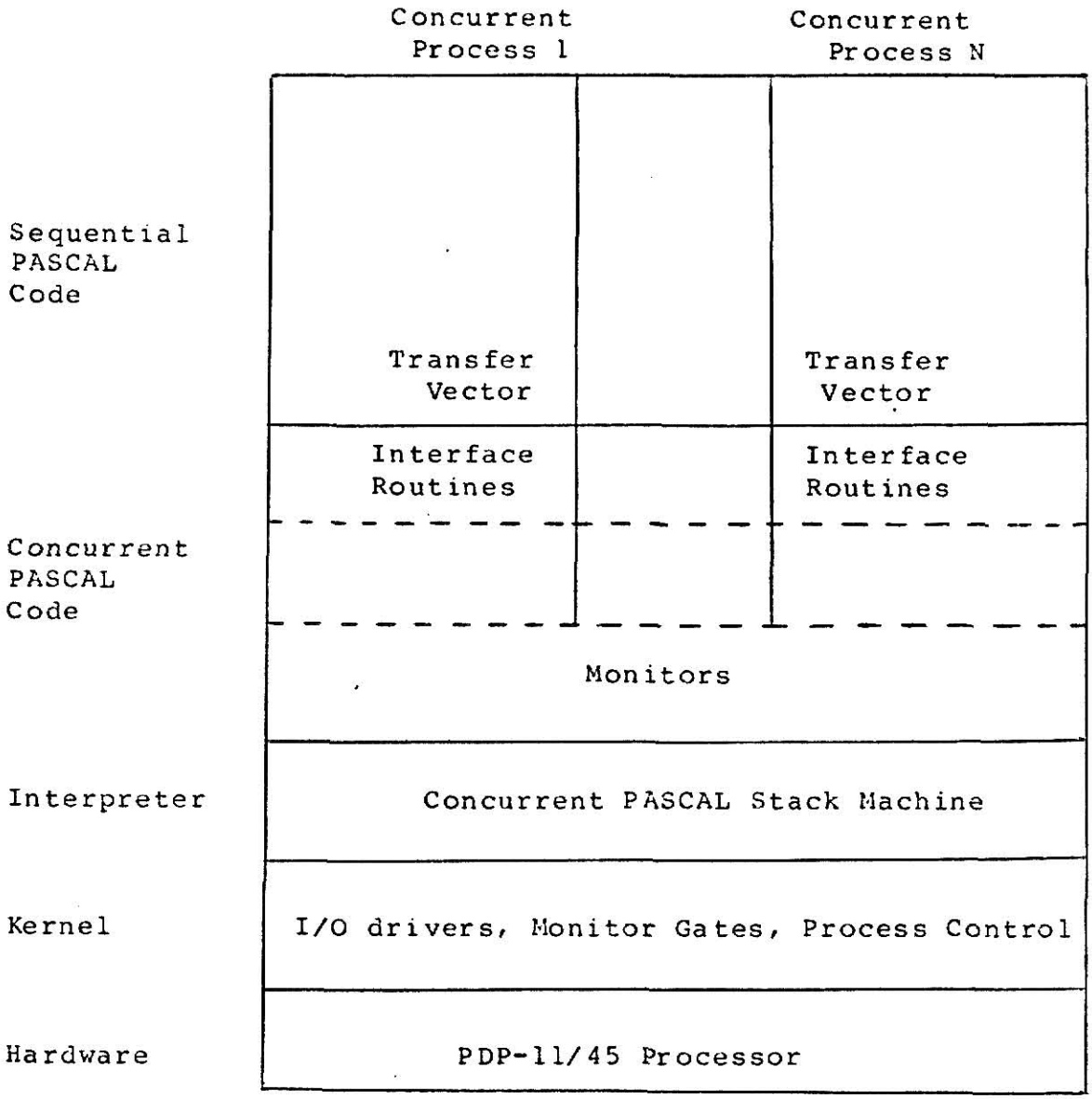


Figure 3.1 Brinch Hansen Concurrent PASCAL Implementation

The logic of the Kernel was presented by Brinch Hansen in PASCAL-like comments to the PDP-11 Kernel. In addition, a very abstract overview was provided in the 'Concurrent PASCAL Machine' manual [BH2]. It seems to us as implementors that an intermediate documentation level (including such things as control flow, calling formats, processing overview by module, and the like) would have been beneficial to the understanding of the Kernel prior to digestion of the code. This level of documentation has been provided for the KSU Kernel and is included in Appendix A.

Virtual Machine Implementation

As a final prelude to the discussion of problems and insights gained in this implementation, we discuss the decision to implement the Concurrent PASCAL system in an operating system environment. The decision itself was dictated by three criteria:

1) the desire to get the system up and running as soon as possible--the use of the operating system allowed for the use of several debugging aids, as well as OS services;
2) the necessity of using the existing OS for other projects involving other languages--doing a stand alone system would have meant scheduling stand alone time not only for development but for use of the system;

3) the unfamiliarity of the implementation staff with writing such things as device drivers and interrupt handlers.

In addition to these factors, several advantages exist for the operating system implementation. First, the most available, if not only, existing operating system written in Concurrent PASCAL at the time of the implementation (completed October 1976) was the SOLO system. This system is a single user multiprogramming system which requires approximately 72K bytes of main storage. With the resources available on our 8/32 (640K bytes, several terminals and two 80 megabyte disks), we are capable of simultaneously accommodating multiple SOLO users. (In practice this has been limited to five.) In addition, users of the other operating system services (COBOL, FORTRAN, editors, assemblers) are unaffected, except perhaps in response time. Users of the Concurrent PASCAL/SOLO system can even take advantage of the operating system services; for example, the Interdata OS Edit program was used extensively in preparing PASCAL programs as the SOLO editor was too crude. The edited versions are then shipped to the PASCAL system (via a virtual card reader facility) for compilation. The structure of the KSU implementation is illustrated in figure 3.2.

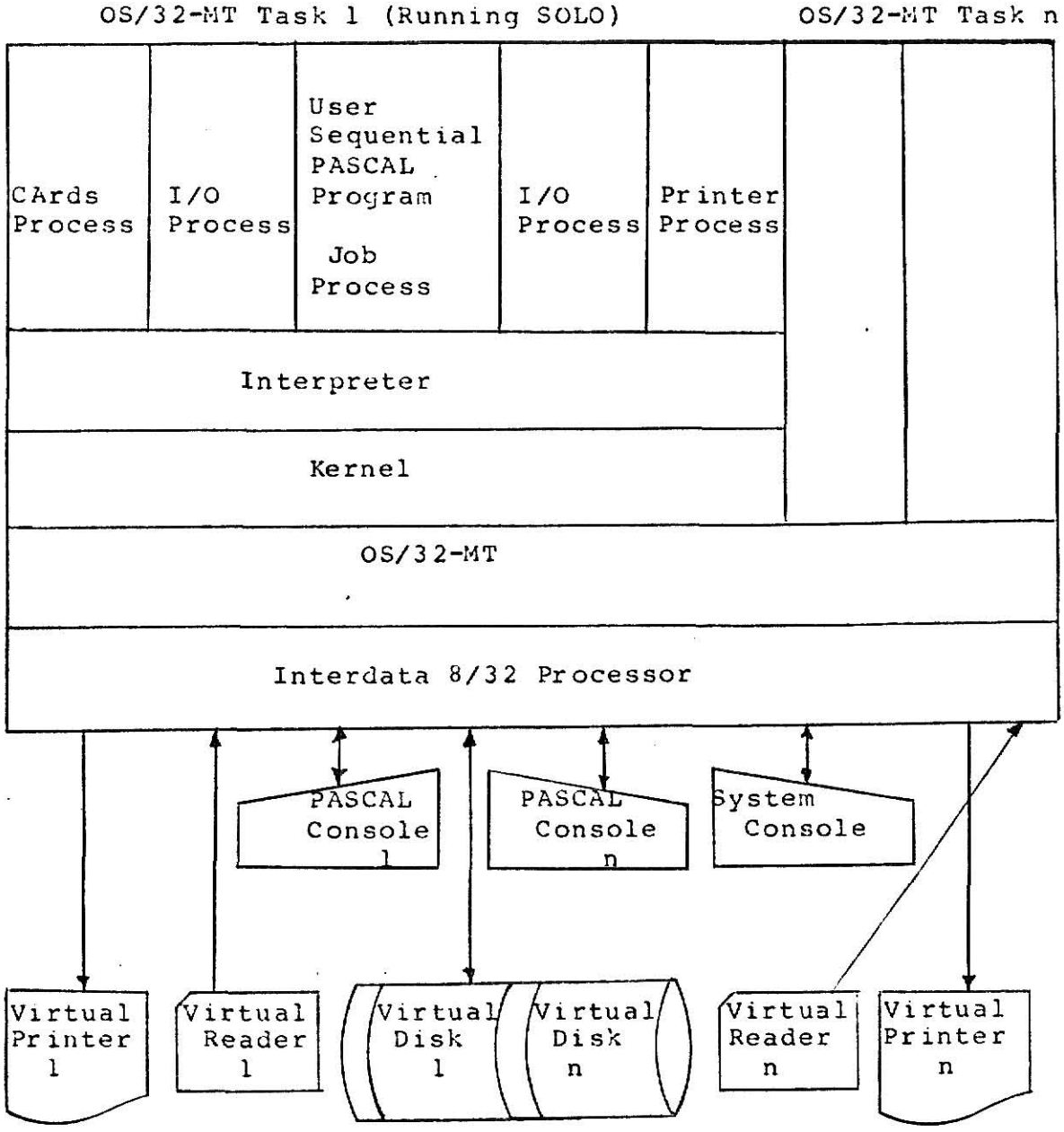


Figure 3.2 KSU Concurrent PASCAL Implementation

Details and Problems of Implementation

In the following sections we discuss some of the problems and fine points of the implementation. The discussion of problems is intended to point out the problems that we encountered and not to fault Brinch Hansen's system. We are very happy with the performance of the Concurrent PASCAL/SOLO system in general, and we are particularly pleased to have a system which is easy to understand and use.

The problems to be discussed include address space mapping, internal representation and virtual code interpretation, and devices and I/O structure. In addition to these discussions of the Concurrent PASCAL system, we discuss some problems generated by the use of Interdata's OS/32-MT operating system [I1].

Address Space Structure

Probably the largest stumbling block in our implementation process was the addressing scheme of the PDP-11. Given a 16-bit internal machine, the PDP-11 designers extended the memory availability by mapping or segmentation registers which provided for multiple 64K byte address spaces, up to a total of 256K bytes of addressability. Brinch Hansen, in the 'Concurrent PASCAL Machine' manual [BH2], complains of this structure, preferring direct addressability. However, he uses it to

implement a 72K byte system (SOLO). Each process in a Concurrent PASCAL program for the PDP-11 has an address space of 64K bytes which is divided into two logical segments: the common segment (consisting of concurrent code, monitor data space, and Interpreter) and the private segment (consisting of the process data space -- including sequential code space). The Kernel has its own address space which includes all Kernel variables. This structure is illustrated in figure 3.3. Although it is known at compile time which segment addresses are in, the compiler generates code in such a fashion that segmentation must be resolved at run time (in particular, pointers are not differentiated from integers at the virtual code level). In our implementation, this scheme caused much consternation. The final approach taken was to simulate the two segment address schemes in software. This is done by comparing any 16-bit address formed from memory with a register containing the largest common address. If the address is larger than this common top, an additional base register is added to form the final address. This is illustrated in figure 3.4. In addition to using two general registers and several instruction cycles, the approach is not elegant. Had the compiler distinguished between integers and pointers, and had provision been made for larger addresses in the stack, the problems that we faced would not have occurred.

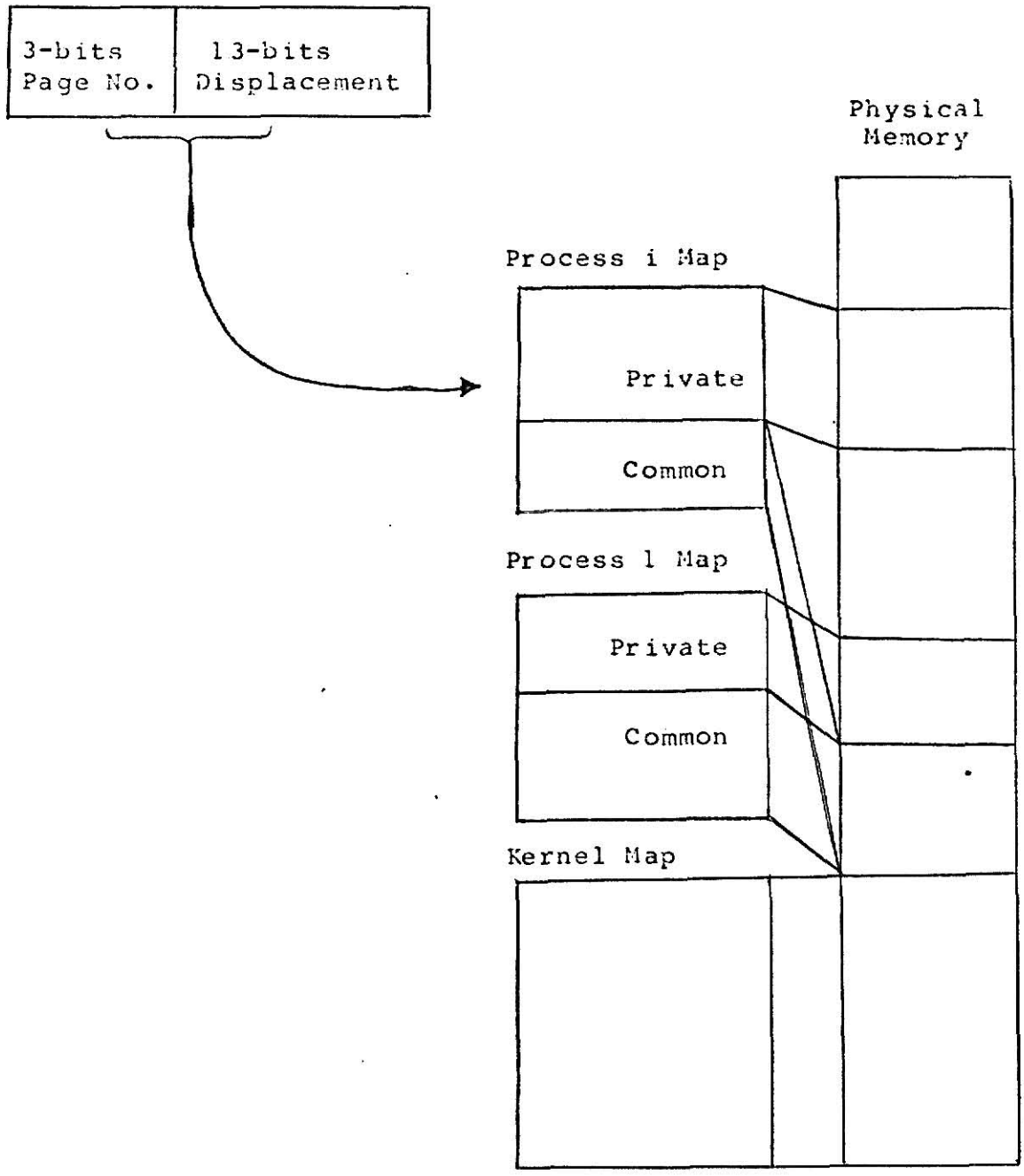


Figure 3.3 PDP-11 Segmentation Structure

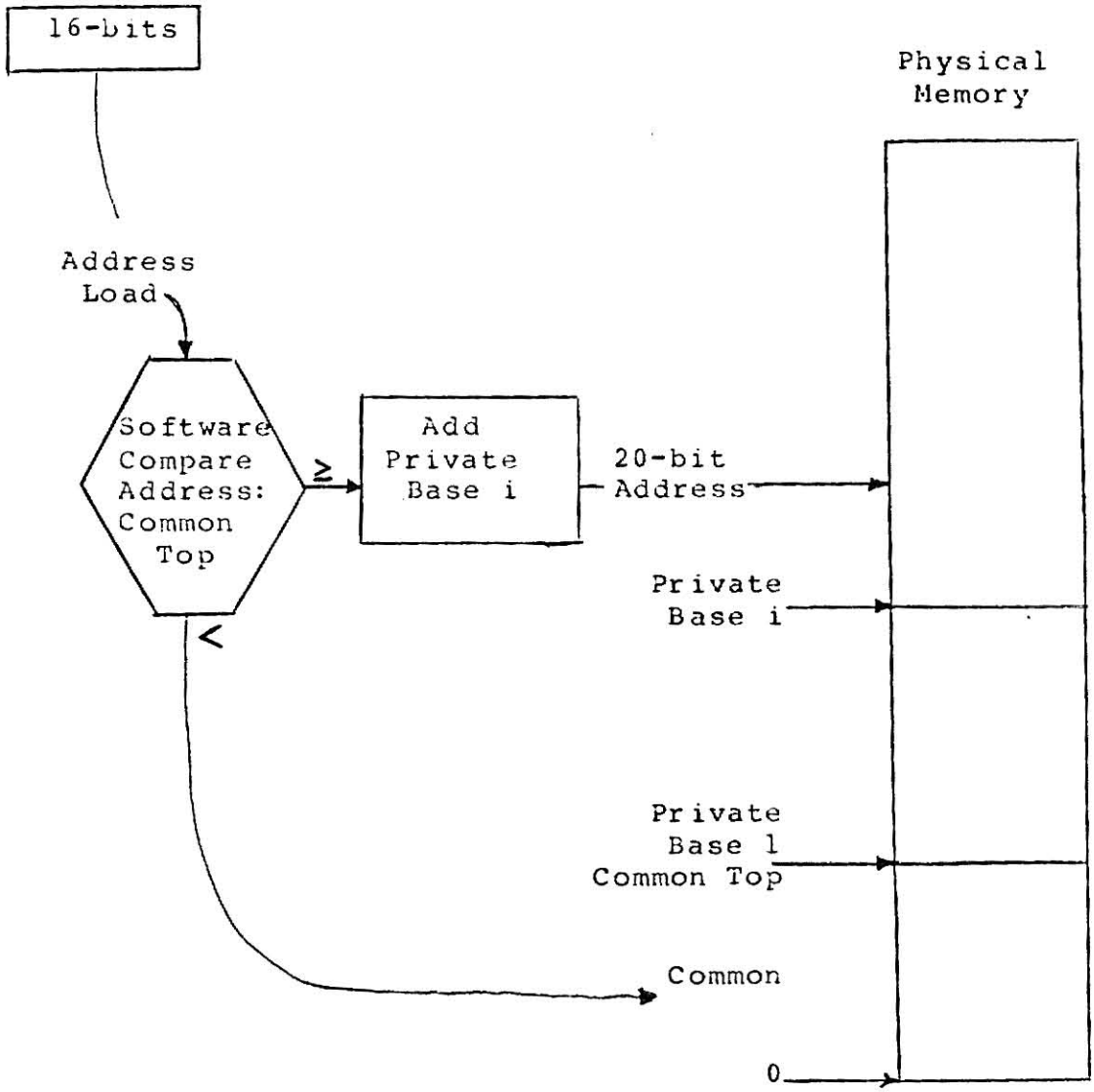


Figure 3.4 I8/32 Software Segmentation

We then would have been able to make use of the large direct addressability, the absence of which Brinch Hansen laments.

Although the Interdata 8/32 also provides segmentation registers, we did not have access to them, since we chose to run our system as a user task. However, the structure of the I8/32 segmentation registers is such that using them for the PDP-11 scheme is very costly. The Interdata provides multiple 64K segments which are distinguished by the top 4 bits of a 20-bit address. Mapping into this structure efficiently from a structure with multiple segments within a 16-bit address space seems very difficult (i.e. many instruction cycles would be required for "twiddling bits".) This mapping is illustrated in figure 3.5.

Virtual Code and Internal Representations

As a prelude to the discussion of the virtual code interpretation, we quote from Brinch Hansen in the 'Concurrent PASCAL Machine' Manual [BH2]:

> In implementing Concurrent PASCAL we followed one simple guideline: A computer should only do obvious things and should do them well. Where compromise was needed we firmly put simplicity first, efficiency second, and generality third. Like any other design rule it needs no justification other than the success that it leads to in practice.

With respect to the portability of the system, we might request that the last two priorities be reversed or at least given equal weight.

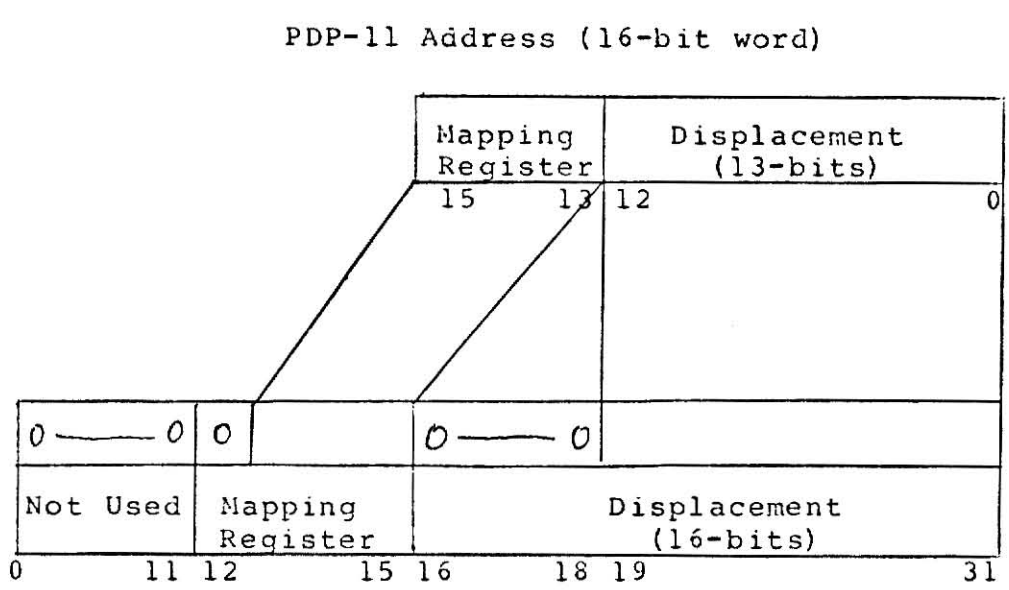


Figure 3.5 PDP-11 to I8/32 Address Mapping

The problems we encountered in relation to the virtual code stemmed from two sources. The first problems were related to the internal representations generated for the PDP-11, and the second to the foreignness of the stack architecture as applied to the Interdata 8/32. It must be noted that the nature of these problems is related to the efficiency and/or elegance (generality) of the system running on the 8/32, and as such are not inconsistent with the goals set by Brinch Hansen.

The two particular peculiarities of the PDP-11 which were a nuisance to our implementation were the ordering of bytes in strings and the ordering of bits in a set. With regard to bytes in a string, the PDP-11 devices output the low order byte of a word before the high byte, and byte addressing reflects this structure. In order to correctly interpret the virtual code as provided by Brinch Hansen without expensive hand decoding, the Interpreter was forced to simulate the byte addressing. However, it was decided that the internal representation of the Interdata system should be preserved on I/O whenever possible (otherwise all I/O would need to be buffered in the Kernel). The upshot of this scheme was that ASCII files on disk and tape ended up with bytes swapped, a minor inconvenience at first which later forced us into changing the internal representation, as Brinch Hansen suggests may be done. Once the compiler is available, the changes themselves are relatively minor and

consist of changing the compiler to exchange bytes in the representation of strings and then recompiling all the programs. However once this was done, Interpreter instructions had to be rewritten, certain device drivers in the Kernel had to be rewritten, and the internal structure of the virtual disk had to be changed. In essence, the process consisted of minor modifications to several parts of the system, not the first thing that an implementor wants to do when a new system is brought up. Delaying the changes, however, allowed several programs to be written which included "byte swapping routines". These routines later had to be deleted. The gnattiness of the whole process suggests that a version of the "SOLO FILES" (that version of the system distributed to non-DEC users) created by the PDP-11 system with internal strings (i.e. within code files) in "natural" (i.e., left to right, non-swapped) order would cause many implementors less pain. This is the current representation in our system.

Associated with the internal representation of strings is the internal representation of sets. Bits are numbered right to left by 16-bit word. Brinch Hansen makes the statement that it is unnecessary to simulate this structure, as the compiler only stores the null set (all zeros) in the code and builds all other sets. Since the Interdata 8/32 architecture includes test bit and set bit instructions which number bits left to right, we used these instructions

to gain an efficient set manipulation scheme. However, upon compiling our first program and attempting to run it, we found, to our dismay, that sets were used by the compilers in creating tag fields for variants, causing all our programs to terminate with a variant error. We modified the set instructions for compatibility with the PDP-11 implementation at a considerable loss of efficiency and simplicity. This was also revised later, along with strings.

Our final comment on code interpretation concerns the "threaded code" approach to interpretation [JB]. The PDP-11 instruction set includes a useful instruction which combines an indirect jump with an autoincrement (i.e., goto store (store(q)); q := q+2;). Combining this with a decode vector in the low addresses of the Interpreter, the PDP-11 implementation performs interpretation at the cost of one instruction (three storage cycles). The equivalent interpretation decode required four instructions on the 8/32. In addition, space and the lack of an adequate macro facility guided us to code the decode sequence once and then branch to it after each virtual instruction, rather than code the whole sequence inline each time. At a later point in the implementation, a microcode routine was added which emulated the PDP-11 instruction. The resulting real (i.e. wall clock) time gain in performance was in the range of 17%. This is at present the only microcoded routine in the

system. It is possible that the microinterpretation of some grossly inefficient routines might further improve performance, but not so drastically. For example, structure copies and compares are currently coded as a software loop, including approximately three instructions per halfword (two bytes) compared or moved. If these routines were microcoded, this overhead could be eliminated. However, the rate of occurrence of structure manipulations in the virtual code would also need to be considered in computing the performance increase. The decode routine is, of course, executed for every virtual instruction.

I/O Structure - Devices

Our final section about the Concurrent PASCAL system itself, concerns the I/O structure of the system. The comments in this area encompass the human readable devices (card reader and printer), the console device, and the fixed length (page) devices (tape and disk).

Some of the problems occurring with the human readable devices were alluded to in the previous section on internal representation. In order to insure that no programming error would occur because of exchanging bytes in the user's space, the (virtual) line printer device buffered its output in the Kernel, exchanging bytes as it transferred from the user's space to the buffer. The (virtual) cardreader device used the user space for its buffering, first reading a card image

into the user space then exchanging the bytes. As noted before, all this superfluous code had to be deleted later when the internal representation of strings was changed.

The console device implementation turned out to be an atrocity. All other devices in the system were buffered and matched nicely the supervisor call I/O facilities which were used in the operating system environment implementation. The console as implemented in the PDP-11 version was a three state (passive, reading, writing) byte-by-byte transfer device. This meant simulating this scheme with buffering in the Kernel. The code to do this was written twice and still turned out to be unsatisfactory from the user (Concurrent PASCAL) level, as most users did not know the details of the buffering and were distressed when their console I/O was erratic because of the absence and/or presence of line feed (LF) characters. A buffered line-at-a-time transfer strategy would have been much more understandable to all concerned.

The final remark with respect to devices concerns the page oriented devices. This is understandable for disks where the physical organization often dictates transfer lengths; however, it seems an arbitrary restriction when applied to tapes, an inherently variable length medium. The fixed lengths caused no problems in the implementation. However, the lack of a truly variable length device hampers the adaptability of the system. We have recently added a

variable length record capability to the Kernel and will implement PASCAL drivers for these types of transfer in the near future.

Operating System Idiosyncracies

This final section on our implementation of Concurrent PASCAL describes the major problems generated by our decision to implement Concurrent PASCAL in an operating system environment, in particular the problems with the Interdata OS/32-MT operating system [I1]. In general, the operating system was supportive to the development. For example, except for console I/O as described above, all I/O consisted of a simple translation of the Concurrent PASCAL I/O function parameter block into an OS/32-MT parameter block. One must, however, take the bitter with the sweet. In the case of OS/32-MT, the bitter came in the form of the file system and device drivers.

One of the chief factors in using the operating system environment for implementation was the possibility of simultaneously activating several SOLO systems on the I8/32. This coupled with the obviously limited physical devices such as card reader and line printer (we have one of each) suggests the use of virtual device facilities (as in VM/370 [IBM], for example), using disk as the storage medium. Several facets of the OS/32-MT file system conspired against these facilities ever becoming available. Of the three file

structuring methods available (contiguous, indexed, and chained) only contiguous files support overlapped I/O and computation at the user level (proceed I/O). As such I/O is implicit in the nature of Concurrent PASCAL, using either of the other two methods defeats the purpose of the system. With our rather limited and fragmented disk storage, it is often difficult to find a large contiguous space for virtual devices. In addition, when other file structures were used as virtual devices, we experienced some "system hangs" giving the impression that the I/O and continue instructions were not being correctly simulated by the file system. As this inconsistancy was never pursued, it is possible that other errors may have given this illusion. (We were plagued in the development stage by hardware errors both in memory and on disk.) The point remains that using these other file structures defeats the purpose of Concurrent PASCAL; and until a suitable alternative arises, such as a new revision of OS/32-MT, we will continue to multiplex the real hardware by computer room consensus, although this sometimes results in intermingled listings and hit/miss card decks. Since our implementation of Concurrent PASCAL was accomplished in conjunction with a project concerning computer networks, a network spooler machine may well provide the alternative.

The final point concerns the OS/32-MT drivers. For some peculiar reason, the execution of a command function to a device does <u>not</u> generate a completion signal. For this

reason and because of the desire to maintain the principle of concurrency, command functions are monitored by polling the device status whenever the Kernel detects that there is an outstanding command. The particular (SOLO) device impacted by this scheme is the tape device. As it happens in the SOLO system, all tape command functions are performed without other concurrent execution. As such, polling takes place only on the occurrence of timing signals (on the order of .1 second), making tape movement extremely slow. In practice, users tend to use the OS utility (OS COPY) to effect skipping over files on tape. The rewind function, however, does work acceptably.

Conclusion

There are several comments applicable to the success of the porting of Concurrent PASCAL to the Interdata 8/32. First, of course, is the design of the system by Brinch Hansen. It is clear that a system requiring so little effort to be moved between vastly differing computer architectures must have been well designed from the outset. In addition, with a single exception (sets and variants), all of the problem points were at least mentioned (if not emphasized) by the implementation notes accompanying the distributed system.

The virtual addressing mechanism used in this implementation is probably the least portable aspect of the

system and will cause the most pain to implementors using systems which support addresses that are larger than 16-bits, or those without mapping structures for their 16-bit addresses. (Remember that SOLO, in its distributed form, uses more than 16-bits worth of addressability!) The device structure of the console device will be a detriment to those implementors using existing I/O drivers which do not support byte-at-a-time transfers. The adaptability of the system to diverse uses is also somewhat limited by the lack of non-page transfer devices. These facilities may be easily added to the system (although the policy of static type checking must probably be violated if truly variable transfers are allowed). The problems of the internal representation of the PDP-11 architecture have been solved in our implementation, and those users with access to our "SOLO files" tape should not encounter those problems in their transportation.

The implementation of this system in an operating system environment was a success, especially with respect to multiple users. It is recommended to implementors with a medium to large scale system who would like to use the SOLO system for research but cannot allocate a whole machine for this purpose. The OS/32-MT operating system provided a generally supportive environment for the development and use of the system. The implementation process itself was pleasant and rewarding and resulted in a tool of significant

value to us for teaching and research.

Architectural Base Considerations

In this chapter we will explore various facets of Concurrent PASCAL and suggest certain architectural features to support the language. The goal of this discussion is to provide an environment in which Concurrent PASCAL may be seen as a viable tool for general purpose operating system design, while maintaining the understandability of the systems so designed. Brinch Hansen proposes that with the advent of Concurrent PASCAL, small special purpose systems may become more viable (that is, economical, reliable, understandable, etc.) than present day large scale systems. This view has not necessarily met with complete acceptance [JS]. However, the idea of special purpose multiprogramming systems running concurrently in the environment of a virtual machine monitor (VMM) on a medium to large scale machine, all written in Concurrent PASCAL does have interesting possibilities. It is support for this type of system that underlies the remainder of this chapter.

Levels of Abstraction (Hierarchy)

The Kernels for Concurrent PASCAL described in the preceding chapters consisted of a single level of software support on conventional architectures for concurrent programming abstractions. This single level of software combined the policies (priority for processor, memory

allocation, etc.) and mechanisms (process exchange, interrupt handling, etc.) of the support into a single entity. Given the precept of providing architectural support for multiprogramming, it seems reasonable to separate mechanism -- which can conceivably be implemented in hardware/firmware -- from policy which should be expressed in software. This separation creates two levels of support below the concurrent programming level, and is illustrated in figure 4.1.

The mechanical aspects of multiprogramming support include such functions as saving and restoring the process state on the occurrence of a process switch (note that this assumes that the architecture knows the representation of a process state not contained in the processor registers) and the transformation of an event signal (interrupt) into a state (blocked to ready) transition of the associated process. Remember that in the model of multiprogramming which supports Concurrent PASCAL, all events must have associated processes to await their occurrence. Policy aspects include the short-term scheduling of processes to resources such as processors, peripherals, and memory. These could conceivably include provisions for multiple processors, virtual memory, capability-based architecture, etc. Remember that at the policy level we are still "under" (supporting) Concurrent PASCAL, so that the restrictions are to provide the correct architecture to that level, as has

been described in the preceding chapters.

The policy level software, by nature, is to be transparent to the Concurrent PASCAL programs and is to be implemented on the non-policy mechanisms of the hardware base. In addition, as verified by the unanimous consensus of the implementations, these functions execute as non-preemptable routines, i.e. basically sequential in nature. As such, a machine dependent (for the purpose of hardware interface only) sequential high level language (such as Sequential PASCAL or EUCLID [BWL]) would be well suited for this level, allowing queueing strategies and fault handlers to be written at a high level and subject to later modification (for experimental, instrumenting, or "tuning" purposes). Using this two level "kernel" scheme, it is possible to provide the correct (virtual) architecture for a Concurrent PASCAL program at the next level in the hierarchy. Three further benefits are gained.

1) State manipulations are handled (efficiently) by the base machine hardware.
2) policies underlying the Concurrent PASCAL architecture are themselves programmable (preferably in a high level language).
3) the underlying language concepts of Concurrent PASCAL need not be modified.

Assuming a Concurrent PASCAL program of n processes, we now have the level structure depicted in figure 4.2.

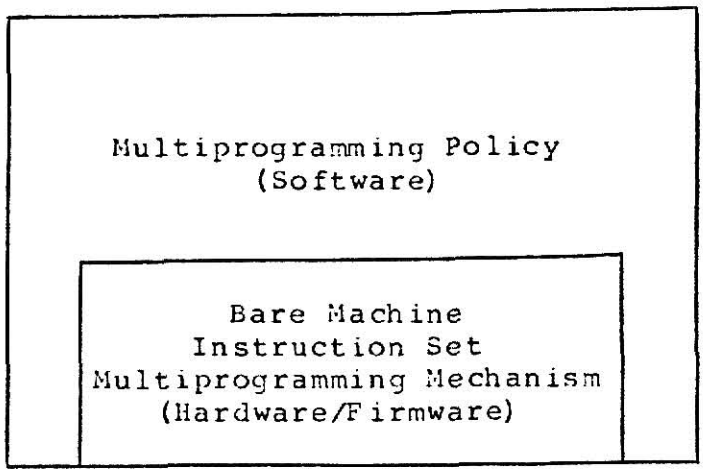


Figure 4.1 Levels of Architectural Support

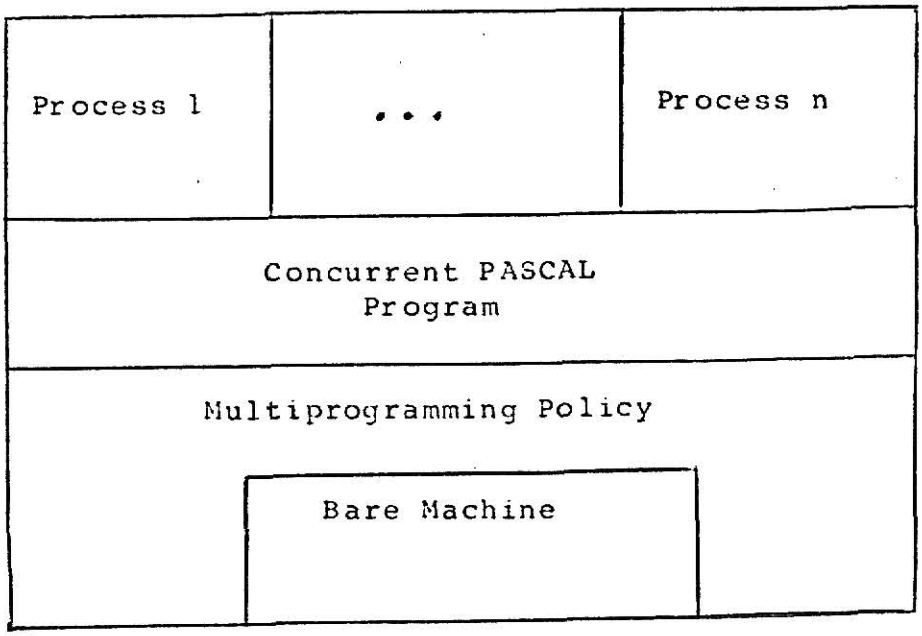


Figure 4.2 Concurrent PASCAL Program

VMM Design Considerations

The three implementations all support the loading and execution of Sequential PASCAL programs by a Concurrent PASCAL process. It seems a logical step to allow the loading, execution and preemption of Concurrent PASCAL programs by processes. This extension to a virtual machine monitor (VMM) design would allow for much flexibility in system design. For example, the existing SOLO operating system could be used as a submonitor system by several users, given the implementation of a VMM in Concurrent PASCAL. The VMM might also provide for the sharing of the re-entrant SOLO code by the various users, as well as providing other services (through the extension of the prefix concept used for Sequential PASCAL to Concurrent PASCAL), such as a data base system, message system, etc.

One problem with VMM designs in present day architectures is that the architecture presented to the virtual machine level is too "low" (i.e. close to the actual machine hardware) for efficient use of multiprogramming. For example, when OS/360 is running as a virtual machine in VM/370 [IBM], and VM encounters a page fault, the entire virtual machine is blocked, even though it is likely that a user program was executing in OS, and there are in fact other OS partitions awaiting execution which could run. This stems from the fact that the underlying architecture is unaware of the state of multiprogramming. It is desirable

that the Concurrent PASCAL levels of architecture should not suffer from this kind of inefficiency, as such the underlying architecture needs to be aware of the existence and representation of the multiprogramming state. Let us consider the impact of VMM design upon the architectural base.

In the implementations of Concurrent PASCAL previously described, the kernels controlled processes which were often linked by threads of process queues. This seems a suitable structure for such a single level system. Now with the possibility of multiple levels, a more complex structure seems necessary. First let us consider the representation of the process structure of the system. This structure is illustrated in figure 4.3. It seems natural to represent this structure as a tree where descendant nodes denote sub-processes of the parent process. The root node of the tree would then represent the "initial process" of the lowest level (level 0) Concurrent PASCAL program. The mechanical portion of the supporting architecture should support the maintenance of this dynamic tree structure, creating and deleting nodes, and its associated threads. The policy portion can then use the structure in determining scheduling and allocation. Note that if virtual memory facilities are provided by the policy level, a fault in one process at level n need not block levels n and above (as in the VM/370 example) but only the faulting process.

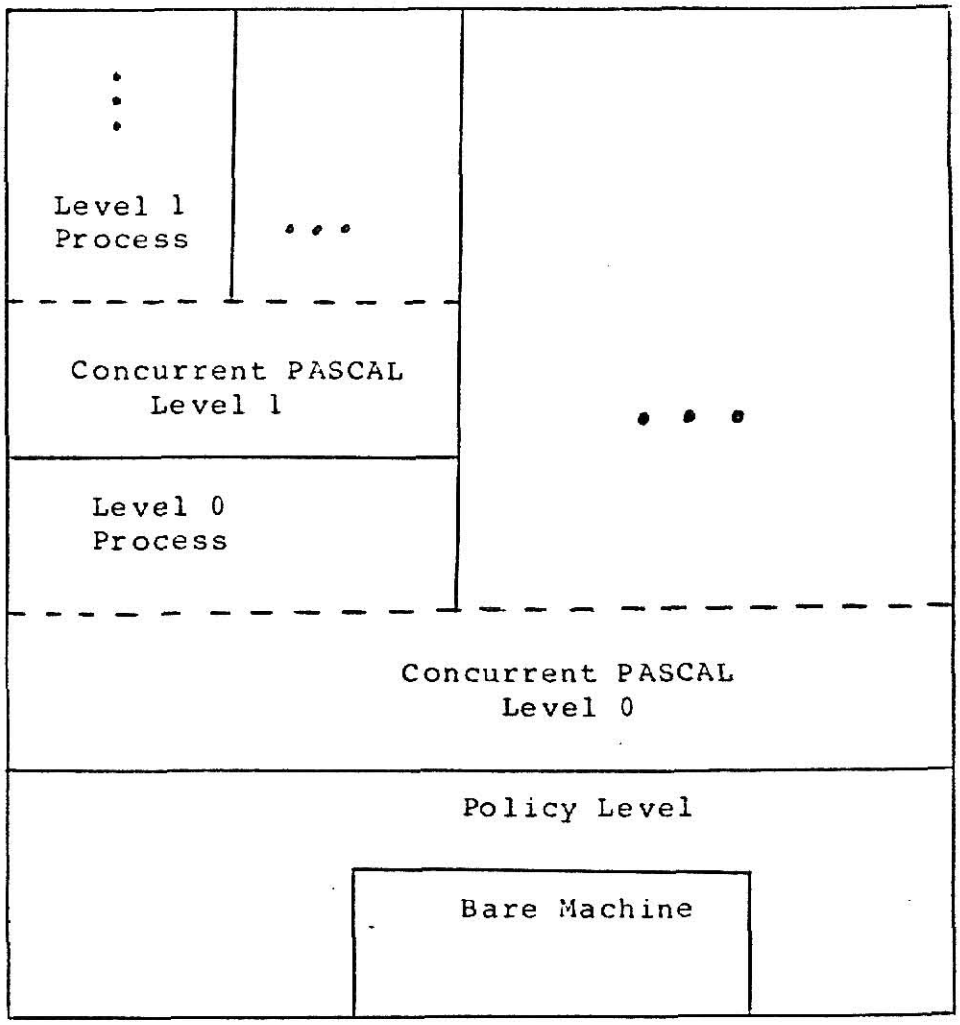


Figure 4.3 Multiprogramming State

This structure allows the policy level much flexibility in resource allocation, particularly processor allocation. For example the level 0 Concurrent PASCAL program might be provided with three priority classes: real time, interactive, and batch. If real time or interactive (sub) processes become unblocked, batch processing could be preempted. A purely timesharing system could be provided by timeslicing level 0 and allowing higher levels priority scheduling if desired. Different strategies for allocation could be experimented with at the policy level without changes to other levels, due to the nature of the Concurrent PASCAL architecture (assuming, of course, that the policy level software is correct). Threads of "ready lists" can be run through the tree in a fashion suitable for the queueing system being used and processes can still be blocked in various ways as in a single level system. This structure is represented pictorially by figure 4.4.

State transitions (Control Model)

The preceding examples show the structure of possible systems which could be implemented within a Concurrent PASCAL architecture. This structure may seem somewhat complex for cost effective architectural support, however, a discussion of the state transitions upon which this structure is based will help to clear up the actual necessary support.

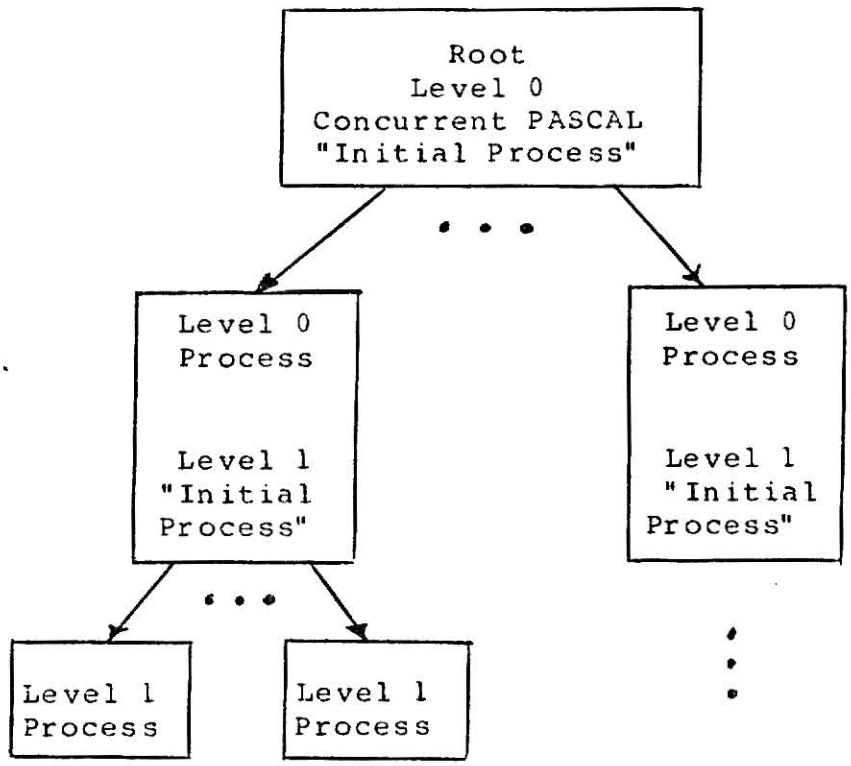


Figure 4.4 Representation of Multiprogramming State

As we have seen within the Concurrent PASCAL implementations, processes become blocked on Concurrent PASCAL queue variables and devices, which are essentially extensions of Concurrent PASCAL queue variables. The extension to multiple levels of Concurrent PASCAL has not changed this function. The occurrence of an unblocking condition (continue of queue variable, device interrupt) should activate the architectural support to reflect the transition from blocked to ready. We provide the architectural base with the knowledge of this transition in the following way. Each process record (PCB) contains reference to the head of the "ready thread" to which its state transition will cause it to be appended. The actual reference can be determined by the policy module at blocking time. The architecture, knowing which thread and the organization (e.g. fifo) of the thread, should be able to easily effect the state transition. This is illustrated in figure 4.5.

Implementation

A design level encoding of a possible implementation of the scheduling ("ready") structure of a Concurrent PASCAL architecture is provided in appendix C. This algorithm combines multiple levels of priority or round-robin (timesliced) scheduling using a tree of "readythreads". The algorithm is designed to be suitable for hardware/firmware

realization, with the underlying policy determined by the structure of the "ready tree" (assumed to be built dynamically by the policy level during system execution). At the process level, the organization of a ready list is first-in-first-out (FIFO). It may be desirable to structure the system such that processes at this level are cooperating (such as the processes in a single SOLO system) and that the implicit scheduling of that cooperation determine scheduling at that level. Because of the large number of processes to be expected in a Concurrent PASCAL VMM implementation, the level structure was added as a mechanism of localization. Actual processor scheduling is accomplished by systematically traversing the levels upon the occurrence of events. The "set method" of determining the next node in the progression seems well suited to hardware realization, both in efficiency and structure (see function return_first_bit -- Appendix C). The level structure also has the property of being exponential in capacity while linear in overhead. Thus, a three level scheme (analagous to Brinch Hansen's three priority classes) could handle x**3 "lists" of processes (where x is a suitably chosen value of the number of descendant nodes - 16 would not seem unreasonable) with essentially little more overhead than Brinch Hansen's algorithm (ready.reschedule). An example "ready tree" is illustrated in figure 4.6.

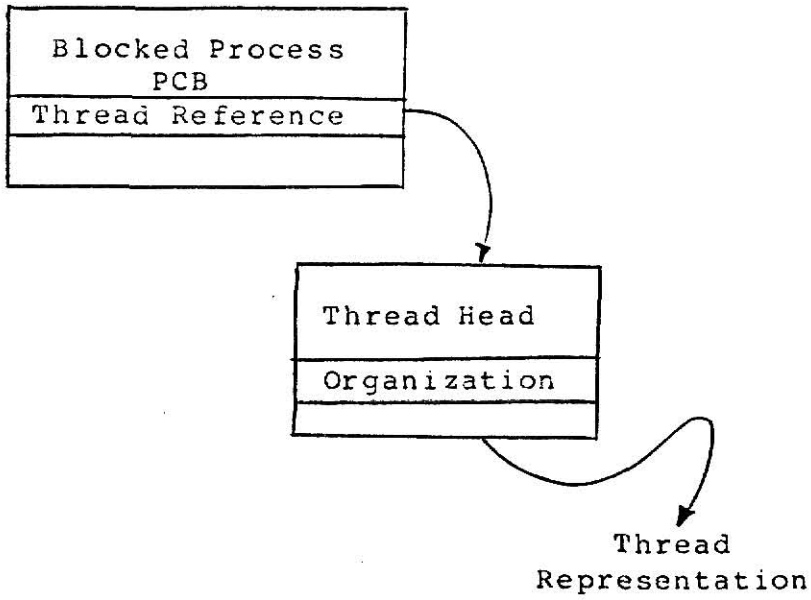


Figure 4.5 Blocked Process Structure

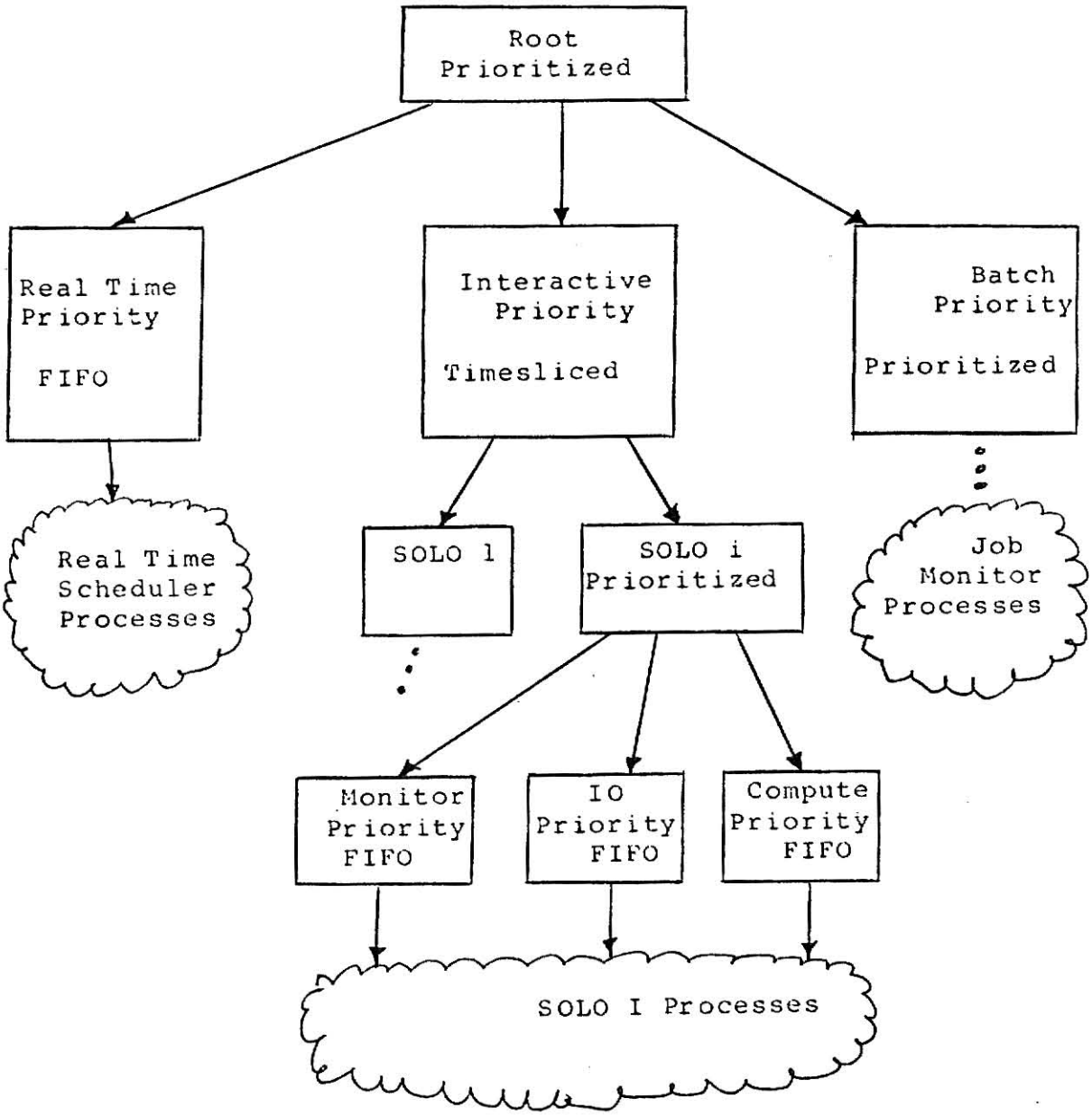


Figure 4.6 Ready Structure

We will now take a brief "walk-through" of the scheduling algorithm for the example in figure 4.6. Assume that all real-time processes are blocked but that some interactive processes are ready and assume further that the current interactive "timeslice" is allocated to SOLO I which has both some slice left and some processes ready at monitor priority. We ignore for the moment the current running process as well as the reason for activating the schedule procedure in the first place. The schedule procedure follows in both intuitive and concrete (PASCAL) forms:

```
procedure schedule(last_process_running)
start
  update time slices for last_process_running
                (using elapsed time timer)
  starting with root node
  traverse the tree
     for each non-leaf node do
        case organization of
           priority:
              next node := highest priority
                           ready child
           timeslice:
              Unless (there is slice left and
                      current child is ready) do
                 pick new current child (starting from
                       last current child)
                 set slice to max for new current
                           child
              end do
              next node:= current child
        endcase
        if nobody ready at this level then idle
          "assertion nobody ready in system"
     end for each non-leaf node
  end tree traverse
  process chosen:= first in FIFO of leaf chosen
  if process chosen is not current process then
     process exchange
  set elapsed timer to zero
return.
```

```
procedure schedule(last_process: processptr);
var level: readyptr; index: indexorzero;
    picked: processptr;
begin
  update_slices(last_process, interval_timer);
  level:= root;
  curslice:= absmaxslice;
  while level^.typethread = lists do begin
     with level^ do
     case organization of
        priority: index:= return_first_bit(0, readybits);
        timeslice: begin
           if (sliceleft <= 0) or not
               (currentindex in readybits) then begin
              currentindex:= return_first_bit(currentindex,
                                    readybits);
              sliceleft:= maxslice;
           end;
           index:= currentindex;
           curslice:= min(curslice, sliceleft);
                 "min function not defined herein"
        end "timeslice"
        end; "case"
        if index = 0 then "idle"
             "assertion: nobody ready in system"
        else level:= level^.children[index];
  end; "while"
  picked:= peek(level^.ready);
  if picked <> currentprocess then
     process_switch(picked);
  interval_timer:= 0;
end; "schedule"
```

First the scheduler updates the time slices using the elapsed time (interval_timer) for the last process executed. Next, starting from the root level, we traverse the tree (using a while loop). The root level is organized by priority and as such an index is computed for the first ready sublist (using return_first_bit(0, readybits);). It is assumed that the child lists are arranged in priority order. In our example, there are no real-time processes ready, so the index returned will be 2 (interactive

priority). Thus children[2] will be chosen for the next while loop progression.

Interactive priority is organized by timeslicing. As specified in our assumptions, currentindex = I, sliceleft > 0 and SOLO I includes ready processes (signified by I in readybits). As such, the next selected index will be I. (If the sliceleft <= 0 then SOLO I would have used up its slice and a new child would be chosen. Similarly, if not I in readybits then no SOLO I processes are ready, and thus another child would be chosen.)

Similarly the next while execution (for SOLO I) in this example is prioritized and as such index is set to 1 (monitor priority) and children[1] is selected. This being a process list (leaf node), the while execution is terminated and a process selected from the FIFO. Process exchange (procedure process_switch) takes place if this process is not the current (running) process.

The schedule procedure is called whenever a state transition (e.g. blocked-to-ready, running-to-blocked) occurs. In addition, these transitions cause the update of the ready state (setting or resetting bits in readybits as necessary). If this algorithm were realized in microcode, for example, with the ready state in main memory and a hardware realization of the find_first_bit function, the overhead for each level transition would be at most nine memory references. This occurs in the case of timeslicing

where all decisions have to be made and all variables changed. The average case should be significantly less (exactly three for all prioritized levels). Three to four memory references per level are also necessary for propagating time slices and ready bits up the tree. In the example system of figure 4.6, the maximum scheduling overhead in memory references per event is less than fifty (50). This is to be compared with the overhead just to save/restore a sixteen register set (thirty two memory references)!

This example and walk-through were designed to show the utility of the scheduling mechanism. Obviously, not all scheduling strategies can be realized by this single algorithm. However, it does seem flexible enough to suit a wide range of purposes (real-time, interactive, and batch) yet simple enough to be realized in hardware/firmware. More complex strategies may be realized through the interaction of the policy level software with its associated overhead. This is analagous to conventional interrupt handling.

For the control specification we have shown the various support functions that reflect the nature of multiprogramming. We now turn to the data model to find what other support may be desirable in the architectural base.

Data Model

The obvious data model for a high level structured language like Concurrent PASCAL is the stack machine. This is reflected in the design of the virtual code in the Brinch Hansen system. The stack mechanism that will be described below is not significantly different from many present day stack machines (particularly the HP3000 [RB]), however, with respect to multiprogramming, the use of the high speed (register) element of the stack is significant.

A self-regulating stack architecture is described, which is a consistent extension of existing general register set architectures. The generalizations allow the compiler writer to let the hardware manage stack temporaries, while allowing general registers for the other optimizations and design features (e.g., frequently used constants/variables, displays in a block-structured language, etc.).

Motivation

The motivation for the design of a stack architecture is basically twofold. First is to provide the compiler writer with an expression stack for the evaluation of temporaries. The use of such a stack decreases the need for register allocation/optimization by the compiler. The second motivation is to create a high speed register buffer for the stack, which will accomplish temporary expression evaluation with minimal main memory references. In addition, with respect to multiprogramming, it is desirable to minimize the

save/restore time necessary for effecting a process exchange.

Stack Model and Discussion

The stack model to be used is that developed by Brinch Hansen in the development of Concurrent PASCAL. This model was, in turn, based on the PASCAL P compiler development.

The stack in the model is a pushdown store which is used for four basic functions:

1) local variables for a procedure activation,

2) linkage for procedure return,

3) parameter passing, and

4) expression evaluation.

This discussion is primarily concerned with the fourth of these uses, i.e., expression evaluation and the optimization of the hardware for temporary evaluation and process exchange. Of the other uses of the stack, the following remarks are in order. Local variables can range from no variables to several integers to many complicated types. In general these will reside in primary memory; they could also be allocated to one of the general registers if the compiler determines that they are highly referenced. They do not usually warrant placement in an (extremely limited) high speed stack. Linkage information entails the state of the machine for a previous (not presently active) procedure. It certainly does not warrant inclusion in a high speed memory.

Parameters warrant special consideration. They consist of evaluations made by the preceding activation for the current activation. This would seem to suggest that their inclusion in a high speed memory would be beneficial. Our studies show, however, that parameters are usually low access items, accessed once, either at the beginning (value) or end (reference) of the procedure. Our studies also show that the average procedure (based on the sequential PASCAL utilities for the SOLO system) uses two or fewer parameters, each accessed once. This usage does not warrant the use of a high speed store.

The study alluded to in the previous paragraph consisted of scanning the code files of the SOLO system utilities. The statistics gathered from the scan included the total amount of temporary stack usage, the number of procedures, and the mean and standard deviation of stack lengths by procedure. This was done twice, once including parameter storage and once for just temporary space (expression stace). The results of the study are presented in Appendix B.

Expression Evaluation

Considering the fourth and, for this discussion, most important use of the stack, we are concerned with the evaluation of expressions using the stack. Two important features of expressions are i) depth of stack (i.e. number

of temporaries) and ii) stack lifetime. The depth of the stack is, of course dependent upon the complexity of the expression and the order of evaluation. Consider, for example, the expression

B + C * (D + E).

The parse tree for this expression is illustrated in figure 4.7. Now using a one-pass, non-optimizing, left-to-right code generator, such as the Hartman compilers, the code for this expression causes a stack depth of four words. However, a depth first tree search would cause a maximum stack depth of two in a stack architecture such as Brinch Hansen's. This stack depth is demonstrated as follows: Following the parse tree to the deepest node (i.e., always following the branch with the most leaves in its sub tree), we generate code as follows:

```
push D
push E
add
push C
multiply
push B
add.
```

At any time in this evaluation, the expression stack consists of either i) two terminals (only at the first evaluation) or ii) a terminal and temporary (i.e., the result of a previous evaluation).

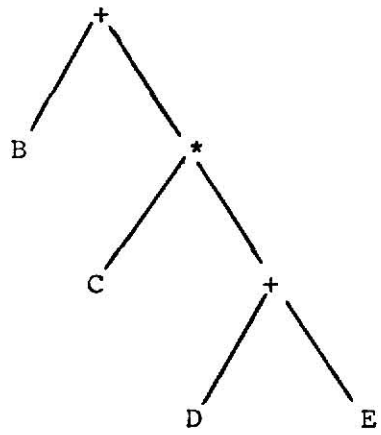


Figure 4.7 Parse Tree for B + C * (D + E)

The lifetime of the expression stack is another interesting concept. In general, ignoring function calls, the expression stack is empty after the evaluation of every PASCAL level statement. This has at least one interesting application; i.e., if the hardware controlling the expression stack is aware of the current depth of the stack, then on the occasion of a process switch, only those high speed registers currently in use by the active process need be saved. In fact, if the switch occurs at the end of a (PASCAL level) statement evaluation, no registers need be saved at all. This could save process switch time and justify a larger high speed stack without incurring excessive process switch overhead. The architecture could conceivably be designed so that process switches occur whenever possible at a point when the temporary expression stack is empty, minimizing overhead for process exchange.

Stack Depth

Using the SOLO system utilities as a data base, the following statistics were gathered about the average depth of the expression stack. For a total of 1674 procedures, the mean stack depth was approximately 15 machine words, with a 3.5 word standard deviation. Some comments are in order about the meaning of these numbers. First of all, we must consider the architecture of the stack machine used by Brinch Hansen from which these figures were produced. For

each assignment, both the address and the value to be assigned must be on the stack. This allows for straightforward left-to-right code generation (i.e., A:=B is implenented as push address of A, push B, copy); however, the stack depth can be shortened by an architecture which supports "copys" and operations with inline addresses (i.e., A:=B could be either push B, copy to A or even copy B to A). Notice, however, that the straightline left-to-right code generation has been eliminated! The tradeoff thus seems to be depth of expression stack vs. ease of code generation (in addition, a deep expression stack implies a little more computational overhead). Consider, in addition, that the ability to do straightforward code generation is of benefit for the creation of cheap compilers for special purpose, application oriented languages. With these tools, compiler optimization can proceed concurrently with use of the unoptimized compiler. Fast compilers can be used for initial development and debug, while optimized versions are used for production programs.

It is interesting that the depth of the Brinch Hansen type code seems to be within range of a sixteen register high speed stack implementation. Code generation by the Hartman compilers is strictly left to right, and all addresses and indices for arrays are formed on the stack. With the right hardware support, even this simply generated code could be executed with moderate efficiency.

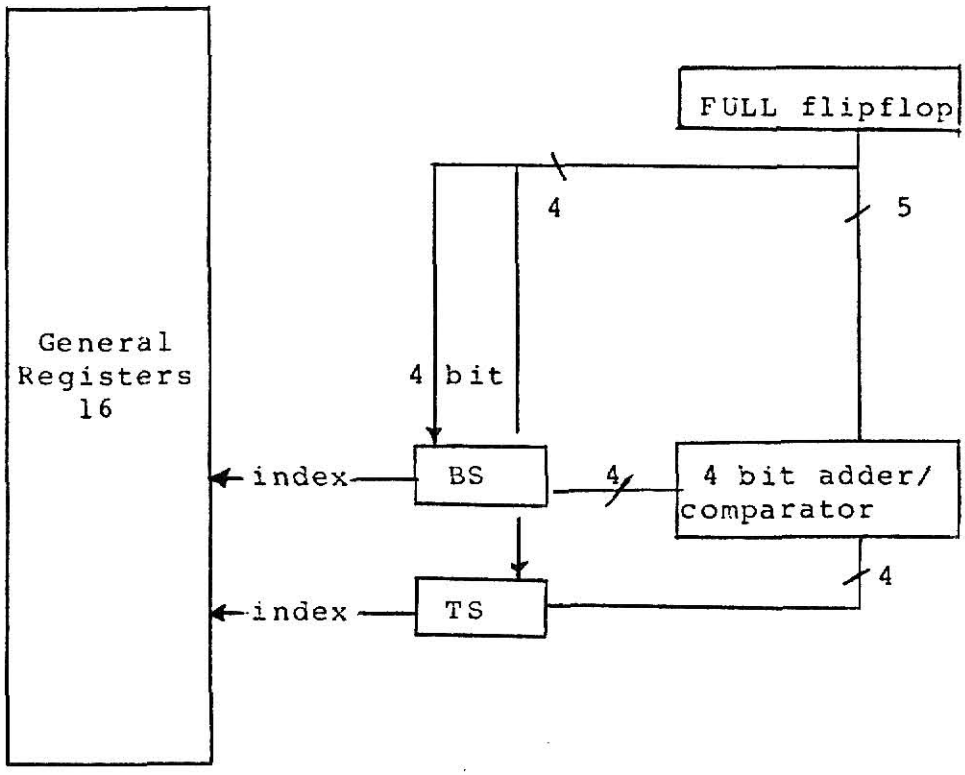


Figure 4.8 16 Register example high speed stack

The Hardware Implementation

The germ of the idea for the hardware implementation of the high speed stack comes from both the "traditional" software stack implementation and the Interdata circular list instructions. The structure is basically the same as the circular list with the following realizations. Available as indices into the high speed register set are two index registers TS and BS, and in addition to these are an adder/comparator which works on these index registers and a flipflop called FULL. (In conventional Interdata architecture, the TS and BS registers are similar to the YS and YD register set. However, they additionally have their own adder for high speed increment/decrement.)

Using a 16 register processor as an example, the structure is illustrated in figure 4.8. Now, continuing with the 16 register example, up to 16 (one word) elements could be pushed onto the stack. In addition, pushes and pops (copys) cause only logical movement of the lower stack elements (i.e., TS is incremented or decremented, but no general register movement is needed). Now stack-to-stack operations are accomplished, using TS and TS+N (number of words in data type) as indices to gate onto ALU busses and may iterate N times to complete the operation (decrementing TS each time and using TS+N as output gating). Note: All arithmetic is modulo 16. A similar scheme applies to stack/logical space operations. Using the four bit scheme,

the index registers may wrap around (in the circular list sense) automatically. The FULL flipflop is used to signify overflow of the stack (which should cause the bottom high speed stack element to be pushed out of the high speed stack to the memory stack, which is its logical extension). An alternative structure would keep a count of the current depth of the high speed stack. This could be of value when saving and restoring the registers, especially as a count of how many registers to restore. We will return to this later.

Logical Stack Structure and Operation

As alluded to above, the high speed expression stack is a logical extension of the memory stack. With this in mind, there are several interesting state transitions which should be considered. As stated above, the overflow of the high speed stack causes the bottom-most element to be saved to memory. This consists of bumping the memory stack pointer S and doing a memory copy. Now, it will also occur at some later time during the expression evaluation, that the purged element will be required for the calculation at hand. This can be recognized by length of HS<=N, where N is the number of words in the data type of the operation. At this point, memory should be referenced to bring back the purged element and the memory stack pointer bumped back.

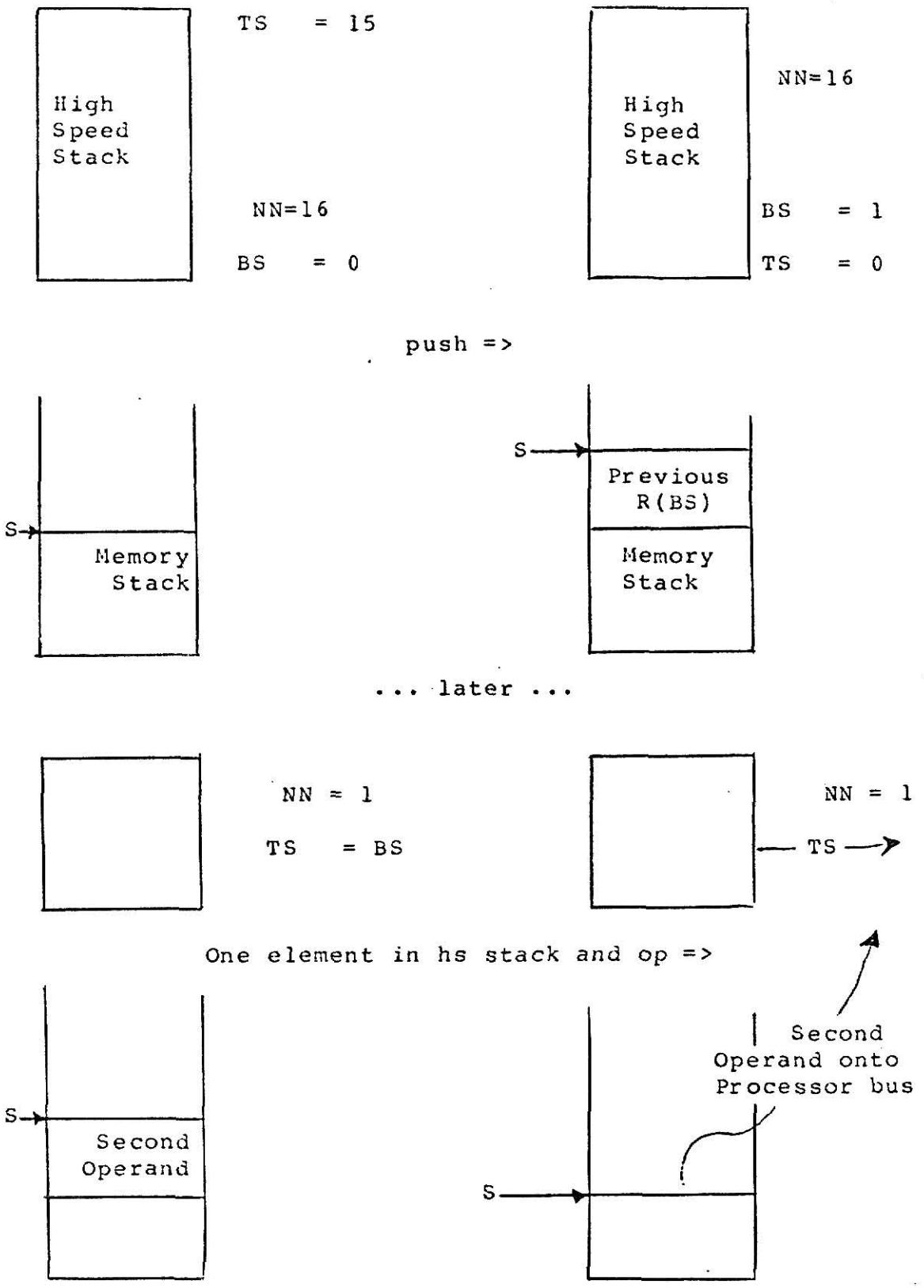


Figure 4.9 Stack Transitions

Pictorially this is illustrated in figure 4.9, where NN is the number of elements in the high speed stack.

Parameter Passing and Procedure Call

A procedure call causes the evaluation of the expressions which are the actual parameters of the procedure activation, followed by transfer to the new procedure. As such, the expression stack consists of only the actual parameters of the activation. This is slightly different for a function call, which may be in the middle of an expression. The called procedure executes the enter instruction, which in this stack model should cause i) the copy from the high speed stack to the memory stack of the parameters, ii) the linkage information (i.e., mark stack) to be saved in the memory stack, and iii) the local variables to be allocated within the memory stack and the local base register (B) to be set. This scheme causes the memory stack to look exactly like Brinch Hansen's stack with its parameter and local addressing structure. Pictorially this is shown in figure 4.10. For a function call within an expression, this is only slightly different, for in addition to the parameters, the high speed stack may also include temporaries from the expression evaluation. In this case the memory stack is shown in figure 4.11. As explained in the discussion of stack overflow, this will cause an access to memory when the temporaries are required for the

expression evaluation.

Implementation of the High Speed Stack in Conventional Computer Architecture

Now that we have provided the motivation and implementation of the High Speed Stack, we extend the discussion to a more complete architecture. We direct the discussion on the basis of a 16 register micro architecture, although the concepts can be extended to any number of registers.

Register Set Segmentation

As set forward in the introduction, the goal of this architecture is to provide both a high speed stack and other registers, such as variable base displays, for example the global and local bases in Brinch Hansen's architecture. Some of the registers are to be used by the compiler writer for whatever optimizations he may choose. Assuming a single register set, this necessitates the segmentation of those registers into two logical sections: a) the high speed stack and b) the displays and other registers. This is illustrated in figure 4.12. The nice wraparound properties of a power of two for the High Speed Stack index registers suggest that the register set be halved, one half being the High Speed Stack and the other programmable by the compiler writer.

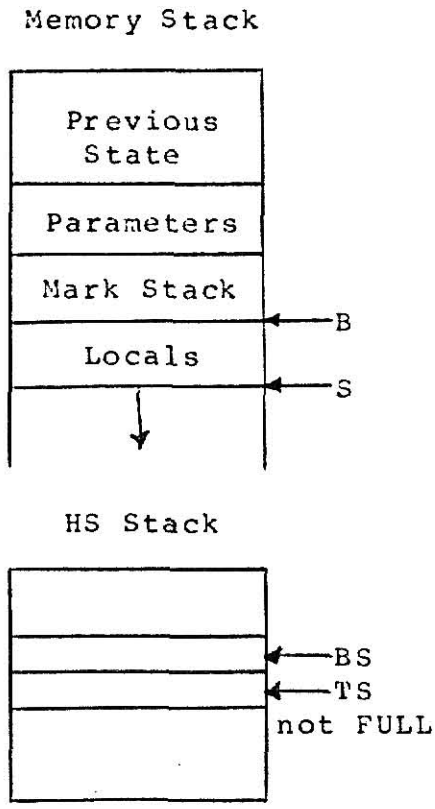


Figure 4.10 Stack at Point of Procedure Call

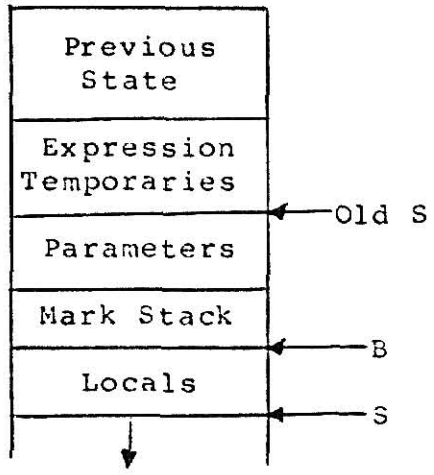


Figure 4.11 Memory Stack for Function Call

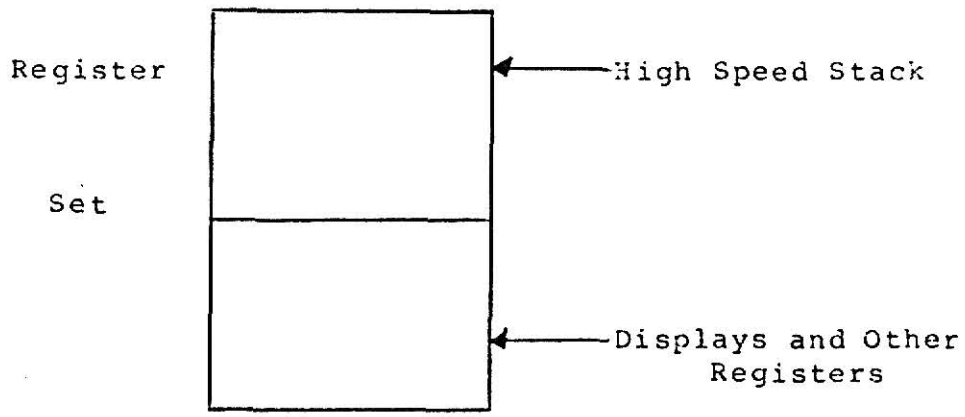
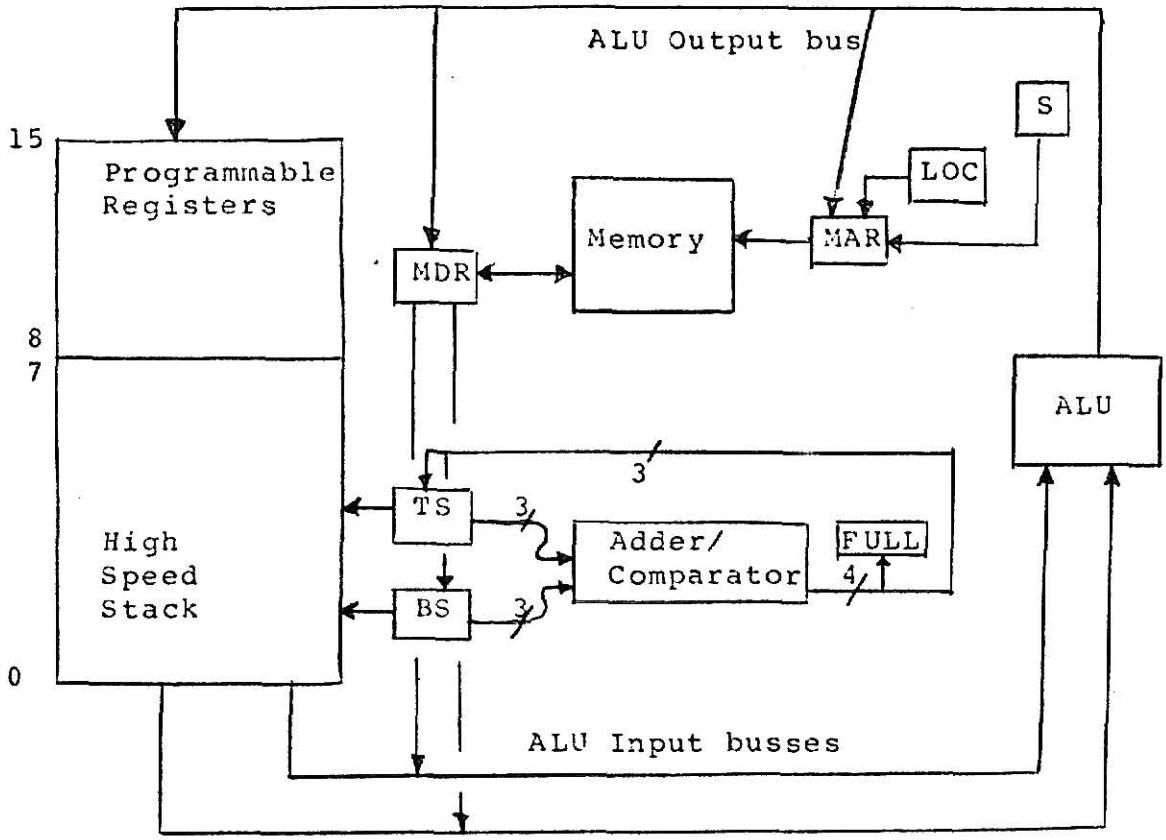


Figure 4.12 Register Set Segmentation



All unmarked busses are word length of underlying machine

Figure 4.13 Processor Architecture

The final architectural drawing is given in figure 4.13. This drawing ignores the instruction fetch and decode hardware, which is assumed to be present, and to include "profile" information which describes data types of operands (in particular the length of the data types).

Let us consider the flow of information and control through this processor for some typical instructions.

Pushdata --The instruction would include the address of the data and the "profile" data type. The address might be indexed by the top of stack, indexed by one of the "programmable" registers, not indexed or immediate data.

Pushing a variable of some data type (i.e., one or more words) onto the stack is an iterative process, one word at a time. Each iteration causes the following:

a) the TS register is incremented by 1 modulo 8;

b) TS is tested for equality with BS when FULL=true;

c) in case of equality, the contents of R[BS] must be copied to the memory stack;

d) FULL <- true;

e) the data word is copied into R[TS].

Operations --Data operations which are stack to stack (i.e., Top_of_stack <- Top_of_stack op second_in_stack) are carried out as follows: Given the length N in words of the data type (from the profile), this is again an N-iteration process. First, the high speed stack is tested for operand

presence. If FULL=false, then neither operand is in the high speed stack and N memory fetches (i.e., pushes) must be initiated. If FULL=true, then the first operand is in the high speed stack. The test for the second operand in stack is determined by the "direction" of the stack. If TS is incremented to push and decremented to pop, then the number of words in the stack is computed by:

```
IF TS>BS THEN
    Num_words_in_stack: = TS - BS
ELSE
    Num_words_in_stack: = 8 + TS - BS
```

This test must be done on each iteration in order to insure that the element is in the stack; if not, it must be fetched from memory and the BS register decremented. The output is always directed to the register TS - N (modulo 8 arithmetic) and the memory stack register popped, if the memory stack was referenced. The TS register is always decremented after iteration.

Copy (Pop) Data --As in pushdata, this instruction specifies the address to copy the data presently at the top of the stack. Indexing may be as in the push, with the exception that the second in stack, rather than the top, may be used for indexing. As with operations, the high speed stack must be tested for operand presence with iteration to effect the copy of N words.

"Programmable" Register Operations --These operations

may be handled in the conventional manner; or if they are to be used only for display (base) registers, they might be given only load/store and index capabilities.

Conclusion

The main emphasis of this chapter has been the effect that multiprogramming and particularly the abstractions of Concurrent PASCAL may exert upon architectural design. It seems that the time is right, with the influences that software is exerting on architecture (re. the influx of recent interest in "stack machines"), for architecture to provide more support for multiprogramming, a de facto standard in the use of computers. It is hoped that this conceptual framework, drawn from the influential development of the Concurrent PASCAL language may be of benefit to this evolution.

Conclusion

In this thesis we have explored three areas. The first area may be described as purely implementational, that is how three implementations accomplished the supporting architecture of Concurrent PASCAL. The second area dealt with portability -- how one implementation was transported to a second implementation on a different computer and the related problems. The third area explored was the considerations which could affect the creation of a Concurrent PASCAL machine, with the realizations of architectural support for concurrent programming abstractions.

The programming language Concurrent PASCAL has, since its recent development, exerted a substantial impact upon the concurrent programming field. The abstractions designed into the language make it not only a viable implementation tool (as shown by the interest in existing and forthcomming implementations) but also as a conceptual tool for system design. These factors, coupled with the control and data structures of PASCAL combine to make a language of considerable interest.

In addition to the abstractions introduced by Concurrent PASCAL, the architecture which supports Concurrent PASCAL is of interest. This can primarily be accounted for by the fact that this architecture may be described as the "bare

bones" of multiprogramming, that is, those elements essential to the nature of multiprogramming and little more. It is through the consideration of three implementations of Concurrent PASCAL that these essential elements of multiprogramming were examined, and the results of this examination suggests features to be incorporated into a truly multiprogramming architecture, an architectural base for Concurrent PASCAL.

[AH1] Hartman, A. A Concurrent PASCAL Compiler for Mini Computers, PhD Thesis, California Institute of Technology, 1976.

[AH2] ---, Sequential PASCAL Report, Information Science, California Institute of Technology, July 1975.

[BH1] Brinch Hansen, P. The Programming Language Concurrent PASCAL, IEEE Transactions on Software Engineering, Vol. 1, No. 2 (June 1975), pp. 199-207.

[BH2] ---, Concurrent PASCAL Machine, Information Science, California Institute of Technology, October 1975.

[BH3] ---, Concurrent PASCAL Report, Information Science, California Institute of Technology, June 1975.

[BH4] ---, The SOLO Operating System, Software -- Practice and Experience, Vol. 6 No. 2 (April - June 1976), pp. 141-206.

[BH5] ---, Concurrent Programming Concepts, ACM Computing Surveys 5, 4 (December 1974), pp 223-245.

[BH6] ---, Concurrent PASCAL Implementation Notes, Information Science, California Institute of Technology, January 1976.

[BH7] ---, Job Control in Concurrent PASCAL, Information Science, California Institute of Technology, March 1975.

[BL] Liskov, B. H. The Design of the VENUS Operating System, Communications of the ACM 15, 3 (March

1972), pp. 144-149.

[BWL] Lampson, B. W. et al. Report on the Programming Language Euclid, SIGPLAN Notices 12, 2 (February, 1977).

[CDC] Control Data Corp., Control Data 6600 Computer System Reference Manual, 1963.

[CH] Hoare, C. A. R. Monitors: an operating system structuring concept, Communications of the ACM 17, 10 (October 1974), pp. 549-557.

[CZ] Cottel, D. and Zaun, J. Concurrent PASCAL Interdata 7/16 Computer, Preliminary release, Naval Undersea Center, December 1976.

[DN] Neal, D., Anderson, G., Ratliff, J., and Wallentine, V. KSU Implementation of Concurrent PASCAL -- A Reference Manual, Kansas State University Department of Computer Science, Technical Report CS 76-16, January 1, 1977.

[ED] Dijkstra, E. W. Hierarchical ordering of sequential processes, Acta Informatica 1, 2 (1971), pp. 115-138.

[EO] Organick, E. I. Computer System Organization, Academic Press, 1973.

[I1] Interdata Inc., OS/32-MT Program Reference Manual, Pub. No. 29-390R04, April 1976.

[I2] ---, Model 8/32 Processor User's Manual, Pub. No. 29-428R01, January 1976.

[I3] ---, Common Assembler Language (CAL) User's Manual, Pub. No. 29-375R05, October 1975.

[IBM] International Business Machines Corp., IBM Virtual Machine Facility/370 Introduction, Order No. GC20-1800-6, October 1976.

[JB] Bell, J. R. Threaded Code, Communications of the ACM 16, 6 (June 1973), pp. 370-372.

[JH] Howard, J. H. Signaling in Monitors, Proceedings of 2nd Int. Conf. on Software Engineering (ACM/IEEE/NBS), October 1976, pp. 47-52.

[JS] Saltzer, J. Review of [BH2], ACM Computing Reviews No. 31,363, Vol. 18, No. 5 (May 1977), p. 184.

[JW] Jensen, K. and Wirth, N. PASCAL User Manual and Report, Springer-Verlag, 1975.

[MB] Ball, M. Personal Correspondence, 1976-1977.

[MBU] ---, PASCAL Compiler for the Univac 1110, Pascal Newsletter, No. 6 (November 1976), Implementation Notes, p. 89.

[NA] Nori, K. V., Ammann U., et. al. The PASCAL 'P' Compiler: Implementation Notes, Institute fur Informatik, Eidgenossissche Technische Hochschule, Zurich, Report No. 10, 1974.

[PW] Poole, P. C. and Waite, W. M. Portability and Adaptability, in Bauer, F. L. Advanced Course on Software Engineering, Springer-Verlag, 1973.

[RB] Blake, R. P. Exploring a Stack Architecture,

Computer, May 1977, pp. 30-41.

[RG] Goldberg, R. P. Architecture of Virtual Machines, Proceedings AFIPS National Computer Conference, July 1973, pp. 74-112.

[SB] Saxena, A. R. and Bredt, T. H. Verification of a Monitor Specification, Proceedings of 2nd Int. Conf. on Software Engineering (ACM/IEEE/NBS), October 1976, pp. 53-59.

[SKB] Silberschatz, A., Kieburtz, R. B., and Bernstein, A. J. Extending Concurrent PASCAL to Allow Dynamic Resource Management, IEEE Transactions on Software Engineering, Vol. SE-3, No. 3 (May 1977), pp. 210-217.

[SS] Schroder, M. D. and Saltzer, J. H. A Hardware Architecture for Implementing Protection Rings, Communications of the ACM 15, 3 (March 1972).

[WM] Wallentine, V., and McBride, R. Concurrent PASCAL -- A Tuitorial, Kansas State University Department of Computer Science, Technical report CS 76-17 November 1976.

[WW] Wulf, W. et. al. The Design of an Optimizing Compiler, American Elsevier Publishing Company, 1975.

Appendix A
Functional Specification of a Concurrent PASCAL Kernel for the Interdata under OS/32-MT

Introduction

This functional specification describes the Kernel for Concurrent Pascal running under OS/32-MT. It describes, class by class, the modules found in the KERNEL program of the system, in order as found in the program listing. This specification teamed with Brinch Hansen's manual 'Concurrent Pascal Machine' and the high level encoding from the Cal Tech Kernel for the PDP 11, should provide for a total understanding of the Kernel for Concurrent Pascal as designed by Brinch Hansen.

In addition to the detailed specification of the Kernel program, the other program modules of the Interdata system (Address space and transient initializer (ADDR), Faults handler (FAULTS) and Interpreter (INTERP)) are each described as a unit in the sequel. The inclusion of an asterick (*) in front of an entry point in the Kernel specifies that routine as an entry point from the Interpreter and that as such it will continue the semantic evaluation begun by the Interpreter. The interaction of the various system programs should become apparent from their individual specifications.

System Conventions

The following conventions are implicit in the following descriptions. These include the usage of processor registers, and subroutine linkage. The Kernel executes in a non-preemptable state, i.e. all generated traps are queued, using the task queue facility of OS/32-MT, until the Kernel is again prepared to handle traps. There are five (5) general registers dedicated to the Kernel, these are registers R0 - R4. The Kernel does not alter any other registers (with the exception of save/restore during process switch). The standard use of the Kernel registers is as follows: R4 - linkage register for subroutines; R3 - Parameter register for subroutines; R2 - scratch register for routines (assumed to be preserved across subroutine calls); R1 - Instance register (set to the current instance of a variable for a class with several instances e.g. queuetype); R0 - always available for scratch (not preserved across subroutine boundaries). The other general registers provide the registers of the virtual stack machine as defined by Brinch Hansen (G, B, S, Q, HD [process head pointer]), scratch registers for interpreter evaluation (W, X, Y, Z), and the software segmentation registers (COMTOP,

CHILDB). Following is a description of each of the Kernel classes.

Class: Traplevel 1.

Entry points: KER.TQTR
NOCODE.

Parameters: implicit input: An (OS/32-MT) system event has occurred; Current running process (HD).

Functional description: The level one trap handler accepts traps (i.e. is the trap entry point) from the extended machine (OS/32-MT). It then decodes the type of trap, causing transfer to the appropriate handling module. NOCODE applies a no-operation (ignore) to unwanted or unexpected traps.

Accessing modules: OS/32-MT.

Accessed modules: IOROUT, TIMERT, Exit.

Global variables/side effects: The user dedicated location UDL.TSKQ is used to find the task queue which is declared in the address space (ADDR).

Class: IO Entry.

Entry point: *KER.IO.

Parameters: input: Data buffer (W), SOLO type parameter block (X), Device index (Y); implicit input: current running process (HD).

Functional description: The IO entry routine is entered from the interpreter to facilitate Concurrent Pascal IO. Associated with each device in the system is an SVC 1 parameter block (DEV.PB) and a (Concurrent Pascal like) queue variable (DEV.USER). The IO routine translates from the SOLO-type parameter block to an OS/32-MT parameter block, and delays the running process on the queue to await IO completion (signaled by the class IO interrupt). In addition this class coupled with the IO interrupt class, buffers data to and from the SOLO console, keeping track of its current status.

Accessing modules: Interpreter.

Accessed modules: RUN.PREM, SIG.AWAT, Exit.

Global variables/side effects: The following global variables are accessed by the IO entry routine: PAS.DEVS (address of device parameter blocks), PAS.NODV (number of devices configured in system), PAS.POLL (polling stack for tape command functions), PAS.HCON (halt I/O parameter block for console), PAS.IBUF, PAS.OBUF (buffers for console input and output respectively), PAS.CSTA, PAS.IBPT, PAS.OBPT (console status, input buffer pointer, and output buffer pointer), PAS.BELL (signal queue for console bell preemption).

Class: IO Interrupt.

Entry point: IOROUT.

Parameters: input: device parameter block of interrupting device (R1).

Functional description: It is the responsibility of the IO interrupt class to translate the occurrance of an IO trap into a continue of the process waiting on the device and to translate the OS/32-NT status into SOLO status.

Accessing modules: Traplevel 1.

Accessed modules: SIG.SEND, RDY.ENTR, RDY.RESC, Exit.

Global variables/side effects: Same as in the IO entry class.

Class: New.

Entry point: NEW.CORE.

Parameters: Input: Amount of space to be allocated(R3).
Output: Address of block(R3).

Functional description: This module allocates space for Kernel data structrues, in particular process records and monitor gates. Space is reserved in low memory for these data structures, in order to insure addressability by 16-bit addresses.

Accessing modules: RUN.INTC, RUN.INTP, GAT.INTG.

Accessed modules: KERERROR.

Side effects/Global variables: The global Kernel variables PAS.TOP and PAS.FREE are adjusted to represent their new values. In addition an error is generated if there is not enough space reserved for the system variable.

Class: Queuetype (Processqueue).

Entry points: QUE.GET
QUE.PUT
QUE.INIT.

Parameters: Input: (GET) Element to be added to end of queue (R3).
Output: (PUT) Element removed from front of queue (R3).
Access: (all) Global queue variable (R1).

Functional description: The class queuetype maintains a doubly linked list of 16-bit process record addresses as a queue. Elements may be added to the bottom or removed from the top of the list. An additional macro function EMPTY was defined in the Brinch Hansen Kernel. In the 8/32 implementation, empty is hand coded inline and consists of testing whether the head of the queue points to itself. The initialize procedure sets up this structure (initially both head and tail point to the head of the queue).

Accessing modules: Signal, Ready, Gate.

Accessed modules: none.

Instances: Instances of queuetypes include the three priority ready lists, the signal queues, and the monitor gate queues (where processes await exclusive access to a monitor).

Class: Signal.

Entry points: SIG.AWAT (await)
SIG.SEND
SIG.INIT.

Parameters: Access: (all) Global signal queue variable(R1).
Inplicit input: (AWAT) Current running process (HD).

Functional description: The signal class maintains (using class Queuetype) a queue of processes awaiting the occurrance of a signal (event). The AWAT (await) entry point causes the currently active (running) process to be deactivated and enqueued on the list. The SEND entry point causes all enqueued processes (if any) to be transferred to the ready list.

Accessing modules: Clock, Console.

Accessed modules: RUN.PREM, Queuetype, RDY.ENTR.

Instances: Instances of the signal class consist of the (software) clock whereby processes may wait for a timing signal and the console bell whereby processes may wait on operator intervention.

Class: Time.

Entry points: TIME.ADD
TIME.INT (initialization).

Parameters: Input: (ADD) Amount of time (in ms) to be added to the timing variable (R3).
Access: (all) Timing variable (R1).

Functional description: The time class implements the timing of system functions, in particular the system clock and process runtimes. A timing variable is a structure consisting of seconds and miliseconds.

Accessing modules: Clock, RUN.UPDT.

Accessed modules: none.

Instances: Each process contains a timing variable as part of its process head which is used to measure process runtimes. A system wide timing variable (PAS.NOW) is maintained by the clock class.

Class: Timer.

Entry point: TIM.TICK.

Parameters: none.

Functional description: Maintains an interval timer (global variable PAS.PERD) for the purpose of process timing. The timer is explicitly reset (to zero) by the module RUN.SERV. All timing values are expressed in miliseconds. In addition, since the Kernel executes in an operating system environment, the timing values are only estimates of CPU utilization.

Accessing modules: Timertrap.

Accessed modules: none.

Global variables: PAS.PERD is the interval timer.

Class: Clock.

Entry points: CLK.INCR (increment)
*CLK.WAIT
CLK.INIT.

Parameters: Implicit input: (WAIT) running process (HD).

Functional description: Maintains a software clock for the system (the interpreter routine realtime access this clock directly). The wait entry point allows a process to wait for the next one second increment of the clock.

Accessing modules: Interpreter, Timertrap.

Accessed modules: Signal, Time.

Global variables: PAS.NEXT (timing signal queue), PAS.NOW (software clock).

Class: Timertrap.

Entry point: TIMERT.

Parameters: Implicit input: The amount of time specified by the TIMER parameter block has expired.

Functional description: This class implements the required clock and timer updates upon the occurrance of a timing signal from the extended machine (OS/32-MT). In the OS/32-MT implementation this class also executes tape polling, as tape command functions do not generate completion traps of their own. In addition, the ready list is rescheduled and a request (SVC 2 code 23) is lodged for the next clock period.

Acessing modules: Traplevell.

Accessed modules: TIM.TICK, CLK.INCR, RDY.RESC, TAP.POLL.

Global variables/side effects: Discussed in functional description.

Class: Core.

Entry point: COR.ALOC.

Parameters: Input: Amount of memory to be allocated(R3).
Output: Address of allocated segment (CHILDB).

Functional description: This class handles the allocation of memory for the common (initial process) and private (all other processes) segments. If memory is overallocated, an error message is generated. All memory is allocated in contiguous, sequential segments (i.e. linearly).

Accessing modules: RUN.INTC, RUN.INTP.

Accessed modules: KERERROR.

Global variables/side effects: PASS.CTOP reflects the current top of allocated memory, PAS.CFRE reflects the current free space. These variables are initialized in the transient initialization phase of the system.

Class: Running.

Entry points: RUN.SERV (serve)
RUN.PREM (preempt)
RUN.UPDT (update)
RUN.POP (pop parameters)
RUN.INTC (initchild)
RUN.INTP (initparent)
*RUN.INPR (init a process)
*RUN.ENDP (endprocess)
*RUN.STOP (stop a process)
*RUN.SYSE (system error)
KERERROR (Kernel error).

Parameters (by entry point):
SERV: input: Process record to be served(R3).
PREM: output: Process record previously active (R3).
POP: input: Number of parameters (R3), (implicit) Stack pointer of parent process(S).
INPR, INTC: input: Heap, variable, and stack lengths (W,X,Y), initial code pointer (Z).
INTP: input: Pointer to code head as defined by concurrent PASCAL compiler (R3).
STOP: input: Process record to be stopped (W), result to be entered in process head (X).
KERERROR: input: error message (R4).
UPDT, ENDP: none.

Functional description: The running class handles the restoration, preemption and maintainence of the currently active process. Each entry point will be treated separately.
SERV: This procedure restores the process registers (state) from the process record.
PREM: This procedure preempts the running process by saving its registers in the process record and purging the current process register (HD).
UPDT: This procedure updates the running process time slice. If this exceeds the maximum slice then overtime is set true and the process runtime is updated.
POP: This procedure pops parameters from the parent process stack, to be passed to the child process.
INTC: This procedure implements the initialization of a child process,

initializing a process record, allocating data space, and initializing registers. The child process becomes the running (active) process.

INTP: This procedure begins the execution of a concurrent Pascal program by initializing the initial process; its process record, registers, and data space and by making it the running process.

INPR: Entry point from the interpreter to initialize a child process.

ENDP: Entry point from the intrepreter to end the (current running) process.

STOP: Entry point from the interpreter to stop execution of the given process with the given result.

SYSE: Entry point from the interpreter to handle an error detected in system (i.e. concurrent Pascal) code.

KERERROR: Entry point for error detected in Kernel.

Accessing modules: (by entry point)

SERV: RDY.SELC.
PREM: IO, SIG.AWAT, RUN.INPR, RUN.ENDP, RDY.RESC, GAT.ENTR, GAT.DELA.
UPDT: RDY.RESC, RUN.PREM.
POP: RUN.INPR.
INTC: RUN.INPR.
INTP: KER.INIT.
INPR, ENDP, STOP, SYSE: INTERP.
KERERROR: NEW.CORE, COR.ALOC, RUN.POP, RUN.INTC.

Accessed modules: (by entry point)

UPDT: TIME.ADD.
POP: KERERROR.
INTC: NEW.CORE, COR.ALOC.
INTP: NEW.CORE, COR.ALOC.
INPR: RUN.POP, RUN.PREM, RDY.ENTR, RUN.INTC.
KERERROR, RUN.SYSE: KER.REIN.
All others: none.

Global variables/side effects: The running class maintains process records (except linkage which is handled by Queuetype) and processor registers. In addition, the following routines access global variables: RUN.SERV resets the interval timer (PAS.PERD). RUN.POP uses PAS.PARM as temporary storage for parameters. RUN.INTP and RUN.INTC each increment the next process index (PAS.NIND) and the process pointer array (PAS.PRCS). In

addition, INTP initializes and INTC accesses the initial heaptop pointer (PAS.HEPT). RUN.INTP sets the concurrent large constant address (PAS.CADR) used by the interpreter. Finally RUN.SYSE and KERERROR access the console device using wait I/O to log error messages prior to system re-initialization.

Class: Ready.

Entry points: RDY.ENTR (enter)
RDY.SELC (select)
RDY.RESC (reschedule)
RDY.INIT.

Parameters: input: (ENTR) Process record to be entered on ready list (R3).
Output: (SELC) Highest priority ready process (R3).

Functional description: The ready class manages the manipulation of the three level (priority) ready list. The lists are maintained using the class Queuetype. The ENTR entry point determines (from the process head) which ready list to use and enters the process record there. The SELC entry point searches for the hignest priority non-empty list and returns the first process on that list. If all three ready lists are empty, the SELC entry point issues a wait for trap (SVC 9 with wait enabled). The trap routine is assumed to recall RDY.SELC at some point (i.e. there is no return from the previous activation). The RESC entry point will preempt the active process if a higher priority process has become ready.

Accessing modules (by entry point):
ENTR: IOROUT, SIG.SEND, RUN.INPR, RDY.RESC, GAT.LEAV, GAT.DELA, GAT.CONT.
SELC: KNEXIT.
RESC: IOROUT, SIG.SEND, TIMERT, RUN.INTC, GAT.LEAV, GAT.CONT.
INIT: not accessed - The address space initializes the ready lists as queuetypes initially empty. This code only comments this initialization.

Accessed modules (by entry point):
ENTR: QUE.PUT.
SELC: QUE.GET, RUN.SERV.
RESC: RUN.UPDT, RUN.PREM, RDY.ENTR.
INIT: QUE.INIT.

Global variables: The global variables PAS.TOP, PAS.RMID, and PAS.RBOT are the heads of the ready queues. They are manipulated using the class Queuetype. PAS.IDLE is a boolean which signifies whether the current Kernel entry was from the idle state.

Class: Gate.

Entry points: *GAT.ENTR (enter)
*GAT.LEAV (leave)
*GAT.DELA (delay)
*GAT.CONT (continue)
*GAT.INTG (init gate)
GAT.INIT.

Parameters (by entry point):
ENTR: input: Gate to be entered (W); implicit input: current running process (HD)
LEAV: input: Gate to be left (W); implicit input: current running process (HD).
DELA: input: Gate (W), (Concurrent Pascal) Queue variable (X); implicit input: current running process (HD).
CONT: input: Gate (W), Queue variable(x).
INTG: input: Gate(W).
INIT: input: Gate(R3).

Functional description: The implementation of a monitor gate variable consists of a boolean flag (open) representing the current state of the monitor and an associated list (of type queuetype) of waiting process records. As all but the initialization are entry points from the interpreter, these routines provide the semantics of monitor entry, exit, delay, and continue. It is important to note that the interpreter instruction continue includes an implicit procedure return, allowing the GAT.CONT routine to enter the continued process onto the ready list without preempting the running process. Continuing an empty queue is an implicit gate exit (leave).

Accessing modules: Interpreter.

Accessed modules: Queuetype, RUN.PREM, RDY.ENTR, RDY.RESC, NEW.CORE.

Instances: Each monitor declared and initialized in the Concurrent Pascal program is dynamically allocated a Kernel gate variable by use of the class NEW. The address of this gate variable is stored in the parameter section of those processes with access to the monitor (as well as in the space of the initial process, allowing the parameters to be popped).

Class: Exit.

Entry points: KNEXIT
KNEXIT2
KNEXIT3.

Parameters: implicit input: current running process.

Functional description: All queued traps are serviced before exiting the Kernel (hence KNEXIT is also part of class traplevel1). If there is no active process (i.e. RUN.PREM was previously called) the exit class selects one. The exit class terminates Kernel execution by returning to the execution (within the interpreter) of the selected or active process.

Accessing modules: NOCODE, IOENTRY, IOROUT, CLK.WAIT, TIMERT, RUN.INPR, RUN.ENDP, RUN.STOP, GAT.ENTR, GAT.LEAV, GAT.DELA, GAT.CONT, GAT.INTG, RDY.SELC.

Accessed modules: (implicit) Traplevel 1, Interpreter, RDY.SELC.

Global variables/side effects: none.

Class: Reinitialization.

Entry points: KER.REIN
KER.INIT.

Parameters: input: Disk address of system to be initialized - either permanent or alternate (MOD(X)).

Functional description: This class implements the reinitialization of the system by halting all outstanding I/O and restoring the address space from disk (Kernel segment), then loading the new concurrent Pascal program and initializing it.

Accessing modules: IO entry (control operation to diskdevice), RUN.SYSE.

Accessed modules: Interpreter, RUN.INTP.

Side effects: All global variables are reset to their initial value.

Class: Interpreter.

Entry points: INITINTR
NEXT.

Parameters: Code pointer (Q), Global (G) and local (B) bases, stack pointer (S), process head pointer (HD), top of common addresses (COMTOP), child base register (CHILDB).

Functional description: The interpreter module effects the execution (interpretation) of the Pascal virtual code. The best documentation of the individual code routines exists in the high level encoding of the routines from the Cal Tech Kernel listing. The parameters childbase and common top are used in the software scheme simulating the addressing mechanism used in the Brinch Hansen system.

Accessing modules: Exit, Reinitialization, Faults.

Accessed modules: KER.IO, GAT.ENTR, GAT.LEAV, GAT.DELA, GAT.CONT, GAT.INTG, RUN.ENDP, RUN.INPR, RUN.STOP, RUN.SYSE, CLK.WAIT.

Global variables/side effects: Global variables accessed are PAS.NOW (real time) and PAS.CADR (large constant address for Concurrent Pascal). In addition several process head variables are accessed (these are described by Brinch Hansen), and the process data space, heap and stack are modified as specified in the Pascal code.

Class: Faults.

Entry points: MEMFT
instr.

Parameters: none.

Functional description: This module handles errors detected by the OS/32-MT system, in particular illegal instruction and memory fault interrupts. There are two possible reasons that such errors might occur. First, if a file is given the attribute SEQCODE or CONCODE (under the SOLO system) but was not generated by the Pascal compilers of the system, and it is attempted to execute this file, one of these errors is bound to occur. In addition, undetected errors in the memory hardware can (and have) also cause these errors. The action taken is to log a message (SVC 2 code 7) to the system console (notifing the operator of possible memory problems) and the return to the Pascal system as an OVERFLOW type error.

Accessing modules: OS/32-MT.

Accessed modules: OVERFL (in interpreter).

Global variables: none.

Class: Transient initializer.

Entry point: INIT.

Parameters: none.

Functional description: This module handles the initial interface between the operating system (OS/32-MT) and the Concurrent Pascal System. Included in these tasks, the transient initializer issues an SVC 2 code 2 (get space) to allocate the entire partition for Concurrent Pascal usage (setting PAS.CTOP and PAS.CFRE). In addition, the transient initializer makes an image copy of the address space on the virtual disk in the Kernel segment (pages 0-23) This image copy is then used for system reinitialization. The transient initializer is overwritten by Concurrent Pascal code after transferring to the Kernel. In the reentrant library version, this is the only Interdata code residing in the task partition.

Accessing modules: OS/32-MT.

Accessed modules: KER.INIT.

Global variables: Pas.CTOP, PAS.CFRE. All other variables are implicitly initialized (DC's in the address space).

Appendix B
Stack Depth of PASCAL Code

Program Name	Stack Used	Procedures	Mean	Standard Deviation
NAVYPREP	172	6	28.666	4.50
PREP	1062	38	27.947	7.07
CPASS6	1410	52	27.115	5.95
SPASS6	1410	52	27.115	5.95
NULL	26	1	26.000	0.00
GGG	862	28	30.785	9.16
SPASSX	912	30	30.399	6.40
CPASS3	2798	96	29.145	5.54
SPASS3	3168	107	29.609	6.60
APPEND	340	12	28.333	2.93
CONSOLE	102	4	25.500	2.51
ERRLIST	268	9	29.777	6.74
CPASSX	974	32	30.437	6.56
MAKETAPE	220	8	27.500	3.50
EDIT	1082	36	30.055	5.90
PRINTER	268	9	29.777	6.74
COPYUSERS	130	5	26.000	3.46
FILE	2110	67	31.492	10.75
DO	852	30	28.399	3.58
CONCAT	464	17	27.294	3.08
BACKUP	544	18	30.222	10.60
CPASCAL	452	17	26.588	2.90
SPASCAL	452	17	26.558	2.90
DISK	196	7	28.000	3.27
CPASS5	2410	81	29.753	6.79
SPASS5	2316	78	29.692	6.88
REMOVE	492	16	30.750	6.69
CPASS2	2222	64	34.718	7.90
SPASS2	2462	70	25.171	7.90
WRITE	274	10	27.399	3.27
MAKETEMP	222	8	27.750	3.28
USER_PROGRAM	916	29	31.586	8.93
STACKDRV	334	11	30.363	5.57
PREPASS1	1470	50	29.399	6.01
CARDS	242	9	26.888	2.47
DRIVER	334	11	30.363	7.09
PRECOB	426	16	26.625	2.70
START	240	9	26.666	2.65
TAPE	346	13	26.615	1.71
CPASS7	1212	46	26.347	3.51
SPASS7	1212	46	26.347	3.51
MOVE	266	10	26.599	1.90
CPASS4	2536	85	29.835	7.25
SPASS4	2520	85	29.647	6.63
STACKGEN	140	5	28.000	2.83

READ	298	11	27.090	2.74
SPASS1	1264	43	29.395	8.46
CPASS1	1238	42	29.476	8.54
IO	316	12	26.333	3.39
RUN	478	18	26.555	2.45
COPY	300	11	27.272	3.38
NEWCPASS2	2250	65	34.615	7.88
PREPPASS2	24	1	24.000	0.00
LIST	594	21	28.285	6.43
TOTAL	49628	1674	29.646	7.00
TOTAL (INCLUDING PARAMETERS)	49846	1639	30.412	8.85

All figures are in bytes (except procedures)
Only final total includes parameters.

Appendix C
Scheduler Algorithm

```
program scheduler;

const maxindex = 16;  "maximum child readylists"
      maxprocesses = 16;  "maximum processes in FIFO"
      absmaxslice = 500;  "maximum time slice"
      minexpressablestate = -32767;
      maxexpressablestate = 32767;

type indextype = 1..maxindex;
     indexorzero = 0..maxindex;
     readyorganization = (priority, timeslice);
     bitstream = set of indextype;
     readyptr = ^ readythread;
     threadtype = (lists, processes);
     slice = 0..absmaxslice;

     FIFO = 0..maxprocesses;  "hardware defined"

     readythread = record
           parent: readyptr;
           index_in_parent: indextype;
           case typethread: threadtype of
             lists: (
               readybits: bitstream;
               children: array [indextype] of readyptr;
               organization: readyorganization;
               currentindex: indextype;
               maxslice, sliceleft: slice);
            processes: (ready: FIFO)
       end; "readythread"

       processptr = ^processrec;

       processorstate = minexpressablestate..
                        maxexpressablestate;

       processrec = record
         state: processorstate;
         readylist: readyptr;
         runtime: integer
       end; "processrec"

var root: readyptr; currentprocess: processptr;
    curslice, interval_timer: slice;
    processorregs: processorstate;

function empty(f: FIFO): boolean;
begin "hardware defined check for empty" end;
```

```
procedure append(p: processptr; var f: FIFO);
begin "hardware defined add to end" end;

procedure remove (var p: processptr; var f: FIFO);
begin "hardware defined  remove from beginning" end;

function peek(f: FIFO): processptr;
begin "hardware defined look at first in FIFO" end;

function return_first_bit(start: indexorzero; stream:
                          bitstream): indexorzero;
"This function returns the index of the first nonzero bit
 in the bitstream set.  The initial bit tested is
 specified by the start parameter.  From the first
 test, tests proceed sequentially."

var next: indextype;
begin
  if stream = [] then
    return_first_bit:= 0
  else begin
    next:= start;
    repeat
      next:= succ(next mod maxindex);
    until next in stream;
    return_first_bit:= next;
  end;
end; "return_first_bit"

procedure update_slices(var p: processptr;
                        interval: slice);
"This procedure propagates the time interval executed
 up the tree from the process p, updating all of the
 affected timeslices."

var nextup: readyptr;
begin
  if p <> nil then with p^ do begin
     runtime:= runtime + interval;
     nextup:= readylist^.parent;
     while nextup^.parent <> nil do begin
        with nextup^ do
           if organization = timeslice then
              sliceleft:= sliceleft - interval;
        nextup:= nextup^.parent;
     end; "while"
  end; "if"
end; "update_slices"

procedure savestate(var currentprocess: processptr);

"This procedure saves the current process state (if there
```

```
 is a current process) in the process record for that
 process."

begin
  if currentprocess <> nil then
  with currentprocess^ do
     state:= processorregs;
end; "savestate"

procedure process_switch(newprocess: processptr);

"This procedure accomplishes process exchange, saving the
 state of the current process, setting the state of the
 processor to the selected process (newprocess) and
 setting the current process pointer to the new process."
begin
  savestate(currentprocess);
  processorregs:= newprocess^.state;
  currentprocess:= newprocess;
end; "process_switch"

procedure schedule(last_process: processptr);

"This procedure accomplishes the scheduling according
 to the levels of priority and timesliced scheduling.
 First the timeslices are updated for the last running
 process. The ready tree is then traversed, picking the
 correct process to be scheduled according to the given
 policy. If this process is not the current process, a
 process exchange takes place.  The interval (elapsed)
 timer is then reset."
var level: readyptr; index: indexorzero;
    picked: processptr;
begin
  update_slices(last_process, interval_timer);
  level:= root;
  curslice:= absmaxslice;
  while level^.typethread = lists do begin
     with level^ do
     case organization of
        priority: index:= return_first_bit(0, readybits);
        timeslice: begin
           if (sliceleft <= 0) or not
                (currentindex in readybits) then begin
              currentindex:= return_first_bit(currentindex,
                                   readybits);
              sliceleft:= maxslice;
           end;
           index:= currentindex;
           curslice:= min(curslice, sliceleft);
                  "min function not defined herein"
        end "timeslice"
```

```
        end; "case"
        if index = 0 then "idle"
             "assertion: nobody ready in system"
        else level:= level^.children[index];
  end; "while"
  picked:= peek(level^.ready);
  if picked <> currentprocess then
     process_switch(picked);
  interval_timer:= 0;
end; "schedule"

procedure propagate_empty(ready: readyptr);

"This procedure propagates an empty ready FIFO up the
 ready tree.  Ready is the leaf node.  Bits are reset in
 the path to the root for each node which this transition
 has caused to go non ready"
var nextup: readyptr;
begin
  nextup:= ready;
  repeat
     with nextup^ do
        parent^.readybits:= parent^.readybits
                          - [index_in_parent];
     nextup:= nextup^.parent
  until (nextup^.readybits <> []) or
         (nextup^.parent = nil);
end; "propagate_empty"

procedure propagate_nonempty(ready: readyptr);

"This procedure propigates ready up the tree.  Ready is
 the leaf node to which an entry has been appended.  Bits
 are set in readybits for the path from this leaf to the
 root."
var nextup: readyptr;
begin
  nextup:= ready;
  repeat
     with nextup^ do
        parent^.readybits:= parent^.readybits
                         or [index_in_parent];
     nextup:= nextup^.parent;
  until nextup^.parent = nil;
end; "propagate_nonempty"

procedure "entry" timeslice_complete;

"This procedure is entered on the occurrence of a timing
 signal signifing the completion of a silce.  It
 calls the schedule procedure"
begin
```

```
"assertion curslice = 0"
  schedule(currentprocess);
end;  "timeslice_complete"

procedure "entry" transition_blocked_to_ready
                        (var p: processptr);
"This procedure is called on the occurrence of an event
 which causes a transition from blocked to ready.
 P is the process which was blocked. P is appended to the
 FIFO to which it points, and ready is propagted up
 the tree. Schedule is then called."
begin
  append(p, p^.readylist^.ready);
  propagate_nonempty(p^.readylist);
  schedule(currentprocess);
end; "transition_blocked_to_ready"

procedure "entry" transition_running_to_blocked
        ("var block: blockingstructure;"
          newready: readyptr);
"This procedure is called when the decision is made to
 block the running process.  The parameter newready
 is the ready list to which the process will be appended
 when unblocked.  Block is the blocking structure
 (not described in this algorithm)"
var last_process: processptr;
begin
"assertion: currentprocess has state in processorregs,
 peek(currentprocess^.ready) = currentprocess"

  savestate(currentprocess);
  "enter(currentprocess, block);
     not defined in this algorithm"
  remove(last_process, currentprocess^.readylist^.ready);
  if empty(currentprocess^.readylist^.ready) then
     propagate_empty(currentprocess^.readylist);
  currentprocess^.readylist:= newready;
  currentprocess:= nil;
  schedule(last_process);
end; "transition_running_to_blocked"

begin "initialization";
  "currentprocess:= initial_process;"
  "root:= initial_process_organization;"
end.
```

AN ARCHITECTURAL BASE FOR CONCURRENT PASCAL

by

DAVID NICHOLAS NEAL

B.A., University of Minnesota, 1972

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1977

Abstract

The programming language Concurrent PASCAL in its design and implementations has exerted a substantial influence upon the fields of operating systems and concurrent programming. The work reported in this thesis extends that influence to the field of computer architecture by analyzing the model of concurrency which supports Concurrent PASCAL. As background to the architectural model, three implementations of Concurrent PASCAL are discussed, including a description of the process of transporting an implementation from one computer to another with its associated insights and problems. Details of the architectural base include discussions of the control and data models. The control model discussion centers around state transitions and scheduling. The data model presents a hardware stack mechanism for the execution of Concurrent PASCAL programs, which is also suitable other block-structured languages within the framework of the concurrent processing.